Effective Mapping of Hypermedia High-Level Design Primitives to Implementation Environments

Walter A. Risi **Daniel H. Marcos** **Pablo E. Martínez López**

LIFIA, Facultad de Informática, Universidad Nacional de La Plata.
C.C.11, Correo Central, 1900, La Plata, Buenos Aires, República Argentina.
E-mail: {walter,daniel,fidel}@lifia.info.unlp.edu.ar
URL: http://www-lifia.info.unlp.edu.ar/

Abstract

As the inherent complexity of hypermedia applications grows, the need for high-level design models and methods becomes imperative. There are several software engineering methods specially tailored for the domain of hypermedia applications, which take into account the special needs of this kind of applications. However, the mapping of these high-level models to the implementation environments often results in a drastic loss of richness.

This paper presents a domain-specific language which can be used to as an intermediary between the software engineering models and the implementation environment. We show, through examples, how primitives found in different hypermedia design methods can be mapped to our language, still preserving the original expresiveness. Our language also provides rendering facilities which allow to obtain working prototypes of the designs in a particular implementation platform.

1 Introduction

The dramatic expansion of the WWW in the recent years has resulted in a considerable interest in developing large hypermedia applications effectively. As the inherent complexity of hypermedia applications grows, the need for high-level design models and methods becomes imperative. There are several software engineering methods specially tailored for the domain of hypermedia and web applications, which take into account the special needs of this kind of applications. Some of these methods are OOHDM [Schwabe and Rossi, 1995], RMM [Isakowitz *et al.*, 1995], W3DT [Bichler and Nussler, 1996] and HyDev [Pauen and Voss, 1998]. Each one is geared towards a particular kind of application.

Despite the several engineering methods available, the design of web applications still is rather low-level. Commonly used approaches relay on several types of editing environments and technologies. These environments have evolved from the very basic web-page editing tools (such as FrontPage[1] [Fro, 1999]), to more complex site-oriented developing environments (see for example NetObjects Fusion[2] [Net, 1999]). However, site-oriented tools still do not embrace high-level abstractions: site-maintenance features are limited to keeping an uniform look in all pages, or having a general view of the structure of the site.

A dramatic point in the acceptance of engineering approaches is the wide gap between high-level design primitives and the implementation environments. Often, the mapping of models to implementation results in a drastic loss of expressiveness. Moreover, several benefits of the high-level models, such as reuse of structures, are lost when performing the actual implementation.

[1] Copyright 1999, Microsoft Corporation.
[2] Copyright 1999, NetObjects Inc.

In this paper we show how this gap can be reduced, by means of HyCom (Hypermedia Combinators) [Marcos *et al.*, 1997, Marcos *et al.*, 1998]. HyCom is a high-level, domain-specific language for hypermedia authoring. HyCom expresiveness allows direct mapping of design primitives, without risking a loss of richness. Moreover, designs expressed with HyCom can be rendered to HTML pages or other implementation platforms automatically. The version we present here is a redesign of the one presented in previous papers, but the overall ideas remain the same.

This paper is structured as follows: in section 2 we introduce HyCom, and the basic features and structure of its latest version. In section 3, we show through examples how design primitives from a well-known method can be effectively mapped into HyCom. In section 4 we review some related work. Finally, in section 5 we draw some conclusions and present our current research lines.

2 HyCom: A Domain Specific Language for Hypermedia

In recent years, there has been a growing interest in using Domain Specific Languages (DSLs) for software development [Hudak, 1996]. DSLs provide several interesting features to programmers, including a very high-level of abstraction, domain-specific tools, reduced developing times, among others. The most important feature is that developers can think in terms of domain specific abstractions. This feature yields programs that are more concise, and easier to understand and maintain than their counterparts in general purpose languages. Well known DSLs are SQL, LaTeX, and HTML. Examples of domain-specific tools are query optimizers in SQL and BibTeX in LaTeX.

An important group in the DSL family consists of the so-called embedded DSLs (DSELs). These languages are embedded in general purpose languages as domain specific vocabularies (often implemented as libraries). From the software developer's point of view, DSEL present important advantages. On one hand, the developer can benefit from having inmediate access to the power of the host language (this is frequently needed, as DSLs often are not used in isolation). On the other hand, the host language acts as a 'glue' for integrating several DSELs embedded in itself. The idea of DSEL was first proposed by Peter Landin [Landin, 1966], who observed that a programming language consists of a domain independent core, and a set of domain specific vocabularies.

HyCom is a DSEL for hypermedia, embedded in the HOT (Higher-Order, Typed) language Haskell [Peyton Jones and Hughes, 1999]. HyCom was designed with the hypermedia developers' needs in mind, and thus provides the frequently desired facilities (e.g. reuse, automatic prototyping). Moreover, Haskell is a very advanced language, thus developers can benefit from its powerful features (e.g. higher-order functions, a powerful type system, libraries). Furthermore, using Haskell allows us to integrate to other existing DSELs (e.g. CGI scripting, Database handling) in a straightforward manner. Finally, several works have shown the virtues of Haskell as a host language [Leijen and Meijer, 1999].

