The Fox Project

A Language-Structured Approach to Networking Software

by Jeremy Buhler

Anyone who has studied the construction of compilers or other large software systems has probably been exhorted to practice modular design. It is an article of faith among today's programmers that big designs must be split into manageable chunks with well-defined interfaces between them. Although we practice this philosophy today in building user programs and operating systems, most networked computers rely on a major software system from which this philosophy is largely absent: the network protocol stack.

In theory, a protocol stack can be split into modules at least as well as any other software system. In practice, modularity in stack design has been sacrificed for raw speed, with the result that networking software, while fast, can be painful to maintain or even understand. In C, most network programmers' implementation language of choice, misunderstandings about how code works often lead to hard-to-trace bugs such as subtle memory corruption. Moreover, network stacks are usually composed of several asynchronously-connected layers; programming errors while connecting these layers can lead to concurrency problems such as race conditions and deadlocks. Trying to modify an existing protocol stack is rightly viewed as a perilous undertaking.

The Fox Project, a research enterprise of Carnegie Mellon University's computer science department, attacks the notion that networking code cannot be both modular and efficient. The project's protocol stack, called FoxNet, is written in Standard ML, a language which provides strong compiler support for modular design. The result is a modular, type-safe protocol stack which is still fast enough for practical use.

The Ideal of Network Programming

> *And the Chief Programmer sent his Analysts to the Users and said,*
> *``Let Specifications be written!''*
> *And there were meetings, and lunches, and telephone calls.*
> *And the Specifications were written.*
> - Michael Coleman, ``Genesis, Release 2.5''

The International Standards Organization (ISO) has adopted a codification of the ideal network protocol stack in a scheme called the *Open Systems Interface (OSI) Reference Model* [5]. The OSI model divides networking software into a seven-layer stack, in which each layer has a distinct function. Messages between two connected machines are sent down the stack of the sending machine, across the network, and up the stack of the receiving machine.

The exact function of each layer of the OSI model is debatable and is not germane to this discussion. However, several important features of the model are widely accepted:

- The protocol stack is decomposed into distinct modules, separated according to functionality. Typically, each major module implements one network protocol.
- Each module in the stack communicates only with the modules directly above and below it through well-defined interfaces. The height of the stack may vary in non-OSI designs, but the top module always talks to user programs, while the lowest software module communicates with the network hardware.
- Modules at a given layer of the model have the same interface.

These aspects of the OSI model describe the theoretical structure of many current networking protocols quite well. For instance, a WWW browser and server might communicate through a stack of the following form:

HTTP
Berkeley sockets
TCP
IP
Ethernet
10-base-T

The OSI model's layered structure is a good starting model for implementing network protocol stacks. If, however, we were to inspect most current networking implementations, we would be hard pressed to identify which portions of the code correspond to which layers, or where the inter-layer communications take place. Why doesn't real networking code reflect the simplicity of the OSI model?

Networking Practice

> *The gap between theory and practice in theory is nowhere near as big as the gap between theory and practice in practice.*
> - anonymous

We can readily identify a major reason that current networking practice does not match the OSI model's theory: performance optimization. A naïve layered protocol stack is extremely inefficient. For instance, if layers are implemented cleanly as asynchronous threads, each inter-layer communication involves the overhead of a context switch. Strict modularity also suggests that each layer maintain its own buffers, so that inter-layer communications must copy data between layers. Inefficiencies like these can destroy the performance of a networking implementation; they are the primary reason that the seven-layer OSI model is sometimes called ``The Seven Deadly Sins of Networking."

A well-known technique for dealing with excessive data copying and context switching is *Integrated Layer Processing (ILP)* [4], a program transformation which may be applied to modular network code. When this optimization is applied, boundaries between layers are systematically destroyed to reduce copying and context switches while preserving correct operation. Because of its complexity and specialized nature, ILP never has been and probably never will be implemented in a C or C++ compiler. As a result, network programmers must apply it to their source code by hand.

After ILP has been applied, a protocol stack performs better; however, its modular structure is largely gone. Unless network programmers can translate mentally between original and optimized code on the fly, they are likely to introduce bugs with each modification of the protocol stack. For stacks written in C or C++, the bugs may lead to subtle memory corruption; bugs in the stack may also manifest themselves intermittently as synchronization errors, such as race conditions or deadlocks. Those with some experience in concurrent programming can testify to the difficulty of finding and eliminating such subtle bugs. Errors in network code are often so difficult to fix that implementing such code has acquired the status of a black art, best left to a few networking wizards.

