Functional Web Applications

Torsten Gipp and Jürgen Ebert

University of Koblenz-Landau
{tgi,ebert}@uni-koblenz.de

Abstract. Web applications are complex software artefacts whose creation and maintenance is not feasible without abstractions, or models. Many special-purpose languages are used today as notations for these models. We show that *functional programming languages* can be used as modelling languages, offering substantial benefits. The precision and expressive power of functional languages helps in developing concise and maintainable specifications. We demonstrate our approach with the help of a simple example web site, using Haskell as the implementation language.

Keywords: Web Application Modelling, Specification, Functional Languages, Haskell.

1 Introduction

A *web application* (or web *site*) is an application that is delivered through the web. Creating such a web site is a complex task. Aside from the most trivial web sites that can be simply written down in one go, 'real' web sites require the application of a sound and consistent engineering approach. The end product must be expandable, reliable, error-free, and, of course, adhere to the given 'specification' perfectly. However, the trade-off between the required development time and the aspired quality is much too often solved by sacrificing the latter.

The Web Engineering discipline suggests using *models* to build abstract descriptions of a web site and to *derive* the end product from these models (e.g., [16]). This becomes particularly useful if the derivation can be done automatically (e.g., [23]), at least to a significant degree, and if the modelling does not impose too much overhead. Our idea is to apply modelling as well, but to do it using a *functional language*.

Example. As an example application we consider a travel booking system that offers its services over the web. The system is called the Travel Agency System (TAS). A customer can search for trips by supplying the origin and destination city together with the desired timeframe for a trip and the system will respond with a list of possible alternatives. Picking one of these provides further details about the selected trip, including the calculated prospective costs, and the customer can choose to book this trip, which will trigger the steps necessary for the financial transaction.

L. Baresi, P. Fraternali, and G.-J. Houben (Eds.): ICWE 2007, LNCS 4607, pp. 194–209, 2007.
© Springer-Verlag Berlin Heidelberg 2007

Every *page* of such a web site must be modelled, stating its inherent compositional *structure* and, most importantly, define which type of *content*, or data, it shall display. This must be done in a precise and abstract way. Here especially, functional languages are an almost natural choice because of their conciseness and power. Section 4.4 will introduce an abstract data type for page descriptions that is used as the backbone for the page models. We chose Haskell [25] as our implementation language, because it is in widespread use and well supported. In principle, however, the implementation of our approach does not depend on any particular language.

For this paper, we assume the reader to have some knowledge of functional programming languages. Due to space limitations, we cannot provide an extensive introduction into functional programming or Haskell. The abstract concepts, however, should be understandable nonetheless. Using Haskell, page specifications look like in this example:

```
tasTripDescription :: PageInfo
tasTripDescription = PageInfo "TASTripDescription" $
  λ params →
    do tasMainTemplate
      (Element "slots" []
        [ Element "heading" [] [Text "Trip␣Description"]
        , Element "navigation" [] [ tasNavigation params ]
        , Element "body" []
          [ (tasTripDescriptionForm [])                          (1)
          , (tasPreferencesList params)
          ]
        , Element "footer" [] [ Text "Footer" ]
        ]
      )
```

This constructs a page that fills the four slots heading, navigation, body and footer of a page template. A page template defines the common structural layout of all pages. The composition of a page is defined by assembling smaller parts, like the function tasTripDescriptionForm (see mark (1)) that describes a web form. Section 4.4 will give the necessary background information in full detail.

Figure 1 shows the hyperlinks between the single pages that constitute the example web site. The dashed arrows suggest the steps of the buying 'workflow' just described. A diagram like this is used as a depiction of the *navigation structure*. This structure models another one of six distinguished *aspects* of the web site. Section 4, the main part of this paper, will give an overview over this and the other important aspects, and section 4.3, in particular, will detail on diagrams that represent the navigation structure. But first, the following section 2 briefly lists the problems that our approach actually solves, and how this is done. We will reference related work as we go, but a coherent look into the state of the art is deferred until section 5. The TAS, our example web site, will be used throughout the remainder of this text to demonstrate our ideas. A preliminary result using the same example was described in a workshop paper [11].