The general architecture of HyCom is depicted in Fig. 1. In the core, we find the HyCom DSEL itself, which defines its main features. We also find additional utility modules, such as database integration. Another important part is the environment specific features (such as HTML) which allow the developer not only to use environment specific components, but also to render designs to working prototypes in that environment. Finally, there are the authoring libraries, which provide application specific abstractions.

2.1 A Brief Overview of Haskell

Being a DSEL embedded in Haskell, HyCom inherits its notational conventions and semantics. Here, we give a very brief overview of Haskell. More details about Haskell can be found in

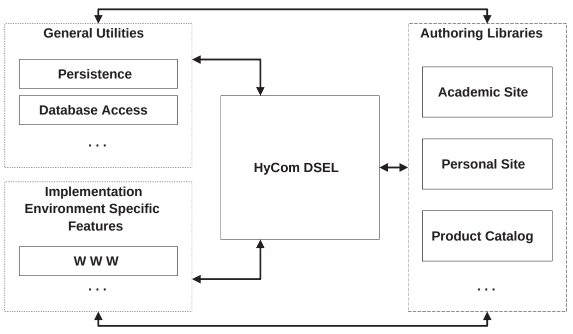

Figure 1: General HyCom Architecture

[Peyton Jones and Hughes, 1999].

Haskell is a functional language, similar to Lazy ML [Augustsson, 1984] or Miranda[3] [Turner, 1985]. The main construct is the function, which is very similar to its mathematical counterpart.

```
-- A function that calculates the minimum of two values. (Comments begin with --).
min x y = if (x<y) then x else y
```

All computations in a functional language are performed by evaluating functions. Function application is denoted by juxtaposition of the function name to its arguments. For example, `min 2 3` is an expression that denotes the application of function `min` to numbers `2` and `3`. The value of the mentioned expression is `2`.

Haskell is also strongly typed, which means that a type is assigned to every valid construct at compile time. Types can be explicitly given by a type signature. Since explicitly giving a type to every construct can be a tedious task, Haskell uses a type inference mechanism. This mechanism can infer, for example, the type of function `min`. Therefore, it is not usually necessary to give type signatures explicitly – though it is good practice, and sometimes may be required. The following is the type signature for the `min` function shown above, meaning that the function takes two `Integer` values and yields another `Integer`.

```
min :: Integer -> Integer -> Integer
```

Actually, the above function can be extended to all values supporting the `<` operation. This can be expressed in Haskell in the following way.

```
min :: Ord a => a -> a -> a
```

The `Ord a => ...` context means that function `min` is defined for every type `a` that is instance of the `Ord` type class. Type classes are a mechanism for overloading, which is similar – though not identical – to the interface construct in Java. Basically, types belonging to a class implement a set of operations defined in that class. For example, instances of class `Eq` implement the `==` operator, and can be compared for equality. The `Ord` type class defines commonly used order operators, such as `<` ('smaller than') and `>=` ('greater than or equal'). Type classes form a hierarchy, meaning that instances of a subclass implement the operators of that subclass, and also the operators of the superclass – for example, `Ord` is a subclass of `Eq`, meaning that elements that can be ordered can also be compared by equality.

Type contexts are used when one must assume something about a certain type. For example, in function `min`, we must assume that for type `a`, operator `<` is defined. When there is no need

[3]Copyright 1986, Research Software Limited

to assume anything about a type, we do not give any context. The trivial example is function `id x = x`, of type `id :: a -> a`. We do not need to assume anything about `a`, so we do not specify any context restriction. This is a simple example of parametric polymorphism in Haskell – in contrast to ad-hoc polymorphism, or overloading, which we already mentioned. In all these cases, we say that `a` is a type variable, since it stands for any type, or for all types satisfying a context.

Some classes have more than one parameter, meaning that overloading is done over two types instead of just one. Therefore, a type `T` can be instance of a certain class `ClassName a b`, with `Integer` as the second argument, and also with `Char` as the second argument. This means that an overloaded function of class `ClassName` – say `op :: a -> b -> a` – will behave differently when applied to an object of type `a`, depending not only on the type of the first function parameter, but also on the type of the second. A context for a multiparameter type class has a similar notation to the one for single parameter classes – for example, for class `ClassName` we could have `ClassName a b => ...` as a context for a certain function.

The most important feature of Haskell is the possibility of defining higher-order functions. Higher order functions may take a function as its argument, and may yield another function as its result. The classical example is `map`, which takes a function `f`, and returns another function that applies `f` to all the components of a list. The following is the type signature for `map` – note that brackets are used to denote lists.

```
map :: (a -> b) -> [a] -> [b]
```

For example, applying `map negate` to the list [1,2,3] yields the list [-1,-2,-3], and applying `map succ` to the same list yields [2,3,4]. Higher-order functions are a very powerful feature, and allow reuse at a very large scale [Hughes, 1989]. Haskell libraries provide a rich set of higher-order functions (such as `map`), but the user can also define her own.

