(1)

Standard ML Signatures for a Protocol Stack

Edoardo Biagioni Robert Harper Peter Lee

October 1993

CMU-CS-93-170

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Also published as Fox Memorandum CMU-CS-FOX-93-01

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. F19628-91-C-0168.

Keywords: Standard ML, computer networks, modules, signatures, types

Abstract

This paper describes the design of a protocol stack implemented in Standard ML. Standard ML's signatures are a language construct which can be used to specify or constrain the interface of a module. The design includes both a generic signature which generalizes all the protocol modules, and individual signatures specific to each protocol module. The specific signatures all inherit from the generic signature. The implementation of each protocol is parametrized, so protocols can be composed into custom protocol stacks. The parameter to each protocol is constrained only by the generic signature, and this lets any protocol instance satisfying a specific signature be used as the parameter to any other protocol. As a result, the design and implementation are highly modular, and syntactic compatibility between modules is checked by the compiler. To provide some context for the discussion of the signatures, some of the details of the implementation are also presented.

Authors' electronic mail addresses are:
`Edoardo.Biagioni@cs.cmu.edu`, `Robert.Harper@cs.cmu.edu`, and `Peter.Lee@cs.cmu.edu`,

1 Introduction

Network communication protocols are designed to be reliable and efficient, transmitting large volumes of data with high throughput and low volumes of data with low delay, and to correctly handle errors introduced by the network. The implementation of network protocols involves programming at both a low level, for example the hardware and interrupts of the machine, and at a high level, for example error detection and retransmission of lost packets. The complexity of the protocols and the complexity and concurrent nature of the system both contribute to making the implementation of network protocols a very complex task.

Careful design and structuring techniques are extremely important in programming complex systems. Besides increasing the reliability of the system by reducing the number of errors and allowing them to be detected earlier, good structure also reduces maintenance costs. Structured design also helps reduce the occurrence of those programming errors that are especially hard to detect and correct, particularly errors in defining and implementing the appropriate performance tradeoffs between latency, bandwidth, and cost that are required of network protocols [7].

In spite of the use of structured programming techniques, the organization of many implementations of networking protocols is still quite poor. There are many reasons for this, but we see the use of the C programming language as a major factor. C, and its descendant C++, are popular in part because they make the programming of low-level operations and data manipulation easy and somewhat portable between different architectures, and because the performance of the compiled code is easily predicted. C however offers only minimal facilities for expressing module structure and completely lacks facilities for enforcing abstraction boundaries. Furthermore, much of the efficiency of C programs is obtained by the use of unsafe low-level operations such as pointer arithmetic. The combination of missing support for abstraction with the presence of unsafe operations makes it possible for many kinds of hard-to-find errors to occur, including type errors, allocation errors, and bad pointers. These errors are seen much less frequently, if at all, when using more advanced languages.

This paper reports on initial progress in implementing network communications in an advanced programming language that offers considerably more support for structured programming, type checking, and enforcement of abstractions than is provided by C or C++. We present some details of an implementation of the TCP/IP suite of network protocols, written in a programming language based on Standard ML (SML) [4]. This pilot study is part of the Fox project which, using modern language design based on formal semantics, seeks better understanding of language design issues and the development of language design principles for real-world programming [2].

The remainder of this paper first summarizes the terminology and concepts used to describe the design and implementation of network communications software. The next section recasts these concepts in terms of ML language structures, followed by a presentation of our system, particularly the high-level design choices we have made. The paper then presents some ideas that can only be applied to our design because we have a structured design in an advanced language, including automatic program transformations to produce more efficient code. We conclude by summarizing both the related work in the field and our own results.

2 Network Communications Systems

The terminology used for network communications systems often depends on the specific type of network communications system being discussed, and is not always used consistently. To fix terminology we give definitions of the terms *layer*, *protocol*, *protocol stack*, and *instance*, which will be used in the remainder of this paper.

A network communications system can be viewed as an implementation of an abstract machine. Application programs (*clients*) communicate through an interface represented by this abstract machine. A client may use this interface as a building block in building further abstractions. The technique of using one abstraction in defining another abstraction is so useful and common in networking that it has been given a name, *layering*. The term *layer* refers to a network abstraction.

In keeping with conventional usage, we use *protocol* to refer to an implementation of a layer, that is, an algorithm that realizes a network abstraction. A *protocol stack* is an implementation of a communication system that is described in terms of a set of protocols, in which each protocol implements the corresponding abstraction in terms of another abstraction upon which it is "layered". If a protocol is layered on top of multiple lower-level abstractions, or conversely if an abstraction supports multiple higher-level protocols, the protocol stack is more properly known as a *protocol graph*, though we do not use the term in this paper.

