Radboud Repository

Radboud University Nijmegen

PDF hosted at the Radboud Repository of the Radboud University Nijmegen

The following full text is a publisher's version.

For additional information about this publication click this link.
https://repository.ubn.ru.nl/handle/2066/237853

Please be advised that this information was generated on 2021-10-21 and may be subject to change.

Typed Directional Composable Editors in iTasks

Bas Lijnse
Radboud University
Nijmegen, The Netherlands
b.lijnse@cs.ru.nl

Rinus Plasmeijer
TOP Software Technology
Mook, The Netherlands
rinus@top-software.nl

ABSTRACT

Generic, type-driven web-based editors have been an integral feature of the iTask Framework (iTasks) since its conception, and even predate it in the form of the iData library. Generic editors enable rapid prototyping, because they allow the generation of graphical user interfaces from any first-order type.

As applications mature, the need for increased control over the look and feel of editors arises. This can be accomplished by creating customised editors. Unfortunately defining custom editors is no trivial task in iTasks. The interface for creating custom editors from scratch is sufficiently powerful, but exposes many implementation details that make it complicated to use. In this paper we present a new interface and composition API for editors in iTasks. This new approach is based on an asymmetric typed interface for editors with separate type parameters for data that is consumed by, and data that is produced by, the web editors. We demonstrate the new possibilities by reconstructing a previously builtin editor as a composition of simpler editors and various other examples.

ACM Reference Format:
Bas Lijnse and Rinus Plasmeijer. 2020. Typed Directional Composable Editors in iTasks. In *IFL 2020: Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages (IFL '20), September 2–4, 2020, Canterbury, United Kingdom,* Olaf Chitil (Ed.). ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3462172.3462197

1 INTRODUCTION

The iTask Framework (iTasks) [6] has been the standard research implementation for the Task-Oriented Programming paradigm for more than a decade. One of the foundations of this system is the ability to generate user interfaces from types using generic programming techniques. In fact, without this ability it would be impossible to write interactive programs at the abstraction level of task compositions. Because user interfaces can be generated for any task of first order type, a program that specifies no user interface concerns, but only tasks and the information flow between them still generates a complete executable multi-user web application.

Generic editor components have been a part of iTasks since its first inception, and have existed before iTasks. iTasks started as an extension of the iData library [7] for creating generic web-based editors, which in turn was based on the GEC (Generic Editor Component) library [1] which provided the first generic user interfaces for desktop applications in Clean.

In iTasks the role of generic editors has evolved from a very explicit definition inherited from iData, to a black-box abstraction that aims to isolate all user interface concerns from the task specifications that express the possible workflows within a distributed multi-user information system.

As useful as generic user interfaces can be for rapid prototyping, when developing real-world applications there are always cases where custom solutions are needed to create a satisfying result. Therefore iTasks has always featured customization and tweaking of the generic user interfaces. Initially by introducing additional types and specializing generic functions, but eventually allowing custom user interface specifications (editor components) to be passed as optional parameters to task specifications.

This mechanism enabled the creation of libraries with a wide variety of customized user interface components, neatly abstracted for a programmer of task specifications. These libraries provide abstract values of type `Editor a`, that can be passed as optional parameter in calls to interaction primitives like `enterInformation` or `updateInformation`.

A task for entering a basic integer, such as the task of entering a grade, will by default generate a validated text field that is guaranteed to produce an integer.

```
1  enterGrade :: Task Int
2  enterGrade = Label "Grade" @>>
3    enterInformation []
```

Figure 1: Task using generic editor as UI

However, by passing an optional `Editor Int` argument, a slider in this case, and a function that maps the value of the editor to the desired task value we can customize the interaction to our liking.

```
1  enterGrade :: Task Int
2  enterGrade = Label "Grade" @>>
3    enterInformation [EnterUsing (\v -> v / 10) slider]
```

Figure 2: Task using customized editor as UI

The `Editor a` type is a convenient abstraction for separating user interface concerns from workflow concerns, but it has limited composability. A value of type `Editor a` represents a black box that

This work is licensed under a Creative Commons Attribution International 4.0 License.

IFL '20, September 2–4, 2020, Canterbury, United Kingdom
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8963-1/20/09.
https://doi.org/10.1145/3462172.3462197

facilitates editing of values of type `a` in a user interface in a model-view approach where `a` is the type of the model. In this black-box model-view approach, composition is limited to "gluing" editors together or to map functions over them to lift them to a different domain. Any form of interaction between editors in a composite user interface is impossible because of the limited interface, leaving programmers wanting a custom editor no other option than to create a monolithic editor for the composition from scratch.

In this paper we present a new composable interface for iTasks editor components. Although its implementation is designed for the iTask framework, we believe our approach extends beyond iTasks, and provides insight into the composition of web-based user interface components in general.

Our main contributions are:

- We present a generic composition mechanism for typed directional editor components.
- We demonstrate its viability by implementing it in iTasks.
- We demonstrate how a previously monolithic editor component can be redefined as a composition of simpler components.

The remainder of this paper is organized as follows: We first introduce the necessary preliminary concepts in section 2. We then analyze the earlier approach to editor components and its limitations by means of an example in section 3. We introduce the necessary changes and additions to address these limitations in section 4. Making use of the new approach we show how the example of section 3 can be redefined compositionally in section 5. We conclude the paper with additional examples in section 6, a discussion of related work in section 7, and closing remarks in section 8.

2 PRELIMINARIES: MODEL-VIEW BASED EDITORS

Before we can look at its limitations, we first have to understand the `Editor` abstraction. That is, what details are we hiding when composing opaque values of type `Editor a`. We also have to understand the role of editors in the larger context of the iTask system.

2.1 The iTask System

The iTask system is first and foremost a platform for Task-Oriented Programming. It facilitates not just the creation of user interfaces, but the complete stack of infrastructure to implement complex multi-user workflows.

The main focus of an iTasks programmer is to decompose a complex collaborative task into a composition of sub-tasks that can be performed by a single person, or an organization consisting of multiple individuals. The task composition is hierarchic as well as dynamic, because larger tasks can be split up into smaller tasks that are executed in parallel, or split up into sequential steps in which case each step is computed from the intermediate results of its predecessors. Tasks that are not decomposed further, are either computations, or interactive tasks in which human decisions or other input are required to advance a task's progress. In these interactive tasks, editors provide the UI to enable this interaction.

In iTasks, the complete UI the user interacts with is therefore an aggregation of the user interfaces of all interactive tasks at a certain point in time that are composed, either directly, or indirectly in parallel. Furthermore, parts are added and removed continuously from this aggregation over time as sequential compositions progress to a next step. A run-time framework manages this reflection of the task's progress in the user's web browser. It handles communication between client and server, the creation and destruction of user interface components and the routing of user events to the associated interactive sub-task (and its editor).

Editors define the user interface for an interactive task both locally and completely. Locally, because they create only the part of the user interface of a single sub-task. But also completely because they define the client-side rendering and processing, as well as the server-side handling of events that have been routed through the shared iTask run-time.

