

# A Lazy approach to separating architectural concerns

Eric Wohlstadter and Prem Devanbu  
Cal Aggie Software Tools Laboratory,  
Dept. of Computer Science,  
University of California, Davis, CA 95616 USA  
email: [wohlstad@cs.ucdavis.edu](mailto:wohlstad@cs.ucdavis.edu)

## Abstract.

To achieve the goal of composable software systems, we define new notions of components and higher order connectors using a higher-order polymorphic functional language. We describe higher order connectors that take other connector functions as arguments and compose them, returning a new connector. The benefits of language features such as curried functions, lazy evaluation, and pure computation lead to an elegant separation of component interaction and functionality. We demonstrate our ideas by showing how they could apply to a client-server architecture and an event service component.

## 1. Introduction

There is an increasing desire to be able to construct programs from reusable software components. The advantages of such software composition are already well documented. Several problems stand in the way of realizing truly composable components. Current programming practices and languages do not cleanly separate the aspects of *computation* from the aspect of inter component *interaction*. Our research is to seek a model of programming that cleanly distinguishes the two and treats components and connectors as first class entities [8]. Systems could then be constructed out of these first class entities by composition. Our initial approach to the problem has been to look at architectural descriptions in a pure, lazy, higher-order functional programming environment. This approach has several benefits, the foremost being that it is easy to type check and is executable. The fact that it is executable allows a software architect to simulate different component-connector configurations with minor modifications to the architecture description. We choose the pure/lazy functional programming language Haskell [4] and explore those parts of the language that allow it to be used as a powerful software composition tool. At first, it may seem a bit odd to model interactions within a pure, lazy language; after all, interactions, by their very nature are strict: they happen not when the compiler wants them to happen, but when we want them to happen! However, the laziness of the underlying implementation mechanism forces interactions to be explicitly represented, which, as we shall see, adds to clarity and checkability.

## 2. Background

A number of the features available in the Haskell programming language are fairly standard among functional languages. These include first-class functions, lambda abstractions, and curried forms. Others such as lazy evaluation and pure computation make it different from languages such as ML or Scheme. Some of the strategies we use in developing higher-order connectors rely heavily on curried functions, laziness, and a pure model of computation so we present a brief explanation.

Currying refers to the construction of new functions by evaluating other functions with only some of their arguments. The new function is identical to the old definition except: the occurrences of supplied arguments in the body are replaced by the formal parameters, and the supplied arguments vanish from the function signature. This becomes extremely useful when functions are used to model the connections between components; it now becomes possible to build up new connection functions by composing others. We can combine this idea with lazy evaluation to allow flexible composition of connection functions. Lazy, or non-strict, programs evaluate expressions only when necessary. A benefit of this is that function arguments that are not used in particular invocations are not needlessly evaluated. This leads naturally to the idea of values as frozen computations. When a function takes an argument of type  $\alpha$ , we can pass in an entire expression that when evaluated reduces to type  $\alpha$ . Happily, the actual evaluation happens only in the body when necessary. If evaluation of some expression produces side effects, lazy evaluation is crucial to preserve the semantics of interaction described by function definitions.

Consider a software architecture setting where we have some component that reads a value of type  $\alpha$  from a connector at certain points during its execution. Using a new notion of connectors as expressions we can give the component an expression *that when evaluated* results in a type  $\alpha$ . It is as if we are giving

the component specific instructions on what to do *when* it wants to read from that connector. The component does not even look at the instructions until it is time to read from the connector; whenever it does need to read, the component just follows the instructions in the order we prescribed. These instructions are not simply one function name but any expression! The Haskell type system separates functions that have side effects from those that are pure by assigning them the type `IO a` where `a` is the actual return type. So any expression of type `IO a` can be thought of as a computation to perform when a value is needed from or put into a connector. We can build them up, freeze them, and pass them around as necessary. In Haskell manipulation of functions with `IO` types is done inside the `do` block. This preserves the order of side-effecting computations by using the sequencing `;` operator. The `<-` extracts the `a` from `IO a` computations and forces the frozen computation to actually take place. All expressions not involving `<-` (the so called put or write expressions) also have type `IO a` (usually `IO ()`<sup>1</sup>). This means some side-effecting computation took place but we did not use the result. These `do` blocks are the area where we describe the interactions of components. The functions without `do` blocks are pure, they describe the functionality.