ILP is one way to optimize network code, but it is by no means the only way. As proof of this fact, a freely available C-based TCP/IP implementation exists which exhibits reasonable performance while preserving its elegant modular structure. The University of Arizona's *x-kernel* [8] is a strictly layered protocol stack with a consistent interface between layers. The x-kernel, though not as fast as ILP-based stacks, exhibits a large speedup over naïve layering. In implementing this stack, the x-kernel programming team maintained consistent modular interfaces largely by mutual consent and good documentation, since the C language provides a relatively weak mechanism for enforcing type safety between modules at compile time.

While the x-kernel's modular design is well thought-out, it is achieved in spite of rather than in cooperation with the project's implementation language. The protections provided by the C and C++ prototype systems can be easily, sometimes inadvertently, circumvented by explicit or implicit type casts. A common case of this problem is the use of type-unsafe `void` pointers (as in ANSI C's `quicksort`) to represent abstract or unspecified types in module interfaces. C++'s class and template systems partially address the need for abstraction, but they lack SML's proven basis in formal semantics [6, 7]. As a result, they are notoriously difficult to implement correctly, and programmers use their full functionality only at their peril. As a result, modular design in C and C++ is often blocked or subverted by the implementation language.

The Fox Project extends the ideas expressed in the x-kernel by enforcing modular design through the Standard ML module system. This strong yet flexible system of type specifications and checks provides a language framework tailored for writing and type-checking modules. Below, we outline the structure of the SML module system and, in parallel, the structure of the FoxNet networking code to show how SML modules support the implementation of a modular network protocol stack.

SML Modules and Their Use in Modular Software

> *And the Chief Programmer said, ``Let the System be divided into parts, and let each part become a Module. And let programming teams be formed and let each be assigned to write a Module.'' And it was so.*
> - Michael Coleman, ``Genesis, Release 2.5''

Standard ML is a *strongly-typed* language. It only accepts *type-safe* programs, in which every data value has a well-defined type. C and C++ programs, in contrast, can and do cast values to any type by either explicit or implicit casting rules. Because SML's strong type system assigns fixed type information to each value, it provides the formal structure necessary to support and enforce modular programming practices within an SML compiler.

The basic SML module system extends the language's type system to include structures and signatures. *Structures* are environments in which one may define and use arbitrary SML values -- such as integers, lists, and functions -- and arbitrary SML types, such as records with named fields and abstract data types. We call the types and values defined inside a structure the *bindings* of its environment, since each type or value is bound to a name which is stored inside the structure.

Because different SML structures define distinct environments, they are useful for modularizing code, rather like separate compilation units in C. (C structures, in contrast, are nothing more than tuples of values; they correspond to SML records.) In order to access values or types contained in a structure, the programmer must either open the structure, merging its contents into the larger runtime environment, or reference the structure by name when accessing its members. Another consequence of encapsulating environments within structures is that different structures may bind different values or even different types to the same name.

The encapsulation that SML structures provide for their contents allows software designs such as network stacks to be cleanly separated into modules; each module simply gets its own structure environment. However, structures alone cannot force modules to conform to interface guidelines. For this latter purpose, SML provides the mechanism of *signatures*.

A signature specifies a set of type and value bindings which may be found inside a structure environment. We say that a structure *satisfies* a signature if every type and value binding listed in the signature is present in the structure. Note that the converse need not be true; that is, a structure may contain values which are not specified in its signature. For this reason, a given structure may satisfy many different signatures, each of which provides a different view of the structure's interior.

Roughly speaking, signatures are to structures as types are to values. However, signatures are useful for more than just providing type information. If a structure satisfies a signature, we may *qualify* the structure with the signature, so that code outside the structure can access only those of its bindings which are visible in the signature. A signature may therefore be used to enforce a specific view of the structure -- its interface. Moreover, if the signature does not specify the implementation of the structure's data types, those types essentially become abstract outside of the structure. Signatures implement a clean boundary between a structure's interface and its implementation. By varying the amount of information exported at signature boundary, the programmer exerts considerable (though not absolute) control over the code's level of data abstraction.

