View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Springer - Publisher Connector

Higher-Order and Symbolic Computation, 14, 309–356, 2001
© 2002 Kluwer Academic Publishers. Manufactured in The Netherlands.

A Network Protocol Stack in Standard ML*

EDOARDO BIAGIONI esb@hawaii.edu
University of Hawai'i at Mānoa, ICS Department, 1680 East-West Road, Honolulu HI 96822, USA

ROBERT HARPER robert.harper@cs.cmu.edu
PETER LEE peter.lee@cs.cmu.edu
Carnegie Mellon University, School of Computer Science, 5000 Forbes Avenue, Pittsburgh PA 15213, USA

Abstract. The `FoxNet` is an implementation of the standard TCP/IP networking protocol stack using the Standard ML (`SML`) language. `SML` is a type-safe programming language with garbage collection, a unique and advanced module system, and machine-independent semantics. The `FoxNet` is a user-space implementation of TCP/IP that is built in `SML` by composing modular protocol elements; each element independently implements one of the standard protocols. One specific combination of these elements implements the standard TCP/IP stack. Other combinations are also possible and can be used to easily and conveniently build custom, non-standard networking stacks. This paper describes in detail the final design and implementation of the `FoxNet`, including many of the details that are crucially affected by the choice of `SML` as the programming language.

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

1. Introduction

The `Fox` project was started in 1991 with the goal of writing systems-level code in high-level programming languages. The purpose of doing this was and is twofold: to improve the state of the art in systems development by using more advanced languages, and to challenge both the design and implementations of advanced languages by applying them to some of the hardest programming problems known. At the time it was hoped that these goals would lead to substantial improvement in the practice of operating system development, as well as to a stronger research and development focus on the performance and expressiveness of advanced programming languages. These improvements are needed for the following reasons:

– An operating system is a complex, often hardware-dependent, non-deterministic, near-real-time program that is expected to be nearly bug-free. In order to interact with hardware and to satisfy the near-real-time property, operating systems are mostly written in the `C` programming language. This design choice conflicts with the requirement for reliability, and together with the complexities of hardware dependencies leads to the necessity for large design and testing efforts. Having a better language for operating system

*This research was sponsored by the Defense Advanced Research Projects Agency, CSTO, under the title "The `Fox` Project: Advanced Languages for Systems Software", Contract No. F19628-95-C-0050.

implementations would reduce the effort, and perhaps lead to better operating system designs.

- Advanced languages are usually designed by small teams, which brings a very welcome focus on clean and concise design but also may lead to gaps in the design corresponding to limitations in the designers' knowledge. Furthermore, advanced languages are often designed to have particular mathematical or theoretical properties, such as machine-independence or determinism, which may not hold in the non-deterministic, hardware-dependent world of operating system implementations. Using such an advanced language to implement a networking protocol stack allows language developers to focus on practical issues such as externalization of data structures and efficient pointer manipulation.
- Implementation of advanced languages is often done by small research teams which naturally focus on research issues. This can lead to amazing breakthroughs, but these are sometimes counterbalanced by a lack of emphasis on practical considerations. Using an advanced language implementation to build a working system with practical, near-real-time constraints points out both the strengths and weaknesses of particular programming language implementations.

The `Fox` project decided to build an implementation of the standard TCP/IP protocol stack [22, 23] using the `SML/NJ` compiler [2] for the Standard ML (`SML`) programming language [17, 18]. The result is the `FoxNet` [4, 7], a standards-conforming implementation of TCP/IP that runs on a variety of hardware architectures, supports everything from devices to web servers, and has a modular, composable implementation. This paper describes the `FoxNet`.

The design and implementation of the `FoxNet` is also a case study in the application of several innovative principles:

- Each protocol building block follows a uniform *system architecture*, described in Section 2. This common architecture allows for nearly arbitrary composition of protocol modules. When a specific protocol needs to be layered above other protocols providing more than specified in the system architecture, the uniform architecture can be specialized as required.
- Different protocols often perform similar functions in managing packets and connections. We have abstracted the commonality of these different modules into a single generic protocol implementation module, the *connection functor*. This module is specialized at compile time to take the correct, protocol-dependent action when specific events occur, and is general enough to be used for such diverse protocols as TCP and IP. The connection functor is described in Section 3.1.
- Section 3.2 describes the `FoxNet` coroutines. This coroutine package is designed specifically for the `FoxNet`, and is implemented entirely in `SML` using native continuations. While the use of continuations to implement multithreading is not novel [29], nor is it novel to `SML` [9, 21], extended use of the `FoxNet` helped us identify and correct a storage leak [6] that is common to most such implementations and that had not been previously identified.
- A fundamental issue in systems programming is the choice of efficient data structures for manipulating potentially large amounts of data. While `C` arrays and pointers are very

efficient, they are also very unsafe, frequently causing errors that are hard to locate and identify. In Section 3.3 we describe word arrays, a data structure we used in the `FoxNet` for safe and potentially efficient access to large amounts of data.