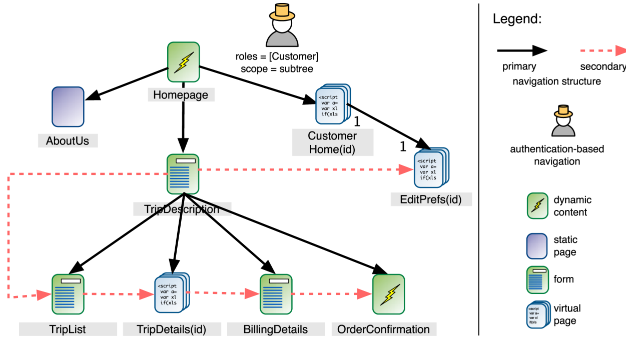

Fig. 1. Navigation structure

2 Benefits

Our approach delivers the following benefits:

– *Declarative description in combination with models allows for executable specifications.*
We employ a functional programming language to write down specifications of web sites, suggesting the combination of model-driven approaches with purely functional ones. Many web sites today are not specified in any way, or if they are, then the specifications are often not precise enough. A declarative description of a web site, however, written in a functional programming language, can serve as both: as a precise description and as an *executable* specification of the end product. The functional language is the perfect vehicle to produce *consistent* specifications.
– *Formalisation of the requirements as early as possible.*
We suggest that the formalisation of the requirements be tried as extensively as possible, which results in the specification being sufficiently close to the requirements. This ensures the consistency of the web site and its coherence with the specification. With functional languages, requirements that are given in a declarative way, like "In a trip description, the date of return must not be earlier than the date of departure.", can be written down directly, in this case by giving a boolean function that checks the constraint on the two date values (see the code in section 4.7).
– *Abstractions can be introduced wherever needed, to master complexity.*
The emerging complexity of a web site and its model is almost necessarily handled by using abstractions. Functional languages provide very powerful ways to introduce new abstractions, which makes them an almost ideal vehicle for these kinds of specifications. Higher-order functions can be used to implement new control structures, for example, and combinator libraries (e.g., as in [13]) or domain-specific embedded languages (DSEL) like Peter Thiemann's WASH/CGI [29] help tremendously by hiding implementation details.

– *The separation of concerns leads to the identification of six core aspects.* We provide a backbone structure for the modelling of web sites by the separation of content, navigation, pages, queries and updates, presentation, and dynamics.
– *Support for testing and simulating.* Functional programs also lend themselves very well to testing. Thus, it is possible to construct test cases directly from the requirements documents. The simulation of a web site visitor is possible as well, as a visit of the web site is nothing more than a sequence of function calls with concrete parameters.

3 Functional Web Sites

The core idea of our approach is the consistent use of a functional programming language to specify a web site. Following a strict *separation of concerns*, we partition the task of describing a web site into a set of separate aspects that can be tackled individually. The identified aspects are

– the content,
– the navigation structure (site map),
– the pages (navigation objects),
– queries and updates,
– the presentation, and
– the dynamics.

Section 4 will provide details for each of them. Each aspect is captured in a model, and the combination of these models provides the overall picture of the entire web site. Since the concerns are tackled separately, each can be treated according to the particular requirements for the respective aspect. The content, for example, is modelled using a conceptual model, which gives an abstract view on the content in form of *concepts* (or classes). The content itself is stored in a graph data structure, which in turn adheres to a (type) schema that is defined by the conceptual model. The navigation structure is captured using a visual language. The pages are modelled using functions that yield the actual page upon evaluation, using concrete parameter values. The dynamics, the queries and updates as well as the presentation are also specified in a functional way.

Thus, the functional language is used extensively. Any specification can benefit from this fact, because the full power of the functional programming language can be used at any level. The introduction of new abstractions is possible at any time, which is very important to master the complexity. As an example, regard the need for some sort of ‘templating’ when describing the single pages of a web site, as many pages will share common parts like design elements or a visualisation of the site map. We simply use a function that fills given fragments of a web page into a page template that contains everything that does *not* change from page to page. Additionally, the template itself is not static. It can contain conditional expressions that allow variations of the template to be handled smoothly. All this

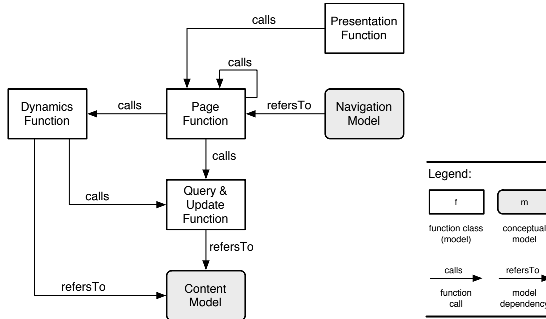

Fig. 2. Model overview

can be embedded into the functional language without syntactic clutter, due to the extensibility of the language via higher-order functions or the definition of new operators. Thus, new abstractions can be defined *in the specification itself*, without the need for any other language.