Another very important feature of Haskell is related to types. Haskell gives the user several ways to define its own types. The first is through type synonyms. Type synonyms do not define new types, but rather they give new names to existing ones. For example, a type synonym declaration like `type Amount = Integer` says that type `Amount` is another name for `Integer`. Truly new types can be defined using algebraic data types. Algebraic data types allow to define types by enumerating its elements. For example, the following is a type declaration for a pair of values of types `a` and `b`, respectively.

```
data Pair a b = Pair a b
```

As it can be seen from the above code, type declarations can be parameterized – this also stands for type synonyms. Moreover, parameters in type declarations can be constrained by type class contexts, in the same way as we did for functions. Note also that we used the same `Pair` in both sides of the equation. This is possible since the left side is denoting how a type is constructed – thus, `Pair` is called here a *type constructor* – and the right side denotes how a data element of that type is built – therefore, `Pair` in the right side is called a *data constructor*. An alternative way for writing the previous declaration is the following.

```
data Pair a b = Pair { fst :: a, snd :: b }
```

This second version is identical to the last one, except that values are labelled, and one can assume that there exist functions `fst` and `snd`, to retrieve the first and second components. With both versions, one can also retrieve the components by pattern matching. Algebraic data types can also be recursive, allowing to define recursive data structures such as lists and trees.

Other interesting feature of Haskell are infix operators. Functions, normally prefix, can also be expressed in an infix fashion, thus gaining expressiveness in certain situations. For example, we can define an infix version of the `min` function already defined.

```
min, (.@.) :: Num a => a -> a -> a
(.@.) = min
```

Now we have an operator `(.@.)` that has the same functionality as `min`, but can be used infix. That means that, for example, `1.@.2` is the same as `min 1 2`. In general, infix operators are a very powerful feature for gaining expressiveness. Furthermore, it is possible to define precedence and associativity for infix operators.

To finish with this brief overview of Haskell, we present local declarations. A local declaration allows to define functions only in the scope of another declaration. For example, the following defines `sum` only in the scope of `double`.

```
double x = sum x x
  where sum x x = x + x
```

This has been only a very short introduction to Haskell capabilities. We have limited ourselves to explaining the features needed to understand this paper. The interested reader can found further information in the references already mentioned.

2.2 The HyCom DSEL

The structure of the HyCom DSEL is given by the type class hierarchy shown in Fig. 2. Note that this hierarchy is not denoting a class-hierarchy in the object-oriented (OO) sense, but rather a type class hierarchy in the sense that we explained in previous section. Also note that when we speak of data object we are referring to a data element of a certain type, and not to an object in the OO sense.

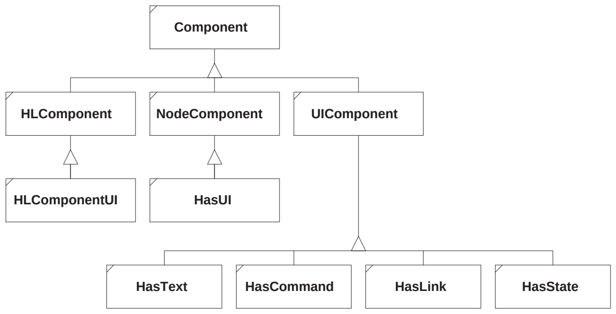

Figure 2: HyCom Type-Class Hierarchy

The base class in the HyCom hierarchy is `Component`. A component is a data object with an associated ID, that allows to univocally identify the object. A component supports operations `getID` and `setID`. Almost every data object in HyCom is a component. Subclasses of `Component`, such as `HLComponent` and `UIComponent` provide hyperlinking and user-interface related operations, respectively.

The most important concept in HyCom is that of combinators and transformers – hence the name, **Hy**permedia **Com**binators. A combinator is a function that takes two or more components, and yields a new one resulting from their combination. A transformer is a function that takes a component, and returns a new one resulting from adding some kind of feature to it. Typical transformers and combinators are the ones used for user interface components.

```
aboveOf,(/=\) :: (UIComponent a, UIComponent b) => a -> b -> Layout a b
leftOf,(<=<)  :: (UIComponent a, UIComponent b) => a -> b -> Layout a b
anchored      :: (UIComponent c, HLComponent l) => c -> l -> Anchored c l
```

The `aboveOf` combinator takes two user interface components, and returns a composed component in which the first argument is placed above the second one. The `leftOf` combinator behaves similarly, but places its arguments one in the left of the other. The `anchored` transformer takes an UI component and a Hyperlink component, and returns an anchor resulting from adding navigational functionality – provided by the link – to the original component.

