iEditors: Extending iTask with Interactive Plug-ins

Jan Martin Jansen[1], Rinus Plasmeijer[2], Pieter Koopman[2]

[1] Netherlands Defence Academy,
Faculty of Military Sciences,
Den Helder, the Netherlands,
[2] Institute for Computing and Information Sciences (ICIS),
Radboud University Nijmegen, the Netherlands
jm.jansen.04@nlda.nl, {rinus, pieter}@cs.ru.nl

Abstract. The iTask library of Clean enables the user to specify web enabled workflow systems on a very high level of abstraction. Details like client-server communication, storage and retrieval of state information, HTML generation, and web form handling are all handled fully automatically. For any task one can also define where it should be processed: on the server, or on the client. Client task are handled by a full Clean interpreter residing at the client side of the web application.
Using standard HTML elements and web browsers also has a disadvantage: they do not offer the same level of interaction as we are used from desktop applications. However, browser plug-ins can be used to fill this gap. Plug-ins not only offer the possibility to play animations, music, and movies, they also offer more interactive functionality e.g. allow users to make drawings. In this paper we explain how plug-ins can be nicely integrated in the iTask system. A very special feature of the integration is the possibility for a plug-in the use Clean functions as call-back mechanism for the handling of events, where call-backs can be handled on the server as well as on the client. Events that demand an immediate response are handled on the client, while others can be handled on the server. As a result we are now able to create interactive iTask applications using plug-ins like graphical editors. Although complicated distributed multi-user applications can be created in this way, reasoning about the program remains easy since all code is generated from one and the same source: the high-level iTask specification in Clean.

1 Introduction

The internet has become an important platform for the deployment of applications. Despite this popularity for an application programmer it is still hard to write web applications. To overcome this the iData [10] and iTask [11] toolkits are developed. They enable the development of web applications at a high level of abstraction, where the programmer can focus on the essence of the application without having to deal with web details like, HTML generation and client-server communication. An iData application automatically generates output (HTML)

and automatically handles user changes made in a HTML form. iTask adds the concept of tasks to iData. An application can now be considered as a structured collection of tasks to be performed by one or more users. In iTask the flow of control and information between tasks can be expressed. To enhance the performance of iTask applications the possibility to handle tasks at the client side of a web application was added. For this the SAPL interpreter [12] was extended to a full Clean interpreter [7].

iData and iTask make use of standard HTML elements. In many cases these standard elements do not suffice. Plug-ins can be used to overcome this. Examples of plug-ins are media players for playing music and movies and Java Applets that offer the possibility to run Java programs at the client side of web applications.

When developing a web application using a plug-in the programmer has to deal with the following issues:

- How to include de plug-in in the web application?
- How to load relevant data into the plug-in?
- How to obtain relevant data from the plug-in?
- How to do specific processing for the plug-in?

For the inclusion of plug-ins in web applications standard solutions in HTML exist. The other issues are mostly handled on an ad hoc basis, depending on the kind of application being developed.

In this paper we focus on a more systematic solution for the last three issues. The focus is on the inclusion of Java Applet plug-ins into iTask applications using generic programming techniques. The presented techniques are not restricted to Java Applets alone but can also be used for communication with other kind of plug-ins, e.g. Flash players. For incorporating Java Applets into iTasks, a generic (read: poly typical) framework is developed. The benefits for an application programmer are:

- A Java Applet plug-in can be used with a minimum of programming effort. Generic functions take care of the conversion of Clean data structures to Java objects and back. They also take care of the communication between the web application and the plug-in;
- One can define call-backs for the plug-in in the Clean application which can be handled either on the server or on the client. Server handling can be used for executing more time consuming functions and client handling can be used for less time consuming functions like mouse event handling (saving client-server communication overhead). For client side evaluation of call-backs the SAPL interpreter is used;
- Plug-in tasks behave like every other iTask; If a suitable plug-in exists or is already made, the application programmer only has to invoke a Clean iTask editor with an initial value of proper type, as usual. No programming of HTML, JavaScript or Java is necessary.

The technical contributions are:

- The implementation of a generic way to exchange data between Clean and Java. On the Clean side this is realized by standard generic print and parse functions. On the Java side this is realized by the Java reflection mechanism;
- The use of Clean and SAPL dynamics [14] for realizing fine grained control over call-back function handling. The call-back closures are serialized to strings that can be de-serialized to SAPL closures and executed;
- The seamless integration of plug-in tasks in the iData and iTask formalisms. This is realized by adapting the HTML generation and data type update functions. This all in a completely transparent way for the application programmer.