2.2 The Editor abstraction

Knowing a little about the iTask context, we can move on to specification of editors in full detail. The 'Editor' type is defined as follows:[1]

```
1  :: Editor a =
2   { genUI ::
3      UIAttributes DataPath *(EditMode a) *VSt ->
4      *(MaybeErrorString (UI, EditState), *VSt)
5   , onEdit ::
6      DataPath (DataPath, JSONNode) EditState *VSt ->
7      *(MaybeErrorString (UIChange, EditState), *VSt)
8   , onRefresh ::
9      DataPath a EditState *VSt ->
10     *(MaybeErrorString (UIChange, EditState), *VSt)
11  , valueFromState :: EditState -> *(?a)
12  }
```

Editor values are explicit dictionaries containing four functions that each handle an aspect of the user interface.

1. `genUI`: An editor specifies how an initial user interface is created based on an optional initial value `EditMode a`. The main result of this function is an abstract specification of the required user interface components, the `UI` value, and an *untyped* representation of the internal state of user interface. The abstract UI definition is rendered in the browser by a client run-time library that also attaches event handlers that relay user events back to the server.
2. `onEdit`: This function processes user events that are sent in a JSON encoded form from the browser. Applying the event produces a new editor state, and a response, the `UIChange` value, to send back to the browser.
3. `onRefresh`: Because iTasks facilitates concurrent editing of shared data by multiple users, editors also have to be able to handle server-sides refreshes of the data while it is being edited by others. Just as the `onEdit` function, the refresh results in an updated editor state and a message to send to the browser. The difference between the two is their input. In this case a typed fresh value is passed in instead of a JSON encoded user event.
4. `valueFromState`: The final function that an editor defines is a parser to convert from the untyped internal state to a typed value. This function yields an optional value `?a` because

[1]The `(?a)` on line 11 is Clean's type notation for optional values. Its values are either `?Just a` or `?None`. The `*` annotation signifies *uniqueness*. Uniqueness typing in Clean allows safe modification of states by guaranteeing a single reference to a value.

editors can allow a temporarily inconsistent unparsable state during editing.

The other recurring parameters in the definition that require an explanation are the `DataPath` and `VSt` typed values. The `DataPath` is a way of identifying parts in a composed editor. They encode a unique path in the tree structure of composed editors. The `VSt` is a unique state that is being threaded to allow editors to use impure operations. This is necessary for special cases where, for example, editors read information from disk, but is not commonly used.

The functions outlined above define the server-side behavior of editors. The client-side behavior is facilitated by the iTasks framework via a set of built-in client-side components. These are implemented in Javascript and can render common components in the browser. The `UI` definition produced by `genUI` can refer to these built-in components to create the necessary parts of an editor. These components also install predefined event handlers that relay user events to the server-side `onEdit` functions.

When the built-in components do not suffice, it is possible to use a minimal component that can be customized by providing a client-side initialization function. The init function, defined in Clean, is compiled to byte-code and interpreted in the browser and can manipulate the UI using the HTML5 API's exposed through foreign function interface to Javascript.

To create for example a minimal button, you need a function that initializes the component's HTML and installs an `onclick` callback function that relays the click to the server.

```
1   onInitUI ::
2    FrontendEngineOptions JSVal *JSWorld -> *JSWorld
3   onInitUI _ me world
4      # world = (me .# "domTag" .= "button") world
5      # (cb,world) = jsWrapFun (onInitDOMEl me) me world
6      = (me .# "initDOMEl" .= cb) world
7   onInitDOMEl me _ world
8      # world = (me .# "domEl.innerHTML"
9          .= me .# "attributes.text") world
10     # (cb,world) = jsWrapFun (onClick me) me world
11     = (me .# "domEl.onclick" .= cb) world
12  onClick me _ world
13     = (me .# "doEditEvent" .$!
14         (me .# "attributes.taskId"
15         ,me .# "attributes.editorId",True)) world
```

As you can see in this example [2] , the interaction with the web browser is untyped and impure. Creating components more complex than buttons or text fields can quickly become very verbose, error-prone, and hard to maintain.

2.3 Model-View design

The editor components as outlined in the previous section are based on a *model-view* design. An editor of type `Editor a` facilitates editing a *model* value of type `a` by mapping it to an untyped *view* which can be manipulated by user events. The `genUI` and `onRefresh` functions map values from the *model* domain to the *view* whereas the `valueFromState` function maps values from the *view* domain back to the *model* domain.

In this *model-view* approach, that uses a single type variable to abstract over the model type, we have two possible ways to create composite editors: Either by using a set of combinator functions, or by defining "wrapper" editors at the level of the individual functions an editor is comprised of.

2.4 Editor combinators

In the *model-view* approach we can define combinator functions that take abstract editors as arguments. One group of combinator functions we can define are those for "gluing" editors together by wrapping them in a container. For example in a panel using a function like `panel2`:

```
1   panel2 :: (Editor a) (Editor b) -> Editor (a,b)
```

When two editors are joined together the model type of the composition is simply a tuple of the model types of the two editors. In iTasks there is a series of such grouping functions for wrapping a number of editors in various containers.

Another group of combinators functions are the functions that map the model type of editors to a different domain. Because the editors define a *model-view* relation themselves, such mappings need to be bidirectional.

For example:

```
1   bijectEditorValue ::
2     (a -> b) (b -> a) (Editor b) -> Editor a
```

Under certain conditions, for example when an editor is only used for viewing data, unidirectional mappings are also possible.

```
1   comapEditorValue ::
2     (b -> a) (Editor a) -> Editor b
```

This second group of combinator functions is useful to simplify the nested tuple types of editors that have been composed with the first group of combinators. Together they can be used to create custom user interfaces to edit values that have a static structure.

The final combinators worth mentioning are the annotation operators «@ and @». These operators are used to annotate editors with static attributes. For example for adding specifying custom styles or layout directive by adding CSS classes or similar properties to an editor.

```
1   editor = container2 textField textField
2     <<@ classAttr ["itasks-horizontal"]
```

2.5 Wrapping editors

Another strategy to create custom user interfaces using an existing `Editor` is to wrap an editor in a custom editor specification. By defining the functions of a new `Editor` such that they use the opaque functions of an existing editor you can create a composition. In this composition parts of the resulting editor are created and handled by the wrapper, while other parts are delegated to the wrapped editor.

This strategy is not used often in iTasks because it requires understanding of the low-level mechanisms of editors to define such wrappers. An important editor that is based on this strategy is the "list editor" (Figures 3 and 4) which we will examine in more detail in section 3.

[2] The # is a let-before construct in Clean that togetwher with the unique `world` state allows safe expression of impure code

2.6 Limitations of Model-View editors

While it is certainly possible to create editor compositions in iTasks, there are inherent limitations that inhibit the ease with which they can be created.

The composition of editors using combinator functions is preferable over the "wrapping" strategy because it allows composition without having to know the low-level mechanics of editors. With the combinator functions editors are composed at an abstract level using well-typed functions that clearly specify the relations between the domains of the editors involved.

However, they are not as expressive as the "wrapping" compositions. The reason why the list editor is defined as a monolithic wrapper is because it is impossible to create using the existing combinators.

What the combinators lack is a way to define dynamic compositions in which the values of component parts are used to modify the value of other parts of the composition, or even extend the composition with additional editors.

