abstract

Task Oriented Programming with Purely Compositional Interactive Vector Graphics

1. Abstract

Task Oriented Programming [24][29] (TOP) is a paradigm that is designed to construct multi-user, distributed, web-applications. The iTask system [28] (iTasks) is a TOP framework that offers three core concepts for software developers.

- **Tasks** which are abstractions of the work that needs to be performed by (teams of) human(s) and software components. A task is a value of parameterized type (Task a). The type parameter a models the task value the task is currently processing. This value can be inspected by other tasks.

- **Shared data sources** (SDS) which are abstractions of information that is shared between tasks. A SDS is a value of parameterized type (ReadWriteShared r w). The type parameters r and w model the read and write values.

- **Combinator functions** that compose tasks and SDSs into more complex tasks and SDSs and combinations of them.

The iTask system is a domain specific language (DSL) that is shallowly embedded in the strongly typed, lazy, purely functional programming language Clean [27][30]. When developing an iTask application, the task developer can concentrate on identifying the tasks, the shared data sources, and their interrelation. The iTask system uses generic programming [5][21] and a hybrid static-dynamic type system [31][32] to generate all required machinery to create an executable. Among the plethora of concerns, the iTask system automatically generates a graphical user interface (GUI) for any conceivable first order model type. For this purpose the iTask system offers a comprehensive set of data types that model common user interface elements. In this way the task developer needs no work.

In this paper we show how the level of abstraction of iTask can remain intact when task developers define new user interface elements. This is done in a number of steps:

- We extend iTask with Images which are vector graphics based renderings. An image of type (Image a) is a rendering of a model value of type a. Images have a span to specify their dimensionality and local coordinate-system (traditional running from left-to-right and top-to-bottom), but there is no global coordinate system in which they are positioned or global canvas on which they are painted.

- We add combinator functions to compose images into more complex images. The absence of a global coordinate system or global canvas allows us to provide only three layout primitives: the overlay (placing images on top of each other), grid (two-dimensional structured layout), and collage (two-dimensional arbitrary layout). Each layout primitive has an optional host image that determines a reference span that is used for layout.

- We obtain interactivity by integrating images in iTask. Any (composite) image of type (Image m) can react to user events and define its behaviour via a pure function of type (m -> m) that alters the image’s model value. This is in accordance to the philosophy of tasks: behaviour only needs to be defined in terms of how tasks depend on the model value of tasks and images.

The implementation of images and its combinator functions in iTask is based on the Scalable Vector Graphics (SVG) standard [10]. The low-level integration of these images in iTask is structured by means of editlets, and the high-level integration is done via the iTask step task combinator function.

Compositional Images

The full paper contains a detailed explanation and motivation for the compositional image library. Figure displays the key elements of the API. For this abstract, we state the key properties of the API.

- Think of a basic image as an overhead-projector slide that is infinitely large. This slide can be rotated, scaled, and skewed. A finite portion of the basic image has visual content, the extent of which is defined by its span. The x-span always extends from left to right, and the y-span always extends from top to bottom.

- Think of a composite image as a stack of overhead-projector slides. This stack can be rotated, scaled, and skewed. When composing images, their span is used to control their relative location. There are three core image combinator functions: overlay to stack images, grid to stack images row-by-row or column-by-column, and collage to stack images and arrange
them to your liking. The commonly occurring layouts beside and above are direct specializations of grid.

- The layout combinator has an optional host image parameter. Think of the host image as the background image relative to which the other images are to be arranged in terms of alignment.
- Images can have tags. This is needed when expressing spans in terms of the span(s) of other parts of the image.
- A (composite) image of type \((\text{Image} \ m)\) can be made interactive by attributing it with a pure function of type \((m \to m)\), thus resulting in a change of image model value. This function is evaluated whenever the user clicks in the image (regardless of the location and transformation of the image).

### Integration in iTask

The integration of interactive, compositional images in iTask concerns the following components:

- The images are mapped to SVG. We face two major hurdles: (i) SVG adopts an imperative-style rendering model, so we must take care to unravel the declarative image specifications and paint them in the right order in SVG; (ii) text dimensions can only be computed at the client-side of the application, so the layout of images can not be performed entirely on the server-side of iTask.
- To establish the server-client side communication, we use iTask editlets.

