Editlets: type-based, client-side editors for iTasks

László Domoszlai[1,2] Bas Lijnse[1] Rinus Plasmeijer[1]

[1]Radboud University Nijmegen, Netherlands, ICIS, MBSD
[2]Eötvös Loránd University, Budapest, Hungary, Software Technology Department
l.domoszlai@cs.ru.nl, b.lijnse@cs.ru.nl, rinus@cs.ru.nl

Abstract

The iTask framework enables the construction of distributed systems where users work together on the internet. It offers a domain specific language for defining applications, embedded in the lazy functional language Clean. From the mere declarative specification of the tasks to do and their interconnection, a multi-user web application is generated which can coordinate the work thus described. User interfaces are generated automatically which is realized by using type-driven generic functions. Although this way of generating user interfaces entails a number of benefits for the programmer, it suffers from the lack of possibility to create custom user interface building blocks. In a precursory work we proposed *tasklets* for the development of custom, interactive web components. However, experimenting with real-world applications indicated that they lack some fundamental properties limiting their usability; these are the tight integration with the type-driven user interface generation, and the capability of working with shared data. In this paper, we introduce *editlets* to overcome these limitations. In addition, editlets also provide a general way to communicate *changes* instead of exchanging the whole data to reduce communication overhead.

Categories and Subject Descriptors D.1.1, H.5.2 [*Applicative (Functional) Programming, Methodology and Techniques*]: User interface management systems (UIMS)

Keywords iTasks, editlet, change based synchronization

1. Introduction

Task Oriented Programming [25, 32] (TOP) is a paradigm for designing multi-user, distributed web-applications. The *iTask system* [31], or iTasks, is a TOP framework that offers a domain specific language which is shallowly embedded in the strongly typed, lazy, purely functional programming language Clean [29, 38].

In the TOP paradigm, the unit of application logic is a *task*. Tasks are descriptions of interactive persistent units of work that maintain a typed value. When a task is executed, it has a persistent value, which may change over time reflecting the current state of the work taken place. This task value can be observed by other tasks. In iTasks, complex multi-user interactions can be programmed in a declarative style just by defining the work that has to be accomplished. The definition is given on a very high level of abstraction and does not require the programmer to provide any user interface definition. Merely by defining the workflow of user interaction, a complete multi-user web application is generated, all the details e.g. the generation of web user interface, client-server communication, state management etc. are automatically taken care of by the framework itself.

The iTask system uses generic programming [5, 18] and a hybrid static-dynamic type system [28, 41] to generate the user interface. From the programmer's perspective, it is achieved in two levels. At the most basic level, the iTask engine can be *asked* to generate a graphical user interface (GUI) for any conceivable first order model type. iTasks uses a predefined set of primitive user interface elements to generate the GUI, a client-side *editor*, for the given type, then dynamically creates an associated primitive task. On the higher level, additional user interface elements are generated *automatically* as tasks are combined together. These elements reflect the actual combinators in use and express the "flow" of the application.

Developing web applications in such a way is straightforward in the sense that programmers are liberated from these cumbersome and error-prone jobs, such that they can concentrate on the essence of the application. The iTask system makes it very easy to develop interactive multi-user applications. The down-side is that one has only limited control over the customization of the generated user interface. In real-world applications, it is often necessary to develop custom user interface elements to achieve special functionality.

To overcome this limitation, in previous work we introduced *tasklets* [14], a special *primitive task* type, for the development of custom, interactive web components. Tasklets are written in Clean and executed in a web browser using a Clean to JavaScript compilation technique [15]. In the browser, they have unlimited access to browser resources through some library functions while on the server they behave like ordinary iTasks tasks.

Using tasklets, we have successfully developed many interactive components for a wide range of applications, but we also experienced certain limitations of the technology:

1. Tasklets cannot work with shared data. As an example, it is not possible to create an interactive map, and enable multiple users to make concurrent modifications to that (e.g. add marks).
2. Tasklets are not compatible with the generic, type-driven user interface generation. User interfaces can be generated automatically for any first order type, in a compositional manner. For example, the system knows, given a user interface for type `t`, how to construct a user interface for a container type `t'` in which type `t` includes, such as a list of `t` or a tree of `t`. However, a user interface created for a tasklet belongs to a particular task, not to a type, and can therefore not be applied in the generic user interface generation.

IFL '14, October 1–3, 2014, Boston, MA, US.

http://dx.doi.org/10.1145/2746325.2746331

3. Tasklets have a simple communication interface based on the exchange of the whole underlying data, which has a high associated communication cost.

Given the experience we obtained with developing real-world applications, we revised our first approach of defining custom interactive components. The new component type is called *editlets*. Editlets solve all the aforementioned limitations while preserving compatibility: in the most basic use cases they give back the functionality of tasklets.

Editlets also have the property that the client-server communication is done in *edits*, which means that the value of the editlet is communicated through changes, that is *incremental updates*, instead of exchanging the whole value at every update. In certain cases this drastically reduces the associated communication cost (consider a source code editor component as an example).

In this paper we show how editlets can be defined, how they work and interact with the other part of the iTask system. This is done in a number of steps:

1. We extend iTasks with editlets. An editlet has an associated value type and consists of a description of the behavior of the component on the client-side, and the logic of creating and applying edits from and to its current value.
2. We develop a simple, but still realistic example of a drawing application, where multiple people can work on the same shared image, to give a taste of editlets.
3. We explain the technical background of editlets along with additional remarks on how they fit in the iTasks architecture.

The remainder of this paper is structured as follows: to set the context, we start with a short introduction of the iTask framework in Section 2. In Section 3 an overview of our old approach, tasklets, is given, where we also identify some general shortcomings of the architecture. Based on these shortcomings, we define new requirements in Section 4. The new editlet architecture is introduced in Section 5, then a small, but illustrative example is developed in Section 6. In Section 7 we briefly discuss the design of the architecture of the client-side execution used by editlets. After a discussion of related work in Section 8, we conclude in Section 9.

The iTask framework has been created in Clean. A concise overview of the syntactical differences with Haskell is in [4]. We assume the reader is familiar with the concept of type-driven generic programming and uniqueness typing [7].

2. Introduction to iTasks

Task Oriented Programming (TOP) is a paradigm that is designed to construct multi-user, distributed web-applications. The iTask system is a TOP framework that offers four core concepts for software developers:

1. *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. The task value may change over time while the task is being worked on (see [32]). The current value can be inspected by other tasks.
2. *Shared data sources* (SDS) which are abstractions of *information* that is shared between tasks. An SDS is a value of parameterized type (`ReadWriteShared r w`). The type parameters `r` and `w` model the *read* and *write* values. The shared data sources enable safe concurrent read/write access for some shared data, and offer change notifications (i.e. a task which depends on a particular SDS will be notified when the SDS has been changed by some other task).

```
:: Task a   // Task is an opaque, parameterized type constructor

// Sequential composition
(>>=) infixl 1 :: (Task a) (a → Task b)
                                  → Task b | iTask a & iTask b

// Parallel composition
(-||-) infixr 3 :: (Task a) (Task a) → Task a | iTask a

// Assigning a task to a user
:: User:==String
(@:) infix 3 :: User (Task a) → Task a | iTask a

// Shared Data Sources
:: ReadWriteShared r w
withShared :: b ((ReadWriteShared b b) → Task a)
                                  → Task a | iTask a & iTask b

// User interaction
enterInformation  :: String   → Task m | iTask m
updateInformation :: String m → Task m | iTask m
viewInformation   :: String m → Task m | iTask m

// User interaction using shared data
updateSharedInformation ::  String (ReadWriteShared r w)
                                  → Task w | iTask r & iTask w
viewSharedInformation   ::  String (ReadWriteShared r w)
                                  → Task r | iTask r & iTask w
```

Figure 1. Combinators and primitive tasks used in the paper

3. *Combinator functions* that compose tasks and SDSs into more complex tasks and SDSs.
4. *Generic interaction* with the users. A TOP framework *generates* user interfaces *generically* for any type of data used by tasks. This means that it is not necessary to design a user interface and program event handling just to enter or view some information.

For programming convenience, a large set of primitive tasks, task combinators, and types are predefined in the iTask library on top of these core concepts. Figure 1 displays a fragment of these tasks, types and combinators, that we use in this paper.