This limitation is a consequence of the model-view approach which is too abstract for this purpose. Because the only thing we know about editors at the abstract level is that they have a (typed) value, we can only define combinators that affect these value domains. We cannot reason about, or specify combinators that modify, the input and output of editors because they are hidden by the abstraction.

3 DECONSTRUCTING THE LIST EDITOR

In the previous section we have already mentioned iTasks's list editor as an example of an editor that cannot be constructed as a composition of simpler editors using the model-view approach. This makes it a suitable candidate to investigate what would be required to make such composition possible using a new approach.

In this section we dissect the monolithic list editor to identify what parts it is made of, and how these affect each other.

First let's look at the definition of the `listEditor` builder function that is provided by iTasks to create list editors.

```
1  listEditor ::
2     (? ([?a] -> (?a))) Bool Bool (? ([?a] -> String))
3        (Editor a) -> Editor [a]
```

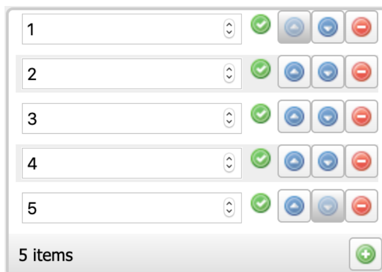

Figure 3: Generic list editor for integers

The list editor is a versatile higher-order editor that creates an editor for lists of a certain type (`Editor [a]`) given an editor for that type (`Editor a`). The first four parameters of the `listEditor` functions are essentially configuration options. They control if new elements can be added, moved around or removed, as well as what label to show. These configuration parameters are necessary because the list editor is a monolithic component which makes it an all or nothing solution. Its parts cannot be reused to create an alternative similar component.

The type `a` in the list editor can be anything. It can be a simple integer, as shown in figure 3, or a complex recursive record structure such as the `Family` type below as shown in figure 4.

```
1  :: Family =
2    { person   :: Person
3    , partner  :: ?Person
4    , children :: [Family]
5    }
6  :: Person =
7    { firstName    :: String
8    , surName      :: String
9    , gender       :: Gender
10   , dateOfBirth  :: Date
11 }
12 :: Gender = Male | Female | Other String
```

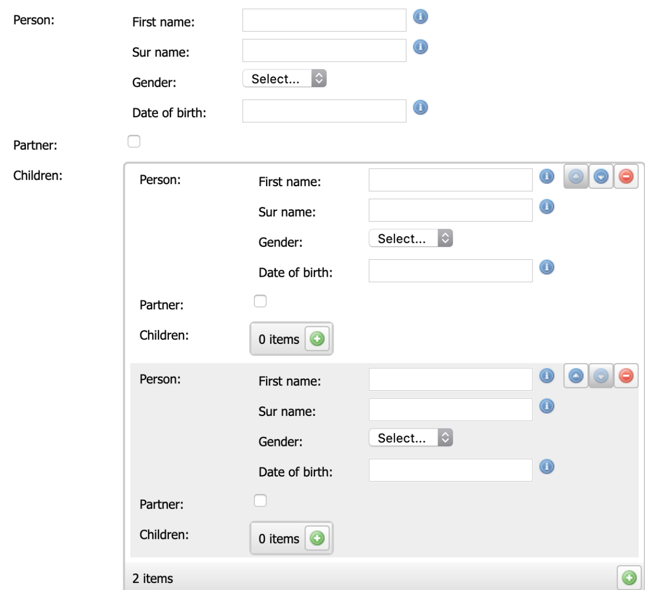

Figure 4: Generic list editor for a recursive type with records

The `listEditor` allows an existing editor to be used multiple times to create a list of values of that editor's type. The additional UI components it adds make it possible to manipulate the spine of this list. There are buttons for adding and removing list items and for changing the order of elements. These operations do not depend on the type of elements in the list and are all handled by wrapping editor component.

3.1 The list items and the list

The first dissection we can make is to separate the list editor into the set of list items and the toolbar beneath it.

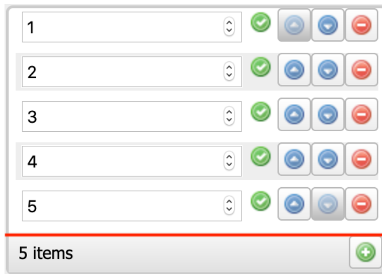

Figure 5: List items and list toolbar

The toolbar represents the part of the editor that is independent of whether there are list items or not. Even when the list is empty, the toolbar remains. This part of the component is concerned with operations on the list as a whole. The label summarizes the list as the total number of elements in it and the button extends the list. Both sub-components are independent of individual list items.

The other part of the editor is dedicated to manipulating individual list items. It reflects the structure of the list and has as many parts as there are list items.

3.2 Repetition of list items

The next cut we can make is to view the list items UI as a repeated component. For each element of a list, the editor provides an identical row that contains the controls for a single list item.

Figure 6: Repetition of items

In a composition mechanism, this part of the editor suggests the need for the possibility to create a composed editor by repeating an existing editor an arbitrary number of times.

3.3 Value and position

If we zoom in on the individual rows we can see that each row has two distinct functions. The left side of the row embeds an existing editor as a polymorphic black-box to allow the value of a list item to be edited. In this case, the generic editor for integers is used which is in itself a composition of a number field and an icon that shows the result of dynamic input validation.

The set of buttons on the right side of the row are used to manipulate the position of the item in relation to the list. Moving an item up, down or removing an item are operations that act on the structure of the list, rather than on the value of the item.

Figure 7: Editing values and editing positions

The group of three buttons can be viewed as composition of three individual button components who are used together to signal what operation should be applied to the row.

3.4 Summary and extension

For the final dissection we have to look back at the toolbar. This component is made up of two parts: The label on the left side, and the button for adding new items on the right. The label summarizes the content of the list. The value of this component cannot be edited by the user, but is updated automatically when the list changes. The implication of this behavior is that a composition mechanism must be able to let the value of one sub-component to be determined by another sub-component.

Figure 8: Summary of, and actions upon the full list

The button is used to modify the list as whole. In this case it is used to add a list element, but it is easy to imagine other operations in the toolbar that modify the list. For example a "reset" button, or a "sort" or "shuffle" button to change the order of the list items. Just as the label, the button in the toolbar implies that one sub-component can determine the value of another sub-component. The direction is reversed though. When the list items change, the label is updated accordingly. When the state of button changes (it is clicked), the list items are updated in response.

3.5 Minimal UI elements

Another way to decompose the list editor is by looking at the minimal building blocks that make up its UI. There are just three building blocks in the entire list editor.

- **Buttons** The most used component is the simple button. It is used for all structural changes on the list.
- **Text labels** The only non-interactive component is the text label in the toolbar.
- **Abstract editors** The values are edited by components which are passed as a parameter in the composition. These are treated as abstract black boxes.

In addition to the smallest building blocks a number of container components are used to group the building blocks. These are essentially all the same except for their appearance. These containers are essential for composition. They facilitate the gluing of components together into a single component.

In the next section we will introduce a single container component that will facilitate all the necessary compositions of components.