We understand an *instance* of a protocol to be the execution of a program implementing a particular protocol on a particular system; there could potentially be several instances of the same protocol within the same system at the same time. In practice, the state of an instance is sufficient for representing the instance. Finally, the *peer* of a given instance `A` is the instance `B` with which `A` is communicating; whenever `A` communicates with itself, `A` is its own peer.

We say that an implementation of a network communication system is *well-structured* if:

- it has a distinct protocol for each layer
- all the peers of each instance of a protocol are other instances of the same protocol,
- the layered structure is realized using modules and explicit interfaces.

The standard example is the TCP/IP protocol stack. The IP layer supports unreliable transmission of data. The TCP layer supports reliable delivery of "correct" data. TCP is built on top of IP. IP itself is layered on top of hardware-specific protocols such as Ethernet or ATM. Data handed by TCP to IP may or may not be delivered to the intended destination, may be corrupted, or may be delivered multiple times. TCP attaches a unique identifier and check bits to each message sent, and exchanges messages with its peers to retransmit any messages that have been lost or corrupted.

It is a simple observation that many implementations of network communication systems are not well-structured according to our definition. One of the explanations we see for this situation is that network software is usually implemented in C, and C fails to support and enforce the concept of modules with clean interfaces, as it fails to support and enforce many of the typing mechanisms that reduce the occurrences of type errors.

Another explanation for this lack of structure is that some optimizations are only possible if layer boundaries are violated. For example, a particularly clean, but inefficient, implementation of a layered protocol would copy the data and perform a context switch every time the data had to cross the boundary between two layers. A more optimized implementation would only copy the data when absolutely necessary, and avoid context switches between layers by using the same thread (where possible) to execute the code for all the layers. Taking this to an extreme means removing all the boundaries between layers. This merging of layers is similar to the loop fusion done by Fortran compilers and to the deforestation techniques used by functional language compilers; it reduces intermediate storage and the operations required to store and retrieve the data, and therefore saves time and space. Unlike these compiler techniques, the layers of communication software are typically merged by hand and optimized at the source code level, in a technique called

Integrated Layer Processing or ILP. The disadvantage of using ILP at the source code level is that it completely destroys the structure of the protocol implementation.

Finally, the definition of protocols developed prior to the standardization of the ISO reference model [8] sometimes fails to satisfy our criteria for being well-structured. This is most blatant in the definition of the TCP protocol, which makes use of some fields from the IP header. This means TCP is actually receiving information from a different layer's peer, which violates the layering principle. As a result of this violation, TCP has to be modified before it can be used on top of a protocol other than IP.

One recent and notable exception to these poorly structured network implementations is the software produced by the x-kernel project [5], which has developed implementations of protocol stacks that are well-structured and highly modular. In the x-kernel, all layers provide the same set of procedures each with the same set of arguments. As a result, a protocol need not know which other layer it is running on top of, and protocols can be layered almost arbitrarily to build custom protocol stacks that can provide the different performance tradeoffs required by different applications.

The x-kernel does not use ILP. As pointed out by Clark and others [1], many of the costs of traditional protocol implementations are due to inefficient implementation of many of the operations required for protocol processing, not to layered structure. By concentrating on avoiding unnecessary context switches and data copying, the x-kernel achieves efficient data transmission and reception while retaining a clean design whose structure is the same as that of the protocol stack [5]. This shows that while integrated layer processing may provide improvements in performance, a well-structured design can achieve good performance even without ILP.

Our work has been inspired by and has taken many ideas from the x-kernel's structured design. Unlike the x-kernel, we have developed formal interfaces for our protocols and expressed them in SML. The next two sections explain the general principles we followed in developing the formal interfaces and the details of the interfaces, and explain how the interfaces fit in our implementation of the TCP/IP protocol suite.

3 Structuring Communication Systems in Standard ML

We have developed a well-structured implementation of a simplified version of the TCP/IP protocol suite[1] in a language based on Standard ML. Our implementation is known as the *FoxNet*. This section gives an overview of the FoxNet, omitting details in the interest of giving a feel for the overall structure and how it is realized in SML.

In our overview we present SML *signatures*, *structures*, and *functors*. An SML signature specifies an interface to a module; a module may implement multiple, different signatures, of which exactly one is the most specific signature. Conversely, a signature is a collection of conditions a candidate implementation of the interface must satisfy. An implementation of an interface specified by a signature is an SML structure; the structure must have a concrete type for every type specified by the signature and a value for every value specified by the signature. A functor is a parametrized structure; the parameters of a functor may themselves be structures. A functor may be instantiated multiple times, with the same or different parameters, to produce different structures.