These will be described in detail in the full paper.

### Case studies

We demonstrate the new iTask approach by means of the following case studies:

- a 1-person pocket calculator,
- a 2-person, distributed, tic-tac-toe game,
- a 2-person, distributed, trax game [11],
- a \(N\)-person, distributed, ligretto card game.

### Related work

Functional programming and GUIs share a long research history [2-4-6-9-11-19-22-23-25-26]. The full paper compares and discusses these approaches in more detail. For this abstract we restrict ourselves to the following observations:

- Regarding compositional images, the work by Henderson [19, 20] has been influential to many compositional approaches, as well as ours. Similar to Henderson’s approach, we abstract from absolute location, but we do not from size. In the context of scalable vector graphics, the latter is not an issue because at any time images can be resized to any demanded size.
- Regarding compositional GUIs, Haggis [15] is similar in their approach to layout and transform GUIs. A difference is that Haggis has a monadic flavour: the GUI elements that are to be combined need to be declared before their handles can be used to arrange them inside layouts. In our approach, the iTask system ‘collects’ the offered images in the task specifications.
- Regarding ‘completeness’, we have not yet made use of all graphics elements that are offered by SVG. Concepts that are currently missing but are intended to be included in the iTask system are Bézier curves, multi-line text blocks, gradients, generalized clipping, and filtering. The layout combinators that we propose were inspired by the Racket image API [14]. The three core layout primitives overlay, grid, and collage of our approach can model them. The current proposal’s event model is certainly incomplete as it covers only user-mouse clicks. We expect that extending the model to deal with the usual set of mouse and keyboard events follows the same approach.

### Conclusions

In the TOP iTask framework multi-user, distributed, web-applications can be developed on a high level of abstraction because the task developer can concentrate on identifying and specifying the required tasks, information, and how they relate, knowing that the iTask framework can generate a suitable web application. The paper shows how this property can also be satisfied when developing applications that require custom built user interface(s). Because images are compositional, the task developer can concentrate on identifying and specifying the required graphical elements,

---

**Figure 1.** The key elements of the Image API.

---

```
:: Image m // Opaque type
:: Span // Opaque type
:: Host m := Maybe (Image m)
:: ImageTag := String
:: FontDef := String
:: ImageOffset := (Span, Span)
:: XAlign = AtLeft | AtMiddleX | AtRight
:: YAlign = AtTop | AtMiddleY | AtBottom
:: ImageAlign := (XAlign, YAlign)
:: GridDimension = Rows Int | Columns Int
:: GridLayout := (GridLayout, GridLayout)
:: GridXLayout = LeftToRight | RightToLeft
:: GridYLayout = TopToBottom | BottomToTop
:: ImageLayout m := [ImageOffset] Image m (Host m) m -> Image m
:: overlay := [ImageAlign] ImageLayout m
:: beside := [YAlign] ImageLayout m
:: above := [XAlign] ImageLayout m
:: grid := GridDimension GridLayout [ImageAlign] ImageLayout m
:: empty := Span Span -> Image m
:: text := FontDef String -> Image m
:: circle := Span -> Image m
:: ellipse := Span Span -> Image m
:: rect := Span Span -> Image m
:: Slash := Slash | Backslash
:: xline := Span -> Image m
:: yline := Span -> Image m
:: line := Slash Span Span -> Image m
:: polygon := [ImageOffset] -> Image m
:: polyline := [ImageOffset] -> Image m
:: rotate := Real (Image m) -> Image m
:: fit := Span Span (Image m) -> Image m
:: fitx := Span (Image m) -> Image m
:: fity := Span (Image m) -> Image m
:: skewx := Real (Image m) -> Image m
:: skewy := Real (Image m) -> Image m
:: px := Real -> Span
:: ex := FontDef -> Span
:: descent := FontDef -> Span
:: textxspan := FontDef String -> Span
:: imagexspan := [ImageTag] -> Span
:: imageyspan := [ImageTag] -> Span
:: columnspan := [ImageTag] Int -> Span
:: rowspan := [ImageTag] Int -> Span
```
Knowing that the image library generates a suitable SVG rendering, via editlets graphically customized tasks are integrated seamlessly in the TOP paradigm.

References