The structure of this paper is as follows. We start with a short survey on the iTask system. We then discuss the issues to be dealt with for including plug-ins. We give two examples of the use of iTask plug-ins. Then the implementation aspects of plug-ins and the exchange of data between Clean and Java are discussed. We compare our solution with other approaches that use client side evaluation and end with some concluding remarks.

2 The iTask toolkit

The iTask toolkit [11] is a web-based combinator library written in the lazy, purely functional programming language Clean. The novel and declarative contributions that this toolkit provides, which cannot be found in the existing commercial systems, are:

- workflows are constructed fully dynamically instead of statically: they can depend on the intermediate inputs and outputs that are yielded by workers and computations;
- workflows can be higher-order, i.e. yield partially evaluated tasks which can be passed around for further evaluation to other workers at other locations;
- workflow cases are specified as pure, strongly-typed functional expressions, using the predefined iTask combinators;
- the workflow application can handle multiple workers, multiple tasks, and multiple clients dynamically, yet everything is controlled by one, single Clean application running on the server;
- the specification of the workflow is executable; all implementation details like web-page generation, web-page handling, client-server communication and database storage handling is handled fully automatically by making intensive use of generic programming techniques [2, 6]: from the types being used the required code is generated fully automatically.

In the iTask system all software is generated from one single source in Clean. To understand what the application is doing, one only needs to look at the iTask specification. It is a specification on a very high level of abstraction which can be read as if we are dealing with an ordinary simple desktop application.

In [12] we showed how we can partially update web pages and reduce the overhead of client server communication in iTask applications by adding Ajax

[4] and client side evaluation of tasks. For the client side evaluation we use the SAPL interpreter [7]. For this the complete Clean application is also compiled to the formalism of the SAPL interpreter and loaded into the SAPL interpreter at start-up of the web application.

2.1 Examples of iTask Applications

To give an idea of the iTask system we give some small examples. Creating a task in iTask is simple. With the `editTask` function one can turn an element of an arbitrary data type into a task. As a result the user can edit the data type element in a web form. The result of the edit action is fed back to the iTask system as an element of the data type and can be further processed. `editTask` has two arguments: the name of the button that the user should press to end this task and the initial value of the editor. Here two examples of the use of this function are given: one for an integer argument and one for an element of type `Person` (together with the definition of `Person`).

```
simpleInt :: Task Int
simpleInt = editTask "Ok" 0

:: Person = { name :: String
            , e_mail :: String
            , dateOfBirth :: HtmlDate
            , gender :: Gender
            }
:: Gender = Female | Male

simplePerson :: Task Person
simplePerson = editTask "Ok" createDefault
```

Fig. 1 shows the resulting editors. Note that Person has `createDefault` as initial

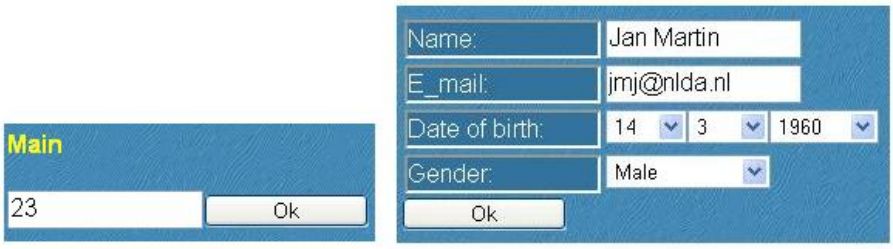

Fig. 1. editTask for Int (left) and Person (right)

value. The fields in the form will now get default values generated by the system. As a second example consider the following multi-user example:

```
addMultiUserTask :: Task Int
addMultiUserTask
=           0  @:: editTask "First number" 0
  =>> \v -> 1  @:: editTask "Second number" 0
  =>> \w -> 2  @:: editTask "Sum" (v+w),
```

User 0 has to enter a number, then user 1 has to enter a second number, then user 2 gets the sum of these number, but can still edit the result. `=>>` is the iTask equivalent of the monadic 'bind' operator. `n @:: task` assigns a `task` to user `n`. The user number must be selected with a drop down box on the page, but can also be associated with a user login.