4 INTRODUCING DIRECTIONAL EDITORS

To address the inhibiting factors that limit the composition possibility of editor components we need to make some fundamental changes to the editor API:

- We abandon the model-view approach and replace it with a read-write interface;
- We extend the container components with the ability to modify its content dynamically;

- We introduce a *loopback* function that enables sub-components to affect each other.

In this section we present the three changes that we made to improve composition. In the next section we will make use of these changes to re-implement the list editor as a composition.

4.1 From Model-view to Read-write

The first change is the most important. We let go of the model-view approach on which editors were based. To be able to link the effects of sub-components in a composition, a single model value is not enough. To do that we need to be able to reason about an editor components input and output instead.

In the model-view approach the input and output are inherently symmetric: An editor is supplied with a model value as input, it is changed by manipulating the view (editing), and finally the changed model is emitted as output. This is even a somewhat simplified view, because models can be shared by multiple users in iTasks.

When you consider the input and output as defining features of an editor, this approach looks even more constraining. Consider for example a drop-down box. In such a component a user selects a single item from a list of possible options. The necessary input for this component is the list of options and optionally an initial selection. The output however, is just the single selected item. In the symmetric approach, the model consists of both the list of options and the selection. This implies that the component's output must include the set of options from which the item was chosen.

In a composition where output of one component determines the input of another component this quickly leads to very complex inputs and outputs being linked together.

We therefore change the abstract interface of editors from `Editor m` to `Editor r w`. The type parameters `r` and `w` define the input (read) and output (write) of the editor components. We use "read" and "write" instead of "input" and "output" to be consistent with the "shared data source" concept in iTasks [3] which also uses reading to mean pulling data towards the user and writing to mean sending (modified) data back to its source.

This change in the abstract definition of editors is reflected in the low level definition as follows:

```
1  : Editor r w =
2   {onReset :: UIAttributes (?r) *VSt ->
3     *(*MaybeErrorString *(UI, EditState, ?w),*VSt)
4   ,onEdit :: (EditorId, JSONNode) EditState *VSt ->
5     *(*MaybeErrorString *(?(UIChange, EditState, ?w)),*VSt
        )
6   ,onRefresh :: r EditState *VSt ->
7     *(*MaybeErrorString *(UIChange, EditState, ?w),*VSt)
8   ,writeValue :: !EditState -> MaybeErrorString w
9   }
```

The `onReset` and `onRefresh` now get a value of the read type instead of the model type. Additionally all event handler functions now produce an optional value of the write type. It is for the editor components to decide when it is necessary to write output.

The new `writeValue` function is the replacement of `valueFromState`. Instead of retrieving the model value from an internal state, an editor can now be forced to compute a value of its write type. This is necessary when editors are grouped together in a composition. When one sub-component writes a value, the others also need to write their value to be able to compute their composite write value.

4.2 Directional built-in editors

In the directional approach all built-in components are updated to have sensible read and write types. Depending on the type of component these types may be symmetric, or asymmetric.

```
1  button :: Editor Bool Bool
2  slider :: Editor Int Int
```

The `button` and `slider` components have symmetric read and write types because they are guaranteed to have a value and do not need additional information to be added in the read type.

```
1  integerField :: Editor Int (?Int)
2  textField :: Editor String (?String)
3  label :: Editor String ()
4  htmlView :: Editor HtmlTag ()
```

Input fields in which free text can be entered are asymmetric and have an optional as their write type. The optional value signifies that the editor can be in a (temporary) invalid state. For example when text is typed into an integer field it is likely preferable to write `?None` instead of an arbitrary integer.

Editors that are used for displaying information only are also asymmetric. Their unit write type `()` signifies that these editors do not produce values.

4.3 Simplified mapping

The read-write model simplifies mapping functions over editors. In the model-view approach mappings had to be bidirectional by definition. In the read-write approach we can choose to map a function over the read values, to map a function over the write values, or to do both.

```
1  mapEditorRead ::
2    (r -> rb) (Editor rb w) -> Editor r w
3  mapEditorWrite ::
4    (wb -> w) (Editor r wb) -> Editor r w
5  mapEditorRW ::
6    (r -> rb) (wb -> w) (Editor rb wb) -> Editor r w
```

As we will see in section 5 some `write` mappings require access to additional information that is available in the editor's `read` values, but is discarded by the editor. For these cases we provide a general combined mapping function that wraps an editor with an arbitrary state that is available during reading and writing.

```
1  mapEditorWithState ::
2    s (r s -> (?rb,s)) (wb s -> (w,s))
3       (Editor rb wb) -> Editor r w
```

4.4 Dynamic containers

The second major change we introduce is an extension of the container components that group multiple editors. In the analysis of the list editor we have seen that this editor adds additional components to manipulate the placement of items of the list, and to add and remove items. This capability can be generalized to all grouping container components:

```
1  :: EditorRef =: EditorRef Int
2  :: ListSubEditorItem a
3    = NewListSubEditor (?a)
4    | ExistingListSubEditor EditorRef (?a)
5
6  groupl :: !UIType
7    ((?rl) [(EditorRef,w)]
8      -> [ListSubEditorItem r])
```

```
9     (Editor r w)
10       -> Editor rl [w]
```

Instead of adding buttons to control the content of the container, the `groupl` function is parameterized with a function that determines the effect on the container's children on every *read* operation.

Using the read value and a list of abstract references to the existing children in the container, this function computes the new container content. For `groupl`, it can reference the existing components and refresh them with new data, it can add new items with or without initial data and it can reorder the items in the container. It can also remove existing items by omitting them in the new container content.

For grouping in tuples there are similar containers:

```
1  :: TupleSubEditorItem a
2    = NewTupleSubEditor (?a)
3    | ExistingTupleSubEditor (?a)
4
5  group2 :: !UIType
6    ((?rt) (?(wa,wb))
7      -> (TupleSubEditorItem ra
8         ,TupleSubEditorItem rb))
9    (Editor ra wa)
10   (Editor rb wb)
11      -> Editor rt (wa,wb)
12
13 group3 :: !UIType
14   ((?rt) (?(wa,wb,wc))
15     -> (TupleSubEditorItem ra
16        ,TupleSubEditorItem rb
17        ,TupleSubEditorItem rc))
18   (Editor ra wa)
19   (Editor rb wb)
20   (Editor rc wc)
21     -> Editor rt (wa,wb,wc)
22 ...
```

For these components it is not possible to reorder, delete or to add items. This is because the types of the parts are potentially different and cannot be interchanged. For these containers there is only a function that determines which components are replaced, refreshed, or left alone when new data comes into the composition.

4.5 Loopback

The final piece of the puzzle we introduce is the `loopbackEditorWrite` combinator function:

```
1  loopbackEditorWrite ::
2    (w -> ?r) (Editor r w) -> Editor r w
```

This simple looking function is parameterized with a single filtering function. Every time the editor writes its value, this function decides to either do nothing and let the write pass, or it computes a new read value which it uses to refresh the editor with. This may trigger a new write value which is again inspected. Naturally, one must be careful to eventually return `?None` when writing the filtering function. Otherwise the loopback will not terminate.

Together with the dynamic container components, this loopback mechanism makes it possible to create editors in which a subset of its write domain is used as a command language that is fed back into the editor and interpreted to modify its sub-components.

