Implementing Software Architectures in Standard ML*

Position Paper

Edoardo Biagioni Robert Harper Peter Lee

Carnegie Mellon University
November, 1994

1 Introduction

Current work on software architecture tends to be of a theoretical nature. By "theoretical" we do not mean to say that the contributions are overtly "formal" or "mathematical"—indeed, most of the ideas are described only in informal terms—but that they propose structuring techniques at a conceptual level rather than as actual running code in a software system.

Our position is that software architectures and system structuring techniques do not have to be described only in theoretical terms. We hold that modern ideas in typed programming languages provide a concrete approach to architectures and composable systems. As a specific example, we present a suite of high-performance network communication protocols, whose architecture is realized directly in an extension of the Standard ML (SML) programming language [5]. Because of the complexity of communications protocols, and also because of the stiff performance requirements, the architecture of the system is necessarily quite sophisticated, involving structuring techniques such as layering of "micro-" and "virtual" protocols [6] and "upcalls" [3]. To express the system architecture, we make essential use of the modules sublanguage of Standard ML, which provides mechanisms such as specification inheritance and instantiation of generic modules. The end result is at once a high level specification of the architecture of high-speed layered network protocols *and* a practical implementation of a communication system.

The main benefit of this approach is that a compiler guarantees the correspondence between the architecture and the running program. This leads

*This research was sponsored by the Defense Advanced Research Projects Agency, CSTO, under the title "The Fox Project: Advanced Development of Systems Software", ARPA Order No. 8313, issued by ESD/AVS under Contract No. F196228-91-C-0168.

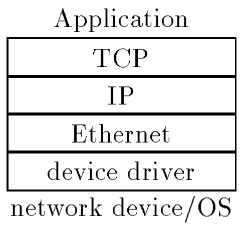

Figure 1: A Standard Protocol Stacks.

to further benefits such as safe and convenient composition of modules and therefore a high degree of reusability of components and reconfigurability of the system. In the case of a network implementation we are able to produce custom protocols quickly and reliably, thereby demonstrating concretely the advantages of a particular software architecture. In addition, this approach allows a smooth transition from one or more prototype (or proof-of-concept) implementations to a production system. By eliminating the distinction between the architecture and the software system, the process of developing a software architecture becomes the most important task in the software development process. This compares favorably to the conventional approach used in many software projects where the development or use of a software architecture is often viewed as a separate, independent, and possibly nonessential task. In short, the programming language helps to bring the matter of software architecture into the spotlight.

The remainder of this paper expands on these claims. In Section 2 we show how SML functors can be used to provide safe and conceptually high-level module composition. Section 3 then describes our use of SML signatures to realize software architectures as valid SML program fragments. All of the examples throughout the paper are taken from actual running programs available *via* anonymous ftp.

2 Architecture of a Network Communication System

In the classical architecture of a network communication system, independent protocols are layered on top of each other, forming a so-called "protocol stack" [4]. Figure 1 depicts the major layers of the widely used TCP/IP protocol stack. Each layer of the protocol stack essentially implements a virtual network capable of sending and receiving data. The functionality of one layer is used by the next layer up to implement a higher-level protocol. For example, in Figure 1, the Ethernet protocol provides low-level transmission of messages on a local area network. This is in turn the basic functionality on which the IP protocol can be specified. IP then provides routing of messages through the Internet.

The implementation of TCP/IP in Standard ML directly reflects this architecture. Each layer in the protocol stack is described by a *signature* which describes the interface presented by the protocol. A protocol signature consists of a collection of types and values (including not just ordinary values such as integers, but also higher-order functions, exceptions, and continuations). Interaction between layers in the protocol stack is mediated by a conformance check, known as *signature matching*, which ensures that an implementation of a protocol provides the types and values specified in the protocol signature. The implementation is free to provide additional functionality beyond what is required by the interface; access to these additional features is provided through enrichments to the basic interface of the protocol. (See Section 3 for more details on the use of signatures in the TCP/IP implementation.)

Each layer in the protocol stack is implemented by a *functor*, a function that builds the implementation of a layer from the implementation of the layer beneath it. The assumptions about the underlying layer are expressed in the signature of the functor parameter; only those features mentioned in the parameter signature are available to the implementation of that layer. The implementation of a layer is obtained by *instantiating* the functor by supplying it with the implementation of the underlying layer. The arguments and results of functors are called *structures*. Conformity between layers is assured by signature matching — the implementation of the lower layer must provide at least the functionality stated in the parameter to the functor. Functor instantiation is akin to dynamic linking, but it is important to recognize that the result of instantiation may involve arbitrary computation, not merely template instantiation.