3 Plug-ins in iTask

Standard HTML elements are often not sufficient to deal with all situations. For example interactive drawing, playing of movies, animations and music are not supported by standard HTML components. To overcome this it is possible to plug in applications into web browsers. Examples of plug-ins are media players to display movies and music and Java Applets or Flash plug-ins for interactive drawing and animation. Plug-ins have to be installed by the user of the browser. Ones this is done they are automatically loaded when needed. The use of plug-ins complicates the development of web applications. The developer has to take care that the plug-in is loaded and that the data needed by the plug-in is passed to the plug-in. In some cases also data from the plug-in has to be passed back to the web application or events occurring in the plug-in have to be handled by the web applications (e.g. mouse events).

In this paper we discuss how plug-ins can be integrated into iTask applications in a seamless way.

3.1 Use of Plug-ins from a Programmers Perspective

In a basic iTask an element of a data type is turned into an editable HTML form by the function `editTask` and the result of editing the form is automatically turned into an updated instance of the data type. This way of working is maintained for plug-ins. More concretely, the information exchanged with a plug-in must also be represented by a data type and the use of the plug-in should lead to an updated instance of this data type. Because `editTask` has no means for distinguishing a data type intended for a plug-in from any other data type and also because we need information about how to load and display the plug-in we have to wrap the content data type into the `PlugIn` data type. Below we first give two examples of how to include plug-ins into iTask applications.

3.2 A Text Editor Plug-in for iTask

As a first example we look at a text editor plug-in. Basic HTML text editor components are rather primitive. More sophisticated text editor plug-ins exist or can be made for example in Java.

The first step for making an iTask application that communicates with a text editor plug-in is the construction of a data type in Clean that represents the information that has to be exchanged between Clean application and plug-in. On the plug-in side there should be a corresponding data type where this data type can be mapped on (more details see Sec. 4). For this example the content data type is rather simple. The content of an text editor is represented by a string:

```
::TextEdit = TextEdit String
```

We wrap this data type into the `SimplePlugIn` data type:

```
::SimplePlugIn ct
= {plugininfo :: PlugInInfo, content :: ct}
```

In this type all information is captured that is needed to construct the HTML representation of the plug-in (`plugininfo`) together with its content (`content`). The following Clean task contains a text editor plug-in:

```
textedittask :: Task String
textedittask =     editTask "Ready" texteditplugin
               =>> \pi -> return_V pi.content

texteditPlugIn :: SimplePlugIn TextEdit
texteditPlugIn = (plugininfo = editorplugin,
                  content    = TextEdit "this is the initial text"}
```

We assume that `editorplugin` contains all information needed to create a HTML representation of the editor plug-in(how can it be loaded, what is its size, how it is initialized, etc.). The content field is filled with the initial content of the editor. When the user clicks the 'Ready' button the content field in the `SimplePlugIn` data structure is automatically updated with the result of the editor and this result becomes available to the remainder of the iTask application. If we compare this example with the previous examples we see that the only difference is that the data type to be edited is now wrapped into the PlugIn data type. All communication with the plug-in is handled automatically.

3.3 A Graphical Editor Plug-in for iTask

In the previous example the plug-in did all necessary processing for constructing its content. We now look at an example where external processing is needed for this construction. The example is a simple graphical editor. We assume we have a plug-in that is capable of displaying simple graphics (lines, ovals, text, etc.) and that can generate events for mouse and button actions. The processing of these events depends on what kind of graphical editor we want to make (vector graphics, diagrams etc.). Of course, it is possible to create a dedicated plug-in for each kind of editor. But by using Clean for doing event handling we can adapt the behaviour of the application by only changing the Clean source, without the need to adapt the plug-in. The key idea is that events are passed to the

web-application by a call-back function call. The call-back function can either be executed on the server by the Clean application or on the client by the SAPL version of the Clean program. The programmer can choose for each type of event where it must be handled (e.g. computation intensive events on the server and often occurring, less processing demanding, events on the client).