Most type definitions of the iTasks combinators contain a context restriction at the end of their type signature, e.g. `| iTask m`, defining a restriction on type variable `m`. It is similar to context restrictions on overloading as can be found e.g. in Haskell. In Clean, context restrictions not only may refer to overloaded functions, it may also refer to generic functions. The context restriction `| iTask m` means, that `m` can be of arbitrary type, provided that a class of generic functions, necessary for the iTasks run-time system, have instances for type `m`. In contrast to the overloaded use, the programmer does not need to define these instances. The Clean compiler can automatically derive instances for generic functions for any conceivable first order model type.

In this paper we only use a few simple combinator functions for sequential and parallel compositions. Task `f >>= s`, created by the monadic `>>=` combinator, first performs task `f`, then the value produced by `f` can be used by task `s` to compute any new task expression. The `-||-` parallel combinator groups two tasks of the same type in parallel and returns the result of the task that is completed first. With the assign operator `@:` a task to work on can be assigned to a specific user. It can be seen as a special case of a parallel task with the property that the task has to be performed by the indicated user.

Many functions are defined on Shared Data Sources, e.g. reading, writing, and observing. In this paper we only make use of the

Figure 2. GUIs generated for values of type `Int`.

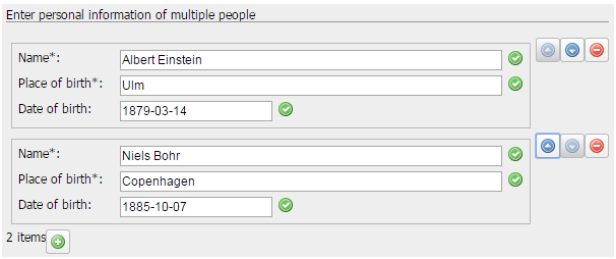

Figure 3. Generated GUI for entering values of type `[Person]`.

`withShared` function. It creates a shared data source in memory with a given initial value and a limited scope. The shared data can only be accessed by the task(s) defined in the continuation function.

In iTasks there are a family of primitive tasks for generating user interface for first order types. The system uses generic programming and a hybrid static-dynamic type system to be able to achieve this functionality. Some of these tasks are `viewInformation`, `enterInformation` and `updateInformation` to display, and to allow the user to enter or edit some data, respectively. The first argument of these functions is a brief description of what the end-user is expected to do. Commonly, these primitive tasks also have a counterpart that work with shared data instead of private, local data. In this paper we will use `updateSharedInformation` and `viewSharedInformation`.

To give a quick glimpse of iTasks, we present two very simple examples. In the first example a user is asked to enter two numbers in a sequence, after which their sum is displayed. One can see in the code how information produced by one task is passed to the next one in a monadic style. The corresponding generated user interfaces for doing the interaction are shown in Figure 2. In this example, GUIs are generated for type `Int`.

```
calculateSum :: Task Int
calculateSum
  =   enterInformation "Enter a number"
  >>= λnum1 →
      enterInformation "Enter another number"
  >>= λnum2 →
      viewInformation "The sum of these number is:"  (num1 + num2)
```

In the following example we define the task `updateAndView` delivering a value of arbitrary type `a`. It starts two tasks in parallel. One is assigned to `user1`, the other one to `user2`. Both tasks communicate via a newly created shared data source with initial value `initialValue`. For `user1` a user interface is created which enables her to update the data source, while `user2` is offered a view to follow the changes made. When a shared value is changed, all tasks that rely on it are automatically informed and refreshed.

```
updateAndView :: User User a → Task a | iTask a
updateAndView user1 user2 initialValue
  =   withShared initialValue
                 (λshared → update user1 shared
                            -||-
                            view user2 shared)

update u sh = u @: updateSharedInformation "enter information" sh
view   u sh = u @: viewSharedInformation   "view information"  sh
```

A task like `updateAndView` is very generally applicable, it can be used for any first order type for which the `iTask` class of generic functions have been generated. The user interfaces that will be generated by the generic functions depend on the concrete type the task is applied with. In the task `twoWorkers` below, `updateAndView` is applied to let Alice and Bob work together, producing a value of type `[Person]`. Using a `derive` statement, an instance for all generic iTasks functions for this type is created by the compiler. Alice is offered an interface to create a list of `Persons` (see Figure 3), and the changes she made can "real-time" be observed by Bob.

```
:: Person =
    { name          :: String
    , placeOfBirth  :: String
    , dateOfBirth   :: Maybe Date
    }

derive class iTask Person

twoWorkers :: Task [Person]
twoWorkers = updateAndView "Alice" "Bob" []
```

Although it is nice that user interfaces can be generated for any first order type, one would like to have an easy way to do something different than the default behaviour and be able to let arbitrary work be done on the client using the latest facilities offered in modern browsers.

Furthermore, in general, an arbitrary number of tasks, varying over time, may view or update shared data. This can of course lead to conflicts when multiple people want to update the same data at the same time. The default behaviour of the system is to ignore a conflicting update. The update is lost and the corresponding task is refreshed showing the most recent information known. Although it is possible to redefine this default behaviour for a specific data source, when rich clients are being used one has to be able to define how to recover from a failing update on the client.

3. The first approach: tasklets

In iTasks, because the user interface is generated and handled automatically, one has only limited control over its customization. Although, for most of the iTasks applications, this is acceptable, our experiment with real-world applications, e.g. the implementation of the Netherlands Coast Guard's Search and Rescue (SAR) protocol [26, 27], indicated that even if the functional web design is satisfactory, custom building blocks may be required for the purpose of user-friendliness.

To overcome this shortcoming, we presented an extension for the iTask system which enables the development of such components, the so called *tasklets* [14]. Tasklets are seamlessly integrated into iTasks to preserve the elegance of functional specification by hiding the behavior behind the interface of a task.

Tasklets are developed in a *single-language* (i.e. Clean), declarative manner and in accordance with the *model-view-controller* user interface design (MVC) [23]. MVC decouples the application logic (the controller), the application data (the model) and the presentation data (the view) to increase flexibility and reuse. Technically speaking, tasklets are embedded applications whose behavior is encoded by means of event handler functions. The event handlers are compiled to JavaScript so that they can be executed in the browser. They have unrestricted access to client-side resources. Using browser resources the tasklet can create custom

appearance and exploit functionality available only in the browser (e.g. HTML5 GeoLocation API). It utilizes the event-driven architecture to achieve interactive behavior. With this extension, iTasks gains similar characteristics to *multi-tier* programming languages like Links [9] or Hop [33, 34], in the sense that the same language is used to specify code residing on multiple locations or tiers, such as the client and the server.

A tasklet consists of the definition of a client-side application which has a state of type `state`, and a function which tells how to extract the result of the tasklet of type `value` from the state:

```
:: Tasklet value = ∃ state:
     { genUI      :: *World → *(TaskletHTML state, *World)
     , resultFunc :: state → value
     }
```

The first part, the definition of the client-side application is generated by the `genUI` non-pure function (the uniquely attributed type `*World` allows access to the external environment), while the transition from `state` to `value` is given by `resultFunc`. Finally, a tasklet must be turned into a primitive task to make it executable by iTasks:

```
mkTask :: (Tasklet value) → Task value
```

The life cycle of a tasklet is the following: it starts when the value of the wrapper task is requested. First, `genUI` is executed on the server to provide the user interface of the tasklet. Then, the definition of the user interface is on the fly compiled to JavaScript and shipped to the browser. In the browser, the application is executed in a tasklet container. As it runs, when its state is changed, `resultFunc` is called to create a new task value that is sent to the server immediately. The life cycle of the tasklet is terminated by the framework when the task value is finally taken by another task.

Tasklets are backed by a JavaScript compilation technique integrated with the Clean compiler. During the compilation of an iTasks application, besides the server executable running in native code, an intermediate representation of the same application in the SAPL [21] language, is also created. This intermediate language is designed to contain only the essential minimum of language features of a lazy, functional language like Clean or Haskell, while preserving the semantics. Furthermore, its syntax is carefully constructed to be easy to handle at source code level. These features enable us to perform fast source code level linking based on any initial expression. The actual JavaScript compilation is done during run-time in a demand driven way: given an expression the depending SAPL code is collected and then compiled to JavaScript on the fly. This technique has the advantage of reducing the size of the generated code to the essential minimum. The compilation technique is explained in more detail in Section 7.