Typically, authoring in HyCom involves defining application specific components – such as conceptual entities, nodes, links –, embedding them in the type class framework shown before, and glueing them using the built-in components and combinators. When we say that a certain feature is 'built-in', we mean that it is provided by HyCom's basic libraries. The following code shows a simple example.

```
data LabMember   = LabMember { name :: String, photo :: FilePath,
                               personalData :: Profile, lab: Laboratory }
type LabMemberC = BasicComponent LabMemberC
```

In the above code, a data object `LabMember` is defined to hold data about members of a laboratory. Type synonym `LabMemberC` is built by applying the built-in type constructor `BasicComponent` to the `LabMember` type. The resulting type is an instance of the `Component` class, since `BasicComponent` provides the required functionality. Next, we define an application specific node, which not only holds data, but also a hyperlink.

```
data MemberNode  = MemberNode LabMemberC MemberLink
type MemberNodeC = NodeComponent MemberNode
```

The application-specific node is defined in the same fashion as the `LabMember`. However, we now use built-in type constructor `NodeComponent`, which implements the basic functionality required for nodes. This yields an instance of the class `NodeComponent`, that we call `MemberNodeC`. The link type `MemberLink` used in the node definition is defined as follows.

```
type MemberLink = PlainLink MemberNodeC MemberNodeC
```

Type `MemberLink` is an application-specific link, defined by applying built-in type constructor `PlainLink` – which provides the basic hyperlinking facilities. A member link is thus a plain link – the most simple type of link –, going from a `MemberNodeC` to another `MemberNodeC`.

Normally, applications from the same conceptual domain use very similar conceptual entities. Therefore, it is not necessary to define new components each time a new application needs to be developed. Instead, the author can use components provided by authoring libraries. Authoring libraries are described in the next section.

2.3 The Authoring Libraries

HyCom takes the DSEL approach to its limit through authoring libraries. As the HyCom DSEL is a hypermedia-specific vocabulary embedded in Haskell, authoring libraries provide application-specific vocabularies embedded in HyCom. In this way, developers of a particular kind of application do not have to define the basic abstractions from scratch, since they are provided by the library.

A typical example is the academic site authoring library. Academic sites often present a similar structure. The underlying conceptual model is similar, and involves entities such as professors, researchers, students, subjects, projects, research-areas, etc. The authoring library already provides types to represent these entities, and also default navigational constructs typically found in this kind of application – for example, a guided tour defined over the professors of a particular area, etc. These reusable components are defined basically in the same way that was shown in the previous section.

Authoring libraries not only allow to reuse components, but also design decisions about the overall hypermedia structure. There exists a set of hypermedia design patterns [Rossi *et al.*, 1997] that provide useful guidelines to enhance the quality of the applications. We are taking account of these patterns in designing the libraries.

Another interesting possibility are authoring assistants. Basically, authoring assistants ask the user several questions about the application to be developed, and generate the basic skeleton of the application. Therefore, authoring assistants complement authoring libraries, by reducing the – often high – initial effort in application development. Authoring assistants are currently under development.

3 Mapping Design Primitives to HyCom

In this section we show examples on how to map design primitives to HyCom. We focus on one of the most widely accepted hypermedia design methods, the Relationship Management Methodology (RMM). We are currently working on a mapping of OOHDM [Schwabe and Rossi, 1995] concepts to HyCom, which we will cover in a future paper.

3.1 A Brief Overview of RMM

The Relationship Management Methodology (RMM) is often referred as the first true hypermedia design methodology. First presented by Isakowitz et al. in 1995 [Isakowitz *et al.*, 1995], it has gone through several enhancements since then. The version we use here is the one presented in [Isakowitz *et al.*, 1998]. In what follows, we give a very brief overview of RMM, and present the example with which we will work from now on. Note that we assume a basic knowledge of RMM. Readers with no previous knowledge of this methodology should look in the following papers [Isakowitz *et al.*, 1995, Isakowitz *et al.*, 1997a, Isakowitz *et al.*, 1997b, Isakowitz *et al.*, 1998].

The RMM is based on the well-known Entity-Relationship (ER) model. The user starts defining a conceptual model of the application, using entities and relationships. This comprises the first step of the method. A very simple ER model is shown in Fig. 3.

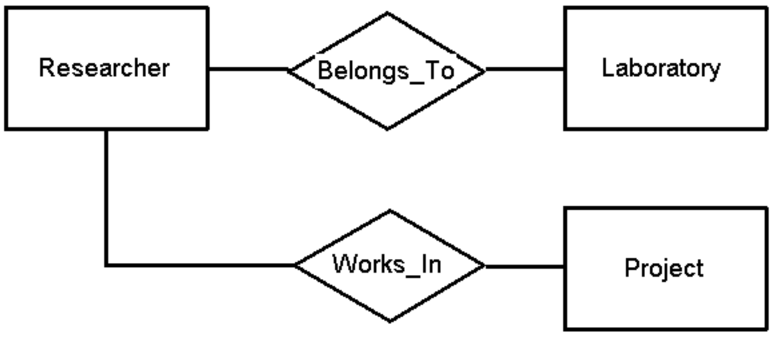

Figure 3: An ER Diagram

The following two steps involve doing a top-down and a bottom-up design of the general navigational structure of the application. The overall goal is to obtain the application diagram. An application diagram consists of M-Slices and links between them. M-Slices are presentational units, and consist of attributes from one or more entities (in particular, each M-Slice has an 'owner' entity), link anchors, access structures (like indexes or guided tours), and other M-Slices. An M-Slice is an abstract representation of a piece of information that is going to be presented in the final application. Top-level M-Slices are actually the nodes of the application.