The Clean source code for a task containing this plug-in is:

```
::GraphState   = NewLine |  NewPolyLine | NewRect | NewOval

::GraphObject  = GraphLine Int Int Int Int | GraphOval Int Int Int Int |
                 GraphRect Int Int Int Int | GraphPolyLine [Pnt]       |
                 GraphButton String
::Pnt          = Pnt Int Int

::GraphEvent   = MouseDown Int Int | MouseDrag Int Int |
                 MouseUp   Int Int | ButtonEvent String

graphtask ::  Task (GraphState,[GraphObject])
graphtask = editTask "Ready" graphPlugIn
            =>> \gpe -> return_V (gpe.state, gpe.content)

graphPlugIn :: PlugIn [GraphObject] GraphEvent GraphState
graphPlugIn = {plugininfo    = grapheditplugin,
               content       = initpicture,
               events        = [],
               state         = NewLine,
               eventHandling = ServerEvents [MouseUp 0 0],
               callback      = doEvents}

// initial picture only contains buttons
initpicture = [GraphButton "Line",GraphButton "PolyLine",
               GraphButton "Rectangle",GraphButton "Oval"]

// Event handler for mouse and button events
doEvents :: (GraphState,[GraphObject]) [GraphEvent] ->
            (GraphState,[GraphObject])
doEvents (_,figs) [ButtonEvent "Line":evs]
= doEvents (NewLine,figs) evs
// Oval, Rectangle, PolyLine in a similar way

doEvents   (NewLine,figs) [MouseDown x y:evs]
= doEvents (NewLine,[GraphLine x y x y:figs]) evs
// Oval, Rectangle, PolyLine in a similar way

doEvents   (a,[GraphLine v w _ _: figs]) [MouseDrag x y:evs]
= doEvents (a,[GraphLine v w x y: figs]) evs
// Oval, Rectangle, PolyLine in a similar way

doEvents (a,figs) [e:evs] = doEvents (a,figs) evs // ignore other events
doEvents (a,figs) []      = (a,figs)              // return result
```

Fig. 2 shows us a screen shot of the application. In this example a drawing is represented by a list of `GraphObjects`. We distinguish several types of figures and simple butons. `GraphEvent` is the type of events that can occur in the editor. We distinguish mouse (down, up, drag) and button events. We assume that the plug-in is capable of displaying elements of `GraphObject` and that it can turn mouse events into elements of `GraphEvent`. `GraphState` is a state data type that maintains that part of the state that is not passed to the plug-in but that is needed by the call-back function. In this example it maintains the type of the figure to be drawn at a mouse down event. The type `PlugIn` contains all information needed for the creation of, and communication with the plug-in:

```
::PlugIn ct et st = {plugininfo    :: PlugInInfo,
                     content       :: ct, events :: [et], state :: st,
                     eventHandling :: EventHandling et,
                     callback      :: (st,ct) [et] -> (st,ct)}
```

- `plugininfo`: information for constructing the HTML representation of the plug-in;
- `content`: content of the editor;
- `events`: events that have to be processed by the call-back function;
- `eventHandling`: indication where events have to be handled. Here only the type matters. In our example `mouseUp` events are handled on the server, all other events are handled on the client;
- `state`: value of the state to be maintained between call-back calls;
- `callback`: call-back function that handles the generated events;

The call-back function takes the current state, the current content and the generated events as input and returns a new state and a new content. The state is maintained for the next call of the call-back. The call-back function is automatically called from the plug-in whenever an event occurs (see Sec. 4). The call-back can either be handled by the client or the server.

For indicating where events have to be handled we use this data type:

```
::EventHandling et = AllServer | AllClient |
                     ClientEvents [et] | ServerEvents [et]
```

`EventHandling` has the actual event type as parameter. The programmer can choose to have all events handled on the client or on the server, or can indicate which events should be handled on the server or on the client.

The user can stop editing the figure by clicking the “Ready” button. The current content and state are now passed to the remainder of the iTask application.

To show that the plug-in task just behaves like a normal iTask we give a small variation of the previous program:

```
graphtask ::  Task (PlugIn [GraphObject] GraphEvent GraphState)
graphtask =               0 @:: editTask "0 Ready" grapheditor
              =>> \ge -> 1 @:: editTask "1 Ready" ge
              =>> \ge -> 0 @:: editTask "Ok" ge
```

In this case two users are involved. User `0` starts with making an initial drawing. The result is passed to user `1`. User `1` can further edit the drawing. If this task is ready the result is passed back to user `0` who can continue editing.

Again we see that the programmer only has to specify the (content, state and event) types and the call-back function needed for handling events. The plug-in iTask again behaves like an ordinary iTask. All communication with the plug-in is handled in a, for the user, transparent way. Also in this case the plug-in should have matching types for the content and events types. The mapping of the elements of these types onto each other is done automatically (see Sec. 4 and 5).

