Towards the Layout of Things

Peter Achten
Institute for Computing and Information Sciences
Radboud University Nijmegen
P.O. Box 9010
Nijmegen 6500 GL, The Netherlands
p.achten@cs.ru.nl

Jurriën Stutterheim
Institute for Computing and Information Sciences,
Radboud University Nijmegen
P.O. Box 9010
Nijmegen 6500 GL, The Netherlands
j.stutterheim@cs.ru.nl

Bas Lijnse
Institute for Computing and Information Sciences,
Radboud University Nijmegen
P.O. Box 9010
Nijmegen 6500 GL, The Netherlands
b.lijnse@cs.ru.nl

Rinus Plasmeijer
Institute for Computing and Information Sciences,
Radboud University Nijmegen
P.O. Box 9010
Nijmegen 6500 GL, The Netherlands
rinus@cs.ru.nl

ABSTRACT

When writing a user interface (UI), the layout of its elements play an important role. Programmers should be able to specify the layout of UIs in an intuitive way, while being able to separate the concern of laying out the UI from the rest of the software implementation. Ideally, the same layout language can be used in multiple application domains, so the programmer only has to learn one set of layout concepts. In this paper we introduce such a general-purpose layout language. We obtain this language by abstracting from a layout language we have introduced in previous work for declaratively defining Scalable Vector Graphics (SVG). We show that this abstract layout language can be instantiated for multiple domains: the SVG library by which the language is inspired, *ncurses*-based text-based user interfaces, and *iTasks*. In all of these cases, a separation of concerns is maintained.

CCS CONCEPTS

• **Computing methodologies** → *Computer graphics*;

KEYWORDS

Graphical User Interfaces, Task Oriented Programming, GUI layout, separation of concerns

ACM Reference format:
Peter Achten, Jurriën Stutterheim, Bas Lijnse, and Rinus Plasmeijer. 2017. Towards the Layout of Things. In *Proceedings of IFL Conference, Leuven, Belgium, August 31-September 2, 2016 (IFL 2016),* 13 pages.
DOI: http://dx.doi.org/10.1145/3064899.3064905

IFL 2016, Leuven, Belgium

DOI: http://dx.doi.org/10.1145/3064899.3064905

1 INTRODUCTION

Every user interface (UI) consists of a collection of possibly interactive UI elements. The layout of the UI can significantly influence the user experience. Being able to lay out appealing user interfaces is therefore important for user-facing software. Ideally, defining such UIs is easy to do for a programmer, and can be done while maintaining a separation of concerns from the business logic of a program.

In previous work (Achten et al. 2014), we have introduced a Scalable Vector Graphics (Dahlström et al. 2011) (SVG) library called *Graphics.Scalable*. With this library, one can create SVG images in a purely compositional way by combining basic SVG elements using a small set of *layout combinators*. Only three basic layouts were specified: *collages*, *overlays*, and *grids*. In a collage, each SVG element, which in turn may be a layout of SVG elements, is given an absolute position, while in an overlay the individual elements can be aligned relative to a parent container. A grid layout can be used to place SVG elements in rows and columns. The SVG library also features two derived layout combinators *above* and *beside*, which are defined as grids of 1 column and row, respectively.

Defining SVG images with these layout combinators turned out to be very practical. In fact, we also wanted to be able to express the layouts of our *iTasks* framework (Lijnse 2013; Plasmeijer et al. 2007) in the same terms. Rather than implementing a new layout language, we opted to *abstract* from our original layout language and use this abstract language to implement the layout combinators for both our SVG library and *iTasks*. At the same time, the new abstract layout language must also be powerful enough to capture other domains. This gave rise to the following questions. How should one specify the spatial *layout* of visual, possibly interactive, components in your program? How should one separate the concern of maintaining the life-cycle of UI components from their chosen layout? How do you *identify* UI components? In this paper we propose a general purpose solution to these challenges.

For application domains that provide direct access to the UI components, it suffices to instantiate the overloaded layout language. Many application domains do not provide direct access to their UI components, however. For instance, in most widget-based APIs the program must first call *handle*-returning actions and use the

obtained *handle* values to control the life-cycle and layout of the created UI components. Thus, the concerns of creating UI components on one hand versus arranging their layout on the other hand is not well separated.

To still provide a separation of concerns, we introduce a way to perform pattern matching on a specific part of the structure of the program. An *annotation function* is applied to this part, which takes an abstract representation of the program's structure over which layout can be specified. Pattern matching on this abstract representation can then be used to identify individual user-interface components. The annotation function returns a layout definition containing some or all of the UI components. With this annotation approach, the specification of layout can be decoupled from the identification of UI components. Naturally, if the annotated program fragment is changed, the identification code must also be re-considered. However, provided that the collection of identified UI components remains identical, this does not affect the layout specification. This also holds the other way around: changing the layout specification does not affect the identification code.

The proposed combination of overloaded layout language and function-annotation works for completely different application domains. We demonstrate this with the following case studies:

(1) The *Graphics.Scalable* library of *iTasks* is the obvious first candidate to consider because the overloaded layout language was derived from it. Because it provides direct access to its UI components, scalable images, it suffices to instantiate the overloaded layout language.
(2) The *ncurses* library[1], available as a *Haskell* package[2], is the second case. With *ncurses* terminal-style "GUI" applications can be created. As with the previous case, *ncurses* provides direct access to the UI components, so it suffices to instantiate the overloaded layout language. However, the application domain is quite different as the programmer needs to divide the available screen estate to the appropriate UI components.
(3) The third case is *iTasks*, but now we wish to arrange the layout of the automatically generated UIs of entire tasks and task compositions. This is an illustrative case of a domain in which the code annotation is required to identify the task UIs that need to be provided with a layout.

We implement the overloaded layout language in both Clean (Plasmeijer and van Eekelen 2002) and Haskell (Peyton Jones 2003). Both iTasks-related case studies are implemented in Clean, while the *ncurses* case is implemented in Haskell.

The remainder of this paper is structured as follows. The overloaded layout language is introduced in Section 2, and the code annotation in Section 3. Section 4 up to Section 6 contain the three above-mentioned case studies. In Section 7 we analyse the properties of this approach and characterize for what kind of systems this is a viable solution. Related work is described in Section 8. The conclusions are found in Section 9.

[1] https://www.gnu.org/software/ncurses/ncurses.html
[2] http://hackage.haskell.org/package/ncurses

```
triptych (cs, rs1, rs2) (a, b, c, d, e)
  = beside [] cs []
      [ above (repeat AtRight) rs1 []
              [a, b] Nothing
      , c
      , above (repeat AtLeft)  rs2 []
              [d, e] Nothing
      ] Nothing
```

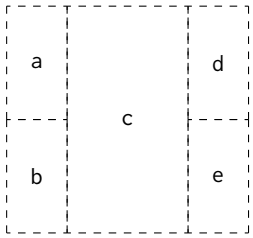


Figure 1: The triptych layout specification

2 THE GENERALIZED LAYOUT LANGUAGE

In this section we introduce the generalized layout language. To get an intuitive understanding of the kind of layouts we wish to be able to specify, Section 2.1 starts with two examples. The layouts in these examples will be used throughout the rest of the paper in various layout domains, without modifications. Important to note here is that different domains typically require arguments of different types for the layout functions. As such, reusable layout specifications need to be parametrised by these arguments as well, rather than just the individual elements that are to be positioned in the layout.