### 3. Simple Client-Server Example

We begin with the simple example of a client-server architecture. We seek to model the interactions of a component with its connections, independently of the functionality of the component [3]. This is achieved by writing server and client functions parameterized by each of their connections and over the functions that manipulate incoming and outgoing values. Thus our server function has three parameters: a `get` function, a `put` function, and the function `f`, which is applied to the input. The function `f` itself can be pure, and its signature is not pre-determined; the construction shown below, then, is higher-order and composable.

```
server get put f = do {
    input <- get;
    put (f input);
    server get put f;
    return ()
}
```

We refer to the `get` and `put` functions as connector functions. They allow `server`, which we view as a component, to communicate with the outside world. They are easy to identify by their type, `IO a`. This specifies that there can be side effects associated with their evaluation. The function `f`, on the other hand, is pure, so we can tell it is part of the component functionality and not a connector function. Parameterizing component functions in this way allows us to build up systems by passing in the right connector functions. Using currying of arguments, one can even provide only some of the connections while leaving others unbound. For example suppose we had the function `getChar` that reads characters from the console. We can make a new component out of our server from the expression `(server getChar)`. This results in a component that only needs one connector function and one additional argument. Manipulating several partially evaluated functions in this manner makes it possible to compose subsystems from components. Lazy evaluation makes it easier to pass functions like `getChar` as arguments and expect they will only be evaluated at the right time. Returning to the server function. The behavior is easy to discern from its body. It grabs some input from the `get` connection, applies the function `f`, applies the `put` connection to the result, and then repeats. Note that `get` and `put` are not restricted to be single functions. They could be compositions of connector functions; laziness will ensure timely evaluation. The client function is similar and its behavior can be seen as relaying information from the user to the server. Note how the connection from the user is decoupled from the connection to the user. We prefer to separate two-way connectors in this fashion in order to maximize reuse.

```
client fromUser toUser toServer fromServer = do {
    input <- fromUser;
    toServer input;
    result <- fromServer;
    toUser result;
    client fromUser toUser toServer fromServer;
}
```

---

<sup>1</sup> `()` is the void type

```

        return ()
    }

```

In the last part of our example we show a piece of code that implements our client-server architecture. The first two lines of the body use primitive operations to create two asynchronous channels that will facilitate communication of the client and the server. Applying the functions `readChan` and `writeChan` to these values create the connector functions that are passed to our components. Additionally we provide the server with some simple arithmetic function that it will apply and the client is passed necessary connections for communication with the console. Finally both client and server are forked into running processes and left to their own devices<sup>2</sup>.

```

main = do {
    toServer <- newChan;
    fromServer <- newChan;
    forkIO(server(readChan toServer) (writeChan fromServer) ((+) 4) );
    forkIO(client ( (getFloat getLine)) (print) (writeChan toServer)
              (readChan fromServer));
    return ()
}

```

#### 4. Towards Composable Connectors

As illustrated above we want to be able to create new connectors by composing existing connection functions together. Our first look into the problem has led us to the notion of higher order connectors. A higher order connector takes other connector functions as arguments and composes them, returning a new connection. The focus in this paper is on simple examples for the purpose of demonstration; however, the concepts generalize for more useful applications such as encryption, compression, or logging connectors. With that in mind let's look at one of the arguments to the client component of the previous example: `(getFloat getLine)`.

```

getFloat getString = do {
    string <- getString;
    return (read string)
}