We have just covered a great deal of theory about modules; it is now time to see how this theory is applied in the Fox Project's code, starting with a brief overview of the top-level design. In FoxNet, each layer of the protocol stack is a module which must conform to an agreed-upon signature for protocols; this signature describes which operations a protocol must provide to get along as part of the FoxNet stack. Anyone can implement a protocol's internals, which eventually end up in a structure, but its interface is fixed by its signature. The SML compiler uses the type information in the protocol signature to catch errors within and between the modules of the protocol stack.

The signature of a FoxNet protocol contains considerable information about how the protocol works. Therefore, we will explore the design of FoxNet in more detail -- not by opening up its structures but by considering what its designers put into its signatures.

The FoxNet PROTOCOL Signature

The PROTOCOL signature defines the properties shared by every protocol defined in the FoxNet stack. Figure 1 shows the important parts of this crucial signature. The little bit of SML syntax required to make sense of the code is explained in the caption.

```
signature PROTOCOL = sig
    eqtype address
    eqtype address_pattern
    eqtype connection

    eqtype incoming_message
    eqtype outgoing_message

    val intialize: 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

    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
```

Figure 1: The FoxNet PROTOCOL signature. Values, which are always followed by their types, are specified by name with the `val` keyword. Arrows denote function types, while asterisks indicate tuple types. `eqtype` is a minor variation on the `type` keyword. Listing adapted from FoxNet sources, after [1].

The interface defined by the PROTOCOL signature includes the major items found in any communications protocol. The first notable items are data types for addresses and connections. Every protocol must have these data types in some form, but the implementation of each type is left up to the module's programmer. The lack of definitions for these types means that no other module may assume anything about the implementations of the objects they represent. (This is almost true; the fact that the first few types are `eqtypes` means that it is safe to compare two objects of each type for equality.) An address may be implemented as a record, a list, or an abstract data type, but to anyone except the module's programmer, the implementation is unknown. A side effect of this hiding of data is that all objects of type `address` must be created inside the module, since nobody outside knows for sure what a value of that type looks like.

The incoming and outgoing messages are unspecified types representing the data packets processed by the protocol module. As above, the implementations of these types are hidden from other modules. The reader may well object that different protocols ought to use the same data type to represent packets, so that inter-module communications may proceed efficiently. This objection is well-grounded, but it is no reason to reveal the guts of the packet data type to the world. The issue of sharing data types will be addressed later on when we consider functors.

The functions `initialize` and `finalize` allocate and free system resources used by the protocol; they take no arguments and so have argument type `unit`, the type containing only the nil value (). The open, close, and abort functions are responsible for building and tearing down connections. It is worth noting that the connection-opening functions themselves take an argument of function type `connection -> incoming_message -> unit`. The unnamed argument function, which we may call `receive`, is responsible for dealing with any incoming messages that appear on the open connection. This method of receiving data is known as an *upcall*, since the lower-level protocol calls the higher-level `receive` to pass a received packet up the stack [2]. The advantage of the `receive` upcall is that a packet can be passed up the protocol stack without a context switch, since an upcall is executed within the context of the lower-level module. Upcalls are thus a performance enhancement which increases the rate at which packets can be received.

The `control` data type is generally composed of a selection of command tokens which affect the operation of a protocol. The protocol's behavior is altered by passing these tokens to the protocol's `control` function. Each protocol also supplies an `info` data type, which contains information on the status of the protocol, and a `query` function to return an object of that data type. These two data types contain most of the protocol-specific parts of the interface; with the right control messages and exported information, a protocol can look like TCP, IP, or HTTP. Unlike the `address` type, whose implementation can usually be hidden from other modules, the `control` and `info` type implementations must be exported (usually as SML `datatype`s) so that others can call these implementations to manage the protocol. They are not specified here because each protocol typically has its own control and information messages, which are specified through a mechanism discussed below.

One more interesting feature of the PROTOCOL signature is its exceptions. Like C++ exceptions, SML exceptions may be used to abort from within a function. Every protocol supports the exceptions shown to recover from errors during its operation. Exceptions can take arguments which indicate the exact nature of the errors which raised them. Other protocols, or even code outside the protocol stack, can catch the exceptions to

provide customized error handling capability.