2.1 Layout language examples

Example `triptych` (Figure 1) illustrates a grid-based layout of a fixed set of items. A triptych consists of a large centre pane `c`, and two side panes. The centre pane and the side panes are placed beside one another using the `beside` layout. The side panes are usually, but not always, half the width of the centre pane. We divide each side pane into two sub-panes `a`, `b` and `d`, `e`. Sub-panes `a` and `b` are placed above one another using the `above` layout, and so are sub-panes `d` and `e`. Sub-panes `a` and `b` are right-aligned, while sub-panes `d` and `e` are left-aligned. It is the responsibility of the application domain to express the correct placements of the elements within the `beside` and `above` layouts. This is done via the parameters `cs`, $\texttt{rs}_1$, $\texttt{rs}_2$. The type of these parameters and the way in which they influence layout can differ per domain.

The second example, `rolodex` (Figure 2), illustrates an overlay-based layout of an arbitrary number of elements. In an `overlay` layout, elements at the front of the list are placed underneath the elements later in the list. The middle item, `c`, must be displayed closest towards the viewer, so it must be placed at the very end of the list of elements of the layout function. The preceding items, `as`, are displayed above `c` on the y-axis and at increasing horizontal offset (x_0, $x_0 + x_1$, $x_0 + x_1 + x_2$, ...) and increasing upward offset (y_0, $y_0 + y_1$, $y_0 + y_1 + y_2$, ...). Similarly, the subsequent items, `bs`\``, are displayed below `c` on the $y$-axis and at increasing horizontal and downward offset. In order to create the correct $z$-axis ordering, `bs`\`` is in reversed order (`bs`) in the list of elements of `rolodex`.

2.2 Implementing the generalized layout language

We generalize the original layout language from *Graphics.Scalable* by means of several type classes, which we will introduce in the rest of this section. Every layout is a function that arranges a finite list of

```
rolodex _ _ []
  = overlay [] [] [] Nothing
rolodex x y things
  = overlay
    (  repeatn na (AtLeft, AtTop)
    ++ repeatn nb (AtLeft, AtBottom)
    )
    (  reverse (take na (zip2 x` y``))
    ++ reverse (take nb (zip2 x` y` ))
    )
    (  as ++ bs ++ [c] )
    Nothing
  where
  n             = length things
  na            = n / 2
  nb            = n - na - 1
  (as,[c:bs`])  = splitAt na things
  bs            = reverse bs`
  x`            = tl (scan (+) zero x)
  y`            = tl (scan (+) zero y)
  y``           = tl (scan (-) zero y)
```

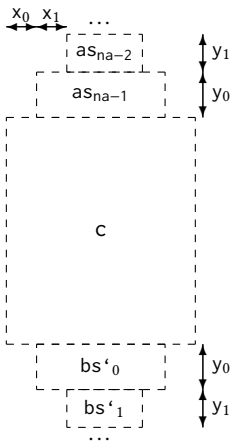


Figure 2: The rolodex layout specification

elements, or *things*, into a new, composite, *thing*. The arrangement is specified by means of a list of *offsets*, that correspond one-by-one with the list of things. If the list of offsets happens to be shorter, then it is padded with default values (*zero* in case of offsets), and if it happens to be longer, then it is truncated. This is a general design guideline throughout the layout language. The meaning of the offsets is determined by a *host*. If a host is present, its coordinate system is used. If it is absent, then the coordinate system is found by taking the bounding box of the dimensions of all things. This amounts to the following type of every layout function:

```
:: LayoutFun offset thing host m
     :== [offset] -> [thing m] -> Maybe (host m) -> thing m
```

The types of *host* and *thing* are parameterized with type variable *m* to accommodate the expected instances. The coordinate system is conventional: the x-axis increases towards the right of a display area, and the y-axis increases towards the bottom of a display area. There is an implicit z-axis that increases towards the viewer. By convention, elements that occur at a higher index in the list of things have a higher z-value, and can thus obscure elements at a lower index position.

The core of the layout language is formed by two multi-parameter type classes: `Overlay` and `Grid`. We introduce two separate type classes because not every application domain handles the two key concerns: ordering things along the z-axis and ordering things within the x- and y-plane. Typically, application domains that do not handle overlapping elements will not support the `Overlay` language. An application domain instantiates the classes by choosing types for the *thing*s, *dim*ensions, *offset*s, and *host*. The *thing* type determines the other types, which is denoted in *Clean* by means of a functional dependency by the prefix ~.

The main concern of the `Overlay` class member functions is to control the layout of things in the z-axis, using the implicit ordering in its list of things:

```
class Overlay thing ~offset ~host where
  overlay :: [(XAlign, YAlign)] -> LayoutFun offset thing host m
  // derived members:
  collage ::                       LayoutFun offset thing host m

:: XAlign = AtLeft | AtMiddleX | AtRight
:: YAlign = AtTop  | AtMiddleY | AtBottom
```

Here, the functional dependency reads as *thing uniquely identifies offset* and *host*. In other words, once the type system knows the type of *thing*, it knows from the functional dependency what the types of *offset* and *host* are. An (`overlay` *aligns offsets things host*) first aligns every $things_i$ according to $aligns_i$ with respect to *host*. The default value for *aligns* is (`AtLeft,AtTop`), and the list is either padded or truncated to match the length of *things*. Second, the position of $things_i$ is tuned with $offsets_i$. The `collage` class member function is a convenience function that has default implementation `collage = overlay []`: the placement of its elements is dictated by the implicit z-axis and their offsets.

The main concern of the `Grid` class member functions is to control the x- and y-axis.

```
class Grid thing ~dim ~offset ~host where
  grid   :: GridDimension GridLayout
            [(XAlign, YAlign)]
            [dim] [dim]
         -> LayoutFun offset thing host m
  // derived members:
  beside :: [YAlign] [dim] -> LayoutFun offset thing host m
  above  :: [XAlign] [dim] -> LayoutFun offset thing host m

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

A (`grid` *dim layout aligns cols rows offsets things host*) places *things* in a grid structure. Its number of columns and rows is determined by *dim*. The order of grid-cells that are selected is determined by *layout*: the `GridMajor` value dictates whether this occurs column-by-column or row-by-row; the `GridXLayout` value determines if the grid is subsequently filled from left-to-right or right-to-left, and finally, the `GridYLayout` value determines if the grid is subsequently filled from top-to-bottom or bottom-to-top. Every $things_i$ is aligned within its grid-cell according to $aligns_i$ (default value is (`AtLeft,AtTop`)). The *cols* (and *rows*) lists add additional constraints on the widths (heights) of the columns (rows). The application domain determines what the default value is. The final position of $things_i$ is obtained by tuning with $offsets_i$.

`Grid` also has convenience layout functions with default implementations:

```
beside as cs
  = grid (Rows    1) (RowMajor,    LeftToRight,TopToBottom)
                     [(AtLeft,a) \\ a <- as] cs []
```

```
above as rs
  = grid (Columns 1) (ColumnMajor, LeftToRight,TopToBottom)
                     [(a, AtTop) \\ a <- as] [] rs
```

In general, a layout language needs a means to refer to the size of its components and perform computations on them (add, subtract, maximum, minimum) in order to construct a correct composition. This requires a *tag* type to identify the component, and a *dim* type that represents the size of the component. This is captured with the multi-parameter type classes `TagOf` and `DimRef`:

```
class TagOf thing ~tag where
  tagOf :: thing -> tag

class DimRef tag ~dim where
  xdim  :: tag     -> dim
  ydim  :: tag     -> dim
  cdim  :: tag Int -> dim
  rdim  :: tag Int -> dim
```

The expression (`tagOf` *thing*) retrieves the tag of *thing*. It is the responsibility of the application domain to assign to each thing one unambiguous tag (the system tag). The application domain can optionally offer a `tag` function to add custom tags to things. Hence, in general, a thing is associated with a non-empty collection of tags. The expressions (`xdim` t) and (`ydim` t) refer to the x-width and y-height of the thing that is tagged with t. The expressions (`cdim` t i) and (`rdim` t j) refer to the x-width of the i-th column or the y-height of the j-th row of the thing that is tagged with t. Tag-expressions are symbolic references. It is the concern of the implementation to take this into account when computing with such values. Unmatched tag expressions always have value *zero*.

Computations with dimensions come with instances for common overloaded arithmetic functions `+`, `-`, `abs`, and `~` (negation). Slightly less usual are the overloaded functions `zero`, `*.`, `/.`, `min`, and `max`. `zero` produces a zero-like value, such as 0 or 0.0 for integers and real numbers, respectively. Expressions (d `*.` k) and (d `/.` k) multiply and divide a dimension d with a scalar value k. Expressions (`min` *ds*) and (`max` *ds*) take the minimum and maximum of a list *ds*.

3 RELATING UI COMPONENTS TO LAYOUT

Not every application domain is suited to instantiate the overloaded layout language of things. There are several issues that make it complicated to do this:

(1) In most widget-based API's the program must first call *handle*-returning actions and use the obtained *handle* values to control the life-cycle and layout of the created UI components. As a result, the layout expression language becomes a layout action language that is interleaved with the UI action language.
(2) One cannot use the UI creation code to identify the resulting UIs because each call of the same code results in a newly created UI instance whose appearance, content, and size diverges over time when compared with the other instances.
(3) Most applications require a UI that evolves dynamically over time: the number of windows, panels, items keep changing to best reflect the current application state and needs of the users. Hence, the layout specification needs to be dynamic as well.
(4) To counter the above limitation, many UI toolkits offer an API to inspect the widget-structure or DOM-structure at run-time. However, these APIs can break the abstraction barrier that is intended by the implementer of UI components.
(5) Some UI approaches implement an automatic layout algorithm while other approaches leave the specification of layout entirely to the programmer. In the first case, the application developer might want to overrule the layout of a particular piece of code and leave other pieces as-is. The solution should work for all of these cases.

We counter the above issues in the following way. First, to deal with issue 5, we introduce a code-annotation. Code without an annotation behaves as dictated by the application domain. Annotated code gets overruled by the specification within the annotation. The annotation specification is a function. This deals with issues 4 and 3. The function is provided with information of the current collection of UI components. The application domain is responsible for providing the information, and can thus protect the program against breaking the abstraction barrier. To resolve issues 2 and 1, we observe that UI components are always organized in a hierarchical way (for instance, windows containing child elements, some of which can be panes that contain further child elements, and so on). The argument of the function-annotation is a rose tree parameterized with the type of UI components of the application domain. The application domain defines the relation between the annotated code and the rose tree.

The rose tree structure of UI components is defined as follows:

```
:: UITree tag = UILeaf tag | UINode tag [UITree]

tagOfUI :: (UITree tag) -> tag
```

The `tag` type parameter uniquely identifies the nodes and leaves of the rose tree. It is the first type parameter of every `DimRef` type class instantiation. The trivial access function `tagOfUI` simply returns the tag found at the root of its argument. The application domain decides which compositions of UI components can be decomposed (`UINode`) or are considered to be atomic (`UILeaf`).

The rose tree structure is the domain of the function annotation. The range type depends on the application domain: types need to be defined to instantiate the type classes `Overlay`, `Grid`, `TagOf`, and `DimRef` of the overloaded layout language. If type T is the type of *things* of a particular application domain, then the layout function has type:

```
my_layout :: (UITree tag) -> T m
```

In this way, the layout language is open ended to allow an application domain to add further constructor functions to create UI elements or transformations specific for that domain. For instance, in SVG based approaches you wish to support rotation and skewing, and in widget based GUIs, you wish to support panels that can be scrolled or resized.

The missing link connects the UI rose tree with the domain of things via the overloaded function `uiOf`:

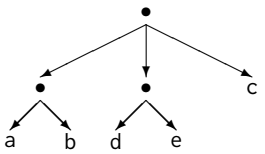
```
class UIOf thing ~tag where
```

```
uiOf :: (UITree tag) -> thing m
```

3.1 UI pattern examples

The first example applies the `triptych` layout to a piece of code. Hence, the annotation needs to identify five UI items. If we know that the program is structured as shown in the image to the right, then the corresponding layout definition can be defined as follows:

```
example1 (cs, rs1, rs2)
  (UINode _ [ UINode _ [a,b]
            , UINode _ [d,e]
            , c
            ])
  = triptych (cs, rs1, rs2)
             (a`,b`,c`,d`,e`)
  where
  [a`,b`,c`,d`,e`] = map uiOf [a,b,c,d,e]
```



In this example we pattern match on a `UITree` value to identify sub-components in the original UI tree. Important note is that sub-layouts `a`, `b`, `c`, `d`, and `e` can be arbitrarily complex user interfaces themselves. We maintain manageability by not pattern matching to the leafs of the tree.

The second example collects all leaf UI items of an annotated piece of code and applies the `rolodex` layout to them.

```
example2 ui = rolodex (repeat (w /. 20))
                      [h /. k \\ k <- [2 ..]] uis
  where
  uis   = uisOf ui
  n     = length uis
  ui    = uis !! (n / 2)
  tag   = tagOf ui
  w     = xdim tag
  h     = ydim tag

uisOf :: (UITree tag) -> [thing m] | UIOf thing
uisOf (UINode _ ts) = flatten (map uisOf ts)
uisOf ui            = [uiOf ui]
```

4 THE LAYOUT OF GRAPHICS.SCALABLE

The overloaded layout language defined in Section 2 is a generalization of the original layout language of *Graphics.Scalable*. As a consequence, this section is brief, and serves mainly as an overview of the two tasks that have to be performed to apply the overloaded layout language to a new domain: identify the domains and the constructor functions.

4.1 Graphics.Scalable domains

First we define the domains that the type classes `Overlay`, `Grid`, `TagOf`, and `DimRef` are instantiated with. These are: the *domain of things*, *dimensions*, *offsets*, *hosts*, and *tags*. Their type definitions are:

```
:: Image m                      // domain of things
:: Span                         // dimensions
:: Offset :== (Span, Span)      // offsets
:: Host m :== Image m           // hosts
:: ImageTag                     // tags
```

The domain of things in *Graphics.Scalable* is captured with the opaque `Image m` type. Every image is infinitely large and perfectly transparent. There is no global coordinate system. With each image a *span box* is associated relative to which visual content is rendered. The dimensions are captured with the opaque `Span` type. Although span values are defined most of the time with `Real` values, they get a 'physical' pixel-based interpretation only when an image gets rendered at a client device. Offsets are a pair of a horizontal and vertical span value. The host is an image that serves as the 'background' image, and its span box is used to deal with the alignments and offsets. Finally, tags are captured by the opaque `ImageTag` type.

With these domains, we obtain the layout language of images:

```
instance Overlay Image Offset Host
instance Grid    Image Span Offset Host
instance TagOf   Image ImageTag
instance DimRef  ImageTag Span
```

Their implementations map to the existing implementation of *Graphics.Scalable*.

4.2 Graphics.Scalable constructor functions

The next step is to define the application domain dependent constructor functions of the domain types. For `Image` these are the common shapes: rectangles, circles, ellipses, lines, and text. Except for text, these shapes are defined via their span box (where circles require only the diameter). The image of a text is defined with:

```
text :: FontDef String -> Image m

:: FontDef = { fontfamily  :: String
             , fontysize   :: Real
             , fontstretch :: String
             , fontstyle   :: String
             , fontvariant :: String
             , fontweight  :: String
             }
```

`FontDef` captures the standard SVG attributes to define a font. The y-span of the text is defined by the `fontysize` field, using a real number. However, the x-span of a text, rendered with a given font, is determined by the client device. This is a major complication when dealing with the layout of text. In *Graphics.Scalable*, the function `textxspan` is a symbolic span-expression that represents the x-span of the given text, when rendered with the given font, on the current client device.