The tasklet architecture enables us not only to create custom interactive components, but it also allows us to execute *arbitrary* tasks on the client. This means that we can *dynamically* choose where a task is to be executed, on the server or on the client [14]. Tasklets are just perfect for the latter purpose, but further experiments revealed that they also have certain limitations concerning the creation of interactive components:

- In iTasks the user interfaces are generated in a generic, type-driven way. The iTask system can be asked to generate a user interface for a value of type `[a]` for example, where `a` is any first order type. However, tasklets are *not integrated* with this type-driven approach. The main problem is that the user interface generated for the type `(Tasklet a)`, should produce a value of type `a` instead of `(Tasklet a)`. Thus, one should rather be able to tell the system that for a given type `a`, instead of creating the generic user interface, use a specific tasklet instance of type `(Tasklet a)` for customization.
- Tasklets *cannot work with shared data*. As an example, it is not possible to create an interactive map, and enable multiple users to make concurrent modifications to that (e.g. add marks). This is a serious limitation as working with shared data sources to achieve collaborative work is one of the main principles of iTasks.
- In certain applications tasklets have a huge associated *communication overhead*. An example of this is a tasklet which implements a syntax highlighted source code editor component. Exchanging the whole source code between the client and server every time when a part of it is changed has a big negative impact on the performance of the whole application.

We concluded that the first two of these limitations are bound to the way tasklets are integrated into iTasks, while the last one is a general limitation of the tasklet architecture. In the light of these limitations, we decided to refine the requirements and create a new component type to implement them.

4. Requirements

This is the extended list of requirements for our new component type, called *editlet*, derived from our previous experiments with tasklets:

1. It should be *general* enough that one can develop with it *arbitrary browser applications*. That basically means that editlets should have unlimited access to browser resources.
2. It should be developed in the *single language* Clean, no matter the platform it will run on.
3. It should be integrated with the *type-driven* approach used in iTasks to generate the user interface. An editlet should be registered in iTasks to be associated with a given type, such that it can be used by the system to replace the generic user interface. In this way, the user interfaces of the container types could be generated generically, while their elements may be customized with an editlet.
4. It should be able to work with *shared data*.
5. It should be able to detect and *resolve conflict situations* when working on shared data.
6. One should be able to develop such an editlet in such a way that one can *minimize the communication overhead* of the given editlet. We also require that the communication overhead related to the architecture of editlets should be as optimal as possible (e.g. the amount of the JavaScript code sent to the browser).
7. Finally, as these requirements indicate additional steps in the building process, e.g. compiling to JavaScript, linking, etc., we want these steps to be fully *integrated with the Clean toolchain*, and therefore, to be completely transparent (c.q. invisible) for the developer and for the end users.

These requirements fall into three different groups. Requirements 1,2,7 put constraints on the definition of the client application and are basically already satisfied by tasklets. Requirements 3,4 impose restrictions on the way editlets are integrated into iTasks. Finally, requirements 5 and 6 affect the design of the communication interface.

In the next section we introduce our revised approach, and present how they meet these requirements.

5. Introduction to editlets

On an abstract level, editlets consist of three parts: (1) the type of the value the editlet produces, (2) a client-side application which

```
:: Editlet value = ∃ state edit:
  { genUI       :: (*World → *(ComponentHTML edit state, *World))
  , appEditClt  :: edit  state *JSWorld → *(state, *JSWorld)
  , genEditSrv  :: value value → Maybe edit
  , appEditSrv  :: edit  value → value
  }
```

Figure 4. Editlet Interface Definition

is a stateful, event-driven application written in a single-language manner in Clean, and (3) a data synchronization interface which describes how to convert the state of the client application to the value of the editlet *via edits* and vice-versa.

On a more technical level, editlets basically consist of a set of functions which manipulate values of three types. These are the type of the `value` that the editlet is supposed to process on the server, the type of the `state` which is maintained by the client application, and, what is new compared to tasklets, the type of the data which is used for the client-server communication, dubbed `edit`.

5.1 The Editlet Interface

Figure 4 shows the type definition of the editlet interface. Editlets are defined by the means of the (`Editlet value`) record type, where the type parameter is the aforementioned `value` type. The other two types, the `state` of the client application and the `edit` type, are of no concern for a programmer applying an editlet, and therefore hidden via existential quantification.

In the editlet interface one has to define four Clean functions: `genUI` which generates the application to run on the client, and the remaining to handle edits: `appEditClt` is executed on the client, while `genEditSrv` and `appEditSrv` run on the server. Client functions potentially alter the `*JSWorld` environment. The server function has access to the 'regular' `*World` environment of any side-effectfull program. The access to the `*World` enables the `genUI` function to do some IO during the initialization of the client application, e.g. for the purpose of templating. The `*JSWorld` environment is also used by the event handler functions to interface with the JavaScript foreign function interface explained in Section 6.2.

5.2 The Client Side Application

The `genUI` function produces the definition of the application to run on the client which is of type (`ComponentHTML edit state`) (see Figure 5). It is an event driven application which has an internal state of type `state`. The application basically consists of some HTML code that will be generated by Clean functions and a list of event handlers to handle the interaction with the end-user.

The actual user interface (`html` field) can be given by any data structure provided that it has an instance of the function class `toHtml`. In this paper we will use an overly simplified ADT to provide HTML definitions. In reality, one probably would like a more sophisticated way to have full, low-level control over the definition of HTML elements. This can be done in an abstract, monadic way like in Wash [37] or by an XML like domain specific language similar to that of Hop [34]. Furthermore, as the `genUI` function of the editlets is non-pure, it enables us to utilize some template mechanism similar to e.g. Yesod [35] or Snap [8]. However, providing any particular tool here is beyond the scope of this paper.

The run-time behavior of an editlet is encoded in a list of event handler functions given in the `eventHandlers` field. It is also possible to create and attach event handlers dynamically, but this facility is not important for the explanation. Event handlers are defined using the (`ComponentEvent edit state`) type. Its only data constructor has three arguments: the identifier of an HTML

```
:: ComponentHTML edit state =
    { html          :: HtmlDef
    , eventHandlers :: [ComponentEvent edit state]
    }

:: HtmlDef = ∃a: HtmlDef a & toHtml a

:: ComponentEvent edit state
       = ComponentEvent HtmlElementId HtmlEventName
                        (ComponentEventHandlerFunc edit state)

:: HtmlElementId:==String
:: HtmlEventName:==String

:: ComponentEventHandlerFunc edit state
       :== (JSObj state *JSWorld →
                    *(state, ComponentEdit edit state, *JSWorld))

:: ComponentEdit edit state
    = Edit edit (Conflict state *JSWorld →
                    *(state, ComponentEdit edit state, *JSWorld))
    | NoEdit

:: Conflict:==Bool
```

Figure 5. The `ComponentHTML` type

element, the name of the event and the event handler function itself. During the instantiation of the editlet on the client, the event handler function is attached to the given HTML element to catch events of the given name.

The event handler functions work on the JavaScript event object (a `JSObj` typed value in Clean) and the current internal state of the editlet. They can change the state by returning a new one, and they can also change the value associated with the editlet by returning an edit, wrapped in a value of type (`ComponentEdit edit state`) (see Section 5.3). The event handlers also have access to the browser resources, e.g. HTML Document Object Model (DOM), to maintain their appearance for example, which is done through a foreign function interface (FFI).

From the point of event handlers, manipulating browser resources is a non-pure behavior. Therefore, FFI functions can be used as IO access is done in Clean, through uniquely attributed types. That is what the unique `*JSWorld` type is used for, in a similar way as the unique `*World` type is used on the server. Introducing a new type to have IO on the client has the advantage that it reflects for different purposes of client and server side code. The server code can access all resources of the server computer, like the file system, not available on the client; at the same time, the client code has external access to a resource accessible only on the client.

5.3 Synchronizing Clients and Server

The synchronization is based on the exchange of `edit` typed values, edits, between the clients and the server. The architecture is very similar to master/slave replication, where the server side shared data takes the role of the master, while the editlet instances in the browsers take the role of slave replicas. In such an architecture, updates are directly committed to the master to avoid distributed synchronization issues.