With the help of signatures and structures, we now have about half a module system and half a protocol stack. On the positive side, structures enforce modular separation, a fact which FoxNet uses to split its stack into layered protocols. With signatures, we can provide and enforce a consistent interface; FoxNet's PROTOCOL signature makes sure that the key elements of a protocol module are implemented and compatible, providing a measure of interchangeability. At the same time, the PROTOCOL signature enforces data abstraction at the protocol interface, masking the details of particular protocol implementations from the rest of the stack.

On the other hand, we have two rather serious problems outstanding. First, while considering the packet types, we deferred the question of how to make sure certain data structures are shared between protocols. This issue is just one example of the bigger question of how we will actually make our different protocol modules interoperate. Second, we have yet to show how to use the PROTOCOL signature to produce an interface for a real, usable protocol. If FoxNet is ever to send a single ping, we had better deal with these issues now.

Signature Inheritance and Functors

Writing a beautiful generic module specification is useless if we cannot specify and link specific protocols into an interoperating stack. We already have some inkling of what an SML protocol module ought to look like. We will now consider how real protocols can be written around the PROTOCOL signature, and how they can be made to work together by the use of functors.

Signature Inheritance

> *And they flowcharted, and they coded, each in his own fashion.*
> *And the Chief Programmer looked upon the work and liked it not.*
> *And the Chief Programmer said, ``Let there be a Standard;''*
> *And there was a Standard.*
> - Michael Coleman, ``Genesis, Release 2.5''

As an example of a real protocol built around the PROTOCOL signature, we consider the specification of FoxNet's IP protocol, part of which is shown in Figure 2. A structure written to conform to this specification also satisfies the generic PROTOCOL signature, but it has additional information which provides an exact description of the IP interface. Once again, most of the necessary SML syntax is explained in the caption.

```
signature IP_PROTOCOL = sig
    type incoming_data

    datatype ip_address =
        Address of {ip: ubyte4, proto: ubyte1}

    datatype ip_address_pattern =
        Pattern of {protocol: ubyte1, source_ip: ubyte4 option}
```

```
    datatype ip_receive = Ip_Packet of {header: ByteArray.bytearray,
                                        data: incoming_data}

    structure Options: IP_OPTIONS

    type ip_connection

    datatype ip_control =
        Set_Default_Gateway of {gateway_ip: ubyte4}
      | Set_Specific_Gateway of {destination_ip: ubyte4, gateway_ip: ubyte4}
      | Disable_Specific_Gateway of {destination_ip: ubyte4}
      | Set_Interface_Address of {interface: string, ip_number: ubyte4}
      | Disable_Interface of {interface: string}
      | (* lots of other IP control options not shown here ... *)

    datatype ip_info =
        Info of {max_packet_size: ip_connection -> int,
                 interfaces: (string * ubyte4 option) list,
                 local_address: ubyte4 -> (string * ubyte4) option,
                 remote_address: ip_connection -> ip_address,
                 time_to_live: ip_connection -> int,
                 packets_sent: int,
                 packets_received: int,
                 failed_sends: int,
                 packets_discarded: int}


    include PROTOCOL
        sharing type address = ip_address
            and type address_pattern = ip_address_pattern
            and type connection = ip_connection
            and type incoming_message = ip_receive
            and type control = ip_control
            and type info = ip_info
end
```

Figure 2. The IP_PROTOCOL signature. Note that most of the abstract types declared in the PROTOCOL signature are now specified using `datatype` declarations. Lists of names and types inside curly braces represent record types. Listing adapted from FoxNet sources, after [1].

The IP_PROTOCOL signature illustrates how a particular protocol evolves from the general protocol interface. We see versions of most of the basic PROTOCOL elements -- addresses, connections, control messages, and status information -- declared and fleshed out. In particular, the unspecified control and information types have

been replaced by detailed specifications which other modules can use to interface with IP.

Of course, if IP_PROTOCOL is supposed to be a specific version of PROTOCOL, the two signatures had better be compatible. For instance, what if a careless programmer forgot to put in an IP-specific equivalent to the generic `address` type? The SML compiler's type checking does not include artificial intelligence; it is *not* smart enough to tell from the variable names alone which declarations in the IP signature correspond to their prototypes in the PROTOCOL signature. After all, we didn't have to call the IP version of `address` ``ip_address" -- it might just as well have been called ``foobar"!