4 Functional Specifications

According to the list of aspects given in the previous section, we will now provide the respective details. Starting with a bird's eye view (section 4.1), the subsequent sections introduce one aspect each.

4.1 Overview

Figure 2 shows the dependencies between the aspects. The aspects are depicted by rectangles. Rounded rectangles represent a conceptual model, normal rectangles represent a set of functions. The *content model* (at the bottom of the diagram) is a conceptual abstraction of the application domain. Relevant terms, or concepts, like, in our example, *customer* or *trip*, are identified and related to one another. We can therefore abstract from the actual content, and we can operate on the 'class' level rather than on the 'instance' level, to use object-oriented terminology. This level of abstraction is exploited in the definition of *query and update functions*. These functions specify the content access by using the terminology provided by the content model. The same abstraction step is done for the *dynamics functions*. They capture the application behaviour or 'business logic'. Both kinds of functions are used in the definition of the single web pages through the *page functions*. They are at the heart of our approach. There is one function for each (kind of) page. The overall navigation structure of the web site is modelled separately in the *navigation model*, and the conversion of the abstract page descriptions into a concrete output language is specified by the *presentation functions*.

We will now give the details, starting with the *content* aspect.

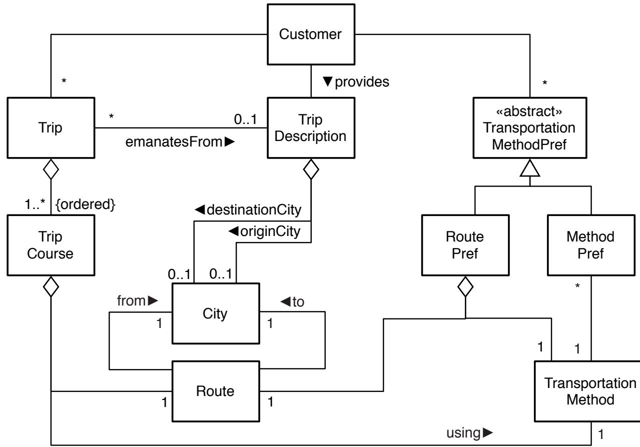

Fig. 3. Content model for the TAS example (attributes omitted)

4.2 Content

As far as the content is concerned, we consider the model (schema) level and the instance level.

Content Model. On the content model level, we capture the application domain by defining a conceptualisation, that is, the model identifies concepts and their relationships. The creation of the content model is an important step towards the understanding and mental structuring of the application domain. It captures what the application is all *about*. For the travel agency system, this includes trips, cities, customers and their preferences, flight schedules, etc.

It is almost consensus to use UML class diagrams as a notation for those conceptual models. This way, concepts (represented by classes), their attributes, and their relationships to other concepts can be visualised conveniently. The class diagram serves as a means of communication between the project participants. The model usually evolves over time, until, after a number of iterations, a sufficiently stable version has emerged.

The content model defines a *terminology* for talking about the application domain. The terms are referenced in other models, thus serving as a building block for the overall set of models. We will define query functions, for example, that access the content by using the abstractions defined in the content model.

Content Instance. The actual content itself is an *instance* of the content model. This means that the type of an individual content object corresponds to a class, and that its relationships adhere to the structure given by the content model. The content itself must be stored, retrieved, and changed. To this end, we employ *graphs*.

Graphs are a powerful mathematical structure that make an almost ideal data repository. We rely on typed and attributed graphs to store the web site's content. Nodes and edges of the graph have a type, and each type directly corresponds to a class or a relationship, respectively, in the content model. Instance attributes are attached to nodes. Node and edge types constitute a graph's *schema*, and the schema defines a *class* of graphs. The graph repository is implemented with the Functional Graph Library (FGL, [5]).

The job of retrieving and changing graph data is performed by the query and update functions, which will be dealt with in section 4.5.

4.3 Navigation Structure

Figure 1 shows the navigation structure, or *site map*, of our example web site. It is a visual representation of the hyperlink structure that connects all pages constituting the 'web'. In our opinion, a visual language is suited very well for showing, to a human, what the structure of the web site actually looks like. Especially during the design phase, moving icons around is easier and more 'intuitive' than writing formulas. This language has been successfully applied in some of our web engineering projects (e.g., [7]). The remainder of this section briefly introduces the language; cf. [10] for further details.