– We and others, especially Derby [11], have also studied in detail the performance of the resulting system. We report our findings in Section 4. Studying the performance is important since one of the concerns when using an advanced language such as `SML` is that the advanced features will result in slow performance. Our study shows that, even though the `FoxNet` is competitive with production systems on some benchmarks, performance is indeed an issue, and we look at some of the ways this issue can be addressed.

The language supported by the `SML/NJ` compiler, including minor extensions that we introduced, is a superset of the Standard ML (`SML`) language. `SML` is type-safe, meaning that the type system supports data abstraction by precluding misuse of values of one abstract type as those of another. Extensions of `SML` that are part of `SML/NJ` provide support for continuations, byte arrays, and raw access to the network device. Since there is little potential for confusion, in this paper we use `SML` to refer both to the standard `SML` language [18] and to its superset, the extended `SML` we used.

Section 2 presents the overall architecture of the `FoxNet`. This is followed in Section 3 by a discussion of selected parts of the implementation, in Section 4 by an analysis of the performance of the `FoxNet`, and in Section 5 by a summary of our experience using `SML` for this project.

2. System architecture

Networking protocols are conventions for communication among interconnected computers. These conventions are elegantly described by layered models such as OSI [10], in which each layer in a stack of protocols defines a particular abstraction of a computer network; each layer logically only communicates with its peers, and only uses the abstraction provided by the next lower layer in defining its own abstraction.

The elegance of the layered model has inspired use of this same model in the design and implementation of networking protocols. Traditional protocol stack implementations are monolithic, and the design and implementation of each protocol can make assumptions about the existence and the details of the design and implementation of other protocols in the stack. In contrast, in a layered implementation of a protocol stack each protocol is designed and implemented in isolation, explicitly declaring any assumptions made about other protocols; these protocols are then successively collected into a protocol stack. Layered implementations allow flexibility in using the component protocols to build special-purpose protocol stacks, can be easier to develop and debug than monolithic implementations since protocols can be tested in isolation, and are generally clearer than monolithic implementations since the dependencies among protocols are explicit.

One project that has had some success in building a layered implementation of standard networking protocols is the x-kernel project. Since the x-kernel inspired the overall design of the `FoxNet`, we describe it in detail in the next section, followed by a detailed description of the design of the `FoxNet` itself.

2.1. The x-kernel

The x-kernel project [19] at the University of Arizona has built a suite of such composable protocols, and used these protocols to build layered implementations of many standard and non-standard stacks. One contribution of the x-kernel is the definition a minimal interface that must be satisfied by all protocols in the x-kernel; this interface is called the *meta-protocol*. For example, a protocol implementation for protocol P that is to be layered on top of another protocol implementation Q can rely on Q satisfying the meta-protocol, and thus use all the functions from Q that are specified by the meta-protocol. The exact semantics of each function differ from protocol to protocol and must be taken into account when building protocol stacks from component protocols. For example, addresses are represented as arrays of bytes, but different layers need addresses of different lengths and would interpret the same string of bytes as different addresses. However, at least syntactically, all protocols implement the same functions with the same arguments, and hence the only constraints on composition are given by the semantics of the protocols. In contrast, in monolithic ("normal") protocol stack implementations, the exact layering is fixed and cannot be changed without substantial re-coding.

A straightforward design for a layered implementation has each protocol P perform send and receive operations on protocol Q below it. The receive operation must block waiting for data, as shown in Figure 1. When data is delivered by Q to P, one of the threads blocked on receive must be restarted and given the data. This requires a thread context switch, which is expensive. The resulting cost makes such a layered implementation infeasible for practical implementations.

The x-kernel avoids context switches by providing each lower layer Q with a function f from each upper layer P; f takes as one of its arguments a packet of data, and does the processing required by P. This technique is known as *upcalls* [8], since the lower protocol Q is calling a function from a higher protocol P. The function that handles incoming data can be thought of as a *data handler*.