To verify the relation between the general and specific protocol signatures, SML uses a technique called *signature inheritance*. This mechanism manifests itself as the `include` directive, which has the same effect as including a verbatim copy of the entire PROTOCOL signature within the IP_PROTOCOL specification. This inclusion takes care of any elements in the specific signature which may be inherited unchanged from the general signature, such as the six exception types and the `open`, `close`, and `send` functions. Indeed, these elements are not mentioned anywhere in Figure 2, since they are copied in automatically during inclusion. This type of inheritance is similar to simple inheritance in C++ class declarations.

To tackle elements such as `address` which are given more specific declarations in IP_PROTOCOL, we use another aspect of signature inheritance known as *type sharing* constraints. A type sharing constraint in a signature declares that two types must be equal (i.e., they must refer to the same set of values) in any module satisfying the signature. If a structure with a constraint-bearing signature implements the two types in the constraint incompatibly, the SML compiler rejects the structure as non-conforming. For instance, one could apply type sharing to equate a list of unspecified type with the specific type `int list`, but it is not possible to equate an integer type with a function type. Sharing constraints only apply within the scope of the signature where they are declared, so a single generic type may be matched to different specific types in different modules' signatures.

In practice, sharing constraints can be used as part of signature inheritance to equate types from a generic signature with their definitions in a specific signature. For example, we solve our address problem nicely by including in the IP_PROTOCOL signature a sharing constraint which equates the generic type `address` with the specific type `ip_address`. In fact, the sharing statements in the IP_PROTOCOL signature equate all the generic types from PROTOCOL with their specific definitions in IP_PROTOCOL.

Signature inheritance informs both the SML compiler and any readers of the FoxNet code as to exactly how a particular protocol signature conforms to the general interface specification. The specific signature, in turn, may be used to qualify the protocol's module, which may therefore be shown to conform to both the generic and specific signatures. Aside from being a coding convenience and a good documentation practice, the signature inheritance mechanism enables the compiler to test a FoxNet signature's compliance to protocol design specs as reliably as it tests the compliance of the code in a protocol's implementation.

Functors and Module Integration

> *Two by two, the Modules were integrated, one with another. And great difficulties were experienced, and many hours of overtime were used, and many cups of coffee were consumed.*

- Michael Coleman, ``Genesis, Release 2.5''

Armed with signatures and signature inheritance, we have been able to specify our module interfaces completely, with assured compliance to a standard generic interface. We will now take a mental coffee break while the illustrious FoxNet programming team actually hacks up the required module code to implement the protocols in our stack. As the FoxNet programmers peck madly at their keyboards, now would be a good time to give them due credit by visiting the Fox Project authors' page at *http://foxnet.cs.cmu.edu/people/*.

After much labor, the FoxNet protocol modules are complete. It is now time to integrate them and produce the final protocol stack. Thus we come again to an important question which was posed above: how do we specify how the FoxNet protocols are to be hooked together?

Functors are the Standard ML method for composing structures according to fixed interface specifications. A functor is a function over structures: pass some structures into the functor as arguments, and a new derived structure comes out. Just as a function has argument types and a return type, a functor has an argument signature to restrict the types of its inputs and a result signature to describe the structure which it returns. Using a syntactic extension, a functor can accept individual types and values as well as structures in its argument list. The process of linking argument structures together to produce the result structure is appropriately dubbed *functor application*.

The design purpose of functors is to specify modules in terms of their dependence on other modules in the system. Typically, a modular design is a hierarchy in which higher-level modules depend on more basic ones for their functionality. In functor-based design, the lower-level modules are passed as arguments to a functor which produces the higher-level module, having made all the necessary links between the modules in the course of application. Of course, the lower-level modules must conform to some expectations on the part of the functor; the compiler uses the argument signature to check these expectations and fails to compile the program if they are not met.

Typically, most of the code in an SML application is written not in structures but in the bodies of functors, since most modules depend on others for their operation. In fact, since a functor can be written with an empty argument signature, it is common programming style to write *every* module, even one with no dependencies, as a functor. The structure of an application like FoxNet is therefore not ``one module, one structure'' but rather ``one module, one functor.'' The input structures for higher-level functors do not even exist until they are created on the fly by the application of lower-level functors.