The *syntax* of the navigation structure diagrams is quite simple. The *primary* navigation structure, given by the solid arrows, defines a tree of page nodes. This tree assigns a unique path to every page. It is also easy to communicate to a web site visitor, who can create a mental image of the site map fairly quickly, which in turn is a very important ergonomic feature.

In the page functions, which will be explained shortly, the primary navigation structure is accessible through a simple, regular term structure.

The *secondary* navigation structure is visualised by dashed arrows. They represent arbitrary links between pages, without heeding the tree structure.

Types of Pages. There are four different types of pages, visually differentiated by four different page icons (cf. the legend in fig. 1). A lightning bolt marks a page as being *dynamic*, i.e., as a page whose relevant content is calculated (and thus potentially varies) at the time of access. In contrast, the content of *static* pages does not change at run-time. The classification of a page as being either static or dynamic is not necessarily unambiguous, because the definition of 'relevant content' is subject to interpretation. The distinction merely serves communicative purposes during modelling. There are no consequences on the implementation level. In our example, the home page is dynamic, and the 'about us' page is declared static.

Pages providing a form to let a web site visitor enter some data can be distinguished by a corresponding *form* icon. In our example, the user can enter and submit a trip description on the TripDescription page.

A small piece of script code on a stacked page icon signifies a *virtual* page that is computed by a script. In contrast to the 'lightning bolt' pages with dynamic content, the script-generated pages are *entirely* calculated by a set of

parameters, where one (the first) parameter defines the name of the page. This is inspired by the skolem functions from Strudel [6]. As an example, consider the CustomerHome(id) page, which represents a set of personalised pages, one page for each customer. The virtual pages do not exist under a pre-defined identifier, like the pages with dynamic content do. They are rather created on-the-fly, every time the page is called with a concrete identifier. We will use the term *instance* to talk about concrete virtual pages. There is one instance for each possible identifier.

These four basic web page flavours can also be mixed on one page. A virtual, a static, or a dynamic page can contain a form (or more than one). Since non-dynamic virtual pages do not make much sense – because this would mean that every instance looked the same and did not make use of the identifying parameter – the lightning bolt adornment will not be applied to virtual page icons, and virtual pages will count as *always* being dynamic.

Technically, pages of all four page types are defined by a page function of the *same* signature. Therefore, the page type chosen in the site map diagram is of no relevance implementation-wise, it is only important conceptually. Section 4.4 will deal with the page functions.

Additional Information about Links. The navigation structure diagram may also contain information on *authorisation-dependent navigation*. In the example, the primary link to CustomerHome(id) is annotated with a role-icon. It states that a web site visitor must possess the role Customer in order to access the page. The scope=subtree declaration expands this constraint to the whole subtree rooted at this page. The alternative value thisPage for scope would prohibit this expansion. The actual mechanism for checking the authorisation of a given user, a given action and a given object is intentionally left open in our approach. We can encompass any matrix-based scheme that maps permissions to roles.

The diagram can also capture the *multiplicity* of links to or from virtual pages. This is useful because virtual pages are like classes in that they represent a set of instances. Thus, we adopted a subset of the UML's multiplicity symbols to define how many instances may be connected. In figure 1, each CustomerHome(id) instance is connected to exactly one instance of EditPref(id).

The actual checking of the constraints and of the authorisation is contained in the associated page functions in form of expressions. They also contain the definition of the links for the secondary navigation structure. Thus, we can employ the full power of the underlying functional language to provide conditional links, whose behaviour or mere existence depends on the system state and other context information.

4.4 Pages

Basic Definitions. Each page that is part of the web site is specified by a *page function*. A first example for a page function was given in the introduction.

Page functions return a value of the abstract type APD, short for *abstract page description*. This type allows for defining a page on an abstract level in terms of hierarchically nested, labelled, and attributed elements (comparable to XML). Here is the definition of this data type (in Haskell):

```
data APD
    = Text String
    | Element Name Attrs APDs
    | Link Name Attrs APDs PageInfo Params
    | Form Name Attrs APDs PageInfo Params
    | Field Name Attrs FieldType APDs String
    | Empty
```

There are six constructors for the APD type. An APD term can be a simple text node (Text); an element (Element) with a name, a list of attributes, and a list of child terms; a link or a form (Link, Form) with a name, a list of attributes, a list of child terms, information about the destination page, and a list of parameters that should be passed to this page; a field in a form (Field) with a name, a list of attributes, the type, a list of child terms, and a default field content; or it can simply be empty (Empty).