In Fig. 4 we show an application diagram corresponding to the ER diagram shown previously. Here we have three top-level M-Slices and links among them. The complete definition of the

Researcher **top** and **name** M-Slices is depicted in Fig. 5. The Researcher top M-Slice has some attributes from its owner (the Researcher Entity), namely rank and photo. It also includes another M-Slice from its owner, the name M-Slice. The lower part of the M-Slice includes features not belonging to the owner entity, such as the Laboratory Logo, which acts as an anchor to the Laboratory top M-Slice. Also, it features an index of projects, anchored in the Project name M-Slice – actually, this is a a particular kind of M-Slice with a single attribute.

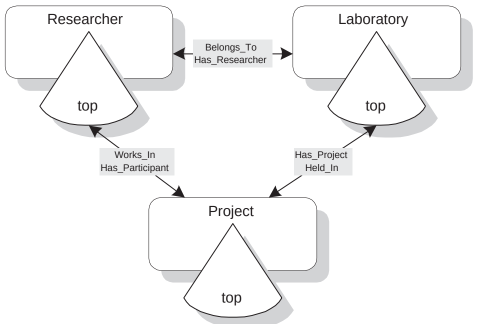

Figure 4: An Application Diagram

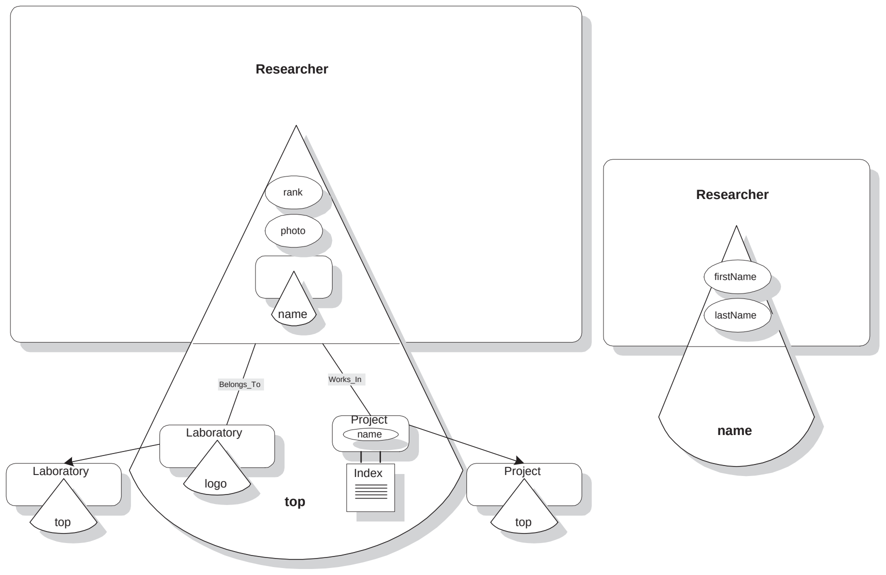

Figure 5: M-Slices **top** and **name** Belonging to the Researcher Entity

The RMM defines four more phases. However, these phases have not been formally defined by its authors, and generally, developers take an ad-hoc approach for these phases [Balasubramanian *et al.*, 1996]. Therefore, we will focus on the phases described above, which are the cornerstone of the method.

3.2 Mapping RMM to HyCom

In this section, we show how to develop a possible mapping of RMM features to HyCom. We use the already presented example to develop the mappings.

The first step is simple, and involves representing the ER model with a set of Haskell types. There are many ways of doing this, and here we show a straightforward one.

```
data Researcher = Researcher {firstName :: String, lastName :: String, rank :: String,
                                  photo :: String, intro :: String, projects :: [Project],
                                  lab :: Laboratory }
```

Here, we decide to include the relations in the entities, since relations have no attributes. This is not necessarily the case, and relations can be also represented by types in the same fashion as entities. We also choose to make entities instances of the component class – this is easily done by providing definitions for `getID` and `setID` operations.

Actually, the developer has to deal with sets of entities, normally stored in a database. Type class `Retriever` allows us to hide the storage details, thus providing an uniform interface to different database implementations. HyCom provides built-in components implementing `Retriever` functionality for file-based resources and relational databases.

```
class Retriever r a where
  retrieveBy :: (a -> Bool) -> r -> IO [a]    -- Retrieve all entities satisfying
                                              -- a predicate.
  retrieve   :: r -> IO [a]                   -- Retrieve all entities.
```

The interesting part consists in representing slices – from now on, we will use the terms slice and M-Slice interchangeably. We want to express slices clearly, but also capitalize on type-checking to prevent defining ill-formed slices. The first thing we do is defining a type class `Owned`, to constrain slice components. Class `Owned` has type information about the owner, and also allows us to retrieve the `ID` of the owner.

```
class Component o => Owned a o where
  owner :: a -> o
```

We can move now to representing slices. Slices have an owner, a set of components belonging to the owner, and a (possibly empty) set of components not belonging to the owner. Thus, we define type `Slice`, which holds a combination of components belonging to the owner, a combination of components not belonging to the owner, and also information about the owner itself. Note that we use type contexts to constrain both sets of combinations.

```
data (Component owner, Owned a owner) => Slice a b owner = Slice a b owner ID
```

The above type constructor `Slice` has three parameters. Note that context `Owned a owner` restricts the component combination of type `a` to have the the same owner type as the owner of the slice – given by the parameter `owner`. Note also that the data constructor `Slice` – on the right side of the declaration – also holds data for an `ID`, which is required to make `Slice` an instance of the `Component` class. Making a comparison with the graphical notation of slices, we can see that the constrained combination – of type `a` – corresponds to the upper portion of the slice, and the other combination – of type `b` – corresponds to the lower portion of the slice.