In the following section we will demonstrate this principle by reconstructing the list editor from the ground up using only an initial set of minimal UI components (buttons and labels) and the concepts introduced in this section.

5 RECONSTRUCTING THE LIST EDITOR

Now that all required additional concepts have been introduced we can demonstrate their use by reconstructing the list editor as a composition.

5.1 An editor for list item operations

We'll start bottom up by creating a simple component: an `itemTools` editor that can be used to trigger operations on the list items as shown in Figure 9.

Our first attempt is to start by grouping three buttons together in a container:

```
1  itemTools :: Editor (Bool,Bool,Bool) (Bool,Bool,Bool)
2  itemTools = container3 button button button
```

These `button` components are of type `Editor Bool Bool`. They can toggled between two states. Both the *read* and *write* types are `Bool` and are used to set (on read) and emit the state of the button when it is clicked (the write).

The type of the composition of the three buttons reflects its structure, but is not very convenient. We therefore use the `mapEditorRead` and `mapEditorWrite` functions to lift the editor to a simpler domain.

We introduce a new type that represents the possible operations we may want to perform on a list item:

```
1  :: ItemCommand = ItemUp | ItemDown | ItemDelete
```

We then define a function that maps the triplet of booleans to an optional value of this type:

```
1  toItemCommand :: (Bool,Bool,Bool) -> ?ItemCommand
2  toItemCommand (True,_,_) = ?Just ItemDown
3  toItemCommand (_,True,_) = ?Just ItemUp
4  toItemCommand (_,_,True) = ?Just ItemDelete
5  toItemCommand _ = ?None
```

If we map this function on the writes of the editor, it writes a single command when one of the buttons is clicked and `?None` if none of the buttons was clicked.

Similarly we can simplify resetting the set of buttons to their initial unclicked state with the following function that we can map over the read values of the editor:

```
1  resetItemTools :: () -> (Bool,Bool,Bool)
2  resetItemTools _ = (False,False,False)
```

When both functions are applied we can define a better, functionally complete version of the `itemTools` editor:

```
1  itemTools :: Editor () (?ItemCommand)
2  itemTools
3    = mapEditorRW resetItemTools toItemCommand
4    $ container3 button button button
```

We can refine our definition of `itemTools` by using the `<<@` operator to annotate its parts with styling attributes to align the components and add some icons.

```
1  itemTools :: Editor () (?ItemCommand)
2  itemTools
3    = mapEditorRW resetItemTools toItemCommand
4    $ container3
5      (button <<@ iconClsAttr "icon-down")
6      (button <<@ iconClsAttr "icon-up")
7      (button <<@ iconClsAttr "icon-remove")
8    <<@ classAttr ["itasks-horizontal"]
```

Figure 9: A first editor for list item operations

5.2 An editor for list items

The next step in the reconstruction is to compose the `itemTools` component we just constructed with an editor for modifying the list values such that we have a complete editor for individual list items. The result of this composition is shown in figure 10. For simplicity's sake we will first use an a concrete integer editor: the predefined `integerField` editor with type `Editor Int (?Int)`. We will later factor out this editor to create the final polymorphic higher-order list editor.

```
1  listItemEditor ::
2   Editor (?Int) (?Int,?ItemCommand)
3  listItemEditor
4    = group2 UIListItem readListItem
5      (integerField <<@ widthAttr FlexSize)
6      (itemTools <<@ widthAttr WrapSize)
7      <<@ classAttr ["itasks-horizontal"
8                    ,"itasks-wrap-height"]
```

If we ignore the style annotations, the composition is very simple. An `integerField` is combined with the `itemTools` editor using the `group2` container. However, if we look closely at the type of the composition there is something unexpected. The *write* type simply pairs the write type of an `integerField` (`?Int`) with the write type of the `itemTools` editor (`?ItemCommand`). The *read* type of the composition is merely `?Int` instead of a tuple such as `(Int,())`.

The read type `?Int` is explained by the second parameter of `group2`: the `readListItem` function.

```
1  readListItem :: (?(?Int)) (?(?Int,?ItemCommand))
2    -> (TupleSubEditorItem Int,TupleSubEditorItem ())
3  readListItem (?Just mbn) ?None
4    = (NewTupleSubEditor mbn
5      ,NewTupleSubEditor (?Just ()) )
6  readListItem (?Just mbn) (?Just _)
7    = (ExistingTupleSubEditor mbn
8      ,ExistingTupleSubEditor (?Just ()) )
9  readListItem ?None _
10   = (NewTupleSubEditor ?None
11     ,NewTupleSubEditor ?None )
```

This function defines the newly introduced fine-grained control over the elements of a composition. It allows us to summarize the potential updates to the composition by the single `?Int` value. If the value is a `?Just` the integer editor is refreshed and the buttons are reset. If the value is `?None` the integer editor is left alone, but the buttons are still reset. This makes it possible to reset the buttons without refreshing the value editor.

Figure 10: The combined editor for list item operations

5.3 An interactive editor for a set of existing items

The next extension of the composition is to repeat the previously constructed `listItemEditor` arbitrary many times to create an editor for a set of list items. We can do this by using the `groupl` container:

```
1  listItemsEditor ::
2     Editor (ListCommand (?Int)) [(?Int,?ItemCommand)]
3  listItemsEditor
4      = groupl UIContainer readList listItemEditor
5      <<@ heightAttr WrapSize
```

Again, the actual work is done in the `readList` function which controls how the component is updated when it reads new data. This function defines the editors behaviour based on values of the `ListCommand a` type:

```
1  :: ListCommand a
2      = ListRefresh [a]
3      | ListModify [?ItemCommand]
4      | ListAppend
```

This type encodes the operations we can apply to an arbitrary set of list items. Refreshing the values, applying item commands or appending new list items. These commands are implemented in the `readList` function:

```
1  readList :: (?(ListCommand (?a))) [(EditorRef,(?b,?
        ItemCommand))] -> [ListSubEditorItem (?a)]
2  readList (?Just (ListRefresh list)) existing
3    = [ExistingListSubEditor ref (?Just val)
4       \\ (ref,_) <- existing & val <- list]
5    ++ [NewListSubEditor (?Just val)
6        \\ val <- drop (length existing) list]
7  readList (?Just (ListModify commands)) existing
8    = modifyList $ zip (fst <$> existing, commands)
9  readList (?Just ListAppend) existing
10   = [ExistingListSubEditor ref ?None
11      \\ (ref,_) <- existing]
12   ++ [NewListSubEditor ?None]
13 readList ?None existing = []
```

Modifying the list is achieved by zipping the list of references to existing list items with the `ListItemCommand` commands and straightforward pattern matching. This is implemented in the `modifyList` function.

```
1  modifyList :: [(EditorRef,?ItemCommand)] -> [
        ListSubEditorItem (?a)]
2  modifyList [] = []
3  modifyList [(ref,?Just ItemDelete):items]
4    = modifyList items //Remove an item
5  modifyList [(ref1,_),(ref2,?Just ItemUp):items]
6    = [ExistingListSubEditor ref2 (?Just ?None)
7      ,ExistingListSubEditor ref1 (?Just ?None)
8      :modifyList items] //Move up
9  modifyList [(ref1,?Just ItemDown),(ref2,_):items]
10   = [ExistingListSubEditor ref2 (?Just ?None)
11     ,ExistingListSubEditor ref1 (?Just ?None)
12     :modifyList items] //Move down
13 modifyList [(ref,_):items]
14   = [ExistingListSubEditor ref (?Just ?None)
15     :modifyList items]
```