```
textxspan :: FontDef String -> Span
```

It should be noted that the SVG image transformation functions (rotate, skew, flip) and image rendering attributes (stroke, opacity) do not alter the span box of the transformed image, but only their rendering. Hence regarding layout, they are irrelevant. However, this is not the case for the scaling functions:

```
fit  :: Span Span (Image m) -> Image m
fitx :: Span      (Image m) -> Image m
fity :: Span      (Image m) -> Image m
```

(`fit` x y *img*) guarantees that the result has precisely width x and height y. (`fitx` x *img*) guarantees that the result has precisely width

Figure 3: Rendering of the `card` image in *Graphics.Scalable*

Figure 4: Screenshot of the triptych example in *Graphics.Scalable*

x, and derives height y proportionally to the current size of *img*. Similarly, (`fity` y *img*) guarantees that the result has precisely height y, and derives width x proportionally to the current size of *img*. These `Span`s can be constructed with, amongst others, the `px` function:

```
px :: Real -> Span
```

The final application domain dependent constructors concern the opaque `ImageTag`. In *Graphics.Scalable*, the top level function to create an image of a server-side value of type `s` and client-side value of type `m` has type:

```
s -> m -> *[(ImageTag, *ImageTag)] -> Image m
```

The list of image tag values is infinitely long and is generated by the image library implementation. The two tags in the tuple are the same tag, but they make different use of Clean's uniqueness types (Barendsen and Smetsers 1996). With this extension of the type system, types can be annotated with a uniqueness attribute `*`. The attribute guarantees that there is ever only exactly one reference to a given unique value. The uniquely attributed version is used in the function `tag`:

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

to guarantee that it adds the tag to the non-empty tag set of at most one particular image. The shared version of tag can be used arbitrarily many times using the type class `DimRef` member functions.

4.3 Graphics.Scalable examples

Let `card :: Span Span Int -> Image Int` create an image of size $w \times h$ that renders a steelblue 'card' on which the number is printed in white (Figure 3).

We use this function to create a triptych of five cards, using the unmodified triptych layout as defined in Section 2.1, (Figure 4) for some given card size `w` and `h`. In the triptych, we proportionally scale the height of the side-panel cards to half the height of the central card.

```
a = triptych ([],[],[])
             ( fity (h /. 2) (card w h 1)
             , fity (h /. 2) (card w h 7)
             , card w h 42
             , fity (h /. 2) (card w h 4)
             , fity (h /. 2) (card w h 2)
             )
```

Figure 5: Screenshot of the rolodex example in *Graphics.Scalable*

As another example, we create a rolodex of cards 1 through 16 (Figure 5), using the unmodified rolodex layout as defined in Section 2.1. The card widths decrease by steps of $0.08w$, so the horizontal offsets increase by steps of $0.04w$. The vertical offsets increase by $\frac{h}{2}, \frac{h}{3}, \ldots$.

```
b = rolodex (repeat (w *. 0.04))
            [h /. k \\ k <- [2 ..]]
            (  zipWith fitx (reverse (take 7 ws)) as
            ++ [c]
            ++ zipWith fitx ws bs
            )
  where
  cards         = map (card w h) [1 .. 16]
  (as, [c: bs]) = splitAt 8 cards
  ws            = [0.92, 0.84 ..]
```

5 THE LAYOUT OF NCURSES

The *ncurses* library is a well-known library that supports command-line interface (CLI) based user interfaces. It allows rendering and placement of text or glyphs, and allows the programmer to specify how keyboard or mouse interaction should be dealt with. To demonstrate the applicability of the overloaded layout language outside the browser, we have implemented it for *ncurses* as well. In particular, it allows the programmer to layout the glyphs. It does not deal with the interaction. Since *Clean* does not currently have bindings for *ncurses*, we ported the layout library to *Haskell*. This allows us to use the `ncurses` package from Hackage. Porting the layout language to *Haskell* is straight-forward and involves only minor syntactical changes. Therefore, we do not show the *Haskell* definition of the type classes.

5.1 The ncurses domains

The type definitions of the *domain of things*, *dimensions*, *offsets*, *hosts*, and *tags* in *ncurses* are:

```
data CursesElem m              -- domain of things
data CursesSpan                -- dimensions
type CursesOffset = (Int, Int) -- offsets
data CursesHost a              -- no hosts for NCurses
data CursesTag    = ...        -- tags
```

In *ncurses*, a terminal is divided in a grid of mono-spaced characters and glyphs. Each character and glyph takes up a single cell in the grid. A string is simply a sequence of (potentially multi-byte) mono-spaced characters. These properties make doing layout for

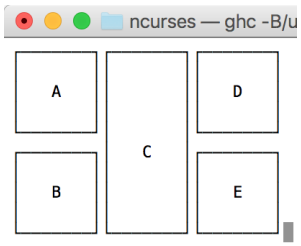


Figure 6: Screenshot of the triptych example in *ncurses*

a terminal easy, since the width and height of each character and glyph is a fixed 1×1. A string is then simply $1 \times n$ in size, where n is the number of characters in the string. In other words, we always know the size of the things that we want to layout. Hence, offsets are integers, each of which represents an on-screen cell. There is no implementation of the `DimRef` class and the host concept for *ncurses*. With the *ncurses* domain types defined, we create the following class instances:

```
instance Overlay CursesElem CursesOffset CursesHost
instance Grid    CursesElem () CursesOffset CursesHost
instance TagOf   CursesElem CursesTag
instance DimRef  CursesTag  CursesSpan
```

The main building block is the `CursesElem`. `CursesElem` values can be created with instances of the `ToCurses` class:

```
class ToCurses a where
  c :: a -> CursesElem ()

instance ToCurses String
instance ToCurses Glyph
instance ToCurses Text
instance ToCurses (CursesElem ())
```

5.2 Examples of ncurses

Using the exact same layout specifications as shown in Section 2.1, we can also lay out *ncurses* elements. Here we will lay out rectangles with a border using the `rect` function.

```
rect :: Int -> Int -> CursesElem () -> CursesElem ()
```

`rect` takes a width and a height, as well as a curses element that is rendered in the centre of the rectangle. Using this function, we can now visualize the triptych example. The triptych code is shown below, with its output in Figure 6.

```
a = triptych ([], [], [])
             ( rect 7 3 $ c "A", rect 7 3 $ c "B"
             , rect 7 8 $ c "C"
             , rect 7 3 $ c "D", rect 7 3 $ c "E")
```

The rolodex example is equally simple. Its code is shown below, with a screenshot of its rendering in Figure 7.

```
b = rolodex (repeat (px 1))
            [px 3, px 2, px 1]
            [ rect 5 3 $ c "1", rect 7 3 $ c "2"
            , rect 9 3 $ c "3", rect 7 3 $ c "4"
            , rect 5 3 $ c "5"]
```

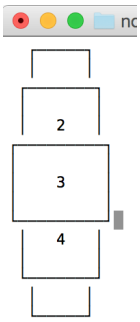


Figure 7: Screenshot of the rolodex example in *ncurses*

6 THE LAYOUT OF TASK UIS

From the perspective of reasoning about layout, the *iTasks* application domain belongs to the 'problem' category discussed in Section 3: tasks are actions for which a UI is generated automatically, the UI is carefully hidden from the application programmer, and the same task gives rise to distinguishable UI instances. Hence, we need to use the function annotation of Section 3 to identify and layout task UI items. The current implementation of *iTasks* does not support overlapping UI items, so it can not support the `Overlay` layout language. It does support the `Grid` layout language.

6.1 Task UI domains

As before, we start to define the *domain of things*, *dimensions*, *offsets*, *hosts*, and *tags* for the *iTasks* application domain. The type definitions are:

```
:: TaskUILayout a                        // domain of things
:: TaskUISize = { minSize     :: UISize // minimum size
                , maxSize     :: UISize // maximum size
                , hasSplitter :: Bool   // user enabled resizing
                }
:: UISize     = FlexSize | WrapSize | ExactSize Int | PctSize Real
                                         // dimensions
:: Offset   :== (Int, Int)               // offsets
:: UIHost m   = InHost                   // hosts
:: TaskUITag                             // tags

class    toUISize a :: a -> UISize
instance toUISize Int                    // convert to ExactSize
instance toUISize Real                   // convert to PctSize
exact       p     = ExactSize p
pct         p     = PctSize (fromInt p)
mkTaskUISize a b c = {minSize = a, maxSize = b, hasSplitter = c}

gDefault{|TaskUISize|} = mkTaskUISize FlexSize FlexSize False
fixUISize   a   = mkTaskUISize (toUISize a) (toUISize a) False
splitUISize a b = mkTaskUISize (toUISize a) (toUISize b) True
```

The opaque type `TaskUILayout` is only used to introduce the member functions of the `Grid` layout language to *iTasks*.

The language of dimensions is much richer than in the previous case studies. As with *ncurses*, layout of task UIs is concerned with dividing screen estate to UI items but without limiting it to the

very simple cell-based approach of terminal-style UIs. By default, the host is divided equally. This can be altered by means of the TaskUISize parameter. It controls the widths of the columns in case of beside, the height of the rows in case of above, and both in case of grid. The minimum and maximum sizes (minSize and maxSize) are specified by means of a UISize value. The default value FlexSize imposes no restrictions, WrapSize is the bounding size of the elements, ExactSize p ($p \geq 0$) is exactly p pixels, and PctSize p ($0 \leq p \leq 100.0$) is $\frac{p}{100}$ of the host size, after rounding to integer pixels. The task layout algorithm computes (column and row) sizes within these constraints. In case the specified minSize < maxSize, the user can be allowed to manually choose a valid size between these values. This is indicated by means of the hasSplitter field. We call a TaskUISize *rubber* if minSize < maxSize and no splitter has been requested, and we call it *splitter* if minSize < maxSize and a splitter has been requested. A splitter user interface element (depending on the client device) is created between column i and $i + 1$ if either the width of column i is *splitter* or if the width of column i is *rubber* and the width of column $i + 1$ is *splitter* (analogous for rows).

Offsets are expressed as pairs of pixels. The UIHost type reflects the twofold purpose of defining task UI layout: *(i)* to assign to each identified task UI a part of the available screen estate (Just InHost), *(ii)* to arrange the relative positioning of the task user interfaces (Nothing). For instance, one can first define the layout of a collection of UI items using Nothing, and thus obtain a composite UI of a certain size, and then place and align it in a smaller part of the screen using (Just InHost).

The TaskUITag type connects the function annotation with the layout. The standard way in *iTasks* to annotate a piece of code is by means of the prefix / postfix tune combinators:

```
class tune b   :: b (Task a) -> Task a
(@>>) infixr 2 :: b (Task a) -> Task a | tune b
(<<@) infixl 2 :: (Task a) b -> Task a | tune b
```

We wrap the function annotation in a new type and turn it into an instance of the tune type class:

```
:: TaskLayout
   = E.a: TaskLayout ((UITree TaskUITag) -> TaskUILayout a)

instance tune TaskLayout where ...
```

Here, the task is annotated with a function that is given a UITree (as defined in Section 3) and produces a layout. With these domains, we obtain the layout language of task UIs:

```
instance Grid   TaskUILayout Int Offset UIHost
instance TagOf  TaskUILayout TaskUITag
instance DimRef TaskUITag UISize
```

6.2 User interfaces in *iTasks*

In this section we look at the internal implementation of user interfaces in *iTasks* in order to identify some of the challenges faced when implementing a layout language for such a sophisticated framework. In *iTasks*, user interfaces and their sub-elements are represented as a rose tree of type UI (shown below). Each node in the tree has a label indicating the type of user interface element, and a map of attributes for that node.

```
:: UI = UI UINodeType UIAttributes [UI]

:: UINodeType
  = UIParallel
  | UIStep
  | UIPanel
  | UIInteract
  | UIButton
  | ..

:: UIAttributes :== Map String JSONNode
```

To minimize network traffic and computation time, *iTasks* updates its user interfaces incrementally, communicating only that what has changed. This incremental communication is represented by the UIChange type, shown below. Changes are applied to the user interface by the applyUIChange function. Its definition is omitted, but its type is included to show which types play a role in user interface updates.

```
:: UIChange
  = NoChange
  | ReplaceUI UI
  | ChangeUI [UIAttributeChange] [(Int, UIChildChange)]

applyUIChange :: UIChange UI -> UI
```

Applying NoChange acts as identity operation, while ReplaceUI simply replaces the entire user interface with a new one. ChangeUI is responsible for updating individual user interface components and is most frequently used.

To apply layout to tasks, we need to integrate the layout language with this user interface update mechanism. Layout is always specified on the static layout defined by the static task composition. At runtime, the layout may change dynamically via the ChangeUI change. These same changes will need to be reflected in the layout, so that the layout rules can also be applied after the user interface has been updated.

The dynamic behavior of user interfaces in *iTasks* complicates the application of layouts. To be able to layout, the components that are being layed out have to exist for the layout to be meaningful. In a fully dynamic setting this can not be guaranteed. We therefore consider the layout language *only* under the following conditions:

- A layout has to be explicitly applied to a part of a UI.
- When a UI is replaced completely with a ReplaceUI change, a layout can rearrange arbitrary sub-UI's into a new UI.
- Subsequent UI changes are only allowed if they modify the content of a sub-UI. If they affect the structure of the UI as transformed by the layout, a run-time error is produced.

With these restrictions the layouting can be achieved as follows. We first consider the ReplaceUI changes. When those occur we uniquely label every node of the UI that is being replaced. Then, using the layout language, we create a UI to UI transformation and apply it to the UI. In the transformed UI we can inspect the labels to build a relocation map that records which parts of the original UI were used and their position in the new UI. On subsequent ChangeUI events, we use the relocation map to detect if the change targets the content of the relocated parts of the UI or the structure

Figure 8: Typical editor GUIs

of the layed out UI. If only the content is affected, we rewrite the `ChangeUI` to target the relocated UI parts.

The current implementation of the task layouts only implements horizontal, vertical and grid layouts, without support for offsets. In other words, we only implement the `Grid` class for *iTasks*. This is not a fundamental limitation, but rather a limitation of the *iTasks* client implementation, which was not designed to work with arbitrary collages. Future versions of the *iTasks* client may add support for free-form layouts.

6.3 Task UI constructor functions

An *iTasks* specification defines the work that needs to be done by end-users and computer systems, each of whom and which have different locations and use different client-devices to perform their work. With each end-user, a collection of tasks is associated. The *iTasks* run-time system collects these tasks, determines the corresponding UI items to be rendered for the particular client device that is used by the end-user at that time, and subsequently assembles a suitable UI for the end-user using a default layout algorithm. There are two classes of tasks that generate UI items: *(i)* the interactive tasks, called *editors*, and *(ii)* task combinators that introduce control items for an end-user to interact with.

6.3.1 Editor tasks. An *editor* is a generic task with which an end-user can *enter*, *view*, or *update* a value of any first-order (custom or pre-defined) type.