To show how functor-based design is applied in hooking together the FoxNet protocol stack, we exhibit the argument signature of the IP functor in Figure 3. Notice that, just as a value declaration in SML is explicitly qualified with a colon followed by its type, a structure or functor is explicitly qualified with a colon followed by its signature.

```
functor Ip (structure Lower: PROTOCOL
            structure B: FOX_BASIS
            sharing type Lower.incoming_message = B.Receive_Packet.T
```

```
            and type Lower.outgoing_message = B.Send_Packet.T
          val lower_resolve: {wanted: ubyte4,
                              local_addr: ubyte4,
                              interface: string}
                             -> Lower.address option
          val lower_hash: Lower.address -> int
          val lower_default_pattern: Lower.address_pattern
          val lower_max_packet_size: string (* interface *) -> int
          val lower_min_packet_size: string (* interface *) -> int
          val initial_interfaces: (string * ubyte4 option) list
          val do_prints: bool): IP_PROTOCOL =
struct
  (* 1200 lines' worth of IP implementation ... *)
end
```

Figure 3. The arguments to the IP functor. Not shown are the 1200 or so lines of code in the body of the functor which actually implement IP using the core SML language. Listing adapted from FoxNet sources.

Most of the arguments to the IP functor describe the properties of the protocol just below it in the stack. In particular, the structure `Lower` which is passed into the functor is a complete instance of the lower-level protocol module. Because the lower-level protocol must conform to the PROTOCOL signature, IP knows how to communicate with the lower interface in a type-safe way without breaking the layer abstraction. Any lower-level protocol (e.g., Ethernet, arcnet, token ring, packet radio) can simply be plugged in, so long as it provides the appropriate interface.

Just as IP accepts a lower-level structure which implements the protocol beneath it, the structure returned by the IP functor can be passed as the lower level of the TCP functor, and so on until an entire protocol stack has been built. In fact, this is precisely how the real FoxNet protocol stack is assembled. At every step in building the stack, the compiler checks that the interface between protocols is satisfied; hence, once the stack has been compiled, the amount of intermodule testing to be done is significantly diminished. In particular, every interface is guaranteed to be type-safe with all necessary arguments present.

We can now tackle a specific question posed earlier in our development: what do we do when two modules need to share a common data structure? The situation which alerted us to this problem was that multiple protocols should share the same packet type; we have seen that the PROTOCOL signature cannot guarantee this sharing without exposing the unsightly innards of the packet datatype. The designers of Standard ML noticed that this kind of problem occurred frequently with the feature known as `abstype` which preceded modules. It was for this reason as much as for signature inheritance that the current system of sharing constraints was invented.

Simple signatures only describe individual modules. They cannot specify that the common substructures of two modules have compatible implementations. Signatures with sharing constraints, on the other hand, can ensure that two modules use compatible definitions of key data types or submodules. We therefore take advantage of sharing constraints by using them in the argument signatures for functors. Specifically, we declare an equivalence between a type within an argument structure and the corresponding type either in another argument

or in the body of the functor. Just as with signature inheritance, the SML compiler accepts type sharing constraints in functor arguments only if they are between compatible types; hence, argument structures with incompatible pieces are simply rejected by the compiler.

As an example of the use of functor argument sharing constraints, consider the FoxNet IP functor. The functor expects not only a lower-level protocol but also a structure, called the *Fox basis*, which contains among other things the definitions of global send and receive packet types shared by all FoxNet protocols. Although the generic PROTOCOL signature does not specify the types of data packets, the FoxNet IP implementation will not work unless the packet types used by lower-level protocols are the same as those in the Fox basis. The sharing constraints in the IP functor arguments make this condition explicit, preventing the linking of IP with a protocol which uses an incompatible packet type.

FoxNet in the Real World

> *Then the Chief Programmer did go unto the Project Manager and said,*
> *``Behold, I bring you tidings of great joy which will come to all Users;*
> *for on this day The System is completed.''*
> - Michael Coleman, ``Genesis, Release 2.5''

Despite what the reader may have concluded by now, this is not a paper about programming languages. Rather, it details a language-structured approach to the design of a modular network protocol stack and the ways in which the SML language resolves issues which arise during the design of the stack. Nothing here would be of more than academic interest to network programmers if the FoxNet stack did not actually *work*.