When a client wants to inform the server that something is changed which might have consequences for the value maintained by the server, it generates an edit which is sent to the server. The server applies the edit to its value, and if this value is shared, the same edit is distributed to the other tasks that share this value, in particular the editlet clients who depend on it. It sounds straightfor-

ward, but there are two serious issues we have to deal with in such a situation:

- First of all, the edit generated by the client, may no longer be valid and therefore cannot be used to calculate the new server value. This can happen if the value stored at the server is shared and the edit is created against a different (old) version of the underlying data. In the meantime the shared data might be changed by some other application, or some other editlet instance. Hence, one can have an *update conflict*, and editlets must be able to resolve this.
- Because update conflicts can happen on the server, the clients should commit updates in a synchronous manner to be sure that the update went through (and to try to resolve the conflict otherwise). However, there can be a significant network latency between the clients and the server and the lag could make the component highly unresponsive.

The first issue is solved by a standard technique of using a version attribute. The server side shared data has a version number which is increased every time the data is changed. When the server sends an edit to a client, it goes along with the current version of the shared data. Later on as the client generates edits, they are sent to the server along with this locally stored version number, so it can be compared with the actual version of the shared data on the server. If the numbers differ, a conflict is detected.

To solve the second issue we decided to use *optimistic concurrency control* [24]. Instead of sending the edits to the server in a synchronous manner, it is sent asynchronously and it is assumed that there will be no conflict. If, however, a conflict situation happens, the client is informed by a callback mechanism. In this way one can either roll back the change, or try again by sending a new edit to resolve the conflict.

In the editlet architecture, client-side edits are generated by the event handlers by means of the (`ComponentEdit edit state`) type (see Figure 5). An edit is provided along with a continuation function which is executed when this edit is accepted or rejected. In the latter case the first argument of the continuation function is set to `True` to indicate conflict.

The edits sent by the client to the server are applied to the server side `value` by the `appEditSrv` function that is used to calculate the new value. This function is, in contrast to its client-side counterpart, pure, as the only responsibility of the server is to hold pure data of type `value`.

In certain cases the server makes use of the `genEditSrv` function which is also defined in the editlet interface. It is used to find out what the difference is between an old and new value that is just calculated. This function is only used when the shared data is *not* changed by an editlet, but some other type of task or application. Assume e.g. that the shared data is stored in a shared file which is overwritten by someone else. In such a case there is no edit available to be distributed to the clients, it must be generated.

When the server has deduced that clients need to be informed about its changed value, it sends the corresponding edit to the client which calculates the consequences for its local state by applying the `appEditClt` function as defined in the editlet interface. It may not only cause a change to the state of the editlet. As it is non-pure, it can adjust its appearance according the received data as well.

To illustrate the role of these functions better, consider the following use case:

1. User interaction on the client triggers the execution of one of the event handlers of the editlet. New data is generated and an edit is created, a value of type (`ComponentEdit edit state`). The edit is sent to the server to synchronize with the server value. The logic of saving the new data in the client state (or roll it back) is encoded in the continuation function which is attached to the edit.
2. The edit, along with the local version number of the shared data, is sent to the server.
3. The server compares the version numbers. If they are the same, the edit is applied by the `appEditSrv` function, the version number is increased, a notification is sent to the client and the edit is distributed to the other instances of the editlet. If the version numbers differ, only a notification is sent to the client to be informed about the conflict.
4. At the client who transmitted the edit, the continuation function is executed to permanently commit or roll back the changes. As a reaction to a conflict, it can decide to send a new edit and the workflow continues with Step 2.
5. At all the other clients, the edit sent by the server is applied by the `appEditClt` function.

The architecture guarantees that the edits and the conflict/success notifications are delivered to the clients in that order as they appear on the server. Based on this guarantee, it is assumed that the `appEditClt` and `appEditSrv` functions never fail. Working with version numbers and edits furthermore has an advantage that the synchronization between clients and server can be achieved with a minimal amount of communication.

5.4 Integrate with iTasks

Finally, iTasks must be made aware of the editlet. For this, one has to overwrite the default UI rendering logic of iTasks by providing an explicit instance of a generic function for the value type of the editlet. Section 6.1 demonstrates how to do that in practice.

6. Editlets by example

In the previous section we sketched out the big picture of the editlet architecture; in this section we show how it works in detail by developing a small, but illustrative example, where users can work together to draw an image.

```
workTogether :: User User a → Task a | iTask a
workTogether user1 user2 initialValue
  =   withShared initialValue
                (λshared → update user1 shared
                           -||-
                           update user2 shared)
```

In this example we also want to explain how update conflicts are being handled. The general applicable task `workTogether` we use here is a variant of the `updateAndView` task as introduced in Section 2, where now both workers work on and update the same shared data such that update conflicts may arise.

```
:: Drawing = Drawing [Shape]

:: Shape = Line   Color        Int Int Int Int
         | Rect   Color Filled Int Int Int Int
         | Circle Color Filled Int Int Int Int

:: Color = Yellow | Red | Green | Blue | Black
:: Filled:==Bool

drawingExample :: Task Drawing
drawingExample = workTogether "Alice" "Bob" []
```

Now we are going to apply this general applicable task to a concrete type in a `drawingExample` where two workers, Alice and Bob, need to work together to produce a drawing of type `Drawing`. A drawing is simple and just consists of a list of shapes of lines, rectangles, and circles.

The drawingExample task can be executed as is, but would offer an interface to Alice and Bob for editing a list of shapes, similar to the list of persons interface we showed in Section 2. It would work and ensure that Alice and Bob can only create type correct drawing values, but our friends would certainly not be very happy with the look of the generated interface. We may assume that instead they would prefer to make drawings using some dedicated drawing application to produce drawings in a natural way. The behaviour should be such that whenever Alice draws something, Bob would see what has been drawn, and the other way around. When they are both drawing at the same time, we need to be able to handle both their contributions without upsetting them.

Changing the visualisation furthermore does not have any other consequences for the tasks being defined. The underlying technology remains completely hidden when type Drawing is used. Yet we want to obtain a nice drawing tool in the browser for free whenever a value of type Drawing is being used by some task. What does the iTask system developer have to do to make this possible?

6.1 Specialization

First, one has to overwrite the default GUI rendering for type Drawing of the iTask system and tell it to use an editlet for dealing with Drawings instead.

```
gEditor{| Drawing |} = withEditlet painterEditlet

painterEditlet :: Editlet Drawing
painterEditlet
  = { genUI      = λworld → (painterGUI, world)
    , appEditClt = updateClient
    , genEditSrv = calculateEditsServer
    , appEditSrv = updateServer
    }
```

This can be realized by defining an explicit instance of the generic gEditor function for type Drawing. Next we have to define all the components of the editlet record: the generator of the client-side painter application painterGUI (see Section 6.3) and the three functions (updateClient, calculateEditsServer, and updateServer) to handle the synchronization between the clients and the server (see Section 6.4).

6.2 The Foreign Function Interface to JavaScript

Before we can explain how the painting application can be defined, we need to explain how the FFI (Foreign Function Interface) to JavaScript and to the DOM of the browser is offered.

Figure 6 shows a subset of types and functions of the iTasks JavaScript FFI which is used in this example. The FFI can be split into two groups. The first group deals with JavaScript values. For this purpose we use the (JSVal a) type. This type is a pointer to a JavaScript value, which real type is *unknown*. Its type parameter is just a phantom type used in higher order APIs. However, we use a special technique to be able to deal with JavaScript values in a type safe manner. This technique is based on overloading the built-in dynamic type system of Clean to generate type information for a given JavaScript value [13]. This type information can be used to pattern match on the actual type of the value. The type information is generated by the fromJSVal function, but, based on this function, unsafe instances are also available to be able to write more concise code in case the type of the value is known reliably.

In the second group there are functions to deal with attributes and methods of objects. These are accessing members of objects (.#), accessing elements of the DOM (getElementById), reading an attribute of an object (.?), assigning value to an attribute of an object (.=) and calling a method of an object (.$?, .$).