We can now close the loop by wrapping the editor with the new `loopbackEditorWrite` function:

```
1  listItemsEditor ::
2   Editor (ListCommand (?Int)) [(?Int,?ItemCommand)]
3  listItemsEditor
4    = loopbackEditorWrite
5      (\w -> let commands = snd <$> w in
```

```
6       if (any isJust commands)
7         (?Just $ ListModify commands) ?None
8       ) $ group1 UIContainer readList listItemEditor
9       <<@ heightAttr WrapSize
```

When any of the list items writes a `ListItemCommand` value other than `?None` (i.e. when a button has been clicked) a `ListModify` command is written back to the editor which interprets the command in the context of the list as whole. The `ListRefresh` and `ListAppend` commands will be added when the `listItemsEditor` is composed with the toolbar component.

Figure 11 shows the component for editing and rearranging a set of list items that we have defined up to this point.

Figure 11: The combined editor for a set of list items

5.4 A toolbar for adding items

The toolbar component is a straightforward composition of a label and a button as shown in figure 12

```
1  listToolbar :: Editor (Int,Bool) (Int,Bool)
2  listToolbar
3    = toolbar2
4      (numberLabel <<@ widthAttr FlexSize)
5      (button <<@ iconClsAttr "icon-add")
6    <<@ classAttr ["itask-listitem-controls"]
7  where
8    numberLabel = mapEditorWithState 0
9      (\n s -> (if (n <> s)
10       (?Just (toString n +++ "␣items"))
11       ?None,n)
12     )
13     (\_ s -> (s,s)) label
```

The type of the composition is the expected symmetric `Editor (Int,Bool) (Int,Bool)` type. The only noteworthy part of this specification is the `numberLabel` component. Because the `label` component works on strings and we do not want to parse the string back to an integer to write the value of the composition, the label is wrapped with the `mapEditorWithState` function to be able to write back the integer value it last read.

Figure 12: The toolbar with label and add button

5.5 The complete list editor

The final steps in completing the composition are the glueing together of the set of list items `listItemsEditor` with the toolbar (`listToolbar`) and simplification of the composition's interface.

The first part can be achieved as follows:

```
1  completeListEditor ::
2    Editor (?(ListCommand (?Int)),(Int,Bool))
3           ([(?Int,?ItemCommand)],(Int,Bool))
4  completeListEditor
5    = loopbackEditorWrite loopback
6    $ group2 UIList read listItemsEditor listToolbar
7    <<@ heightAttr WrapSize
8  where
9    read ?None _ =
10     (NewTupleSubEditor ?None
11     , NewTupleSubEditor ?None)
12   read (?Just (mbx,y)) ?None =
13     (NewTupleSubEditor mbx
14     ,NewTupleSubEditor (?Just y))
15   read (?Just (mbx,y)) (?Just _) =
16     (ExistingTupleSubEditor mbx
17     ,ExistingTupleSubEditor (?Just y))
18
19   loopback (items,(num,True))
20     = ?Just (?Just ListAppend,(num, False))
21   loopback (items,(num,c))
22     | length items <> num
23       = ?Just (?None,(length items,c))
24     | otherwise = ?None
```

The pattern is the same as in previous compositions. The `read` function controls how the parts of the composition are updated, and the `loopback` function defines when information should be fed back into the composition. In this case that needs to be done when the add button is clicked, or when the number of items is no longer consistent with the number displayed in the toolbar. This happens when the item delete buttons are used to remove items form the list.

The final step we need to complete the editor is to simplify the interface:

```
1  listEditor :: Editor [Int] (?[Int])
2  listEditor
3    = mapEditorRead
4          (\l -> (?Just $ ListRefresh $ ?Just <$> l
5      ,(length l,False)))
6    $ mapEditorWrite (\(l,t) -> sequence $ fst <$> l)
7    $ completeListEditor
```

The editor's read values are mapped from a list of integers to a list of updates for the individual parts. The function mapped over the write values isolates the list values by removing the `ItemCommand` values. It also applies a monadic `sequence` that simplifies the resulting list of optional integer values to a single optional list of integers.

5.6 A parameterized polymorphic list editor

The `integerField` editor that we have used in this section was a convenient but arbitrary example. Nowhere in the compositions do we use a function that is limited to integer editors. We can therefore generalize the `listEditor` editor to a higher order editor that is parameterized with an editor for editing the list items.

The definition of `listEditor` then becomes:

```
1  listEditor :: (Editor r (?w)) -> Editor [r] (?[w])
2  listEditor editor
3    = mapEditorRead
4          (\l -> (?Just $ ListRefresh $ ?Just <$> l
5      ,(length l,False)))
6    $ mapEditorWrite (\(l,t) -> sequence $ fst <$> l)
7    $ completeListEditor editor
```

Similarly we need to parameterize the `completeListEditor`, the `listItemsEditor`, and the `listItemEditor` compositions.

6 ADDITIONAL EXAMPLES

The list editor we featured in the previous sections illustrates all aspects of the new editor approach. It is however only one example.

In this section we offer a number of additional examples of composite user interface components expressed using the new editor composition API.

6.1 Simple pagination

A common minimal UI example is a counter which displays a number which can be incremented by clicking a button. We'll show a little complex, but more generally useful, editor for paging through large sets of information.

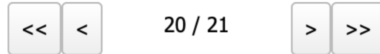

Figure 13: A simple pagination component

This editor works on a tuple of two integers: The current page number, and the total number of pages. The buttons allow you to increment or decrement the current page, or to go to the first or last page directly. The type of this component is therefore `Editor (Int,Int) (Int,Int)`.

The essence of the pager component is the label that shows the current page and the total number of pages. Because that information also needs to be available to determine the pager's behavior we need a label that is not just a string, but keeps the original data. We therefore first generalize the label with state that we also used in the list editor's toolbar:

```
1  labelSt :: (a -> String) a -> Editor a a
2  labelSt toString st
3    = mapEditorWithState st
4      (\x _ -> (?Just (toString x),x))
5      (\_ x -> (x,x)) stringlabel
```

The remainder of the component is little more than the grouping of the label with a set of buttons and the definition of a feedback loop:

```
1  pager :: Editor (Int,Int) (Int,Int)
2  pager
3    = mapEditorWrite (\(Left x) -> x)
4    $ loopbackEditorWrite (either (const ?None) ?Just)
5    $ mapEditorRead
6      (\ct -> (False,False, ct, False,False))
7    $ mapEditorWrite update
8    $ container5
9      (boolbutton <<@ textAttr "<<")
10     (boolbutton <<@ textAttr "<")
11     (labelSt
12       (\(c,t) -> toString c +++ "␣/␣" +++ toString t)
           (0,0)
13       <<@ styleAttr "text-align:␣center"
14     )
15     (boolbutton <<@ textAttr ">")
16     (boolbutton <<@ textAttr ">>")
17   <<@ classAttr ["itasks-horizontal"]
18 where
19   update (first,prev,(cur,total),next,last)
20    | last && cur < total = Right (total, total)
21    | next && cur < total = Right (cur + 1, total)
22    | first && cur > 1 = Right (1, total)
23    | prev && cur > 1 = Right (cur - 1, total)
24    | otherwise = Left (cur,total)
```

The `container5` groups the label with the two buttons on each side. The `mapEditorRead` on line 5 sets the the label and resets all buttons. The `mapEditorWrite` on line 7 and the `update` function inspect the button states and encodes the response as an `Either (Int,Int) (Int,Int)` that is interpreted by the `loopbackEditorWrite` on line 4. When no button is clicked a `Left` value represents the current value of the editor. When a button is clicked a `Right` value carries the new value that is fed back into the editor. Finally the `mapEditorWrite` on line 3 strips the `Left` constructor from the values that are let through by the `loopbackEditorWrite`.