[1]Our simplified implementation would be a full TCP/IP if it implemented TCP and IP *options* and IP *fragmentation*. The TCP and IP standards specify not only headers that accompany every packet, but also optional sub-headers known as options. Since options are not required for normal operation, and since most implementations never use them, we have chosen not to support them for now. For the same reasons we do not yet support reassembly of IP fragments.

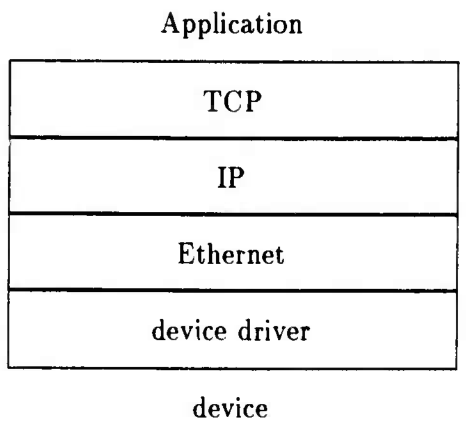

Figure 1: Standard TCP/IP Protocol Stack

3.1 Protocol Building Blocks

Our protocol stack is built by instantiating functors. Each functor takes as an argument the instantiated functor(s) implementing the layers below it. For example, the typical stack containing TCP, IP, Ethernet, and a device interface is built as follows.

```
functor Tcp (structure Lower: PROTOCOL): TCP_PROTOCOL = ...
functor Ip (structure Lower: PROTOCOL): IP_PROTOCOL = ...
functor Ethernet (structure Lower: PROTOCOL): ETHERNET_PROTOCOL = ...
functor Ethernet_Device (): DEVICE_PROTOCOL = ...

structure Device_Instance = Ethernet_Device ()
structure Ethernet_Instance = Ethernet (Device_Instance)
structure Ip_Instance = Ip (Ethernet_Instance)
structure Tcp_Instance = Tcp (Ip_Instance)

val connection = Tcp_Instance.active_open (...)
```

The protocol stack built by these statements is shown in Figure 1.

Each of the protocol functors is parametrized by a structure which must satisfy a particular signature called PROTOCOL. This parameter implements the lower layer protocol. At the same time, each individual functor has an interface defined by a different signature, such as IP_PROTOCOL or TCP_PROTOCOL. These *specific* signatures each define the interface for one particular protocol. The PROTOCOL signature is *generic* and defines the minimal set of features that every protocol should satisfy. By only using PROTOCOL to constrain the parameter, the various functors agree to use only this minimal set of features from whatever protocol is provided as a parameter.

All of the specific signatures are derived from the generic signature:

```
signature PROTOCOL = sig ... end
signature TCP_PROTOCOL = sig
  ...
  include PROTOCOL
    sharing type ...
  ...
end
```

By including the generic signature in their definition, each of the specific signatures inherits all the types and values declared in the PROTOCOL signature; sharing constraints bind types declared in PROTOCOL to types that are declared in the specialized signature. Because the specialized signatures inherit all the types and values of PROTOCOL, they are enriched descendants of the same signature. Any functor that satisfies a specialized signature also satisfies the PROTOCOL signature, and any instantiation of such a functor can be used as a parameter to other protocol functors.

Because the specific signatures are derived from the generic signature, and because the compiler strictly enforces adherence to signatures, the design of the generic signature is of central importance and affects the implementation of all the protocols. The generic signature is similar to the concept of abstract data type, and the specific signatures then correspond to concrete data types implementing the abstract data types.

Our PROTOCOL signature is analogous to the x-kernel's *meta-protocol*, which describes the set of operations that are supported by individual protocols.

One of the principles in our design is that all the types in the generic signature are left unspecified, and the types in the specific signatures are more specified and unique for each instance. Another feature of our design is that the generic signature specifies the arguments and return values of every function, so the caller of a function knows what arguments to provide and what results to expect even without knowing which specific signature a protocol satisfies. Finally, the sharing constraints in the specific signature bind the types defined in the generic signature to the local types which the specific signature defines.