Some auxiliary declarations are used: The name of an element (Name) is a string. Attributes and parameters (Attrs, Params) are modelled as lists of key/value-pairs. Elements as well as forms and fields can contain child nodes, so they use APDs as a container for a list of arbitrary APD terms. A PageInfo term contains information about a single page, comprising an identifier, and a *page function*, which is the function that returns the APD for that page. Since functions are first-class objects in a functional language, the term is able to contain the proper function itself.

A page function basically maps a set of parameters to an APD. The current system state, including the session information, is passed along as an implicit parameter with the help of the Haskell StateT monad transformer (cf. [10]).

Links and form destinations are defined in terms of PageInfos. This implies that links are represented by terms in an APD structure, attached with a reference to the actual page function they link to. This guarantees link consistency.

Example. The following code for the function tasTripDescriptionForm represents a form for entering a trip description. The form itself is not a complete page, but rather just a building block. It is used inside another page function, tasTripDescription, that was already shown in the introduction. Figure 4 shows a rendered version of the form after a transformation to HTML (cf. section 4.6 for information about how this transformation is specified).

```
tasTripDescriptionForm :: StatefulPageFunc
tasTripDescriptionForm _ = do
    (graph, session) ← get
    return $
        Form "TripDescriptionForm" []                                   (1)
            [ Text "From:"
```

From: New York
To: New York
Departure:
Return:
Maximum Price:
Maximum Duration:
Sorting Order: SortByPrice

Submit

Fig. 4. Example page fragment

```
          , Field "originCity" [] (OptionListField $ getOriginCities graph) [] ""      (2)
          , Text "To:"
          , Field "destCity" [] (OptionListField $ getDestCities graph) [] ""
          , Text "Departure:"
          , Field "dateOfDeparture" [] SimpleField [] ""
          -- some similar fields omitted
          , Text "Sorting␣Order:"
          , Field "sortingPreference" []
              (OptionListField $ map show possibleTripSortingPreferences) [] ""
          , Field "submit" [] SubmitField [] ""
          ]
          tasTripList []
```

The form is defined using the Form constructor (see mark (1)). The form's content is a list of Text and Field elements, which stand for a simple text label or a corresponding input field, respectively. The example code unveils two demonstrations of *re-use*:

1. The possibility for a page function to also define a *fragment* of a page, rather than a complete one, seems trivial and minor. In fact, however, this implies that pages can be assembled from smaller building blocks, which is a very important feature for encouraging re-use of page fragments.
2. The definition of helper functions like getOriginCities (see mark (2)) helps cleaning up the specification. Here, this function encapsulates the access of instance data that is stored in a graph.

Next to the list of form fields we can see the link to tasTripList. This is the PageInfo function that gets called when the form is actually submitted (the action handler).

The same principle applies to ordinary links between pages defined by the secondary navigation structure, as following a link is only a special case of submitting a form. One can regard a link as a form without any fields. Hyperlinks are defined with the Link constructor, in the same manner as forms.

Templates. The introduction already mentioned page templates as a useful abstraction element for pages with recurring content. We use a transformation-based approach to implement templating: A template function transforms a given input APD into an output APD in a filter-like manner, provided that the input conforms to some simple constraints. It must provide a "slots" element that contains a list of named slots. These slots are then merged with the template. It is the template function's job to define how the slots are actually rearranged, and thus it defines the general structure of all pages that use this template.

A template function traverses the given APD term and processes the slots it knows about. Typically, the slots' content is copied into a new APD term that represents the output page. Generally, arbitrary transformations on the input are possible.

Thus, templates are an example of an abstraction that is introduced in order to reduce complexity. This easy abstraction is possible due to the functional language.

4.5 Queries and Updates

We rely on graphs to store the content data. This has many advantages compared to other data structures or even to storing the data in an external database system. Since the content model is given in terms of classes and associations, it is possible to use almost any kind of representation for the underlying content repository.

One strong point for graphs is the possibility to employ powerful graph *querying* to retrieve values from it. In our implementation, we defined a simple querying interface to graphs.

Consider, as an example, the following query that retrieves the list of all available cities:

```
queryAllCities :: AttributedGraph → [String]
queryAllCities g =
    nodesToValues
        g
        (λ lbl → getValue lbl "id")
        (query g (nodes g) [ constrainByType "City" ])
```

Without diving into the implementation details, you can see from the signature that this function returns a list of strings, given a concrete Graph g. It does so by first selecting all nodes that are of type City, and then mapping a function that extracts the value of the id attribute over this list of nodes, resulting in the desired list of city names.