```
enterInformation  :: String [EnterOption    m]
                  -> Task m | iTask m
viewInformation   :: String [ViewOption     m] m
                  -> Task m | iTask m
updateInformation :: String [UpdateOption m m] m
                  -> Task m | iTask m
```

A descriptor string is used to inform the end-user of the purpose of this task. The `EnterOption`, `ViewOption`, and `UpdateOption` parameters can be used to provide a custom rendering function for the value of the editor. The use of these options does not influence the way editors are placed in a layout, so we do not discuss them further. With `enterInformation`, the end-user creates a new value of type `m`. In case of the other two editors, an initial value is provided. The UI of `viewInformation` only displays the value, and the UI of `updateInformation` allows the user to alter it. Figure 8 shows typical renderings of these elements (using the descriptors `"enter"`, `"view"`, `"update"` and the initial value `42`):

The above interaction tasks are naturally applied in cases where the value to be interacted with is carried along with the control flow. However, data sources also exist outside of the control flow, and require interaction as well. The following sibling interaction tasks do this:

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

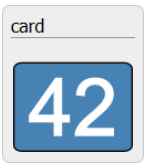


Figure 9: Graphical rendering of cards.

Instead of an initial value to work on, they manipulate a *shared data source* (Domoszlai et al. 2014b). Changes to the shared data source are propagated to all interaction tasks that are connected thusly.

Editors can be customized in case the default rendering is inadequate. This is done via the `(Enter/View/Update)Option` parameter. For any editor one can choose to map the value to another first-order domain that is automatically rendered. A UI can also be defined from scratch for `update(Shared)Information` using *editlets* (Domoszlai et al. 2014a). In particular, it is possible to use an `Image` (Section 4) to render the content of an editor. Here, we customize an `Int` editor to show its content as a `card`:

```
edit_card  = updateInformation       "card" [asImage o card w h]
edit_card` = updateSharedInformation "card" [asImage o card w h]
asImage :: (Image a) -> UpdateOption Int Int
```

The result of either of these editors is shown in Figure 9. How `asImage` can be implemented is explained in the paper that introduces our compositional vector graphics library (Achten et al. 2014).

In all of the above cases, the UI of an editor is accessed as a leaf in the UI rose tree.

6.3.2 Task combinators. All possible ways of collaboration boil down to two core task combinators: *sequential* and *parallel* composition.

Sequential composition, denoted with `>>*` and pronounced as *step*, is basically a generalized, guarded version of the standard monadic `>>=`, *bind*, operator in the presence of task values that evolve over time.

```
(>>*) infixl 1 :: (Task a) [TaskCont a (Task b)]
               -> Task b | iTask a & iTask b
:: TaskCont a b
   =      OnValue         ((TaskValue a) -> Maybe b)
   |      OnAction Action ((TaskValue a) -> Maybe b)
   | E.e: OnException     (e             -> b) & iTask e
   |      OnAllExceptions (String        -> b)
:: TaskValue a = NoValue | Value a Stability
:: Stability :== Bool
```

The UI control elements originate from the guarded `OnAction` task continuations. The `Action` parameter indicates that a user can interact with the application via a clickable user interface element,

such as a button or menu item. These actions are co-located with the UI of the left-hand side task argument of >>*. For the purpose of the layout proposal in this paper, we consider them to be an integral part of the UI that is matched on. The UI belongs to the leaf constructor of the UI rose tree.

Parallel composition, denoted with parallel, captures the collaboration of a (possibly dynamic) number of tasks. The progress between these tasks is accessible to both the participating tasks as well as any external context (such as a guarded task continuation of the >>* combinator). Without going into too much detail, we briefly walk through the signature of parallel:

```
parallel :: [(ParallelTaskType,ParallelTask a)]
            [TaskCont [(TaskTime,TaskValue a)]
                      (ParallelTaskType,ParallelTask a)]
         -> Task [(TaskTime,TaskValue a)] | iTask a
```

The ParallelTaskType governs the end-user-ownership of the task. A task can be *embedded* or *detached*, thus enabling task distribution between co-workers. The ParallelTask is a task function that is provided with access to the current status (task values and meta-information) of the collaborating tasks via a shared data source. The task continuation (TaskCont, see >>* above) can add new tasks to the collection of tasks (tasks can also be removed). For the purpose of the layout proposal in this paper, parallel composition is an ordered sequence of UI items from the perspective of the current end-user. Each UI is a sibling node within the node constructor of the UI rose tree. A UI belonging to another end-user is empty, having dimensions of zero size.

Many task compositions have a simple static structure and can do without the rather elaborate signature and interface of parallel. For instance, some frequently occurring combinations are:

```
(-&&-) infixr 4 :: (Task a) (Task b) -> Task (a,b) | iTask a
                                                   & iTask b
(-||-) infixr 3 :: (Task a) (Task a) -> Task a     | iTask a
allTasks        :: [Task a]          -> Task [a]   | iTask a
anyTask         :: [Task a]          -> Task a     | iTask a
```

(t_1 -&&- t_2) evaluates two tasks in parallel until both have a stable task value, and allTasks generalizes this to a list of tasks. (t_1 -||- t_2) evaluates two tasks in parallel until either one has a stable task value, and anyTask generalizes this to a list of tasks. In these cases, each of the task UIs are retrieved via the node constructor of the UI rose tree.

For any derived task combinator, the corresponding UI rose tree structure must be documented.

6.4 Task UI examples

We use the triptych layout specification to place four interactive tasks with which the end-user can edit integer values, around a task that displays the sum of these values as a blue card, using edit_card`). Analogous to customizing the card tasks, we introduce interactive tasks that edit a particular element of a list of values at some index location i:

```
edit_elt  i = updateInformation       ("edit␣" +++ toString i)
                                      [upd_elt i]
edit_elt` i = updateSharedInformation ("edit␣" +++ toString i)
                                      [upd_elt i]
```

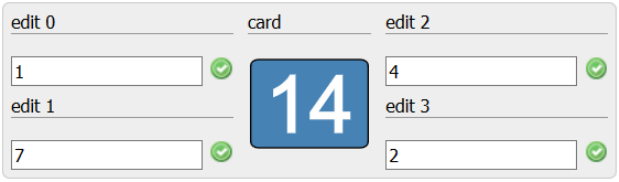


Figure 10: Screenshot of the triptych example in *iTasks*

```
upd_elt   i = UpdateWith (flip (!!) i) (flip (updateAt i))
```

We define the task structure as follows:

```
task_triptych :: (ReadWriteShared [Int] [Int]) -> Task [Int]
task_triptych sds
    = edit_card` sum sds
        -||-
      anyTask [edit_elt` i sds \\ i <- [0 .. 3]]
```

The desired task layout is obtained by adding the following layout annotation to the above expression:

```
withShared [1,7,4,2] task_triptych <<@ TaskLayout my_layout

my_layout :: (UITree TaskUITag) -> TaskUILayout a
my_layout (UINode _ [c, UINode _ [a,b,d,e]])
   = triptych ([],[],[]) (a`,b`,c`,d`,e`)
   where
   [a`,b`,c`,d`,e`:_] = map uiOf [a,b,c,d,e]
```

Figure 10 shows the resulting task UI layout.

To illustrate the flexibility of the approach, suppose that somebody wishes to exploit the following equivalence:

```
t -||- anyTask ts = anyTask [t : ts]
```

and thus alters task_triptych as follows:

```
task_triptych` :: (ReadWriteShared [Int] [Int]) -> Task [Int]
task_triptych` sds
   = anyTask [edit_card` sum sds : edit_elt` i sds \\ i <- [0 .. 3]]
```

Although the programs are equivalent, their structure is different. We only need to alter the pattern match accordingly:

```
withShared [1,7,4,2] task_triptych` <<@ TaskLayout my_layout`

my_layout` :: (UITree TaskUITag) -> TaskUILayout a
my_layout` (UINode _ [c,a,b,d,e])
   = triptych ([],[],[]) (a`,b`,c`,d`,e`)
   where
   [a`,b`,c`,d`,e`:_] = map uiOf [a,b,c,d,e]
```

to obtain the same desired layout of task UIs.

To illustrate a layout of task UIs that deploys the much richer language of dimensions, here is an example of an irregular layout that occurs often in the *iTasks* system itself (in Figure 11 a dashed line indicates a splitter).