To show the modularity of our functors, we can build a non-standard protocol stack with TCP running directly over Ethernet.

```
structure Tcp_Over_Ethernet = Tcp (Ethernet_Instance)
```

This is a special-purpose protocol that reliably sends arbitrary-length packets, but only between hosts that are on the same Ethernet. With such a protoco., it might be desirable to turn off TCP checksums, which are expensive to compute and not as strong as the Ethernet's Cyclic Redundancy Check.[2] By adding to the TCP functor a new parameter that specifies whether TCP should compute checksums, we can specify different behavior for different instances of TCP.

```
functor Tcp (structure Lower: PROTOCOL
             val compute_checksum: bool): TCP_PROTOCOL = ...
structure Tcp_Instance = Tcp (Ip_Instance
                              val compute_checksums = true)
structure Tcp_Over_Ethernet = Tcp (Ethernet_Instance
                                   val compute_checksums = false)
```

[2]That is, the probability of an undetected error is higher when using only the TCP checksum than when using only the Ethernet's CRC.

In practice, each functor has several parameters, as can be seen in Figures 5–7.

With functors, we thus have truly modular building blocks; the "glue" that holds them together are the functors' parameters. This achieves code reuse, one of the goals of software engineering, and makes it easy to build specialized protocol stacks for higher performance without sacrificing modularity and type safety. The example of a non-checksumming TCP over Ethernet describes a protocol stack that is potentially much faster than the regular TCP/IP and therefore very attractive for applications that only need to communicate across a single Ethernet.

3.2 Signature PROTOCOL

In Section 2 we defined a network communications system as an implementation of the "layer" abstraction. The **PROTOCOL** signature, shown in Figure 2, defines a standard interface for such layers. Note that this signature is a closed specification, that is, completely self-contained. This signature is a formal definition of an interface, and syntactic conformance of each protocol implementation with this signature is enforced automatically at compile time.

The types defined in **PROTOCOL** are all unspecified; they must either be bound (by **sharing**) in the specific protocol signatures, or be declared by the functors.

The type **address** is used to identify a peer of the protocol or application layered above the protocol represented by this signature; an **address_pattern** is a specification of a set of possible peers from which a connection request will be accepted; and a **connection** is a handle used to send data and to close or abort a connection. The type of incoming messages is the same across all FoxNet protocols, and bound to the abstract data type **Receive_Packet.T**; this abstract data type provides efficient access to the data and to the message's headers and trailers. The same is true for the **Send_Packet.T** abstract data type, to which the type of incoming messages is bound. The **control** and **info** types are discussed below.

The values specified by **PROTOCOL** are all functions.

A protocol may need to acquire system resources before it can operate. The lowest layer of the FoxNet, for example, needs to acquire Mach ports so it can communicate with the Mach device driver. These resources must be explicitly released once the application no longer needs them, because the resources include hardware devices or operating system resources that will not be released by the SML garbage collector; the **finalize** call releases all resources held by the protocol. The corresponding **initialize** call allocates the resources. Both functions may be called multiple times; resources are only allocated on the first **initialize** and released on the matching **finalize**. The count of unmatched initialize calls is returned by both functions.

A connection can be opened either by **active_open** or **passive_open**; both of these functions take as one of their arguments a packet handler function that will be called when a packet is received. The **connection** value returned by either of the **open** calls can be used to **send** any number of packets, and finally to **close** or **abort** the **connection**. The difference between **close** and **abort** is that the latter will terminate even if the peer is no longer reachable.

There is no **receive** function corresponding to **send**; data is received when the protocol implementation calls the handler that was given as a parameter to one of the open functions. Since the lower layer protocol calls handlers provided by the higher layer, this method of receiving data is known as an *upcall*. Upcalls for receive have the advantages of simplicity, since the protocol implementations do not need to deal with suspending a function call and waking it up again once the data arrives, and of high performance, since in normal cases it is possible to get a packet in at the lowest level of the protocol stack and transfer it up to the top of the stack with no context switches and no buffering. For these reasons, we use upcalls for receive in all our protocols.

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

  val send: connection -> outgoing_message -> unit

  type control
  type info
  val control: control -> unit
  val query: unit -> info