```

`getLine` is a primitive provided by the Haskell libraries. It provides us with a connector to the console for reading strings. What we want however is a connector to the console that yields floats. Using the higher order connecting function `getFloat` we are able to convert any connection producing a string into a float result. Note that `getFloat` does not depend on how the string result was obtained. It is completely unaware that the function `getLine` exists and the argument to `getFloat` may consist of any composition of other connection functions that give back a string. Thus, to filter a connection we can just snap a filter connector to other communication mechanisms. This idea generalizes into the `filterCon`.

```

filterCon oldCon filterFun = do {
    input <- oldCon;
    return (filterFun input)
}

```

`filterCon` takes as arguments an old connector and a filter function. It describes the sequencing of computations that we would like to use for our new connector<sup>3</sup>. This filter connector is the first step in realizing composable components for software architectures.

#### 5. Event-Service Example

---

<sup>2</sup> We could enhance the readability of these architectural descriptions by using named arguments. This would also allow flexibility in the reordering of carried arguments. Such features are present in the language Ocaml.

<sup>3</sup> Monad [11] enthusiasts will note the similarity between `filterCon` and the IO bind operation. You can write `oldCon >>= return.filterFun` instead of defining `filterCon`.

Sometimes we need to combine more than one connection function with a higher order connector. This will be seen in an event service [2] example. An event service allows other components to register interest in the certain event notifications by providing callback functions that execute when a notification is received. In the `eventService` component a new notice is first received on the `getNotice` connection and then a list of current subscribers is taken from `getSubs`. This list consists of callback functions provided by the subscribers. Each callback can be guarded using pattern matching to request interest in only certain event types<sup>4</sup>. Using a functional map<sup>5</sup> we evaluate all the callbacks on the new notice and those that pass the pattern matching are executed.

```
eventService getNotice getSubs = do {
  notice <- getNotice;
  callbacks <- getSubs;
  mapM (flip id notice) callbacks;
  eventService getNotice getSubs;
  return ()
}
```

In order for a subscriber to communicate with the event service it needs some way of appending it's callback function onto the current list<sup>6</sup>. This is done with the `eventSubscribe` connector.

```
eventSubscribe getCallbacks putCallbacks newSub = do {
  callbacks <- getCallbacks;
  putCallbacks (newSub:callbacks);
  return ()
}
```

This higher-order connector carries two connectors as arguments before becoming a connector that takes in one argument. The responsibility of binding the final `newSub` argument is left for the subscriber and will vary for each call to `eventSubscribe`. The first two arguments, on the other hand, are bound only once. `getCallbacks` retrieves the current callback list, which might be stored in shared memory, in some buffer, or out on the network (we are not interested). It then appends the new subscription callback and uses `putCallbacks` to store the new list. Notice how it is now the responsibility of the programmer constructing the system to ensure some coherence between `getCallbacks` and `putCallbacks`. This particular connector construction does not enforce them to refer to the same location.

## 6. Related Work

There are many different ways to view the problem of component composition; these can be seen in the variety of approaches. Software architecture research has emphasized on the abstract modeling of systems. Tools based around different ADL's are used to aid in analysis [1], to provide simulations [7], or generate implementations [9]. There are even ADL's devoted to particular architectures [10]. Our work is aimed at moving software architecture ideas closer to the programming language level so that real components can be built from the ground up as an alternative to being partially synthesized from ADLs and partly coded up in conventional languages. Our belief is that statically checked, polymorphic languages provide powerful facilities for modeling architectures, and that the powerful type checking and optimizations built into the language can be leveraged. In addition we feel that by modeling architectures in a lazy language (force-feeding, as it were, strict interactions into a pure language) we (almost paradoxically) make the modeling of interactions a more explicit aspect.

The flexible packaging work done by Deline [3] proposes a separation between the ware, that which encapsulates the component's functionality, and the packager, which manages the details of interaction. This is achieved by writing specifications from which code for the components is generated.

---

<sup>4</sup> We have left out the detail of the pattern matching. It is similar to the idea used by JEDI [2] and the pattern matching provided by Haskell makes it easy to implement.