```
my_layout (a,b,c,d,e)
  = beside [] [splitUISize (pct 15) (pct 85)] []
      [ above [] [defaultSize, splitUISize (pct 10) (pct 30)] []
              [a, b] (Just InHost)
      , above [] [ fixUISize (pct 10)
```

```
            , splitUISize (pct 20) (pct 40)] []
        [c, d, e] (Just InHost)
] (Just InHost)
```

UI element `b` can be resized by the user between 10% and 30% of the height of the left column. The top bar, `c` has a fixed height of 10% of the height of the right column. UI element `d` can be resized by the user between 20% and 40% of the height of the right column.

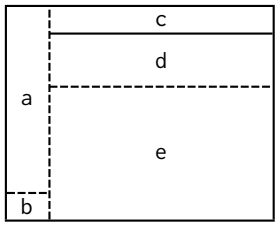


Figure 11: An irregular layout of tasks `a` ... `e`

7 ANALYSIS

In this section we justify the claim that we have proposed a "general purpose solution" for the challenges of specifying spatial layout of UI items, separating the concern of maintaining the life-cycle of UI items from their chosen layout, and identifying UI items.

In application domains in which every constructor function has the property that identical calls yield indistinguishable UI items, it is sufficient to instantiate the overloaded layout language. The application domains *Graphics.Scalable* (Section 4) and *ncurses* (Section 5) satisfy this property. In (Achten et al. 2014) we explain how *Graphics.Scalable* has been inspired by the mature *Racket* image API (Felleisen et al. 2009; image.rkt 2014). For this application domain the overloaded layout language can also be instantiated. There is an interesting, and deliberate difference, between the ways image dimensions are handled. In the overloaded layout language, the class `DimRef` introduces a tagging system to identify images of which dimensions need to be found. In *Racket*, this is done more directly. Paraphrasing its `image-width` function signature in *Graphics.Scalable*:

```
image-width :: (Image m) -> Int
```

Such a function only makes sense in a context where image-generating functions yield indistinguishable images. An implementation of the `DimRef` member functions can use the images themselves for the tags.

In application domains that rely on action-based constructor functions, more effort is required to integrate it with the overloaded layout language. Virtually every widget-based library is action-based: the programmer is required to call *handle*-creating actions that have as immediate side-effect that the corresponding widget object is created. The returned handle is used later on in the program to alter its properties, such as dimensions, position, stacking order, visibility, accessibility, and finally, to delete it (manually, or via a finalizer mechanism). Instead of immediately having a side-effect and create a widget, the implementation of such approaches should be altered to create an *intermediate representation of the widget*, or, if the back-end allows it, create an invisible widget. The intermediate representation can then be used afterwards to apply the layout language to. Once the layout has been computed, either the widgets can be created at the correct positions, sizes, and stacking order using the intermediate representation or made visible after settings its other properties.

In this context, *Clean* Object IO (Achten and Plasmeijer 1995; Achten and Wierich 2000) takes position between these two extremes: it offers both a declarative GUI representation language that can serve as an intermediate representation language *and* it offers an action-based API to create any of these GUI elements. The GUI representation language of Object IO uses a rather complicated layout language and can be replaced entirely by the overloaded layout language described in this paper. Its action based functions can be altered as described above, allowing the code annotation to be introduced. The UI rose tree that is required is a rather straightforward projection of the hierarchical structure of the intermediate representation.

Finally, the *iTasks* case study demonstrates that the approach is applicable also for systems in which no handles, or similar values, are created.

The code annotation works for any application domain, regardless whether it deploys an existing layout strategy or none at all, leaving layout at the discretion of the UI constructor functions. When the implementation has been altered to a two-phase process as described above, it is clear which elements get influenced by the code annotation. For these elements, the new layout can be computed and protected against further manipulation by passing it to other code annotations or layout strategy as a UI rose leaf.

The example in Section 6.4 shows how separation of concerns is achieved in the solution. The code annotation works as a *pivot*. If the task structure is altered, only the UI rose tree pattern-match changes along, and the layout specification is untouched. If the layout specification is altered, only its call within the code annotation changes along, and the task structure is untouched. From this point of view, the approach shares the same advantages and disadvantages as standard pattern-matching in functional programming languages.

As shown in this paper, layout specifications can be reused across multiple domains. This implies that the layouts that specify the relative position of their elements are portable. However, the effort required in porting an application to a different user interface back-end remains largely the same, since the reusable layout specifications need to be parametrised by domain-specific hints.

8 RELATED WORK

In traditional, widget based, GUI libraries, the application code uses actions to generate one GUI component at a time, together with some kind of identification value that must be used to control the life cycle of that GUI component. Examples of such approaches are *Haggis* (Finne and Peyton Jones 1995), *TkGofer* (Claessen et al. 1997), *wxHaskell* (Leijen 2004). In these approaches, GUI component creation and identification is not separated at all. The identification values are required to control the layout of the elements. The layout language is the familiar set of structured layout, i.e. placing GUI components horizontally, vertically, and in a grid. In *Clean* Object IO, the creation of GUI components and their identification is turned

around: pre-conceived identification values are used to identify GUI elements within a shallowly embedded DSL that describes entire, composite, GUI structures. This improves the separation of concerns. In all of these approaches, rules need to be defined when an identification value is used that does not (temporarily or forever) correspond with a GUI element. The relation between identification value and GUI component is *fragile*. The code annotation that we propose in this paper does not suffer from this fragility, but it comes at the expense of introducing a dependency between the concrete task structure and the UI rose tree.

The seminal *Fudgets* (Carlsson and Hallgren 1993) system used a purely combinator based approach to specify GUI applications in order to move away from the traditional widget abstraction towards a functional style of programming. A GUI component, *fudget*, is a stream processor that can be glued together with other GUI components to form a more complex stream processor, using combinators. A default layout algorithm takes care of placing the GUI components. The layout can be tuned at the fudget combinators when the default is not appropriate. Thus, this approach has the similar disadvantage as the old layout mechanism of *iTasks*, viz. that tuning the layout clutters the original fudget structure. The school of *functional reactive programming*, *FRP*, suffers from the same issue. FRP examples are *Fran* (Elliot and Hudak 1997), *FranTk* (Sage 2000), *Fruit* (Courtney and Elliott 2001), and *Yampa* (Courtney 2004). In FRP, a GUI component processes a signal, which is a continuous, time-varying value. GUI components are glued together to process more complex signals. Just like *Fudgets*, layout is specified at the combinator level, and therefore interferes with the original combinator structure.

iTasks is one of many systems that utilize the web infrastructure to create distributed, interactive applications. In *Wash/CGI* (Thiemann 2002), the application developer uses a monadic abstraction to create forms and an identification value to access their content. Therefore, as with traditional widget based approaches, GUI component creation and identification are tightly coupled.

At a higher level of abstraction, we find *Hop* (Serrano et al. 2006) and *Links* (Cooper et al. 2006). *Hop* uses a stratified approach, and offers two separate, cooperating, languages, one for the web server and one for the web clients. *Links*, as *iTasks*, uses a single language approach, but unlike *iTasks*, the application developer needs to use the keywords *server* and *client* to coordinate where which part of the application should be executed. Both approaches differ from *iTasks*, in which the entire program is compiled to target the server, using the 'ordinary' *Clean* code generator, and one to target the client, using a *JavaScript* compiler (Domoszlai et al. 2011). The designers of *Hop* and *Links* have seamlessly blended HTML with functional features to define behavior in a callback style, making it look familiar to developers who know their HTML. In addition, both approaches offer access to the HTML structure via DOM-manipulating functions, providing the developer with low-level access to the created GUI. Low-level access to the GUI can break the final user interface, is sensitive to changes of implementation, and should concern only the interactive task that creates the GUI. In the approach proposed in this paper, the application domain determines which abstraction barriers should be kept intact and can protect against this simply by offering a composite GUI as a leaf value in the UI rose tree.

9 CONCLUSIONS

In this paper we have introduced an overloaded, general purpose, language of *the layout of things*. Instead of inventing yet another layout language, we have generalized the `Graphics.Scalable` image layout language. For some application domains, this suffices to specify the layout of its inhabitants. We have shown this for scalable vector graphics (*Graphics.Scalable*) and a terminal style GUI toolkit (*ncurses*). For application domains in which the inhabitants can only be found indirectly, or even do not exist, we have introduced a code annotation with which the application developer can *pattern match* the structure of UI that is generated, and define an appropriate layout. We have shown this for handling layout of arbitrarily complicated UI components (*iTasks*).

Separation of concerns is achieved in this way in *iTasks*. The task engineer can concentrate on defining the appropriate task structure, knowing that a default task layout is always available. She can replace 'equals by equals', as illustrated in Section 6.4, knowing that it is always possible to define a custom task layout via the code annotation. Tuning the layout of a task involves defining a pattern matching code to find the task UIs and creating a custom task UI layout. If the task structure is changed, then it is likely that the task tree pattern must be changed as well, so there is a price to pay. However, in this way fragile task references do not exist (see Section 8) and the task layout definition does not have to be changed if the collection of task UIs is the same.

In the current proposal we have not dealt with the possibility to introduce 'harmless' UI content such as frames to visually delimit parts of the user interface, or expressive labels to guide the application user, and so on. We conjecture that this can be dealt with via domain specific constructor functions in combination with the constructor functions for the host.

The long term vision is that in the specification of software the way to specify things and their layout are completely orthogonal issues. We think that this paper is a first step towards this goal.

ACKNOWLEDGMENTS

We thank the reviewers for their constructive feedback. This research is partly funded by the Royal Netherlands Navy and TNO.

REFERENCES

Peter Achten and Rinus Plasmeijer. 1995. The ins and outs of Concurrent Clean I/O. *Journal of Functional Programming* 5, 1 (1995), 81–110.

Peter Achten, Jurriën Stutterheim, László Domoszlai, and Rinus Plasmeijer. 2014. Task Oriented Programming with Purely Compositional Interactive Scalable Vector Graphics. In *Proceedings of the 26Nd 2014 International Symposium on Implementation and Application of Functional Languages (IFL '14)*, Sam Tobin-Hochstadt (Ed.). ACM, New York, NY, USA, Article 7, 13 pages. DOI: http://dx.doi.org/10.1145/2746325.2746329

Peter Achten and Martin Wierich. 2000. *A tutorial to the Clean Object I/O library (version 1.2).* Technical report CSI-R0003. Institute for Computing and Information Sciences, Radboud University Nijmegen, The Netherlands . 294 pages.

Erik Barendsen and Sjaak Smetsers. 1996. Uniqueness typing for functional languages with graph rewriting semantics. In *Mathematical Structures in Computer Science*, Vol. 6. 579–612.

Magnus Carlsson and Thomas Hallgren. 1993. 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.

Koen Claessen, Ton Vullinghs, and Erik Meijer. 1997. Structuring graphical paradigms in TkGofer. In *Proceedings of the 2nd International Conference on Functional Programming, ICFP '97*, Vol. 32(8). ACM Press, Amsterdam, The Netherlands, 251–262.

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2006. Links: web programming without tiers. In *Proceedings of the 5th International Symposium on Formal Methods for Components and Objects, FMCO '06*, Vol. 4709. Springer-Verlag, CWI, Amsterdam, The Netherlands.

Antony Courtney and Conal Elliott. 2001. Genuinely functional user interfaces. In *Proceedings of the 5th Haskell Workshop, Haskell '01*.

Antony Alexander Courtney. 2004. *Modeling User Interfaces in a Functional Language*. Ph.D. Dissertation. Yale University, USA.

Erik Dahlström, Patrick Dengler, Anthony Grasso, Chris Lilley, Cameron McCormack, Doug Schepers, and Jonathan Watt. 2011. *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*. Technical Report REC-SVG11-20110816. W3C Recommendation 16 August 2011.

László Domoszlai, Eddy Bruël, and Jan Martin Jansen. 2011. Implementing a non-strict purely functional language in JavaScript. *Acta Universitatis Sapientiae* 3 (2011), 76–98. Issue 1. http://www.acta.sapientia.ro/acta-info/C3-1/info31-4.pdf

László Domoszlai, Bas Lijnse, and Rinus Plasmeijer. 2014a. Editlets: Type-based, Client-side Editors for iTasks. In *Proceedings of the 26Nd 2014 International Symposium on Implementation and Application of Functional Languages (IFL '14)*, Sam Tobin-Hochstadt (Ed.). ACM, New York, NY, USA, Article 6, 13 pages. DOI: http://dx.doi.org/10.1145/2746325.2746331

László Domoszlai, Bas Lijnse, and Rinus Plasmeijer. 2014b. Parametric Lenses: Change Notification for Bidirectional Lenses. In *Proceedings of the 26Nd 2014 International Symposium on Implementation and Application of Functional Languages (IFL '14)*. ACM, New York, NY, USA, Article 9, 11 pages. DOI: http://dx.doi.org/10.1145/2746325.2746333

Conal Elliot and Paul Hudak. 1997. Functional Reactive Animation. In *Proceedings International Conference on Functional Programming, ICFP '97*. Amsterdam, Netherlands, 263–273.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2009. A Functional I/O System * or, Fun for Freshman Kids. In *Proceedings International Conference on Functional Programming, ICFP '09*. ACM Press, Edinburgh, Scotland, UK.

Sigbjorn Finne and Simon Peyton Jones. 1995. Composing Haggis. In *Eurographics Workshop on Programming Paradigms in Graphics*. Springer, Maastricht, the Netherlands, 85–101.

image.rkt 2014. image.rkt. (Dec 2014). http://docs.racket-lang.org/teachpack/2htdpimage.html

Daan Leijen. 2004. wxHaskell: a portable and concise GUI library for Haskell. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*. ACM, Snowbird, Utah, USA, 57–68. DOI: http://dx.doi.org/10.1145/1017472.1017483

Bas Lijnse. 2013. *TOP to the Rescue – Task-Oriented Programming for Incident Response Applications*. Ph.D. Dissertation. Institute for Computing and Information Sciences, Radboud University Nijmegen, The Netherlands . ISBN 978-90-820259-0-3.

Simon Peyton Jones (Ed.). 2003. *Haskell 98 language and libraries: the revised report*. Cambridge University Press.

M. J. Plasmeijer, P. M. Achten, and P. W. M. Koopman. 2007. iTasks: executable specifications of interactive work flow systems for the web. In *Proceedings of the 12th international conference on functional programming, ICFP'07*. ACM Press, Freiburg, Germany, 141–152.

Rinus Plasmeijer and Marko van Eekelen. 2002. Clean language report (version 2.1). (2002). http://clean.cs.ru.nl.

Meurig Sage. 2000. FranTk - A declarative GUI language for Haskell. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. ACM, New York, NY, USA, 106–117. DOI: http://dx.doi.org/10.1145/351240.351250

Manuel Serrano, Erick Gallesio, and Florian Loitsch. 2006. Hop, a language for programming the web 2.0. In *Proceedings of the 11th International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '06*. Portland, Oregon, USA, 975–985.

Peter Thiemann. 2002. WASH/CGI: server-side web scripting with sessions and typed, compositional forms. In *Proceedings of the 4th International Symposium on the Practical Aspects of Declarative Programming, PADL '02 (Lecture Notes in Computer Science)*, Shriram Krishnamurthi and Raghu Ramakrishnan (Eds.), Vol. 2257. Springer-Verlag, Portland, OR, USA, 192–208.