Some slices only hold data from its owner. For this particular kind of slice, we provide type `PureSlice`. This is similar to the `Slice` type, except that it does not hold the combination of components not belonging to the owner.

```
data (Component owner, Owned a owner) => PureSlice a owner = PureSlice a owner ID
```

Other interesting representations of RMM concepts are types `Att` which models attributes or single attribute M-Slices –, `RMM_Anchor` – which models anchored components within slices –, and `RMM_Index` – which models indexes. We do not show the definitions of these components here. All these types are instances of the `Owned` class, allowing to prevent formation of invalid structures.

We mentioned previously that a slice holds combinations of components, either belonging to the owner or not. Combinations are modelled with combinator functions – the main concept underlying HyCom. We provide two types of combinations: combinations of components of the same owner – `ownerCombined` –, and combination of any components – `sliceCombined`. These combinators are modelled with types, and functions to create data objects of those types.

```
data (Component o, Owned a o, Owned b o) => OwnerCombined a b o = OwnerCombined a b
ownerCombined,(+--+) :: (Owned a o, Owned b o) => a -> b -> OwnerCombined a b o
data SliceCombined a b = SliceCombined a b
sliceCombined,(-##-) :: a -> b -> SliceCombined a b
```

These combinators prevent the user from mixing components belonging to the slice's owner with the ones not belonging to it. Note that in `ownerCombined` we used type contexts in a similar way that in the `Slice` declaration: the `Owned a o` and `Owned b o` contexts ensure that components from different owners are not combined, and that the resulting combination also belongs to that owner. Note also that we provided infix versions of the combination functions.

Some other interesting functions are the ones used to build anchors, indexes, and single attribute slices.

```
rmmAnchor :: (HLComponent l) => a -> l -> RMM_Anchor a l
rmmIndex  :: (Component owner, HLComponent l, Owned a owner) =>
               (owner -> a) -> (owner -> l) -> [owner] -> o -> RMM_Index a l o
att       :: (c -> a) -> c -> Att a c
```

Next, we show the HyCom based definition for the Researcher top M-Slice. We define a function `researcherTop`, which takes a `researcher` entity, a `lab` entity and a list of `projects`, and returns the Researcher top M-Slice. Note that we use functions `att`, which is used to build attributes – attributes have a value, and also information about their owner entity –, and `linkTo`, which is used to build a link given a destination slice. We do not show the definitions for these functions here.

```
researcherTop researcher lab projects = slice fromOwner other researcher

  where fromOwner = (att rank researcher)        +--+
                    (att photo researcher        +--+
                    (researcherName researcher)

        other     = (rmmAnchor (laboratoryLogo lab) linkToLab)  -##-
                    (rmmIndex projectNameAtt linkToProject projects researcher)

        linkToLab  = linkTo (laboratoryTop lab)
        linkToProject p = linkTo (projectTop p)
```

Compare the above code with the slice definition previously shown. Notice that the HyCom/Haskell based version has not lost the original expressiveness. The `fromOwner` local declaration corresponds to the upper part of the slice graphical notation, and the `other` declaration corresponds to the lower part. Furthermore, using types prevents the user from defining ill-formed slices.

The next step consists in defining how the M-Slice is going to be seen in the interface. That can be done using HyCom UI Components and combinators. We define a function that takes a `researcherTop` M-Slice, and returns an UI component. In the following, we assume that we have functions for retrieving the internal components of the slice, such as the name M-Slice, the rank attribute, etc – those functions can be easily defined by pattern matching. We also make use of some other UI components, such as `divisionLine`, `rmmIndexUI` (UI for indexes), `researcherNameUI` (UI for Researcher name M-Slice), for which we do not show the definitions.

```
researcherTopUI researcherTop =  header           /=\
                                 personalData     /=\
                                 researchInterests
  where header            = laboratoryLogoUI logo /=\
                            divisionLine
```

```
personalData      = researcherNameUI theName          /=\
                    (textLabel rank <=< image photo) /=\
                    divisionLine
researchInterests = (textSize 4 (textLabel "Projects")) /=\
                    (rmmIndexUI indexOfProjects)
nameSlice         = getNameSlice researcherTop
projectIndex      = getProjectIndex researcherTop
logo              = getLogoSlice researcherTop
photo             = getAttrContent (getPhotoAttr researcherTop)
rank              = getAttrContent (getRankAttr researcherTop)
```

Again, notice that even when defining the UI, the original design constructs (slices, attributes) are still present. The resulting UI component can be compiled, automatically yielding a prototype in HTML or other platform. The prototype HTML page corresponding to the Researcher top M-Slice can be seen in Fig. 6 – notice that we have marked the UI representations of the RMM constructs, to allow easy comparison with the HyCom code.