The function queryAllCities is used in the definition of getOriginCities and getDestCities.

Updating the graph is done analogously, by defining a function that returns the changed graph as its result. The calling function then puts this new graph into the session context, replacing the old one.

4.6 Presentation

The presentation model is given by defining one or more mappings (presentation functions) from an APD to the corresponding presentation level language. In the case of a web application that is to be rendered by a user agent that understands XHTML, a simple transformation of the regular APD into XHTML was implemented as a Haskell function. Alternatively, the APD could be converted to any other XML dialect first, and subsequent transformations may be done with technologies like XSLT [3]. All conceivable possibilities are open at this point, and the approach can be easily adapted to a great number of run-time systems. Note that the actual transformations can be selected at run-time, even on a page-to-page basis, or according to context information. This opens the path to adaptive web applications, encompassing customisation, personalisation, and multi-mediality.

Our implementation uses a straight-forward transformation of an APD into XHTML (cf. the example in figure 4). Links and form actions are coded into simple URL query strings. As a proof-of-concept, this is sufficient, but we would like to enhance this by integrating a proper web server (see the outlook in section 6).

4.7 Dynamics

The 'business logic' or *dynamics* of a web site is captured in the requirements documents. Use cases, for example, are employed to describe the behaviour of the site and which interaction steps are possible.

Using the terminology defined in the content model, many statements concerning the behaviour can be formalised. We suggest to do this with functional specifications. This way, the dynamics is broken down into well-specified functions that can be glued together in the page function definitions.

As a simple example, consider that, for the TAS, the requirements state that a trip description must always be well-formed, meaning that "(a) The cities denoted by originCity and destinationCity are not equal; and (b) dateOfReturn is later than dateOfDeparture." (that is, the travel agency is not happy if you order a trip of length zero, and they don't offer time travels either). This statement is captured by a function:

```
checkWellformedness :: TripDescriptionRecord → Bool
checkWellformedness td =
  (originCity td ≠ destinationCity td)                              (1)
  && (
    if (isJust (dateOfReturn td)) -- is the return date provided at all?
    then (fromJust (dateOfReturn td) > (dateOfDeparture td))        (2)
    else True
  )
```

Line (1) tests statement (a), and line (2) lets the function return True if, and only if, statement (b) holds as well.

The function is used in the page that shows the trip list (see line (1)):

```
tasTripList :: PageInfo
tasTripList = PageInfo "TASTripList" $
  λ params →
    do (graph, session) ← get
      navigation ← tasNavigation params
      tripDescr ← return $ validateTripDescription params (graph, session)
      if (isJust tripDescr && checkWellformedness (fromJust tripDescr))        (1)
        then do
            trips ← return $ prepareTrips (fromJust tripDescr) (graph, session)
            -- remainder omitted
```

5 Related Work

Relying on models for describing and specifying web sites has quite a long tradition. Overviews and comparisons of the most prominent approaches are given e.g. in [18], [8], and [16]. The approaches can be very coarsely classified by their 'foundations': Some focus on object-oriented models, others rely on entity-relationship models, and again others put documents into the center of interest. The most influential 'schools' are the graph-based Strudel approach [6], the TSIMMIS project [2], the ER-based RMM [14], Araneus [21], HDM [9] and OOHDM [26], WebML [1], and UWE [17]. The integration of the access to models into a programming language by using a domain specific language is reported in [24].

Significant effort has been put into developing and describing diverse methodologies for web site generation, of which none, to our knowledge, relies as much on functional specifications as we do. We envision a synergetic potential for the integration of our findings into existing approaches, or, vice versa, the integration of selected parts of the aforementioned approaches into ours. This vision was the reason for our approach being as abstract and as extensible as possible. The idea of integrating the models by making them functions, which is unique to our approach, clearly works best when *all* models are specified as functions.

It is interesting to compare the various notations used in the respective approaches. Some approaches rely on proprietary notations for some of the diagram types, especially for the hypertext models. A majority of the current approaches employs the UML (and its extension mechanisms) for the notation of diagrams. The main reasons stated for using UML are the availability of tools ([12, p. 2]), the fact that the UML is well-documented ([19, p. 2]), and the coherence gained by using UML for a web application that is connected to other systems that are already modelled using UML ([4, p. 64]). As of today, one can state that using UML class diagrams for the notation of entity-relationship views simply is standard practice.