<sup>5</sup> `mapM` applies a function to all the elements of a list and returns all the results in a new list. The expression `flip id` is used because we are applying one argument to a list of functions instead of the other way.

<sup>6</sup> `:` is the list concatenation operator.

Code generation is also the preferred technique for the aspect-oriented languages [6]. We have tried to provide this separation without the use of additional software tools.

Design patterns, specifically, the Decorator [5] allows flexible, dynamic composition of features. However, if a Decorator pattern were used as a basis for compositional connectors, and implemented in a language without powerful higher-order type system, one would have to resort to basic connectors that transmit a general object (such as `java.lang.Object`) and thus be forced to resort to typecasting. The advantages of static type-checking would no longer be available.

Another effort towards composable systems is by middleware developers. The benefits of implementing systems as CORBA, COM, or JavaBeans components have been widely reported. Typically however one is forced into a particular implementation of component connection. Indeed, each of these schemes can be viewed as a specific kind of connector, in contrast to the flexible connector composition mechanism we seek.

## 7. Conclusion

We have presented a model of programming that uses separation of interaction behavior and functionality to enhance reuse. The goal of providing composable connectors is achieved through the notion of higher-order connectors. The connector functions that are used by the components in this paper provide a uniform model for accessing more primitive connector types such as shared memory or asynchronous channels. Since we chose a functional programming environment our view of components and connectors is inherently functional in nature. It is expected that readers will have differing views on whether this is an advantage or disadvantage. Some object-oriented approaches place components as objects with a well-defined interface. However, interfaces provide for communication only through procedure calls, which is a special case of a connector. Our approach is to parameterize components by all possible mechanisms by which the component can communicate. The tools associated with more mature architectural design frameworks provide useful information through a variety of analysis techniques. We have not addressed many of the benefits that these tools provide. The initial research goal of this project is to experiment with those aspects of component-based programming that are available at the language level. Therefore we have to try to avoid writing libraries or defining new abstract data types. In the future it is our intent to demonstrate larger examples using our programming model and develop stronger notions of architectural families.

## References.

- [1] Robert Allen and David Garlan. "Formalizing Architectural Connection". In *Proc. of the 16<sup>th</sup> International Conf. on Software Engineering*, May 1994.
- [2] A. Carzaniga, E. Di Nitto, D. S. Rosenbloom and A. L. Wolf. Issues in Supporting Event-based Architectural Styles. 3rd International Software Architecture Workshop (ISAW3), Orlando FL, 1998.
- [3] Robert Deline. "Avoiding Packaging Mismatch with Flexible Packaging". In *Proc. International Conf. on Software Engineering*, 1999.
- [4] Paul Hudak, John Peterson, Joseph Fasel. "A Gentle Introduction to Haskell". <http://www.haskell.org/tutorial/>
- [5] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. "Design Patterns: Elements of Reusable Object-Oriented Software". Addison-Wesley Longman Inc. 1995.
- [6] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin. Aspect-Oriented Programming. In *ECOOP'97 proceedings*. Finland.
- [7] David C. Luckham. "[Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events](#)". DIMACS Partial Order Methods Workshop IV, Princeton University, July 1996
- [8] Mary Shaw. "Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status". In D.A. Lamb (ed) *Studies of Software Design, Proceedings of a 1993 Workshop*, Lecture Notes in Computer Science No 1078, Springer-Verlag 1996, pp. 17-32.

[9] Mary Shaw, Robert Deline and Gregory Zelesnik. "Abstractions and Implementations for Architectural Connections". In *Proc. International Conf. On Configurable Distributed Systems*. 1996.

[10] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead and J. E. Robbins. A component- and message-based architectural style for GUI software. *IEEE Transactions on Software Engineering*, 1996, 22(6), pp. 390-406.

[11] Philip Wadler. In M. Broy, editor, *Marktoberdorf Summer School on Program Design Calculi*, Springer Verlag, NATO ASI Series F: Computer and systems sciences, Volume 118, August 1992. Also in J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, Springer Verlag, LNCS 925, 1995.