To illustrate the usage of the interface, we can already define some utility functions to deal with a HTML5 canvas element

```
:: JSVal a // Pointer to a JavaScript value of some type
:: JSObject
:: JSObj :== JSVal (JSObject) // Unknown JavaScript value

// Convert foreign value to Clean dynamics
fromJSVal :: (JSVal a) *JSWorld → *(Dynamic, *JSWorld)

// Unsafe functions converting foreign values
jsValToString :: (JSVal a) → String
jsValToInt    :: (JSVal a) → Int

// Handling JavaScript objects:

class JSObjAttr a
instance JSObjAttr String
instance JSObjAttr Int

:: JSObjSelector

// Select an attribute of object
class (.#) infixl 3 s :: s t → JSObjSelector | JSObjAttr t
instance .# (JSVal o)
instance .# JSObjSelector

getElementById :: String → JSObjSelector

// Read an attribute, assign a value, call a function
.? :: JSObjSelector *JSWorld → *(JSVal r, *JSWorld)

(.=) infixl 2 :: JSObjSelector v → *(*JSWorld → *JSWorld)
(.$?) infixl 1 :: JSObjSelector a
       → *(*JSWorld → *(JSVal r, *JSWorld)) | ToArgs a

// Drop return value of the application function
(.$) infixl 1 :: o a → *(*JSWorld → *JSWorld) | .$? o & ToArgs a
```

Figure 6. The subset of JavaScript FFI used in the paper

which is used by our component to draw graphics. Its getContext method returns a rendering *context*, a built-in HTML5 object, with many properties and methods for drawing paths, boxes, circles, text, images, and more. The definition of some of the utility functions are neglected as they are straightforward to implement based on the others:

```
// Context provides methods and properties for drawing on the canvas
:: JSCanvasContext
:: Context :== JSVal JSCanvasContext

// Return the context of a canvas given by HTML id
getContext :: String *JSWorld → *(Context, *JSWorld)
getContext canvasId world
    = (getElementById canvasId .# "getContext" .$? ("2d")) world

// Clear a canvas for redrawing
clearCanvas :: Context *JSWorld → *JSWorld

// Draw a line using the given color, coordinates and context
drawLine :: Context Color Int Int Int Int *JSWorld → *JSWorld
drawLine context color x1 y1 x2 y2 world
  # world = (context .# "beginPath" .$ ()      ) world
  # world = (context .# "strokeStyle" .= color) world
  # world = (context .# "moveTo" .$ (x1, y1)   ) world
  # world = (context .# "lineTo" .$ (x2, y2)   ) world
  # world = (context .# "stroke" .$ ()         ) world
  = world

drawRect   :: Context Color Filled
                      Int Int Int Int *JSWorld → *JSWorld
drawCircle :: Context Color Filled
                      Int Int Int Int *JSWorld → *JSWorld
```

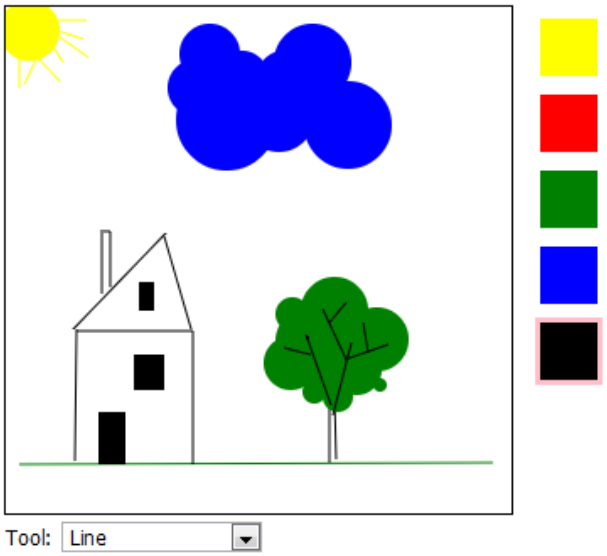

Figure 7. The Painter Application Component

```
// Draw an arbitrary shape using the specialized functions
draw :: Context Shape *JSWorld → *JSWorld
```

These functions are intuitive to read and write as their structure mimics the structure of their imperative counterparts. The names of the operators are also chosen carefully that the individual calls resemble to the corresponding JavaScript commands. Still, the functions are written in the single-language manner in Clean enabling access to all the features of a pure, lazy functional language.

6.3 The Painting Application

The application we want to create on the client is shown in Figure 7. Below the actual canvas, the user can choose the tool from a drop-down list, and the current color can be chosen by clicking on one of the small boxes next to the canvas.

To keep the example illustrative, it is not allowed to delete or modify an already drawn shape. That would add much complexity to both the user interface and the synchronization parts, thus hampering the comprehension of the main idea. Handling conflict situations, depending on the actual task, can be arbitrarily hard, and there is no "proper" logic to do it. Editlets offer a general mechanism to build *any* customized conflict resolution logic.

Since in the editor on the client one can only add, but cannot remove or modify shapes, the incremental updates can be given by the list of newly added shapes. Therefore, `[Shape]` is used as the `edit` type for this application.

The following record type is used as the state of the client-application:

```
:: PainterState = { selectedTool  :: Tool
                  , selectedColor :: Color
                  , currentOrigin :: Maybe (Int, Int)
                  , currentShape  :: Maybe Shape
                  }

:: Tool  = TLine | TRect | TRectF | TCircle | TCircleF
:: Color = Yellow | Red | Green | Blue | Black
```

The `selectedTool` and `selectedColor` fields contain the currently selected tool and color, respectively. For understanding the remaining two fields, `currentOrigin` and `currentShape`, we need to explain how the drawing of a given shape takes place.

A very simple approach could be to ask the user to select two points on the canvas and the shape is drawn between these points. It is very tempting to follow this approach, mainly because of its simplicity. However, in the same time, it would render our component completely unusable, as it is hard to draw images for a human without constant visual feedback. Thus, the usual approach is taken: drawing of a shape starts when the user presses the left mouse button, and ends when it is released. As long as the left button is pressed, while the user moves the mouse, the current appearance of the shape is continuously updated on the screen.

Therefore, the `currentOrigin` field contains the start coordinates of the shape being drawn, while the `currentShape` contains the drawn shape itself. They only contain an actual value while the left mouse button is pressed.

Finally, some explanation on the implementation of this mechanism. In JavaScript, it can most easily be done by using a temporary drawing canvas to overlay a permanent canvas. The permanent canvas contains all the shapes which are already "committed", while the shape being drawn is put on the temporary canvas, which can be cleared and redrawn as it contains only that one. When the user releases the mouse button, the final shape is copied to the permanent canvas.

The actual user interface is created by the `painterGUI` function. It generates the two canvases discussed before, small boxes as HTML DIV elements for the color selectors to the right of the canvases, and a drop down list just under the canvases for choosing the current tool (please note that the code below is slightly simplified for presentation purposes):

```
painterGUI :: ComponentHTML [Shape] PainterState
painterGUI = { html          = DivTag [] [canvases:editor]
             , eventHandlers = eventHandlers
             }
where
  canvases =
    DivTag [StyleAttr "position: relative; float: left;"] [
              CanvasTag [IdAttr "pcanvas"] [],
              CanvasTag [IdAttr "tcanvas",
                         StyleAttr "position: absolute;"] []
            ]

  editor = [
    DivTag [StyleAttr "float: right;"] (map selector colors),
    DivTag [StyleAttr "clear: both;"] [],
    DivTag [] [Text "Tool:",
               SelectTag [IdAttr "tool"] (map tag tools)]
   ]

  tools  = [TLine, TRect, TRectF, TCircle, TCircleF]
  colors = [Yellow, Red, Green, Blue, Black]

  tag tool = let text = toString tool
             in OptionTag [ValueAttr text] [Text text]

  selector color = DivTag [IdAttr (mkId color),
    StyleAttr ("background-color:" +++ toString color +++ ";")] []

  colorEvent color
    = ComponentEvent (mkId color) "click" (onSelectColor color)

  eventHandlers = map colorEvent colors ++ [
        ComponentEvent "tcanvas" "mousedown" onMouseDown,
        ComponentEvent "tcanvas" "mouseup"   onMouseUp,
        ComponentEvent "tcanvas" "mousemove" onMouseMove,
        ComponentEvent "tool"    "change"    onChangeTool
    ]
```

The list of event handlers is associated with the elements of the user interface: one for each of the color selectors, one for the drop down

list of the available tools, and one for each of the mouse events we use to handle drawing.