As a specific example, our network system defines a functor for the TCP protocol:

```
functor Tcp_Protocol (structure Ip: IP_PROTOCOL
                      val checksum: bool) : TCP_PROTOCOL = ...
```

(The code for this functor, which would appear on the right-hand side of the equal sign, has been omitted.) The functor `Tcp_Protocol` implements the TCP protocol as a function of an implementation of the IP protocol. The requirement that TCP be implemented atop IP is expressed by the signature `IP_PROTOCOL` ascribed to the argument of `Tcp_Protocol`. In addition to an IP protocol, `Tcp_Protocol` is also parameterized by a boolean value indicating whether or not headers checksums should be computed. Once provided with an IP protocol and a boolean value, the functor `Tcp_Protocol` computes a new implementation of the TCP protocol; this is indicated by the signature `TCP_PROTOCOL` ascribed to the result of the functor.

The complete TCP/IP protocol stack can be implemented in Standard ML as a collection of functors, each of which implements a layer in the stack. Here are a few such functors, in addition to `Tcp_Protocol` given above:

```
functor Dev_Protocol (): DEVICE_PROTOCOL = ...
functor Eth_Protocol (structure Dev: DEVICE_PROTOCOL)
                         : LINK_PROTOCOL = ...
functor Arp_Protocol (structure Transport: LINK_PROTOCOL)
                         : ADDRESS_RESOLUTION = ...
functor Ip_Protocol (structure Arp: ADDRESS_RESOLUTION)
                         : FULL_IP_PROTOCOL = ...
functor Icmp_Protocol (structure Ip: FULL_IP_PROTOCOL)
                         : IP_PROTOCOL = ...
functor Udp_Protocol (structure Ip: IP_PROTOCOL)
                         : UDP_PROTOCOL = ...
```

To build a standard TCP/IP protocol stack, the following sequence of functor instantiations is performed:

```
structure Dev  = Dev_Protocol ()
structure Eth  = Eth_Protocol (structure Dev = Dev)
structure Arp  = Arp_Protocol (structure Transport = Dev)
structure Ip   = Ip_Protocol (structure Arp = Arp)
structure Icmp = Icmp_Protocol (structure Ip = Ip)
structure Tcp  = Tcp_Protocol (structure Ip = Icmp
                               val checksum = true)
```

Non-standard protocol stacks are often useful. For example, for testing purposes it is useful to simulate the physical network in software. This can be achieved by building a "test wire" functor that creates a pseudo-device driver (matching the signature `DEVICE_PROTOCOL`) upon which the rest of the stack can be built.

```
functor Test_Wire (): DEVICE_PROTOCOL = ...
structure Tst = Test_Wire ()
structure Eth  = Eth_Protocol (structure Dev = Tst)
```

Another example of a non-standard protocol stack is to build TCP directly on top of the Ethernet protocol, bypassing the IP layer. This would be useful in situations where reliable transmission of arbitrary length messages is needed, but routing through the Internet is not necessary (*i.e.*, all communication will be conducted on a local area network). A naïve attempt to build such a protocol stack might proceed as follows:

```
structure Tcp  = Tcp_Protocol (structure Ip = Arp
                               val checksum = false)
```

However, this neglects the requirement that TCP can only be built on an implementation of IP, and not directly over ARP. Since ARP fails to satisfy these requirements the above attempt is rejected by the compiler during signature matching, providing an explicit indication of which aspects of IP are not provided by ARP.

To build TCP directly on Ethernet, we must define a "micro-protocol" that provides the minimal IP functionality required by TCP from Arp.

```
functor Ip_from_Arp (structure Arp: ADDRESS_RESOLUTION)
                         : IP_PROTOCOL = ...
```

Using this functor we can build the intended non-standard protocol stack:

```
structure Mini_Ip  = Ip_from_Arp (structure Arp = Arp)
structure Tcp  = Tcp_Protocol (structure Ip = Mini_Ip
                               val checksum = false)
```

Checksumming is suspended because in the absence of internet routing, we can rely on the Ethernet checksum to ensure the integrity of TCP headers.

3 Expressing Abstractions Using Signatures

The key to composability in a layered system architecture is to arrange for each layer to satisfy a standard interface. If every component satisfies the standard interface, and only requires that the layer below it satisfies this interface, then the layers may be composed arbitrarily.

In network communication systems, the use of this composable layered architecture was pioneered by the x-kernel project [6]. One of the main developments of the x-kernel is the standard interface for protocols, which is called the *metaprotocol*. Of course, in the x-kernel the metaprotocol is only an informal structuring concept which must be translated into working C code. In contrast, we are able to express the metaprotocol in formal terms as the SML signature called `PROTOCOL`, which is shown in Figure 2.

Specific protocols are defined as enrichments of the generic `PROTOCOL` signature. For example, Figure 3 shows an excerpt of the `IP_PROTOCOL` signature. In this signature, the use of the `include` and `sharing` declarations specifies that `IP_PROTOCOL` enriches and refines the generic `PROTOCOL` signature. Thus, any implementation of `IP_PROTOCOL` automatically matches also `PROTOCOL`; essentially, this specifies a subtyping relationship between the specific and generic protocol. (A thorough discussion of this signature is given in [2].)