To put the FoxNet protocol stack through its paces, readers may access the Fox Project's Web Server at *http://foxnet.cs.cmu.edu/*. The protocol stack used on this machine is FoxNet, written entirely in Standard ML. The server is built entirely around the code discussed above; it implements a complete protocol stack, from HTTP on top right down to the network device driver, which interfaces directly to the Mach 3.0 microkernel.

No discussion of a new network protocol stack would be complete without mentioning its real-world performance (indeed, many network programmers seem to care about little else). FoxNet was not adapted from any existing protocol stack but was written from scratch. To date, the code has been worked on for a little over two years and is far from completely optimized, though improvements continue. FoxNet's IP and UDP throughput is on par with many current networking implementations, so long as it does not have to fragment outgoing packets. Fragmented IP and UDP, along with TCP, exhibit considerably lower throughput. Latency tends to be high as well. Still, FoxNet performance is within an order of magnitude of the BSD-based x-kernel and is constantly improving as new optimizations are applied.

Inefficiencies in FoxNet's implementation may be blamed in part on a lack of compile-time optimizations, particularly the presence of redundant array bound checks in inner loops, and in part on the inevitable (though decreasing) expense of modular code over code transformed with ILP. On the other hand, FoxNet does *not* suffer from excessive data copying or context switching, the two primary reasons why most protocol stacks had their structures destroyed by ILP in the first place.

Lest the reader feel that a somewhat slow networking implementation is no great achievement, consider the

following facts. Unlike some high-performance implementations, the FoxNet code does not exhibit core dumps, memory leaks, type errors, or synchronization problems. Most of the Standard ML language has been *proven* type-safe, correct, and self-consistent [6, 7], reducing the likelihood of both user- and compiler-introduced bugs. Thanks to its modular structure, the FoxNet code is not especially difficult to understand, even for non-wizards. The SML module system further ensures that the code can be safely modified and optimized, even by a team of programmers, without introducing hordes of interface violations into the system. Frankly, the ever-shrinking cost in lost performance seems a small price indeed for FoxNet's reliability and maintainability benefits.

FoxNet 1.0 for Mach 3.0 is available for anonymous ftp at ftp.cs.cmu.edu, in the project/fox subdirectory. It comes with a compiler for Standard ML which is based on the popular Standard ML of New Jersey compiler. A set of performance figures is included in the distribution. A new version of FoxNet should be released in late summer of 1995. The Fox Project team continues to show that modular, maintainable, correct networking code is compatible with real-world implementations and realistic performance. In short, FoxNet shows that language-supported modular design of a network protocol stack works.

The author wishes to thank Robert Harper, a principal investigator of the Fox Project, along with Shriram Krishnamurthi and Matthias Felleisen of Rice University, for their invaluable aid in preparing this article.

References

1

Biagioni, E., Harper, R., Lee, P., and Milnes, B.G. Signatures for a Network Protocol Stack: A Systems Application of Standard ML. In *Proceedings of 1994 ACM Conference on Lisp and Functional Programming* (June 27-29, Oraldo, Fl.) SIGPLAN/SIGACT/SIGART, New York. 1994, pp. 55-64.

2

Clark, D.D. The structuring of systems using upcalls. In *Proceedings of the SIGCOMM '90 Symposium*, Sept. 1990.

3

Clark, D.D., and Tennenhouse, D.L. Architectural considerations for a new generation of protocols. In *Proceedings of SIGCOMM '90 Symposium*, Sept. 1990.

4

Coleman, Michael. Genesis Release 2.5. Electronic version downloaded May 1995 as gopher://wiretap.spies.com/00/Library/Humor/Nerd/genesis.txt . RHSFTRW (The Real World), 1984.

5

Day, J.D., and Zimmerman, H. The OSI Reference Model. *Proceedings of the IEEE* 71 (Dec. 1983) pp. 1334-1340.

6

Milner, R., and Tofte, M. *Commentary on Standard ML*. MIT Press, Cambridge, Mass., 1990. (For paperback, go here.)

7

Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML*. MIT Press, Cambridge, Mass., 1990.

8

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