  exception Initialization_Failed of string
  exception Protocol_Not_Initialized of string
  exception Invalid_Connection of connection * address option * string
  exception Bad_Address of address * string
  exception Open_Failed of address * string
  exception Packet_Size of int
 end (* sig *)
```

Figure 2: Signature PROTOCOL

The control and query functions are used to provide custom information or operations for each protocol. For example, IP must be configured to use a specific address as a *default gateway*, whereas TCP needs to be told when a packet has been consumed and more packets can be accepted from the network. Likewise, each protocol has information it can return, information that may be different from that of every other protocol. By binding the generic control and info types to appropriate specific types in the specific signatures, we specify a different set of operations for each protocol. If such an operation is needed by a higher-level protocol, we pass an encapsulation of that function as one of the parameters to the higher-level protocol functor; when building custom protocols, only the encapsulation of the lower-level protocol function as a parameter to the higher-level functor needs to be re-implemented.

All the functions are designed to return to their caller in the course of normal operations. When an unusual event is detected, however, we raise an exception. The protocol signature defines six generic exceptions that may be raised by any protocol implementation. Protocol implementations only raise exceptions from this group, and in fact each function has a specific set of exceptions it may raise.

3.3 A Specific Protocol Signature

As an example of a specific protocol signature we use the signature for the IP protocol, shown in Figure 3. This signature binds the four types address, address_pattern, control, and info to types meaningful for IP, and the two types incoming_message and outgoing_message to the two types that we use for protocols generally.

The IP address and address pattern are declared as record-valued *datatypes*: the datatype declaration makes the types unique for each structure matching this signature, and the records make the structured values self-documenting. We use record-valued datatypes extensively in our signatures.

The IP protocol definition [3] specifies a 4-byte *IP number* which identifies a host, and a 1-byte field which identifies the particular layer above IP to which packets will be given. Our IP address type specifies both values; this combination uniquely identifies the peer of the protocol which uses IP to send data. Note that the types Byte1 and Byte4 are not part of Standard ML; they are extensions we have identified as useful for systems programming.

The IP address pattern always specifies a field to identify the layer above IP, and optionally specifies the remote host by its IP number. This flexibility is used to passively wait for packets for a specific protocol; the packets can be either from a specific remote host, or from any host. Once a packet matching the pattern is received, the passive open completes and the packet is delivered to the specified handler.

The control type offers four control operations. Two allow the enabling and disabling of a specific interface; the local address for the interface must be specified when enabling an interface. The other two control operations provide for the specification of a gateway (also known as *router*) for IP packets that must be forwarded for other networks. The gateway can be set either for all packets that can't otherwise be routed, or specifically for a particular destination address. This control operation can be used when an *ICMP redirect* packet is sent by a gateway currently in use specifying a better gateway for a particular remote IP address.

The info type returns a variety of information used both by the layers above IP and for monitoring the system. In the first category are the maximum packet size and the local address, which are used by TCP and UDP (via functor parameters; Ethernet exports corresponding functions, which can be used for Tcp_Over_Ethernet). The other values returned are useful for monitoring

```
signature IP_PROTOCOL =
 sig
  datatype ip_address =
           Address of {ip: Byte4.U.ubytes, proto: Byte1.U.ubytes}
  datatype ip_address_pattern =
           Pattern of {protocol: Byte1.U.ubytes,
                       source_ip: Byte4.U.ubytes option}

  datatype ip_control =
           Set_Default_Gateway of {gateway: Byte4.U.ubytes}
         | Set_Specific_Gateway of {destination: Byte4.U.ubytes,
                                    gateway: Byte4.U.ubytes}
         | Set_Interface_Address of string * Byte4.U.ubytes
         | Disable_Interface of string
  datatype ip_info =
           Info of {max_packet_size: ip_address -> int,
                    interfaces: (string * Byte4.U.ubytes option),
                    local_address: Byte4.U.ubytes
                                -> (string * Byte4.U.ubytes),
                    packets_sent: int,
                    packets_received: int,
                    packets_discarded: int}

  include PROTOCOL
   sharing type address = ip_address
       and type address_pattern = ip_address_pattern
       and type incoming_message = Receive_Packet.T
       and type outgoing_message = Send_Packet.T
       and type control = ip_control
       and type info = ip_info
 end (* sig *)
```

Figure 3: Signature for the IP Protocol Layer

the system. Note that some of the values returned are functions. In general, it would be either too expensive or outright impossible for every call to **query** to compute all the values which may be of interest. Returning a function is a form of lazy evaluation (call by need) that allows the caller to ask for exactly the data required.

4 Protocol Implementations

4.1 A Sample Protocol Implementation

In the FoxNet, we define a functor for each protocol and instantiate the functors to generate implementations of protocol signatures. Figure 4 shows an example of such an implementation. The purpose of the **Check** functor is to implement the check protocol,[3] which discards packets that have been corrupted in transit. To detect which packets have been corrupted, the protocol adds a two-byte header to every outgoing packet and places in this header the two-byte checksum of the packet's data. The checksum is recomputed by the receiver, and packets with incorrect checksum are discarded. The checksum is computed in 16-bit one's complement arithmetic, and the value of +0 (which is 16 zero-valued bits) is sent as −0 (which is 16 one-valued bits). Any error in the network that causes the packet to be randomly corrupted is likely[4] to cause the checksum to no longer match the data, and the packet will be discarded. Packets set to all zeroes by the network are also discarded.

In the x-kernel, small protocols such as the **Check** protocol and their implementations are referred to as *micro-protocols*.

The implementation of the **send** function is complex in the interest of high performance. The function takes two arguments. Unlike other functions, e.g. **active_open**, in which multiple arguments are specified as a single formal argument which is a tuple, **send** takes two formal arguments (cf. Figure 2). Formally, the function is applied to its arguments one at a time; applying the function to the first argument returns a function that will consume the remainder of the arguments and return the value.

The computations that **send** must perform can be divided into those that do not depend on the packet being sent and those that do. Our implementation, and in fact any optimizing implementation, completes the computations that only depend on the connection argument before returning a function in which only the computations that depend on the packet being sent are performed. The returned function is a special-purpose **send** that is optimized for this connection. In our case the only computation that does not depend on the second argument is the evaluation of a **send** for the lower layer that is optimized for the lower-layer connection; this can be a very substantial optimization if the lower layer's **send** function does substantial computation given only its first argument, and if the function returned by send is called more than once. If the function is called multiple times, we have effectively moved the initial computation out of the loop and amortized its cost over all calls of the returned function; the advantage of doing this as we have is that we have been able to specify this without affecting the program's modular structure, and in fact this functor is a completely modular building block. This kind of optimization is very natural in SML, and would at best be very awkward to program in a traditional imperative language.

In a FoxNet protocol stack, each functor has as one of its parameters a structure representing an unspecified implementation of the next lower level protocol in the stack, and builds the protocol

[3]This is not a standard protocol.

[4]With probability approximately $1 - 2^{-16}$