6.2 Text input with suggestions

Another common UI component worth showing is a text field with an attached list of suggestions that a user may choose a suggested value from to set the text field with.

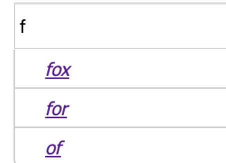

Figure 14: A text input with suggestions

What makes this component interesting is that it is often used with asynchronous retrieval of the suggestions while something is typed in the text field. When the suggestions are the result of, for example, a database query, then it takes a significant amount of time to retrieve the suggestions in relation to the speed with which the user types in the text field. When the suggestions arrive, the text field will already have a new value because the user kept typing in the mean time. This means that we need to be able to construct a composition of a text field with a list of suggestions in which updating the suggestion list is independent of updating the text.

The new container combinators parameterized `read` function now enables the construction of such compositions.

```
1  textFieldWithSuggestions ::
2    Editor (?String, [String]) (Either String String)
3  textFieldWithSuggestions
4    = mapEditorWrite write
5    $ group2 UIContainer read stringfield suggestionlist
6  where
7    read ?None _ =
8      (NewTupleSubEditor ?None
9      ,NewTupleSubEditor ?None)
10   read (?Just (mbs,options)) ?None =
11     (NewTupleSubEditor mbs
12     ,NewTupleSubEditor (?Just options))
13   read (?Just (mbs,options)) (?Just _) =
14     (ExistingTupleSubEditor mbs
15     ,ExistingTupleSubEditor (?Just options))
16   read _ _ =
17     (ExistingTupleSubEditor ?None
18     ,ExistingTupleSubEditor ?None)
19
20   write (_,?Just choice) = Right choice
21   write (s,_) = Left s
```

The independence of the text field is reflected in the compositions read type (`(?String, [String])`). The first element of tuple is optional and can be `?None` to independently update the list of suggestions without affecting the text field.

The component itself is a straightforward grouping of a text field of type `Editor String String` and an editor for choosing a string from a list of strings of type `Editor [String] (?String)`.

The `read` function, and especially the alternative on line 13 enable the independent updating of the suggestions.

Finally, the `write` function mapped over the editor on line 4 maps the write values of both parts to single right-biased `Either` value. When no choice has been made it is a `Left` with the value of the text field, and when an item is chosen in the suggestion list, it is a `Right` with this choice.

When we have a pure function that can provide the suggestions (of type `String -> [String]`), we can create a drop-in replacement editor for standard text fields using the loopback combinator.

```
1  textFieldWithSuggestionFunction ::
2    (String -> [String]) -> Editor String String
3  textFieldWithSuggestionFunction suggestions
4    = mapEditorRead (\s -> (?Just s,suggestions s,?Just s))
5    $ mapEditorWrite (\(s,_) -> either id id s)
6    $ loopbackEditorWrite loopback
7    $ mapEditorWithState ?None
8      (\(mbs,opts,state) _ -> (?Just (mbs,opts),state))
9      (\w state -> ((w,state),state))
10   $ textFieldWithSuggestions
11 where
12   loopback (Left cur,?None)
13     = ?Just (?None,suggestions cur,?Just cur)
14   loopback (Left cur,?Just prev)
15     | cur <> prev
16       = ?Just (?None, suggestions cur, ?Just cur)
17     | otherwise = ?None
18   loopback (Right choice,?Just prev)
19     = ?Just (?Just choice,[], ?None)
20   loopback _ = ?None
```

This editor adds a state of type `?String` to the editor in which the the value of the text field is duplicated. This allows the `loopback` function to detect changes to the text field and to update the list accordingly. When a choice is made in the suggestion list, all suggestions are cleared.

The `mapEditorWrite` and `mapEditorRead` hide the full complexity of the editor's type and simplify its interface to an editor of type `Editor String String`, the same type as a regular text field.

6.3 A complete calculator

In our final example we go beyond the intended use of editor components, namely to create UI components to customize interactive tasks in an iTasks task composition, and specify a small stand-alone pocket calculator application as an editor component.

Figure 15: A simple pocket calculator

In this example we demonstrate the use of function types in intermediate editors to bypass the use of encoded commands as we have used in the list editor (e.g. `ItemCommand` and `ListCommand`). Instead we simply create editors that produce functions that encode the desired state changes directly.

The calculator consists of two main parts. The display section and the button pad. The display section holds two integer values: one for the (intermediate) result, and a second one for entering an operand value which is applied when one of the operator buttons is pressed. The button pad is a grid of buttons to enable the manipulation of the two integer values.

The display section is an editor of type `Editor (Int,Int) (Int,Int)` and is created by combining two integer fields. Because we will not allow the fields to be edited directly, we can be certain that the fields always have a value and safely map the partial `fromJust` function over the fields.

```
1  calculatorDisplay :: Editor (Int,Int) (Int,Int)
2  calculatorDisplay = container2 field field
3  where
4    field = mapEditorWrite fromJust
5      (integerField <<@ enabledAttr False)
```

The button pad is more interesting. Because each button represents a function to be applied to the pair of integers we create an editor of type `Editor () (?((Int,Int) -> (Int,Int)))`. The read type is `()` and is used to reset the editor. The write type is the function to apply wrapped in a maybe type. It is `?None` when no button is pressed yet, and `?Just` when a button is pressed.