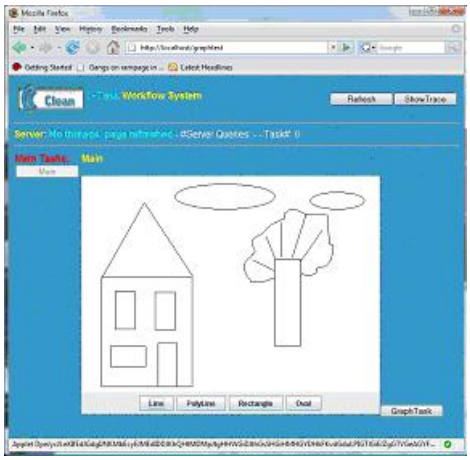

Fig. 2. Screen shot of the Drawing Application

4 Implementation of Plug-ins

From the examples it is clear that the use of plug-ins is straightforward for the application programmer. In the implementation of plug-ins we face a number of challenges:

- How to fit a plug-in into the iData/iTask architecture?
- How to exchange data between server, plug-in and client?
- How to invoke call-back functions from the plug-in for the server and client?

We concentrate on the implementation of `PlugIn` with call-backs. The implementation of `SimplePlugIn` is a simplified version of this.

4.1 Fitting a Plug-in into the iTask/iData architecture

Each plug-in has a specific representation in HTML that describes the plug-in to be loaded and some initialisation parameters like size, identification, etc. In this

HTML representation it is also possible to add extra parameter tags. We use these extra tags for storing other information we need. Examples are the content, state, information about the call-back function, information about where to handle events (client or server). We should generate this HTML representation from the `PlugIn` data type. For this generation we need a closer look at some details of iData and iTask. The iData library is based on the following two principles:

- Each data type can be turned into an (editable) web form in a generic way by using the `gForm` function. The web form is generated on the server and transported to the browser as (part of) a web page.
- Each user edit action in such a web form is automatically transformed into an updated instance of its underlying data type by the generic `gUpd` function. The user edit action is uploaded to the web server and further handled by generic functions that update the data type.

The programmer can change the default generation of web forms by giving a specific instance of the `gForm` function for a data type. Also the generic updating for data types can be adapted by the programmer by specifying a specific implementation of `gUpd` for a data type. This is exactly what is needed for creating a plug-in from the `PlugIn` data type. We have to adapt `gForm` in such a way that the HTML needed for the plug-in is generated. `gUpd` has to be adapted in such a way that the results of a plug-in are included into the original data representation and that, if necessary, the call-back function is executed. The Clean data structures `SimplePlugIn` and `PlugIn` should contain all information necessary to realize this.

All communication between server, client and plug-in is done via a number of JavaScript functions, like this is done for Ajax and Client side handling of iTasks (see [12]). These functions are generic in the sense that they do not depend on the specific plug-in. The JavaScript functions are responsible for passing data from plug-in to server and client and vice-versa, but also for making the call-back and deciding where the call-back must be handled. In JavaScript the following functions callable from the plug-in are available:

- `syncServer(String content)`, to be called for a direct synchronization of the plug-in with the server. The argument is a (serialized) string version of the content type;
- `doCallback(String content, String events)`, to be called whenever an event occurs. The first argument is the serialized version of the current content. The second arguments is a serialized version of the list of the events that must be processed. This JavaScript function now takes case of making the actual call-back (see Sec. 4.3).

The plug-in itself should implement the function `setContent(String content)`, callable from JavaScript, to set its content, again with as argument a (serialized) string representation of the content. It is common for plug-ins to have a JavaScript interface.

4.2 Data exchange between client, plug-in and server

Although we have Clean programs running at both the server and client side, the internal representations of data types are completely different. Therefore we use the generic print and parse functions for the exchange of data between the Clean programs at the server and client side.

For data exchange with the plug-in we also use the generic print and parse functions on the Clean side (on server and client). This means that also the plug-in must have a generic way to parse and unparse the strings representing the event and content data types. In Sec. 5 we discuss how this is realized for Java.

4.3 Handling call-backs

Call-backs can be made to either client or server. The plug-in makes the call-back by calling the (generic) Javascipt function `doCallBack` with the serialized content and events as arguments. The JavaScript function determines whether the call must be handled on the server or the client. For the server case the `PlugIn` data type is updated in a similar way as for an ordinary update for iData [10]. The adapted version of `gUpd` applies the call-back function before updating the `PlugIn` data structure.