```
functor Check (structure Lower: PROTOCOL
                sharing Lower.incoming_message = Receive_Packet.T
                    and Lower.outgoing_message = Send_Packet.T): PROTOCOL =
 struct
  datatype address = Address of Lower.address
  datatype address_pattern = Pattern of Lower.address_pattern
  type connection = Connection of Lower.connection
  type incoming_message = Receive_Packet.T
  type outgoing_message = Send_Packet.T

  val initialize = Lower.initialize connection
  val finalize = Lower.abort connection

  fun send_packet lower_send packet =
       let val size = Send_Packet.size packet
           val raw_check = Send_Packet.checksum packet (0, size)
           val checksum = if raw_check = 2u0 then 2uxffff else raw_check
           val extended = Send_Packet.extend_header packet 2
           val final = Send_Packet.set_byte2 extended raw_check
       in lower_send final end
  fun send (Connection connection) = send_packet (Lower.send connection)

  fun receive handler packet =
       let val packet_sum = Receive_Packet.byte2 packet 0
           val data = Receive_Packet.hide_header (packet, 2, "check")
           val size = Receive_Packet.size data
           val raw_check = Receive_Packet.checksum data (0, size)
           val checksum = if raw_check = 2u0 then 2uxffff else raw_check
       in if checksum = packet_sum then handler data
          else () (* bad checksum, discard packet *)
       end

  fun active_open (Address address, handler) =
       Connection (Lower.active_open (address, receive handler)
  fun passive_open (Pattern pattern, handler) =
       let val (c, address) = Lower.passive_open (pattern, receive handler)
       in (Connection c, Address address) end

  fun close (Connection connection) = Lower.close connection
  fun abort (Connection connection) = Lower.abort connection

  type control = unit
  type info = unit
  fun control () = ()
  fun query () = ()
 end (* struct *)
```

Figure 4: The Implementation of a Modular Checksum Protocol

```
functor Tcp (structure Lower: PROTOCOL
                structure Aux: TCP_AUX
                sharing type Lower.outgoing_message = Send_Packet.T
                    and type Lower.address = Aux.address
                    and type Lower.address_pattern = Aux.address_pattern
                    and type Lower.incoming_message = Aux.incoming_message
                val initial_window: int
                val compute_checksums: bool
                structure Scheduler: COROUTINE
                structure Event_Queue: EVENT_QUEUE
                structure B: FOX_BASIS
                val do_prints: bool
                val do_traces: bool): TCP_PROTOCOL =
 struct
  ...
```

Figure 5: The TCP Functor's Parameters

using as a basis the types and values implemented by the structure. The **Check** is a perfect example of this strategy. Besides expecting the lower level to satisfy the **PROTOCOL** signature, the **Check** functor also requires that the incoming and outgoing message types be the specified ones; a sharing constraint lets the compiler verify this assertion. Since the **Check** protocol is very simple, this functor needs no parameters besides the structure implementing the lower-layer protocol; the more realistic protocols we present below need several additional parameters.

4.2 The Parameters of the TCP Functor

The functor implementing TCP has not one but eight parameters, in addition to the structure implementing the lower-level protocol. The last two parameters are used for debugging and will not be discussed here. The three parameters before that are structures that are used by the implementation of TCP and are usually defined as general library utilities outside of TCP. The two parameters before these are configuration parameters to control the behavior of this implementation of TCP. Finally, the **Aux** structure implements all the functions which TCP requires of its lower levels.

The functions of the **Aux** structure are defined in the **TCP_AUX** signature, shown in Figure 6. TCP needs to hash on lower-layer addresses and convert addresses to strings (the latter for debugging). TCP also needs a default pattern so it can passively open connections with the lower layer and thereby receive any packets that come in with the correct protocol identifier. This identifier is a constant the single-byte value when we run TCP over IP, but may be different if we are running a non-standard protocol stack. The **info** function returns all the information that TCP needs from the lower layer protocol's header of any incoming packet, and the **check** function returns the pseudo-checksum required for an outgoing packet. For non-standard protocols, the value of the checksum could conceivably be always the same (always zero) and independent of the packet and destination. Finally, the maximum size of a packet, the **max_mtu**, is also available so TCP can properly segment the data it is sending. This size may depend on the interface over which the packet will be sent, which is why the function takes the remote address as an argument.

Note that in the TCP functor we share all the types of **Aux** with the corresponding types of