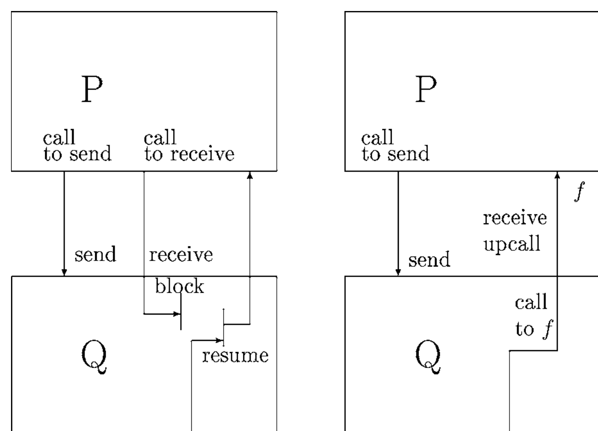

Figure 1. Context switch is required unless receive is an *upcall*.

The results obtained by O'Malley and Peterson [19] show that, because there is no thread to be woken up and consequently no context switch, upcalls allow layered implementations to be as efficient as conventional, monolithic implementations. Instead of having a thread per blocked receive call, a single thread is created for every data frame received from the network. This thread shepherds the packet until it is either delivered to the application (the topmost layer), processed by the networking stack, or stored for future use.

The FoxNet follows the x-kernel in having a layered implementation using upcalls and in creating a new thread for each upcall.

Some characteristics of the x-kernel are worth noting. The implementations of individual protocols are coded in C, but this language does not provide mechanisms for expressing the meta-protocol, for checking conformance of protocols with the meta-protocol, and for the automatic renaming of exported functions needed when protocols are composed. Protocol composition in the x-kernel is achieved externally to the programming language, through a specialized language for defining protocol composition and through associated tools which process these definitions and do essentially a linking pass on the protocol implementations. These tools are sufficient to get the job done, but do relatively little checking and hence do not even guarantee that every protocol in a protocol stack syntactically satisfies the meta-protocol. If this invariant should be broken, subtle problems may appear at run-time. The programming language also limits the design of the meta-protocol; in particular, it does not allow specification by inheritance of specialized versions of the meta-protocol and does not support higher-order functions. As discussed in Section 2.3, higher-order functions can be used to define richer and more appropriate interfaces than those used by the x-kernel.

2.2. Signature hierarchy

The FoxNet is a layered implementation of standard networking protocols, and in overall design somewhat resembles the x-kernel. One crucial difference is that the FoxNet is implemented in SML. SML has a rich module language with interface definitions (*signatures*) as integral part of the language, and with the compiler automatically checking that implementations actually conform to their signatures. Composition of protocols to form a protocol stack is also expressed using the modules language, and the compiler checks not only conformance of protocols with the meta-protocol, but also any additional constraints that one protocol may have placed on its lower protocol. Finally, in the FoxNet the meta-protocol is expressed explicitly as a signature, the PROTOCOL signature. This section describes how the signatures are used in composing protocols to form a protocol stack. Later sections motivate the specific features of our PROTOCOL signature.

The PROTOCOL signature has a large number of abstract types, and is therefore generic: many different implementations may satisfy it by instantiating the abstract types differently. Its generic nature makes the PROTOCOL signature the equivalent of the meta-protocol of the x-kernel. The meta-protocol is used to specify what is implemented by each protocol P, and what P can expect to see in protocol Q that P is layered over. This works well as long as the generic PROTOCOL signature provides all the guarantees that P needs from Q. Sometimes, however, P needs additional guarantees. For example, the standard protocol TCP needs

```
signature NETWORK_PROTOCOL = sig
  include PROTOCOL
  type network_connection_extension =
        {pseudo_checksum: unit -> Word16.word}
  sharing type connection_extension =
             network_connection_extension
  val key_to_address: Connection_Key.T
                  -> Network_Address.T
end
```

Figure 2. NETWORK_PROTOCOL signature.

access to data from the IP protocol to compute its checksum. For other reasons, TCP also needs to be able to compute an address given a connection key.[1]

We can extend the PROTOCOL signature to produce a new signature that specifies these additional guarantees. The extended signature declares everything that the PROTOCOL signature declares, but may place additional constraints on types that PROTOCOL leaves abstract, and also may provide additional operations. To illustrate this point we show how to extend the PROTOCOL signature to support the operations required by TCP. This extension is done by making an abstract type, `connection_extension`, concrete, and by adding a new function, `key_to_address`. We will call this new signature NETWORK_PROTOCOL, and the definition is shown in Figure 2.