Referring back to the protocol functors from Section 2, we see that each functor provides an implementation of a specific protocol signature, each of which is a refinement of the `PROTOCOL` signature. Further refinements are also possible and useful. For example, the ICMP protocol requires access to aspects of the IP protocol that are not used by the transport protocols. These aspects are exposed in the signature `FULL_IP_PROTOCOL` signature, which is defined as an enrichment of the `IP_PROTOCOL` signature.

```
signature PROTOCOL = sig
  eqtype address
  eqtype address_pattern
  eqtype connection
  type incoming_message
  type outgoing_message

  val initialize: unit -> int
  val finalize: unit -> int

  val active_open:
      address * (connection -> incoming_message -> unit) -> connection
  val passive_open:
      address_pattern * (connection -> incoming_message -> unit)
         -> (connection * address)
  val close: connection -> unit
  val abort: connection -> unit

  val send: connection -> outgoing_message -> unit

  exception Initialization_Failed of string
  exception Bad_Address of address * string
  exception Open_Failed of address * string

 ...
end (* sig *)
```

Figure 2: Excerpt of the PROTOCOL signature.

```
signature IP_PROTOCOL = sig
  type ip_number
  type ip_protocol
  datatype ip_address =
      Address of ip: ip_number, proto: ip_protocol
 ...
  include PROTOCOL
   sharing type address = ip_address
       and type incoming_message = ip_incoming
 ...
end
```

Figure 3: Excerpt of the IP_PROTOCOL signature.

The example of the previous section involving the layering of TCP over ARP (thereby bypassing ICMP and IP) demonstrated the usefulness of signature matching. Because both the `ADDRESS_RESOLUTION` and `IP_PROTOCOL` signatures inherit from `PROTOCOL`, the `Glue` functor that allows layering of TCP over ARP can be small; if the two signatures were dramatically different, the `Glue` functor would have to be correspondingly large and complicated.

Unlike modern protocols, the TCP/IP suite is not fully modular. In particular the TCP protocol definition assumes that it is built over IP. This necessitates the use of "glueing" functors between layers in *ad hoc* protocols. More modern protocol design would dictate that each layer depend only on a generic protocol beneath it, which would have the effect of eliminating the need for the glue functors.

4 Conclusions

We have shown that Standard ML allows the architecture of a system to be expressed in concrete terms. This in turn ensures that the implementation will be faithful to the architecture, and that this correspondence will continue to hold as the architecture and implementation evolve.

In our experience, the development of a software architecture into the software product can be maintained by the type system of the programming language in a way that is both semantically sound and eminently practical. The soundness is guaranteed by the properties of the type system (which are formally defined), and the practicality is demonstrated by the fact that our network system achieves competitive throughput and latency [2] over an Ethernet.

The usefulness of conventional approaches to software architectures [1] for software development and maintenance is questionable because these architectures require the maintenance of two separate and unrelated specifications, the architecture itself and the source code of the program. In contrast, software architectures expressed as SML signatures and functor signatures are integrated with the source code of the program and necessarily maintained together, since the compiler automatically requires and checks the conformance of the source code with the architecture. As a result the software architecture must be used during maintenance as well as for the initial design and implementation. Our experience is that evolution and maintenance of the system often involves a redesign of the architecture to fit the updated specification, and continues with the corresponding update of the source code. In such a process, the compiler becomes essentially a specification checker, ideal for pointing out specific places in the implementation that require updating.

Software architectures do not have to be theoretical. A practical application of principles from modern typed programming languages such as Standard ML has the effect of enhancing the rigor and therefore the value of the work in software architectures, as well as software engineering in general.

References

[1] Robert Allen and David Garlan. Formal connectors. Technical Report CMU-CS-94-115, Carnegie Mellon University, March 1994.

[2] Edoardo Biagioni, Robert Harper, Peter Lee, and Brian Milnes. Signatures for a network protocol stack – a systems application of Standard ML. In *1994 ACM Conference on Lisp and Functional Programming*, Orlando, FGV, June 27-29 1994.

[3] David D. Clark. The structuring of systems using upcalls. In *Proceedings of the 10th Symposium on Operating System Principles*, pages 171–180, Orcas Island, Wash., December 1985.

[4] J. D. Day and H. Zimmerman. The OSI reference model. *Proceedings of the IEEE*, 71(12):1334–1340, 1983.

[5] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[6] Sean W. O'Malley and Larry L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2), May 1992.

[7] RFC 1057. RPC: Remote Procedure Call protocol specification, version 2. Sun Microsystems, Inc.