```
signature TCP_AUX =
 sig
  type address
  type address_pattern
  type incoming_message
  val hash: address -> int
  val makestring: address -> string
  val default_pattern: address_pattern
  val info: incoming_message
         -> src: address, checksum: Byte2.U.ubytes, data: Receive_Packet.T
  val check: address -> Byte2.U.ubytes
  val max_mtu: address -> int
 end (* sig *)
```

Figure 6: The Signature for the TCP Auxiliary Structure

Lower. This allows TCP to use values from one structures as arguments to functions from the other, and therefore allows the two structures to work together. The compiler is responsible for ensuring that the sharing constraints are actually met by the two structures provided as arguments.

TCP has two levels of flow control. The lowest and most drastic level is implemented by withholding acknowledgements from the peer if TCP is overloaded. The more refined level uses a mechanism called a *sliding window.* In each acknowledgement packet TCP indicates how many bytes it is ready to receive; the peer is not supposed to send more than specified by this value. The **initial_window** parameter controls the window specified by this TCP at connection setup time; 4096 is a common value, though 65,000 is often used to achieve higher performance. When data is received, it is the responsibility of the data handler (by way of a **control** operation) to make sure that TCP is told when it may accept more data and therefore increase the window.

We have mentioned in an earlier example the function of the **compute_checksum** parameter.

TCP specifies a number events that should execute after timers have expired and concurrently with other events: the retransmission of unacknowledged packets and the transmission of acknowledgements are both controlled by timers. The **Scheduler** provides convenient functions such as **fork** and **sleep** that TCP can use as primitives to avoid having to explicitly schedule these events. The **Event_Queue** structure provides synchronization primitives for those cases where synchronization is needed, and the structure B[5] gives access to other structures which form the standard library for the FoxNet code.

4.3 The Parameters of the IP Functor

The first IP functor parameter is the lower-level protocol instance, as is common to all protocols. This structure is constrained to have the incoming and outgoing message types bound to particular types, as for the **Check** functor.

The **resolve** function parameter would normally be a function which uses the Address Resolution Protocol [6] to discover the lower-layer address for the given higher-layer address. Since IP knows which interface it is going to use to send the packet, it provides that information to the

[5]We have used a short name because the structure is used very frequently.