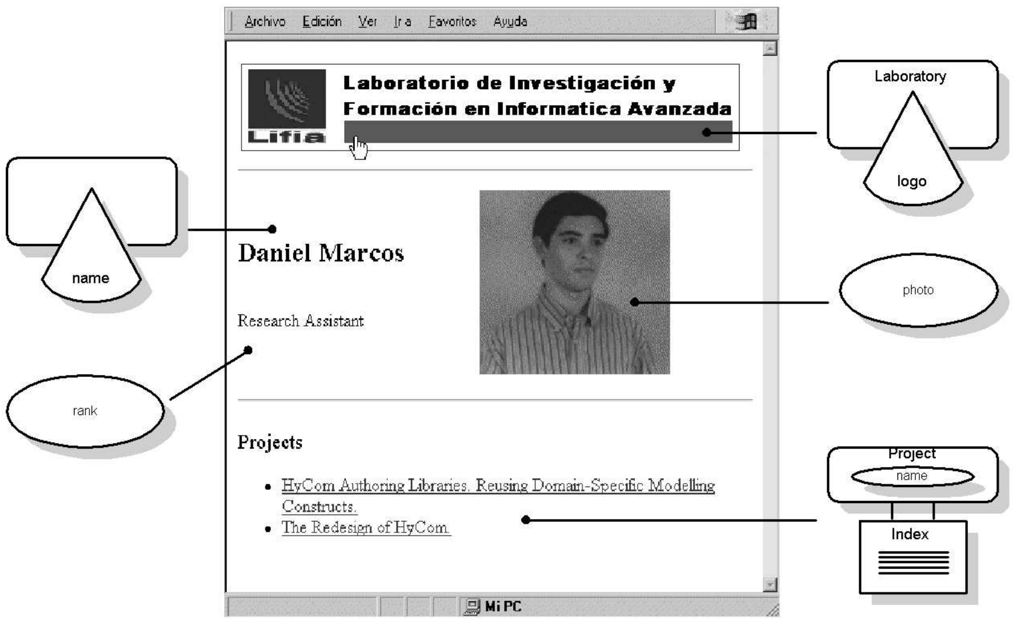
Archivo Edición Ver Ir a Favoritos Ayuda
Laboratorio de Investigación y Formación en Informatica Avanzada
Lifia
Laboratory
logo
name
Daniel Marcos
Research Assistant
photo
rank
Projects
HyCom Authoring Libraries. Reusing Domain-Specific Modelling Constructs.
The Redesign of HyCom
Project
name
Index
Mi PC

Figure 6: Web Page Corresponding to the Researcher Top M-Slice.

4 Related Work

Several works have faced the problem of the mapping of design primitives to implementation environments. However, we claim that none of them embraces the high-level principles present in HyCom. Most of the existing works emphasize implementation-oriented features, such as dynamic generation of web pages, but the expressiveness and abstraction level of those approaches are often left unattended.

The WebComposition approach and its XML-based implementation, the WebComposition Markup Language (WCML) [Gellersen *et al.*, 1997, Gaedke *et al.*, 1998] are in some way, an object-oriented counterpart to HyCom. However, we believe that the object model adopted is poor – objects support very limited functionality. Moreover, WCML lacks features like type checking to avoid the definition of invalid constructs. HyCom also differentiates from WCML for being an embedded language – which allows easy integration with existing DSELs.

Another approach is that of the OOHDM-Web [Schwabe and de Almeida Pontes, 1998] environment, that proposes mapping OOHDM design primitives to relational tables, HTML templates, and Lua scripts. We claim that the mapping of objects to tables as proposed by OOHDM-Web still results in a loss of richness.

5 Conclusions and Future Work

The wide gap between high-level design primitives and implementation environments often results in a dramatic loss of expressiveness. Benefits obtained from the high-level models are actually lost when performing the implementation. In this paper, we pointed out how this gap can be reduced by means of HyCom, a DSEL for hypermedia. HyCom has a very high level of abstraction, and allows design primitives to be mapped in a very expressive way. Moreover, the resulting mapping benefits from type checking and prototyping facilities provided by HyCom.

Our current research lines include the development of solid authoring libraries. We plan to build meta authoring-libraries for the most widely used hypermedia design methods, and base the concrete authoring libraries on those methods. This would allow us to take advantage of those engineering models for documenting the libraries. We also plan to take advantage of hypermedia design patterns in designing the libraries.

Other interesting aspects for research involve the development of software-based development environments. Our primary target is the development of authoring assistants, to complement the authoring libraries. An interesting possibility is that of designing a CASE tool based in hypermedia design methods, using HyCom as the underlying language to describe designs.

References

[Augustsson, 1984] L. Augustsson. A compiler for lazy ML. In *Proceedings of the ACM Symposium on Lisp and Functional Programming, Austin*, pages 218–27, August 1984.