Our approach is based on functional specifications. We aim at integrating the advantages of this 'way of thinking' into existing web engineering practice. To the best of our knowledge, only very little effort has been put into this direction. Producing HTML and XML with a functional language in a type-safe way is, e.g., investigated in [27], [28]. This is expanded by work directed towards the

specification of XHTML-based, interactive web applications (esp. [30], [13]). A very inspiring solution for Scheme is described in [20].

6 Summary and Conclusion

We described an approach to web site modelling by using functional languages. It is very important to use models as a basis for the development of web sites. We practise a separation of concerns and identify six core aspects that have to be considered for modelling, namely the content, the navigation structure, the pages, the queries and updates, the presentation, and the dynamics. The content model and the navigation structure are captured using 'traditional' object-oriented, conceptual models and a simple, graphical language, respectively. The other four aspects, however, are formalised using a functional language. In our examples, we used the functional programming language Haskell to write down these functional specifications.

The functional programming language can unleash its full power for the benefit of concise, easily maintainable, and re-usable specifications. Furthermore, the specifications are also executable, which is an advantage over the potential 'gap' between an abstract model, given in one language, and a manually crafted implementation written in another. The inherent features of a functional language allows for powerful and new abstractions wherever they are needed, which is almost a necessity to master the complexities of real-world applications. Relying on a wide-spread implementation language like Haskell facilitates the specification even further, because a great array of data structures and function libraries are already available.

Our future work will be directed towards the extension and streamlining of our approach. The specifications could benefit from improving the usage of type information for the content that is stored in the graph. Currently, we rely on simple, string-based labels for the types, while a real type-system, possibly built using the specification language, would be desirable. The same idea applies to the typing of the output documents; here the integration of a domain-specific embedded languages (DSEL) like WASH/HTML (for XHTML documents) might be tried, possibly sacrificing some of our approach's generality.

We would also like to integrate our current implementation with either HSP [22] or WASH/CGI [30], two very powerful web server solutions written in Haskell. Both approaches offer substantial benefits, as they correctly deal with the bookkeeping of states and sessions, and also with the user jumping backwards in the browser history, or cloning the browser window. This integration should also provide an opportunity to test the scalability of our approach.

The template functions we use are an example for an ad-hoc extension that becomes possible because the functional programming language allows to do it. The notion of templates and slots could be sharpened by using a separate data structure for templates.

An end-user who wants to specify a complete web site needs better tool support than a text editor to write Haskell programs with. Libraries with commonly

needed auxiliary functions is not enough. A *visual language* for specifying the pages, for example, could be used by a graphical tool to *generate* the functional specifications. The visual language might be less powerful than the functional one, but it might suffice for the majority of the cases. An interesting compromise between user-friendliness and expressive power is sketched in [15], proposing a more user-friendly way of working with functions in a spread-sheet software. We would also like to investigate the integration of existing graphical modelling languages, so to avoid inventing yet another new visual language.