We can use the same strategy for the client side, but we can (and must) do it in a much more efficient way. Because we have an interpreter at the client side we can execute an arbitrary Clean expression. We use this to directly execute the call-back function call and to make a direct update of the content of the plug-in. In this way we short-circuit the use of `gUpd` and the whole iTask machinery needed for finding out which task is updated [11]. This optimisation is absolutely necessary for often occurring events with small (less processing intensive) event handling functions like mouse drag events. For making the direct call-back on the client we use Clean dynamics [14]. With dynamics it is possible to serialize a Clean expression (closure) to a string, store the string somewhere, retrieve the string at a later moment, turn it into a Clean expression again and execute it. We extended the dynamics features of Clean in such a way that it is also possible to serialize an expression in a Clean executable and de-serialize it in the SAPL interpreter and execute the expression there. This is a powerful feature, because it makes it possible to migrate execution of a Clean program from server to client. We already used this feature for the client side evaluation of iTasks[12]. In this case the serialized call-back function is stored in one of the parameters of the plug-in HTML representation. Not only the call-back function itself is serialized, but also the parse and unparse functions for its arguments (content, state and events). This is necessary because the arguments are passed to the call-back as serialized strings from the plug-in via the `doCallBack` function and the result must be passed back in serialized form too.

The actual call to SAPL is made with the generic `doSaplCallBackWrap` function that gets the serialized call-back, parse and unparse functions and the serialized state, content and events as arguments. It de-serializes all functions and

uses them to: parse the call-back arguments, apply the call-back function and unparse the state and content again. The content is handed back to the plug-in directly (via `setContent`) and the state is maintained as an HTML parameter in the plug-in representation in the web-page.

4.4 Evaluation of Effiency of handling call-backs

In the graphical editor application we used the call-back function to handle mouse down and drag events in the SAPL interpreter. Mouse drag events often occur in quick sequences (in the order of 5-10 events per second). The whole call-back machinery was capable of keeping track of these events on a 1.6 GHz Core Duo 2 machine. Attempts to handle the drag events by the server lead to a crash of the browser due to a client-server communication overload.

5 Implementation for Java Applets

A special kind of plug-ins are Java Applets [1]. All modern web browsers offer the possibility of Applet plug-ins. In this way it is possible to incorporate complex Java applications into web pages. We already used the Java Applet mechanism for loading the SAPL interpreter at the client side of iTask for handling client tasks. Although Java Applets can offer rich functionality they are less popular because it is difficult to integrate them with the remainder of a web application. Using the iTask plug-in techniques we have a generic strategy which simplifies the communication with Java Applets.

To include a Java Applet in an iTask application we have to deal with the following issues:

- We have to find a way to map Clean types on corresponding Java data types;
- We have to take care that we offer the interface needed for communication with JavaScript.

Because it is possible to call JavaScript functions from Java and Java functions from JavaScript the second issue can easily be realized.

5.1 Mapping Clean and Java Data Types onto each other

In order to exchange information between a Clean and Java application there must be a way to map a Clean data type onto a Java data type and back. To save the programmer from writing boilerplate data transformation code we included generic code in Clean and Java to handle this data transformation. Not all Clean and Java data types can be mapped onto each other. For example, Java only support first order data types (it has no function types). For Java we have the following restrictions.

- A class may be a subclass of another class or implement an interface, but all superclasses and the class itself must obey the rules mentioned here;
- For a class member field the following types are allowed:
  - primitive types (`int`,`long`,`float`,`double`,`boolean`,`char`);

  - the `String` type;
  - all subtypes of `List` (they are all mapped on a Clean list;
  - other Java types that obey these rules;
- Other types, like `Map`, other container types and arrays are not (yet) allowed at this moment. For these classes an ad-hoc mapping must be made (like this is done for `List`).

From the Clean point of view the automatic conversion of Clean data types to Java types is restricted to first order data types that can be described by standard Algebraic Data Types. Records are not allowed yet, but they can be simply added. If a Clean type is mapped onto a Java type hierarchy the fields of the Clean type should match the union of all fields in the class hierarchy in the order of the hierarchy (fields of superclass before fields of subclass).

Consider the following algebraic data type definition in Clean:

```
::typename t1 .. tk = C1 t11 .. t1n_1 | .. | Cm tm1 .. tmn_m
```

`t1..tk` are type parameters, `C1..Cm` contructor names and `tik` types or type parameters. We map this type definition to a Java interface and `m` Java classes:

```
interface typename {}
class<t1,..,tk> C1 implements typename {t11 a1; .. t1n_1 an_1;}
...
class<t1,..,tk> Cm implements typename {tm1 am; .. tmn_m an_m;}
```

Each constructor is represented by a separate Java class with as name the constructor name and with as fields the arguments of the contructor with their type.