As the user clicks on one of the color selectors, the border of the selected color box is highlighted and the state is changed according to the new selection, but no edit is generated. The selected color is in the first argument of the event handler as partially applied functions are used for this purpose:

```
:: PaintEventHnd :== JSObj PainterState *JSWorld
  → *(PainterState, ComponentDiff [Shape] PainterState, *JSWorld)

onSelectColor :: Color → PaintEventHnd
onSelectColor color e state world
  # world = foldr (setBorder "white") world allBoxes
  # world = setBorder "pink" (e .# "target") world
  = ({state & selectedColor = color}, NoEdit, world)
where
  allBoxes = map (getElementById o mkId) colors
  setBorder color el world
      = (el .# "style" .# "borderColor" .= color) world
```

When the tool is changed, we just read the new identifier and set it in the state:

```
onChangeTool :: PaintEventHnd
onChangeTool e state world
  # (idx, world) = .? (e .# "target" .# "selectedIndex") world
  # (os, world)  = .? (e .# "target" .# "options") world
  # (tool, world) = .? (os .# jsValToInt idx .# "value") world
  = ({state & selectedTool = fromString (jsValToString tool)},
     NoEdit, world)
```

The drawing begins when the user pushes the mouse button. Then we read the current position of the mouse and set it in the state to indicate that drawing is in progress:

```
getCoordinates :: JSObj *JSWorld → *((Int, Int), *JSWorld)
getCoordinates e world
  # (x, world) = .? (e .# "layerX") world
  # (y, world) = .? (e .# "layerY") world
  = ((jsValToInt x, jsValToInt y), world)

onMouseDown :: PaintEventHnd
onMouseDown e state world
  # (coordinates, world) = getCoordinates e world
  = ({state & currentOrigin = Just coordinates}, NoEdit, world)
```

The actual drawing happens when the mouse is moved *while* the mouse button is pressed. Thus, in the `onMouseMove` event handler function, first the presence of the `currentOrigin` coordinates must be checked:

```
onMouseMove :: PaintEventHnd
onMouseMove e state world
  = case state.currentOrigin of
      Just coordinates = onDrawing coordinates e state world
      Nothing          = (state, NoEdit, world)
```

If they are set, then using these coordinates and the current coordinates of the mouse, along with the current color and tool, we can create a shape to draw it to the *temporary* canvas (which is cleared before that). Finally, the shape is saved in the state as it will be needed to finalize the drawing:

```
onDrawing :: (Int, Int) → PaintEventHnd
onDrawing (ox, oy) e state world
  # ((x, y), world) = getCoordinates e world
  # shape = case state.selectedTool of
      TLine    = Line   state.selectedColor       ox oy x y
      TRect    = Rect   state.selectedColor False ox oy x y
      TRectF   = Rect   state.selectedColor True  ox oy x y
      TCircle  = Circle state.selectedColor False ox oy x y
      TCircleF = Circle state.selectedColor True  ox oy x y
  # (tempcontext, world) = getContext "tcanvas" world
  # world = clearCanvas tempcontext world
  # world = draw tempcontext currentShape world
  = ({state & currentShape = Just shape}, NoEdit, world)
```

The drawing is finalized when the mouse button is released: the temporary canvas is cleared and the shape saved by `onDrawing` is copied to the permanent canvas:

```
onMouseUp :: PaintEventHnd
onMouseUp e state world
  # (tempcontext, world) = getContext "tcanvas" world
  # world                = clearCanvas tempcontext world
  # (edit, world) = case state.currentShape of
      Just shape
        # (context, world) = getContext "pcanvas" world
        = (addShape shape, draw context shape world)
      Nothing
        = (NoEdit, world)
  = ({state & currentOrigin = Nothing, currentShape = Nothing},
     edit, world)
```

This is the only point of user interaction when the value associated with the editlet can change: when the user releases the mouse button, and the mouse moved since the button was pressed, that is the `currentState` field of the state contains a shape. In this case the shape is copied to the permanent canvas and an edit is generated:

```
addShape :: Shape → ComponentEdit [Shape] PainterState
addShape shape = Edit [shape] callback
where
  callback True state world
        # (context, world)  = getContext "pcanvas" world
        = (state, addShape shape, draw context shape world)
  callback False state world
        = (state, NoEdit, world)
```

As it is explained in Section 5, the client part of the editlet applies edits to the shared server value in an asynchronous manner. There is a continuation function associated with the edits which is executed when the edit is finally applied or rejected. In this particular example, we take a highly optimistic approach: the new shape is drawn to the canvas in the same time when the edit is created. Later on, if the edit is rejected, the shape is drawn to the canvas again (to be the top most shape on the canvas again) and the same edit is tried again. If it is accepted we are fine, the shape is already the top most on the canvas.

When the notification of the rejection of a previous edit is triggered on the client, the client state had already been synchronized with the server value (the edit(s) causing the conflict on the server are applied to the client state). Further considering that an edit, in this particular case, can describe the addition of new shapes only (thus does not depend on the current value to be applied to), it is safe to reapply the rejected edit at that point.

6.4 Synchronization Functions

So far, we have a function to generate the user interface (the `painterGUI` function), which also describes when and how to generate edits on the client. To finish our editlet, we need to provide the rest of the synchronization interface. These are the functions for the `genEditSrv` (the function `updateClient`), `appEditSrv` (the function `calculateEditsServer`) and `appEditClt` (the functions `updateServer`) fields of the `(Editlet value)` record type.

```
updateClient :: [Shape] PainterState *JSWorld
                                  → *(PainterState, *JSWorld)
updateClient edit state world
  # (context, world) = getContext "pcanvas" world
  = (state, foldl (λworld s = draw context s world) world edit)
```

```
calculateEditServer :: Drawing Drawing → Maybe [Shape]
calculateEditServer (Drawing oldss) (Drawing newss)
  = case drop (length oldss) newss of
            []   = Nothing
            edit = Just edit

updateServer :: [Shape] Drawing → Drawing
updateServer ns (Drawing ds)  =  Drawing (ds ++ ns)
```

In this particular case the state of the client-application, the `PainterState` does *not* actually need to store a client-side counterpart of the server value. We do not need to traverse that data any time on the client, only have to draw the shapes to the canvas in the correct order. It is guaranteed that edits are delivered in the proper order to the clients. Hence the `updateClient` function just updates the user interface by drawing the new shapes to the canvas, and does not modify the state value.

On the server, an edit sent by one of the clients, a list of new shapes being added, is handled by the function `updateServer`. It just appends the list of new shapes to the shapes that already have been drawn and collected on the server in the value of type `Drawing`.

Finally, the `calculateEditServer` function also exploits the fact that the original drawing cannot be modified: it calculates the difference between an old and a new drawing by determining the number of shapes that might have been added.

7. The architecture of client-side execution

A crucial point of a single-language solution is the way the JavaScript code is produced and handled. In a single-language setting, client and server code is mingled, and, unless there is special syntactic construction introduced in the language for indicating which code is intended for client or server, the code cannot be separated during compilation. This can have the consequence that the whole application, including the code which is only relevant to the server, is compiled to JavaScript and shipped to the browser. This results in an explosion of code that not only causes huge communication overhead, but also a waste of browser resources. From security perspective, shipping unnecessary cross-compiled server code to the client, would also expose the structure of the server, which can help to reveal any potential weaknesses.

To overcome this issue in iTasks, we developed a special JavaScript compilation technique integrated with the Clean language. The compilation technique has four key components: (1) the SAPL [21] language, (2) a compiler extension, (3) run-time support, and (4) the SAPL compiler infrastructure [12], which is a library to handle SAPL source code. This library supports low level functions e.g. parsing SAPL source code, program transformations, and it also provides high level functionality, e.g. linking. A full-featured SAPL to JavaScript compiler [15] is also implemented.

The first component, the SAPL language, is an intermediate language designed to contain only the essential minimum of language features of a lazy, functional language like Clean or Haskell, while preserving the semantics. Furthermore, its syntax is carefully constructed such that it can be easily handled at source code level. These properties makes it perfect for efficient source code level *linking*, and for fast *cross compilation* as a source language.

The second component is a Clean compiler extension. During the compilation of an iTasks application, besides the server executable running in native code, an intermediate representation of the same application in the SAPL language is also created. This extension is seamlessly integrated with the Clean compiler, it just transparently creates a directory, along with the native binary, which contains all the SAPL source code.

The third component is the run-time support. During the execution of an application, the Clean run-time can be asked to provide the SAPL source code needed for the evaluation of an *arbitrary* Clean expression.

Using the SAPL expression, the last component, the SAPL library, is utilized to recursively collect the SAPL code the expression depends on, using the SAPL source code of the application. Then, the collected SAPL source is ran through an iTasks specific, *per client* caching mechanism. Its task is to filter out the functions which have already been sent to a given client in the corresponding session. The SAPL functions which are not yet on the client, are on the fly compiled to JavaScript and shipped to the browser.

This overall architecture enables to reduce the communication cost to the potential minimum and to preserve as much browser resources as possible.