A Example: Pipes

Below we present an implementation in SML of an example mentioned in [1]. We present this in two parts, the first being the signatures, and then functors giving an implementation. The signatures would be obtained by a direct transcription of the kinds of (unimplemented) specifications given by Allen and Garlan, and thus constitute a definition of the "software architecture."

Informally, the program splits the input stream into two streams, capitalizes one of the streams, changes the other to lowercase, and produces a merge of the resulting streams.

Software Architecture

The following are signatures for a basic stream type with essential operations and for specialized stream operations such as split, merge, and filter, followed by the functor signature for the main functor. The functor signature clearly expresses the architecture of the program and the connections between its modules: for example, the sharing constraints make it clear that all modules operate on the same kind of stream and that the input and output types of the two filters must be the same.

```
signature STREAM = sig
  datatype 'a stream = End
                     | Next of (unit -> ('a * 'a stream))
  val map: ('a -> 'b) -> 'a stream -> 'b stream
  val force: 'a stream -> 'a list
 end (* sig *)

signature SPLIT = sig
  type 'a stream
  val split: 'a stream -> 'a stream * 'a stream
 end (* sig *)

signature MERGE = sig
  type 'a stream
  val merge: 'a stream * 'a stream -> 'a stream
 end (* sig *)

signature FILTER = sig
  type 'a stream
  type input
  type output
  val filter: input stream -> output stream
 end (* sig *)
```

```
funsig MAIN (structure Lower: FILTER
             structure Upper: FILTER
             structure Split: SPLIT
             structure Merge: MERGE
             structure Stream: STREAM
               sharing type Lower.stream = Upper.stream =
                            Split.stream = Merge.stream =
                            Stream.stream
                   and type Lower.input = Upper.input
                   and type Lower.output = Upper.output) =
               sig
                type 'a stream
                val capitalize: char stream -> char stream
               end
```

Implementation

Each functor implements one or more of the modules of the final program. The functors are then instantiated (linked) to produce structures, which are executable. The `Filter` functor is instantiated twice to produce the two structures `Upper` and `Lower`.

```
functor Main (structure Lower: FILTER
              structure Upper: FILTER
              structure Split: SPLIT
              structure Merge: MERGE
              structure Stream: STREAM
                sharing type Lower.stream = Upper.stream =
                             Split.stream = Merge.stream =
                             Stream.stream
                    and type Lower.input = Upper.input
                    and type Lower.output = Upper.output) =
 struct
  fun capitalize stream =
       let val (l, u) = Split.split stream
       in Merge.merge (Lower.filter l, Upper.filter u) end
 end (* struct *)
```

```
functor Stream (): STREAM =
 struct
  datatype 'a stream = End
                     | Next of (unit -> ('a * 'a stream))
  fun map f End = End
    | map f (Next thunk) =
       let val (element, rest) = thunk ()
           fun suspension () = (f element, map f rest)
       in Next suspension end
  fun force End = []
    | force (Next thunk) =
       let val (element, rest) = thunk ()
       in element :: force rest end
 end (* struct *)

functor Even_Split (structure Stream: STREAM): SPLIT =
 struct
  type 'a stream = 'a Stream.stream
  fun skip1 Stream.End = Stream.End
    | skip1 (Stream.Next thunk) =
       let val (_, rest) = thunk ()
       in next1 rest end
  and next1 Stream.End = Stream.End
    | next1 (Stream.Next thunk) =
       let val (value, rest) = thunk ()
           fun suspension () = (value, skip1 rest)
       in Stream.Next suspension end
  fun split stream = (next1 stream, skip1 stream)
 end (* struct *)

functor Even_Merge (structure Stream: STREAM): MERGE =
 struct
  type 'a stream = 'a Stream.stream
  fun merge (Stream.End, second) = second
    | merge (Stream.Next thunk, second) =
      let fun first () =
               let val (el, rest) = thunk ()
               in (el, merge (second, rest)) end
      in Stream.Next first end
 end (* struct *)
```

```
functor Filter (structure Stream: STREAM
                type input
                type output
                val filter: input -> output): FILTER =
 struct
  type 'a stream = 'a Stream.stream
  type input = input
  type output = output
  val filter = Stream.map filter
 end (* struct *)

structure Basic_Stream = Stream ()

structure Split =
    Even_Split (structure Stream = Basic_Stream)

structure Upper = Filter (structure Stream = Basic_Stream
                          type input = string
                          type output = string
                          val filter = CType.toUpper)

structure Lower = Filter (structure Stream = Basic_Stream
                          type input = string
                          type output = string
                          val filter = CType.toLower)

structure Merge =
    Even_Merge (structure Stream = Basic_Stream)

structure Main = Main (structure Lower = Lower
                       structure Upper = Upper
                       structure Split = Split
                       structure Merge = Merge
                       structure Stream = Basic_Stream)
```