For the actual exchange of data we use the standard generic print and parse functions at the Clean side (`gPrint` and `gParse`) and refelction on the Java side.

5.2 Java Applet Plug-In Interface

At the Java side we implemented the class `CleanJavaCom`, which contains member functions that can be used for parsing and unparsing objects and functions for handling the communication with JavaScript. The `CleanJavaCom` class is generic and can be used for every Java Applet to be included in an iTask application.

```
public class CleanJavaCom<CT,ET> {
  private String writeClassToString(Object object) throws Exception {...}
  private Object readClassFromString(String inp) throws Exception {...}
  public CT getContent() {...}
  public void setContent(String ser_content) {...}
  public void handleEvents(List{ET> events) {...}
  public void syncServer() {...}
  ...
}
```

The class is parametrised by the Java versions of the content (`CT`) and event (`ET`) types.

- `writeClassToString` generates a string representation of an object that exactly fits the Clean representation;
- `readClassFromString` parses a string representation generated by `gPrint` to the corresponding Java object;
- `getContent` to be used by the remainder of the Applet to obtain the content;
- `setContent` to be called from JavaScript to set the content. The content string is de-serialized by a call to `readClassFromString`. The result can be obtained by a call of `getContent`;
- `handleEvents` to be called by the Applet after one (or more) event(s) have occurred. The function takes care of serializing the content and events and calling the `doCallBack` function in JavaScript. The result is passed back by a call from JavaScript for `setContent`.
- `syncServer` can be called by the Applet to synchronize its content with the Clean application. This is necessary if the plug-in updates its content in another way then by calling `doCallBack`.

For the implementation of `writeClassToString` and `readClassFromString` the Java reflection mechanism is used.

6 Related Work

The iTask toolkit allows high-level specification of multi-user workflows. Forms are generated generically from type information, which considerably decreases the amount of HTML programming. Complex dynamic workflows can be created that can be evaluated at both the client and the server without any restriction. In this paper we extended the iTask toolkit with a generic framework for the inclusion of plug-ins, with the possibility to make call-backs from the plug-in to the Clean application. Call-backs can be executed either on client or server. We are not aware of any other functional system that has these features. However, there are functional approaches for handling web pages. Links [3] and its extension formlets is a functional language based web programming language. Links compiles to JavaScript for rendering HTML pages, and SQL to communicate with a back-end database. A Links program stores its session state at the client side. In a Links program, the keywords client and server force a top-level function to be executed at the client or server respectively. In Links processes can be spawned, and these processes can communicate via message passing. Client-server communication is implemented using Ajax technology, like we do. In iTask processes are not created explicitly as in Links programs. In the iData and iTask toolkits, forms are generated generically for every data type, whereas in Links and formlets these need to be coded by the programmer.

Another functional language based web programming language is Hop [13, 9]. Just as Links, Hop is compiled to JavaScript. It implements a strict separation between programming the user interface and the logic of an application. The main computation runs on the server, and the GUI runs on the client(s). These components can invoke each other (from GUI client to server via function calls, from server to client via signalling events). In particular the latter feature increases the expressive power of web applications, which are usually driven by

the browser client side. Additional server logic can be invoked as a Hop service, which makes the design very modular. This has been implemented with Ajax. In the iData and iTask toolkits, we do not require a stratified language approach to divide our attention to GUI programming versus application logic, because the GUI is mainly generated generically. The relation between plug-in and Clean application is comparable to that of GUI and server in Hop, because of the explicit calls the plug-in can make to the Clean application.

The Flapjax language [8] is an implementation of functional reactive programming in JavaScript. Many of its features are comparable with those of Hop, and indeed both are designed to create intricate web applications. The main difference with our approach is that the iTask system is geared for distributed, multi-user, workflow systems in which the coordination and interaction of work is defined in a highly declarative style. The enabling technology of client-side evaluation in the iTask toolkit is SAPL. In principle the complete Clean application can be compiled to SAPL. This enables our approach to use the full expressive power of Clean to perform intricate computations at the client side for tasks and call-backs. A much more restricted approach has been implemented in Curry [5]: only a very restricted subset of Curry is translated to JavaScript to handle client side verification code fragments only.