8. Related work

There are three main groups of works that are closely related to editlets. These are the multi-tier programming languages, including our previous approach with tasklets, JavaScript cross-compilers, and the theory of change based bidirectional transformations.

Multi-tier programming languages

Several other languages address multi-tier programming. In the imperative world the most modern approach is the Google Web Toolkit (GWT) [1], Google Dart [3] and Node.js [36]. GWT utilizes a Java to JavaScript compilation technique for building complex browser-based applications. GWT fosters classical GUI programming where widgets can be developed using a programming model comparable to that of editlets.

The Dart language and the Node.js framework take a different approach. They enable multi-tier programming by providing a run-time environment of their languages for both client and server side. The language of Node.js is JavaScript, which is native in the web browsers; the framework also provides a run-time environment, including IO libraries, for the server side. Dart is a programming language developed by Google specially designed for web application engineering. On the client, it compiles to JavaScript, on the server it is executed by a Dart virtual machine.

These aforementioned systems have a more general approach than iTasks and editlets, but they still share the idea of using the same language on both client and server side and implicitly bridging the communication between them.

Hop [33, 34] uses a declarative approach. It is a dedicated web programming language with a HTML-like syntax built on top of Scheme. Hop uses two compilers, one for compiling the server side program and one for compiling the client-side part. The client-side part is only used for executing the user interface. Hop uses syntactic constructions for indicating client and server part code. The application essentially runs on the client and may call services on the server. In contrast, an iTasks application essentially runs on the server and may execute services, editlets, on the client.

Links [9] and its extension Formlets is also a functional language-based web programming language. Links compiles to JavaScript for rendering HTML pages, and SQL to communicate with a back-end database. In a Links program, the keywords `client` and `server` force a top-level function to be executed at the client or server respectively.

The iTask framework differs from the latter two by fostering a non view-centric approach even in the component development. Links and Hop have extended syntax for embedding XML descriptions in the language; this is used to mix the user interface definition and the behavior of the application. During editlet development the model-view-controller user interface design is enforced to separate these roles.

Another important difference is that editlets blur the boundaries of different tiers. Links uses location annotations, Hop utilizes special syntactic construction to denote the target tier of a given function or expression. In editlets this is implicit (basically the controller role runs in the browser) but unconcerned. If a function is pure, it does not matter where it is executed. If it is not pure, the available resources are controlled statically by the signature of the function.

As for iTasks, we already compared our previous approach, tasklets, to editlets in Section 3. However, there are also earlier implementations of similar features utilizing a Java written SAPL interpreter [21] as a browser plug-in. The iEditors [22] enables the development of interactive web UI elements as editlets do, however, it does not allow direct access to browser resources, therefore its applicability is restricted to functionality provided by the plug-in. As a consequence, it does not have the single-language property either, because for some functionality the plug-in has to be extended using Java. There also had been client-side task evaluation attempts for an early version of iTasks using the same plug-in based interpretation technology [30]. However, our approach, to give one general solution for both of the problems is a novel strategy.

JavaScript cross-compilers

JavaScript cross-compilation is a subject that has drawn much attention in the last years as web applications getting richer and richer to improve the web experience. Virtually every modern programming language has at least one JavaScript cross-compiler, thus we limit ourselves to the comparison of some very closely related technologies and concentrate on the high level architecture only (an explanation of the compiler can be found in [15]).

The most relevant technologies are the aforementioned Links and Hop languages. Both languages are functional just like Clean, however unlike these languages, Clean is a *lazy* functional language. Although this property has a big impact on the actual compilation technique, it only slightly affects the high level architecture. The main difference between them is rather that these languages are specially designed as multi-tier. This has a consequence that the client and server side code is distinguished by special syntactic constructs at source code level. Thus, the client and server side code can be separated during compilation time. However, this does not take into consideration the dynamic behavior of the application. In our architecture, only those functions are shipped to the browser which are actually requested at *run-time*; the difference can be significant in some executions.

Another relevant technology is GHCJS [2], the most advanced JavaScript cross-compiler for the Glasgow Haskell Compiler (GHC). As Haskell and Clean are from the same family of the functional languages, and they are actually very similar in many viewpoints, it is worthwhile to compare the architecture of its flagship JavaScript cross-compiler and the architecture of our cross-compiler.

In contrast to our solution, GHCJS takes a module based approach. During compilation, along with the object files, the JavaScript version of the modules are also generated. The final client-side "executable" is produced by simply combining the JavaScript versions of all the referenced modules together. Although this approach definitely has the advantage that no runtime component is necessary in the architecture, it also suffers from producing an explosion of code: Haskell applications tend to use many libraries and many generically created functions which are blindly merged resulting in an enormous amount of JavaScript code.

Although most of the cross-compilers use a kind of simplified core language as the source of the compilation, the idea to use this intermediate core language for linking as well, thus reducing the size of the output, is a novel idea as far as we know.

Change based bidirectional transformations

In general, bidirectional transformations are a mechanism for maintaining the consistency of two related sources of information [10]. In the field of bidirectional transformations, the most closely related work is the so called bidirectional *lenses* [16]. Within lenses, *edit lenses* [19, 39] bear the most resemblance to our data synchronization interface.

In a nutshell, edit lenses define bidirectional transformations between pairs of connected structures, where each of the two structures may contain information that is not present in the other (also known as symmetric lenses [20]). Moreover, edit lenses work with descriptions of changes to structures, rather than with the structures themselves. In practice that means that with each of the connected structures there is an additional associated data structure, an *edit language*, to define the changes of the original structure. The actual changes, the *edits*, of the structures are converted to each other by a bidirectional transformation, the edit *lens*, in a *stateful* manner.

It has two main differences comparing with editlets: (1) our case is *not symmetric* in the sense that the value held by the server does not contain information that is not synchronized with the clients; (2) there is only one edit language.

However, from another perspective, these are not important differences. If the lens is applied on the client (in the `appEditClt` function), which is stateful, then for the external observer the state of the edit lens and one of edit languages is hidden. This means that the necessary synchronization functions of an editlet can be easily defined based on an existing edit lens implementation.

Finally, another group of works addresses the generation of edits [11, 17, 40]. Although these are interesting for our case, adapting such a framework is beyond the scope of this paper.

9. Conclusion and future work

In this paper we have presented an extension to iTasks, for the development of interactive web components in a single-language manner. This extension is based on our previous experiment in the same topic, called tasklet. We have identified several shortcomings of this previous approach based on experiments with real-world applications, what we used to set new requirements. Based on this new set of requirements, we have designed a new component type, called editlet, which is superior to tasklets from many perspectives. The main properties of the editlet architecture are the following:

- Editlets enable the development of *arbitrary browser applications* in the *single language* Clean.
- Editlets are integrated with the *type-driven* approach used in iTasks to generate the user interface. The default behavior of the user interface generation can be overwritten by registering an editlet to be associated with a given type.
- Editlets can *work on shared data*, moreover, it is done in a way which enables the developer to *deal with conflicting updates* efficiently in an asynchronous manner to keep the user interface responsive.
- The client-server communication is done in changes, called edits, instead of exchanging the whole shared value. In certain applications, it dramatically reduces the communication cost.
- Editlets are based on an advanced client-side execution architecture to reduce the amount of generated JavaScript code to the minimal possible. The JavaScript cross-compiler is integrated transparently in the Clean tool chain.

Although editlets is an iTasks extension, these properties make it interesting in a global perspective. The client-side execution architecture, the edit based communication interface and the type-

based approach are all interesting properties in their own right, and can be integrated with other languages and frameworks.

As for the future work, we are planning two major tasks. First we would like to upgrade iTasks SDSs to be based on edits as well. The edit type would be attached to the value type by functional dependencies to be global for the whole system, and the SDSs would be updated by edits even on the server. In this way we could get rid of the `genEditSrv` function from the editlet definition.

The other task affects the JavaScript cross-compiler. Currently it is not safe in the sense that during code generation some sensible data may be shipped to untrustworthy clients. We would like to overcome this issue by investing in techniques similar to those developed for the Links language [6].

References

[1] GWT: The Google Web Toolkit site. URL `http://code.google.com/webtoolkit/`.

[2] GHCJS. URL `https://github.com/ghcjs/ghcjs`.

[3] Dart: structured web programming, 2011. http://www.dartlang.org/.