[Balasubramanian *et al.*, 1996] V. Balasubramanian, Michael Bieber, and Tomás Isakowitz. Systematic Hypermedia Design. October 1996.

[Bichler and Nussler, 1996] Martin Bichler and Stefan Nussler. Modular Design of Complex Web-Applications with W3DT. In *Proceedings of the 5th Workshop of Enabling Technologies: Infrastructure of Collaborative Enterprises (WET ICE '96), Stanford.* IEEE Computer Press, 1996.

[Bieber and Isakowitz, 1995] M. Bieber and T. Isakowitz, editors. *Communications of the ACM*, volume 38 (8). ACM Press, August 1995.

[Fro, 1999] FrontPage Home Page, Microsoft Corporation. URL: `http://www.microsoft.com/FrontPage/`. 1999.

[Gaedke *et al.*, 1998] M. Gaedke, M. Beigl, and H. W. Gellersen. Mobile Information Access: Catering for Heterogeneous Browser Platforms. In *Proceedings of the International Workshop on Mobile Data Access in Conjunction with 17th International Conference on Conceptual Modelling (ER98), Singapore*, 1998.

[Gellersen *et al.*, 1997] H. W. Gellersen, R. Wicke, and M. Gaedke. WebComposition: An Object-Oriented Support System for the Web Engineering Lifecycle. In *Computer Networks and ISDN Systems 29, Special Issue of the 6th World-Wide Web Conference, Santa Clara, CA, USA*, 1997.

[Hudak, 1996] Paul Hudak. Building Domain-Specific Embedded Languages. 1996.

[Hughes, 1989] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[Isakowitz *et al.*, 1995] T. Isakowitz, E. A. Stohr, and P. Balasubramaninan. RMM: A Methodology for Structured Hypermedia Design. [1995], pages 34–44.

[Isakowitz *et al.*, 1997a] T. Isakowitz, A. Kamis, and M. Koufaris. Extending The Capabilities of RMM: Russian Dolls and Hypertext. In *Proceedings of the 30th Annual Hawaii International Conference on System Sciences*, 1997.

[Isakowitz *et al.*, 1997b] T. Isakowitz, A. Kamis, and M. Koufaris. Reconciling Top-Down and Bottom-Up Design Approaches in RMM. In *Proceedings of the Workshop on Information Technologies and Systems (WITS-97), Atlanta, GA*, December 1997.

[Isakowitz *et al.*, 1998] T. Isakowitz, A. Kamis, and M. Koufaris. The Extended RMM Methodology for Web Publishing. 1998. Working Paper IS-98-18, Center for Research on Information Systems.

[Landin, 1966] Peter Landin. The Next 700 Programming Languages. *Communications of the ACM*, 9(3):15–164, February 1966.

[Leijen and Meijer, 1999] Daan Leijen and Erik Meijer. Domain Specific Embedded Compilers. Submitted to 2nd USENIX Conference on Domain-Specific Languages, 1999.

[Marcos *et al.*, 1997] Daniel H. Marcos, Pablo E. Martínez López, and Walter A. Risi. Expresando Hypermedia en Programación Funcional. In *Proceedings of the Second Latin American Conference on Functional Programming (CLAPF97), La Plata, Buenos Aires, Argentina*, 1997.

[Marcos *et al.*, 1998] Daniel H. Marcos, Pablo E. Martínez López, and Walter A. Risi. A Functional Programming Approach to Hypermedia Authoring (Poster). In *Proceedings of ACM International Conference on Functional Programming (ICFP98), Baltimore, Maryland, USA*, 1998.

[Net, 1999] NetObjects Home Page. URL: `http://www.NetObjects.com`. 1999.

[Pauen and Voss, 1998] Peter Pauen and Josef Voss. The HyDev Approach to Model-Based Development of Hypermedia Applications. In *HyperText 98 Workshop, 1st International Workshop on Hypermedia Development: Process, Methods and Models*, 1998.

[Peyton Jones and Hughes, 1999] Simon Peyton Jones and John Hughes. Report on the Programming Language Haskell '98. Technical report, Yale University, February 1999. Available online: `http://www.haskell.org/report`.

[Rossi *et al.*, 1997] G. Rossi, D. Schwabe, and A. Garrido. Design Reuse in Hypermedia Design Application Development. In *Proceedings of the ACM International Conference on Hypertext (HT97), Southampton*. ACM Press, April 1997.

[Schwabe and de Almeida Pontes, 1998] Daniel Schwabe and Rita de Almeida Pontes. OOHDM-WEB: Rapid Prototyping of Hypermedia Applications in the WWW. Technical report, Dept. of Informatics, PUC-Rio, 1998.

[Schwabe and Rossi, 1995] D. Schwabe and G. Rossi. The Object-Oriented Hypermedia Design Model. [1995], pages 45–46.

[Turner, 1985] D. A. Turner. Miranda - a non-strict functional language with polymorphic types. In J. P. Jouannaud, editor, *Conference on Functional Programming Languages and Computer Architecture, Nancy*, pages 1–16. Springer, 1985.