We construct the button pad as a matrix consisting of rows of buttons.

```
1  calculatorPad :: Editor () (?((Int,Int) -> (Int,Int)))
2  calculatorPad = matrix
3     [[sh 1,sh 2,sh 3,op "x" (*)]
4     ,[sh 4,sh 5,sh 6,op "-" (-)]
5     ,[sh 7,sh 8,sh 9,op "+" (+)]
6     ,[sh 0,ce,cl,pm]
7     ] horArt vertArt
8  where
9    sh i = functionButton (toString i)
10      (\(v,t) -> (v,10*t + i))
11   op s f = functionButton s (\(v,t) -> (f v t,0))
12   ce = functionButton "CE" (\(v,t) -> (v,0))
13   cl = functionButton "C" (\(v,t) -> (0,0))
14   pm = functionButton "+-" (\(v,t) -> (v,~t))
15   horArt = classAttr ["itasks-horizontal"]
16   vertArt = classAttr ["itasks-vertical"
17                       ,"itasks-wrap-height"]
18
19 functionButton :: String (a -> a)
20   -> Editor () (?(a -> a))
21 functionButton label fun
22   = mapEditorRW
23     (\() -> False)
24     (\b -> if b (?Just fun) ?None)
25     (button <<@ textAttr label
26             <<@ widthAttr (ExactSize 30))
27
28 matrix :: [[Editor () (?(a->a))]] UIAttributes
       UIAttributes
29   -> Editor () (?(a->a))
30 matrix es h v = rows [rows e h \\ e <- es] v
31
32 rows :: [Editor () (?(a->a))] UIAttributes
33   -> Editor () (?(a->a))
34 rows [] _
35   = mapEditorRW (const ()) (const ?None) (group UIEmpty)
36 rows [e:es] h
37   = mapEditorRW (\() -> ((),()))
38                 (\(f1,f2) -> maybe f2 ?Just f1)
39     (container2 e (rows es h) <<@ h)
```

The final step is to combine the display section with the button-pad using a loopback function that applies the function to the pair of integers as soon as a button is pressed.

```
1  calculator :: Editor (Int,Int) (Int,Int)
2  calculator
3    = mapEditorRW (\s -> (s,())) fst
4    $ loopbackEditorWrite
5      (\(s,mbf) -> (\f -> (f s,())) <$> mbf)
6    $ container2 calculatorDisplay calculatorPad
```

This calculator example highlights a property of editors that we have not yet paid much attention to. All event handling of (composed) editors happens server-side. In multi-user collaborations this is usually desirable, because progress on tasks should be available for sharing as soon as possible. However, for small self contained applications like this calculator or a date picker, we could handle most events client-side. Only the value should be synchronized with the server.

7 RELATED WORK

The space of web-based UI components and libraries for creating and composing them is large. It is dominated by Javascript (or dialects like Typescript) frameworks such as Angular [5], React [4] and Vue.js [9]. While these frameworks are useful in a common web-development technology stack, with separated front-end and back-end systems, the editor components that are the focus of this paper are constrained by the context of their embedding in iTasks. Instead of empowering front-developers to build complete client-side applications, iTasks aims to abstract from web technologies as much as possible. Custom editors are a fallback for those interactive tasks for which a generic UI is insufficient.

The extension of iTasks editors presented in this paper is therefore, not so much an alternative to web-based UI frameworks in general, but a continuation of the work of Domoszlai et al. on "Editlets" [2]. This work first addressed the need for more expressive custom user-interface components, their focus was however on running client-side code to enable interaction with the DOM and third-party Javascript libraries. Composition of "Editlets" was not yet possible. The customization and client-side execution capabilities have, since their introduction as editlets, been integrated in the iTask editor concept and improved upon. The compositionality introduced in this paper makes it possible to combine editlets with builtin editors as well as type-based generic editors.

Even though our contribution is specific to the iTask context, we can compare our work to related approaches in functional languages. In this space a common used approach for user interface specification is functional-reactive programming (FRP), using frameworks such as Reflex [8]. Our approach is quite similar to FRP, with the biggest difference that we have only a single abstraction (`Editor`) instead of separate concepts (`Event`, `Behaviour`, `Dynamic`). In general, editors in our approach are closest to `Dynamic`s in FRP, but some of the simpler ones are closer to extended versions of `Event`s or `Behaviours`. We need a single abstraction however, because the bulk of the `Editor` values in an iTask program, are produced by a type-generic function. This function can provide an editor for any (first-order) type. The guarantee that this is always possible, enables the separation of concerns between workflow and user-interface implementation, that iTasks provides. Because a UI can always be generated, it can be an optional parameter which enables the postponement of decisions about the UI. This allows programmers to consider task decomposition and workflow in isolation. If iTasks were to adopt the FRP concepts as separate types, it would imply having multiple type-generic functions for each FRP concept, which would force an iTask programmer to choose between those. Hence, she would be forced to consider UI decisions upfront. Nonetheless, composing iTask editors using the approach presented in this paper will appear similar for programmers with FRP experience. For example, the reduction of the three boolean producing buttons to a single `ItemCommand` value in the list example is the same as a `leftmost` composition of multiple events in Reflex.

8 CONCLUSION

In this paper we have presented a new composition mechanism for user interface components in iTasks (editors). We have shifted from a model-view based approach with a typed model value, to a directional read-write approach with typed inputs and outputs. Additionally we have increased control over container components and added a loopback combinator to enable local behavior.

Together, these changes enable arbitrary composition of editor components while maintaining compatibility with iTask's type-generic editors. Effectively one can now construct user interfaces for iTask tasks by mixing and matching generic editors, builtin widgets, and low-level DOM interactions (editlets) in custom compositions.

We have shown the feasibility of our approach by implementing it in iTasks, and have demonstrated the expressiveness of our approach by reconstructing a formerly monolithic real-world component as a composition of simpler components.

REFERENCES

[1] P. M. Achten, M. C. J. D. van Eekelen, and M. J. Plasmeijer. 2004. Compositional model-views with generic graphical user interfaces. In *Practical aspects of declarative programming:6th international symposium, PADL'04 (LNCS, Vol. 3057)*, Jayaraman (Ed.). Springer, Texas, USA, 39–55.
[2] László Domoszlai, Bas Lijnse, and Rinus Plasmeijer. 2014. Editlets: Type-based, Client-side Editors for iTasks. In *Proceedings of the 26Nd 2014 International Symposium on Implementation and Application of Functional Languages* (Boston, MA, USA) *(IFL '14)*, Sam Tobin-Hochstadt (Ed.). ACM, New York, NY, USA, Article 6, 13 pages. https://doi.org/10.1145/2746325.2746331
[3] László Domoszlai, Bas Lijnse, and Rinus Plasmeijer. 2014. Parametric Lenses: Change Notification for Bidirectional Lenses. In *Proceedings of the 26Nd 2014 International Symposium on Implementation and Application of Functional Languages* (Boston, MA, USA) *(IFL '14)*. ACM, New York, NY, USA, Article 9, 11 pages. https://doi.org/10.1145/2746325.2746333
[4] Facebook Inc. 2020. React - A JavaScript library for building user interfaces. https://reactjs.org/.
[5] Google. 2018. AngularJS. https://angularjs.org/.
[6] Bas Lijnse and Rinus Plasmeijer. 2011. iTasks 2: iTasks for End-users. In *Lecture Notes in Computer Science*, Marco MorazÃ¡n and Sven-Bodo Scholz (Eds.), Vol. 6041. Springer, Berlin, 36–54. http://dx.doi.org/10.1007/978-3-642-16478-1_3
[7] M. J. Plasmeijer and P. M. Achten. 2006. The implementation of iData: a case study in generic programming. In *Selected papers of the 17th international symposium on the implementation and application of functional languages, IFL'05 (LNCS, Vol. 4015)*, A. Butterfield, C. Grelck, and F. Huch (Eds.). Dublin, Ireland, 106–123.
[8] Reflex Contributors. [n.d.]. Reflex FRP. https://reflex-frp.org/.
[9] Evan You. [n.d.]. Vue.js - The Progressive Javascript Framework. https://vuejs.org/.