[4] P. Achten. Clean for Haskell98 programmers - a quick reference guide, July 13 2007. URL `http://www.mbsd.cs.ru.nl/publications/papers/2007/achp2007-CleanHaskellQuickGuide.pdf`.

[5] A. Alimarine. *Generic Functional Programming - Conceptual Design, Implementation and Applications*. PhD thesis, Radboud University Nijmegen, 2005. ISBN 3-540-67658-9.

[6] I. G. Baltopoulos and A. D. Gordon. Secure compilation of a multi-tier web language. In *In ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2009*, pages 27–38, 2009.

[7] E. Barendsen and S. Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. In *Mathematical Structures in Computer Science*, volume 6, pages 579–612, 1996.

[8] G. Collins and D. Beardsley. The snap framework: A web toolkit for haskell. *IEEE Internet Computing*, 15(1):84–87, 2011. ISSN 1089-7801. .

[9] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *Proc. of the 5th International Symposium on Formal Methods for Components and Objects*, FMCO'06, 2006.

[10] K. Czarnecki, J. N. Foster, Z. Hu, R. Lmmel, A. Schrr, and J. F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In R. F. Paige, editor, *ICMT*, volume 5563 of *Lecture Notes in Computer Science*, pages 260–283. Springer, 2009. ISBN 978-3-642-02407-8. URL `http://dblp.uni-trier.de/db/conf/icmt/icmt2009.html#CzarneckiFHLST09`.

[11] Z. Diskin, Y. Xiong, and K. Czarnecki. From state- to delta-based bidirectional model transformations: the asymmetric case. *Journal of Object Technology*, 10:6:1–25, 2011. ISSN 1660-1769. . URL `http://www.jot.fm/contents/issue_2011_01/article6.html`.

[12] L. Domoszlai. The SAPL compiler infrastructure. URL `http://wiki.clean.cs.ru.nl/SAPL`.

[13] L. Domoszlai and T. Kozsik. Clean up the web! - rapid client-side web development with clean. In *The Beauty of Functional Code*, pages 133–150, 2013.

[14] L. Domoszlai and R. Plasmeijer. Tasklets: Client-side evaluation for iTask3. In *Domain specific languages, summer school*, DSL'13, 2014.

[15] L. Domoszlai, E. Bruël, and J. Jansen. Implementing a non-strict purely functional language in JavaScript. *Acta Universitatis Sapientiae*, 3:76–98, 2011. URL `http://www.acta.sapientia.ro/acta-info/C3-1/info31-4.pdf`.

[16] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3), May 2007. ISSN 0164-0925. . URL `http://doi.acm.org/10.1145/1232420.1232424`.

[17] H. Giese and R. Wagner. From model transformation to incremental bidirectional model synchronization. *Software and Systems Modeling (SoSyM)*, 8(1):21–43, February 2009. .

[18] R. Hinze. A new approach to generic functional programming. In T. Reps, editor, *Proceedings of the 27th International Symposium on Principles of Programming Languages, POPL '00, Boston, MA, USA*, pages 119–132. ACM Press, 2000.

[19] M. Hofmann, B. C. Pierce, and D. Wagner. Edit lenses. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Philadelphia, Pennsylvania*, Jan. 2012.

[20] M. Hofmann, B. C. Pierce, and D. Wagner. Symmetric lenses. *Journal of the ACM*, 2014. To appear; extended abstract in POPL 2011.

[21] J. Jansen, P. Koopman, and R. Plasmeijer. Efficient interpretation by transforming data types and patterns to functions. In H. Nilsson, editor, *Proceedings of the 7th Symposium on Trends in Functional Programming, TFP '06*, pages 157–172, Nottingham, UK, 19-21, Apr. 2006. ISBN 978-1-84150-188-8.

[22] J. Jansen, R. Plasmeijer, and P. Koopman. iEditors: extending iTask with interactive plug-ins. In S.-B. Scholz and O. Chitil, editors, *Revised Selected Papers of the 20th International Symposium on the Implementation and Application of Functional Languages, IFL '08*, volume 5836 of *LNCS*, pages 192–211, Hatfield, UK, 2011. Springer.

[23] G. E. Krasner and S. T. Pope. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, Aug 1988. ISSN 0896-8438. URL `http://dl.acm.org/citation.cfm?id=50757.50759`.

[24] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981. ISSN 0362-5915. . URL `http://doi.acm.org/10.1145/319566.319567`.

[25] B. Lijnse. *TOP to the Rescue – Task-Oriented Programming for Incident Response Applications*. PhD thesis, Radboud University Nijmegen, 2013. ISBN 978-90-820259-0-3.

[26] B. Lijnse, J. Jansen, R. Nanne, and R. Plasmeijer. Capturing the netherlands coast guard's sar workflow with itasks. In D. Mendonca and J. Dugdale, editors, *Proceedings of the 8th International Conference on Information Systems for Crisis Response and Management, ISCRAM '11*, Lisbon, Portugal, May 2011. ISCRAM Association.

[27] B. Lijnse, J. Jansen, and R. Plasmeijer. Incidone: A task-oriented incident coordination tool. In L. Rothkrantz, J. Ristvej, and Z. Franco, editors, *Proceedings of the 9th International Conference on Information Systems for Crisis Response and Management, ISCRAM '12*, Vancouver, Canada, Apr. 2012.

[28] T. van Noort. *Dynamic Typing in Type-Driven Programming*. PhD thesis, Radboud University Nijmegen, May 2012. ISBN 978-94-6108-279-4.

[29] R. Plasmeijer and M. van Eekelen. Clean language report (version 2.1). `http://clean.cs.ru.nl`, 2002.

[30] R. Plasmeijer, J. Jansen, P. Koopman, and P. Achten. Declarative Ajax and client side evaluation of workflows using iTasks. In *Proceedings of the 10th International Conference on Principles and Practice of Declarative Programming, PPDP '08*, pages 56–66, Valencia, Spain, 15-17, July 2008.

[31] R. Plasmeijer, P. Achten, P. Koopman, B. Lijnse, T. Van Noort, and J. Van Groningen. iTasks for a change: Type-safe run-time change in dynamically evolving workflows. In *PEPM '11 : Proceedings Workshop on Partial Evaluation and Program Manipulation, PEPM '11, Austin, TX, USA*, pages 151–160, New York, 2011. ACM.

[32] R. Plasmeijer, B. Lijnse, S. Michels, P. Achten, and P. Koopman. Task-Oriented Programming in a Pure Functional Language. In *Proceedings of the 2012 ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, PPDP '12*, pages 195–206, Leuven, Belgium, Sept. 2012. ACM. ISBN 978-1-4503-1522-7.

[33] M. Serrano and C. Queinnec. A multi-tier semantics for hop. *Higher-Order and Symbolic Computation*, 23:409–431, 2010. ISSN 1388-3690. URL `http://dx.doi.org/10.1007/s10990-010-9061-9`.

[34] M. Serrano, E. Gallesio, and F. Loitsch. Hop: a language for programming the web 2.0. In *ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA'06, 2006.

[35] M. Snoyman. *Developing Web Applications with Haskell and Yesod*. O'Reilly Media, Inc., 2012. ISBN 1449316972, 9781449316976.

[36] L. M. Surhone, M. T. Tennoe, and S. F. Henssonow. *Node.Js*. Betascript Publishing, Mauritius, 2010. ISBN 6133180196, 9786133180192.

[37] P. Thiemann. WASH/CGI: Server-side Web Scripting with Sessions and Typed, Compositional Forms. In *Proc. 4th Int'l Symposium on Practical Aspects of Declarative Languages*, PADL'02, Jan 2002.

[38] J. van Groningen, T. van Noort, P. Achten, P. Koopman, and R. Plasmeijer. Exchanging sources between Clean and Haskell: a double-edged front end for the Clean compiler. In J. Gibbons, editor, *Haskell'10 : proceedings of the third ACM Haskell symposium on Haskell*, pages 49–60. ACM, 2010.

[39] D. Wagner. *Symmetric Edit Lenses: A New Foundation for Bidirectional Languages*. PhD thesis, University of Pennsylvania, 2014.

[40] M. Wang, J. Gibbons, and N. Wu. Incremental updates for efficient bidirectional transformations. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 392–403, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0865-6. . URL `http://doi.acm.org/10.1145/2034773.2034825`.

[41] A. van Weelden. *Putting types to good use*. PhD thesis, Radboud University Nijmegen, 17, Oct. 2007. ISBN 978-90-9022041-3.