```
functor Ip (structure Lower: PROTOCOL
               sharing type Lower.incoming_message = Receive_Packet.T
                   and type Lower.outgoing_message = Send_Packet.T
               val resolve: wanted: Byte4.U.ubytes,
                             local_addr: Byte4.U.ubytes,
                             interface: string
                          -> Lower.address option
               val lower_hash: Lower.address -> int
               val lower_default_pattern: Lower.address_pattern
               val lower_max_packet_size: unit -> int
               val lower_min_packet_size: unit -> int
               val hide_received_header: bool
               structure Event_Queue: EVENT_QUEUE
               structure Scheduler: COROUTINE
               structure B: FOX_BASIS
               val do_prints: bool): IP_PROTOCOL =
 struct
  ...
```

Figure 7: The IP Functor's Parameters

resolve function.

The **lower_hash** and **lower_default_pattern** are equivalent to the corresponding TCP parameters.

The packet size parameters are used by IP to make sure the packets it is sending are legal for the lower layer. When we add fragmentation, IP will be able to fragment packets that are too large. For now, IP raises the **Packet_Size** exception if handed a packet which is too large. Packets that are too small are handled correctly by adding bytes to the end of the packet to make it be at least as large as the minimum size. On receipt, the peer removes any bytes inserted by the remote IP or by the operating system (Mach also inserts pad bytes to make the packet size a multiple of 4 bytes) to reproduce the original packet.

The **hide_header** parameter is similar to the TCP's **compute_checksum** parameter. If the protocol above IP does not need to take a look at IP's header, the parameter should be set to true. In the more normal case of TCP and UDP, the higher-layer protocol does need to access the IP protocol's header; for TCP specifically, the **Aux** structure actually accesses the header, extracts the values needed by TCP, then hides the header before handing the remainder of the data to TCP. This keeps the code for both TCP and IP independent of the layer below and above, respectively, and concentrates all protocol stack dependencies in the much smaller and simpler **Aux** structure.

The last four parameters of the IP functor are similar to the corresponding parameters of the TCP functor.

The parameters for the Ethernet functor are a subset of those for the TCP and IP functors.

4.4 Initial Results

We have completed an initial implementation of the TCP/IP protocol stack. This includes implementations of the TCP, IP, ARP, and Ethernet protocols, and the implementation of an Ethernet device interface. The implementation is built on top of the Mach 3.0 microkernel, and is coded entirely in an extension of Standard ML with primitives to provide access to the **mach_msg** primitive of the Mach kernel which provides most of the services traditionally provided by system calls.

To date, we have tested the entire protocol stack on a simulator. We have also successfully run preliminary tests of most of the protocol stack between Ethernet-connected hosts.

We have also run performance benchmarks that involve sending and receiving data on a minimal protocol stack consisting only of the lowest layer of the stack, the Ethernet device interface. The results indicate that we can send more quickly than we can receive, and that we can correctly receive data at speeds above 3 megabits/second on a DECstation 5000/125 host. We fully expect to improve this performance in the future, though of course the Ethernet will be saturated at 10 Megabit/second. A thorough analysis of the performance of the entire protocol stack and individual layers is planned, with the results to be presented in a future report.

5 Concluding Remarks

We have designed and built a modular implementation of a standard and widely used network communications protocol. The structure of the system into independent modules with a well-defined interface between protocols ensures that protocols are composable into a variety of configurations without sacrificing type security. The design hinges on a generic protocol signature which contains exactly those types and values that we expect every protocol implementation to support. This signature is enriched and specialized to produce the specific signatures for the individual protocols. Protocol implementations which satisfy a specific signature also satisfy the generic signature. Our protocol implementations are parametrized to run on top of any protocol that satisfies the generic signature, and can therefore be easily composed to form different special-purpose protocols.

The TCP/IP implementation is a first step towards establishing the feasibility and usefulness of advanced programming languages for developing real-world systems, in particular network communications systems. While the performance of our implementation in SML does not yet match that of the most highly tuned C implementation, we are encouraged to have already achieved reasonable throughput, with only preliminary tuning or performance analysis. Clearly, with respect to performance, much work and analysis is still required, and it is our plan to carry out this work now.

Besides performance, software quality considerations are also important. Here, we can already claim to have implemented a system with important reliability benefits. In particular, the language guarantees that many common bugs are not possible; our system will not "dump core". Although we have not taken steps to measure such software-engineering aspects such as reliability and maintainability of the system, we are also convinced that our choice of language provides substantial benefits here as well. In order to provide some evidence of this, we have been maintaining development logs, in an attempt to record the nature and frequency of bugs that occur during the development and maintenance of the system. We are able to report that the log, thus far, is rather thin.

6 Acknowledgements

We would like to thank Brian Milnes for his many important contributions to the design and implementation of the FoxNet. We would also like to thank Nick Haines, Greg Morrisett, and Eliot Moss for their assistance and contributions to the FoxN .

References

[1] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An analysis of TCP processing overhead. *IEEE Communications*, 27(6), June 1989.

[2] Eric Cooper, Robert Harper, and Peter Lee. The Fox Project: Advanced development of systems software. Technical Report CMU-CS-91-178, School of Computer Science, Carnegie Mellon University, August 1991.

[3] USC Information Sciences Institute. Internet protocol. RFC 791, September 1981.

[4] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, 1990.

[5] S. W. O'Malley and L. L. Peterson. A dynamic network architecture. Technical report, Arizona, 1993.

[6] David C. Plummer. An ethernet address resolution protocol, or converting network protocol addresses to 48.bit ethernet address for transmission on ethernet hardware. RFC 826, November 1982.

[7] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4), August 1988.

[8] H. Zimmerman. OSI reference model – the ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications*, COM-28(4):425–432, April 1980.

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890