References

1. Ceri, S.: Web Modeling Language (WebML): a modeling language for designing Web sites. Computer Networks (Amsterdam, Netherlands: 1999), 33(1–6), pp. 137–157 (2000)
2. Chawathe, S., Garcia-Molina, H., Hammer, J., Ireland, K., Papakonstantinou, Y., Ullman, J.D., Widom, J.: The TSIMMIS project: Integration of heterogeneous information sources. In: 16th Meeting of the Information Processing Society of Japan, pp. 7–18, Tokyo, Japan (1994)
3. Clark, J.: XSL Transformations (XSLT) Version 1.0. W3C Recommendation (1999) `http://www.w3.org/TR/xslt`
4. Conallen, J.: Modeling Web application architectures with UML. Communications of the ACM 42(10), 63–70 (1999)
5. Erwig, M.: Inductive graphs and functional graph algorithms. Journal of Functional Programming 11(5), 467–492 (2001)
6. Fernández, M., Florescu, D., Levy, A.Y., Suciu, D.: Declarative specification of Web sites with Strudel. VLDB Journal 9(1), 38–55 (2000)
7. Fleer, J.: Entwurf und Implementierung eines erweiterbaren Web-Portals. Studienarbeit, University of Koblenz-Landau, Koblenz (2005)
8. Fraternali, P.: Tools and approaches for developing data-intensive Web applications: a survey. ACM Computing Surveys 31(3), 227–263 (1999)
9. Garzotto, F., Paolini, P., Schwabe, D.: HDM – a model-based approach to hypertext application design. ACM Transactions on Information Systems 11(1), 1–26 (1993)
10. Gipp, T.: Functional Web Site Specification. Logos Verlag Berlin, Berlin (2006)
11. Gipp, T., Ebert, J.: Web engineering does profit from a functional approach. In: Koch, N., Vallecillo, A., Rossi, G. (eds.) Workshop on Model-driven Web Engineering (MDWE 2005). Proceedings, pp. 40–49. University of Wollongong (July 2005)
12. Gorshkova, E., Novikov, B.: Exploiting UML extensibility in the design of web information systems. In: Proc. Fifth International Baltic Conference on Databases and Information Systems, pp. 49–64, Tallinn, Estonia (June 2002)
13. Hanus, M.: Type-oriented construction of web user interfaces. In: PPDP'06. Proc. of the 8th International ACM SIGPLAN Conference on Principle and Practice of Declarative Programming, pp. 27–38. ACM Press, NewYork (2006)
14. Isakowitz, T., Stohr, E.A., Balasubramanian, P.: RMM: A methodology for structured hypermedia design. Communications of the ACM 38(8), 34–44 (1995)
15. Jones, S.P., Blackwell, A., Burnett, M.: A user-centred approach to functions in Excel. SIGPLAN Not. 38(9), 165–176 (2003)
16. Kappel, G., Pröll, B., Reich, S., Retschitzegger, W(eds.): Web Engineering: Systematische Entwicklung von Web-Anwendungen. dpunkt.verlag, Heidelberg (2004)

17. Knapp, A., Koch, N., Zhang, G., Hassler, H.-M.: Modeling business processes in web applications with ArgoUWE. In: Baar, T., Strohmeier, A., Moreira, A., Mellor, S.J. (eds.) UML 2004 - The Unified Modeling Language. Model Languages and Applications. LNCS, vol. 3273, pp. 69–83. Springer, Heidelberg (2004)
18. Koch, N.: A comparative study of methods for hypermedia development. Technical Report 9905, Ludwig Maximilians-Universität München (November 1999)
19. Koch, N., Kraus, A., Hennicker, R.: The authoring process of the UML-based Web engineering approach (june 2001) (on-line) http://www.dsic.upv.es/ west/iwwost01/files/contributions/NoraKoch/Uwe.pdf
20. Krishnamurthi, S., Hopkins, P.W., McCarthy, J., Graunke, P.T., Pettyjohn, G., Felleisen, M.: Implementation and use of the plt scheme web server. Higher-Order and Symbolic Computation (2007)
21. Mecca, G., Merialdo, P., Atzeni, P.: Araneus in the era of XML. IEEE Data Engineering Bulletin 22(3), 19–26 (1999)
22. Meijer, E., van Velzen, D.: Haskell server pages - functional programming and the battle for the middle tier. Electronic Notes in Theoretical Computer Science, 41(1) (2001)
23. Meliá, S., Kraus, A., Koch, N.: Mda transformations applied to web application development. In: Lowe, D.G., Gaedke, M. (eds.) ICWE 2005. LNCS, vol. 3579, pp. 465–471. Springer, Heidelberg (2005)
24. Nunes, D.A., Schwabe, D.: Rapid prototyping of web applications combining domain specific languages and model driven design. In: ICWE '06. Proceedings of the 6th international conference on Web engineering, New York, NY, USA, pp. 153–160. ACM Press, NewYork (2006)
25. Peterson, J., Chitil, O.: The Haskell Home Page. (December 2004) http://www.haskell.org/
26. Schwabe, D., Rossi, G.: The object-oriented hypermedia design model. Communications of the ACM 38(8), 45–46 (1995)
27. Thiemann, P.: Modeling HTML in Haskell. In: Pontelli, E., Santos Costa, V. (eds.) PADL 2000. LNCS, vol. 1753, p. 263. Springer, Heidelberg (2000)
28. Thiemann, P.: A typed representation for HTML and XML documents in Haskell. Journal of Functional Programming 12(4 and 5), 435–468 (2002)
29. Thiemann, P.: An Embedded Domain-Specific Language for Type-Safe Server-Side Web-Scripting. ACM Transactions on Internet Technology 5(1), 1–46 (2005)
30. Thiemann, P.: Web Authoring System Haskell (WASH) (February 2007) http://www.informatik.uni-freiburg.de/~thiemann/haskell/WASH/