7 Conclusions

Plug-ins are often an essential part of more interactive web applications. In this paper we discussed a generic way for including plug-ins in iTask. All communication between iTask application and plug-in is on the level of exchanging and updating data types, which is entirely consistent with the normal way iTask works. Plug-in tasks behave like ordinary tasks. No adaptations of iTask where necessary to incorporate them, only a orthogonal extension of the `gForm` and `gUpd` functions. Communication with a plug-in is realized by putting the data types used for the exchange of information into a PlugIn wrapper type and, if necessary, specifying a call-back function in Clean.

An important feature is that the plug-in can make call-backs to the Clean application that can be handled on either server or client. This gives the programmer fine-grained control over the behaviour of the plug-in without the need to adapt the plug-in itself. In this way the plug-in only has to implement functionality that cannot be implemented in Clean.

A small number of JavaScript functions are provided that act as an intermediair between Clean application and plug-in. For Java Applets a straightforward to use generic class is provided that handles all interaction of the plug-in with Clean including the (de)serialization of data types and the handling of call-backs. Plug-ins of other type should implement the JavaScript interface and the (de)serialization of the data types used for the exchange of information.

We have maintained the declarative approach of the iTask toolkit. Everything is still generated from an annotated, single source specification with a low burden on the programmer because the system itself switches automatically between client and server side evaluation of tasks and call-backs when this is necessary.

The iTask system integrates all mentioned technologies in a truly transparent and declarative way.

References

1. *The Java Home Page*. java.sun.com.
2. A. Alimarine and M. J. Plasmeijer. A generic programming extension for Clean. In *Implementation of Functional Languages, IFL 2001*, volume 2312 of *Lecture Notes in Computer Science*, pages 168–185. Springer, 2001.
3. E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *Proceedings of the 5th International Symposium on Formal Methods for Components and Objects, FMCO '06*, volume 4709 of *Lecture Notes in Computer Science*, pages 266–296. Springer, 2006.
4. J. Garrett. Ajax: A new approach to web applications. www.adaptivepath.com/ideas/essays/archives/000385.php.
5. M. Hanus. Putting declarative programming into the web: Translating Curry to JavaScript. In *Proc. of the 9th International ACM SIGPLAN Conference on Principle and Practice of Declarative Programming (PPDP'07)*, pages 155–166. ACM Press, 2007.
6. R. Hinze. A new approach to generic functional programming. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2000.
7. J. M. Jansen, P. Koopman, and R. Plasmeijer. Efficient interpretation by transforming data types and patterns to functions. In H. Nilsson, editor, *Proceedings Seventh Symposium on Trends in Functional Programming, TFP 2006, Nottingham, UK, 19-21 April 2006, The University of Nottingham*, volume 7 of *Trends in Functional Programming*. Intellect Publisher, 2006.
8. S. Krishnamurthi. The Flapjax site, 2007. www.flapjax-lang.org.
9. F. Loitsch and M. Serrano. Hop client-side compilation. In *Trends in Functional Programming, TFP 2007, New York*, pages 141–158. Interact, 2008.
10. R. Plasmeijer and P. Achten. The implementation of iData. In A. Butterfield, C. Grelck, and F. Huch, editors, *Implementation and Application of Functional Languages, 17th International Workshop, IFL 2005, Dublin, Ireland, September 19-21, 2005, Revised Selected Papers*, volume 4015 of *Lecture Notes in Computer Science*, pages 106–123. Springer, 2006.
11. R. Plasmeijer, P. Achten, and P. Koopman. iTasks: Executable specifications of interactive work flow systems for the web. In N. Ramsey, editor, *Proceedings of the 2007 ACM SIGPLAN International Conference on Functional Programming, Freiburg, Germany, October 1-3, 2007*, volume ICFP'07 of *International Conference on Functional Programming*, pages 141–152. ACM, 2007.
12. R. Plasmeijer, J. M. Jansen, P. Koopman, and P. Achten. Declarative Ajax and client side evaluation of workflows using iTasks. In *Principles and Practice of Declarative Programming, Valencia, Spain, July 2008*, volume PPDP 08, 2008.
13. M. Serrano, E. Gallesio, and F. Loitsch. Hop: a language for programming the web 2.0. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2006), Portland, Oregon, USA, October 22-26 2006*, pages 975–985, 2006.
14. A. van Weelden. *Putting Types To Good Use*. PhD thesis, Radboud University Nijmegen, the Netherlands, 2007.