The signature NETWORK_PROTOCOL first includes PROTOCOL, declaring everything that PROTOCOL declares. The next declaration defines a new type abbreviation, `network_connection_extension`. This type abbreviation has one component, a function which computes the pseudo-checksum needed by TCP. NETWORK_PROTOCOL then specifies, using the `sharing` declaration, that this type abbreviation is the same as the abstract type `connection_extension` defined in PROTOCOL. A value of type `connection_extension` is defined by the PROTOCOL signature to be part of the `connection`. Hence, wherever the type `connection` appears in the PROTOCOL signature, in the signature NETWORK_PROTOCOL it carries a component of type `network_connection_extension` which has `pseudo_checksum` as its component. In addition, a module satisfying NETWORK_PROTOCOL also has a function `key_to_address`. In all other respects the signature NETWORK_PROTOCOL is identical to the signature PROTOCOL. Because NETWORK_PROTOCOL specializes and extends PROTOCOL but is otherwise identical to it, *any module satisfying* NETWORK_PROTOCOL *will automatically also satisfy* PROTOCOL. This should be kept in mind in the following discussion.

By repeatedly including a signature and specializing it we obtain a hierarchy of signatures. The hierarchy for the standard TCP/IP protocol stack is shown in Figure 3.

This hierarchy of signatures is not unlike a class hierarchy, though a class hierarchy may include executable code at each level, whereas a hierarchy of signatures is entirely independent of implementation. In other words, a class hierarchy is usually a hierarchy of implementation, but a hierarchy of signatures is a hierarchy of specification.

Several of the protocol signatures, e.g., ethernet and ARP, are derived directly from PROTOCOL. The signatures NETWORK_PROTOCOL and TRANSPORT_PROTOCOL are signatures for classes of protocols rather than for individual protocols. In turn, two signatures are derived

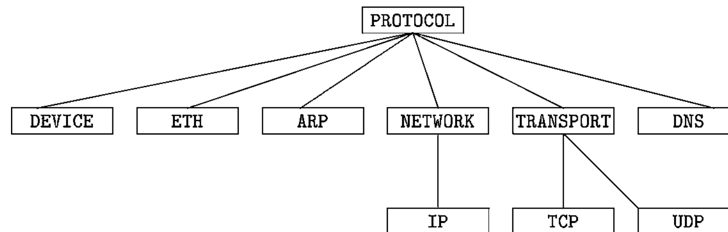

Figure 3. Hierarchy of protocol signature for TCP/IP.

from the TRANSPORT_PROTOCOL signature, and one from the NETWORK_PROTOCOL signature. The point of having such intermediate protocol signatures was mentioned above: to allow layering over protocols specifying particular operations without necessarily uniquely identifying the lower protocol.

As a further example, consider an application that is written to run unchanged over any transport protocol. If this application relies only on the TRANSPORT_PROTOCOL signature, then it can potentially run unchanged on top of either TCP or UDP. The two protocols have significant differences that are not expressed in the signature,[2] so arbitrarily layered protocols are not guaranteed to work correctly. Nonetheless, the signature does describe constraints that aid in composing only correct stacks.

The next section details our design choices in designing the PROTOCOL signature.

2.3. *The* FoxNet PROTOCOL *signature*

This section describes the development of the meta-protocol for the FoxNet. Successive figures in this section (Figures 4 to 8) show successive refinements of the PROTOCOL signature.

The simplest possible PROTOCOL signature has a send and a receive operation, as shown in Figure 4.

This signature defines two abstract types, address and data. An *abstract* type is one that is not further specified here; each type given as abstract in the protocol can be implemented by any *concrete* (that is, fully specified) type. Semantically an address identifies one or more peers in the communication, and different protocol layers will use different concrete types to instantiate this abstract type.

```
signature PROTOCOL = sig
  type address
  type data
  val send: address -> (data -> unit)
  val receive: address -> data
end
```

Figure 4. PROTOCOL signature, version 1: send and receive.

The `send` operation takes an address and returns a function specialized to send data to the given address. The `receive` operation takes an address and returns the data when the data is available. These functions correspond to the left-hand side of Figure 1.

An improvement to the signature in Figure 4 would recognize that programs often call `send` and `receive` many times for any one address. In Figure 4, `send` computes a function specialized for sending to a specific host, but `receive` does not; we would like a way to allow both `send` and `receive` to be specialized for communication with a given address. Furthermore, there is no way for either function to allocate or de-allocate cleanly any shared global storage that might be needed to coordinate with other invocations of the same function. For example a protocol providing reliable transmission must coordinate separate executions of `send` and `receive` to make sure multiple streams sent to the same host are kept distinct, and to determine when the peer has received a particular item of data.

A well-known answer to both issues is to define a `connection` type. Values of this type provide a context for communication with peers at a given address, and must be explicitly allocated and de-allocated. Such a connection refers to local state only, and is not necessarily an end-to-end connection such as TCP would provide. For example, it can be used to send IP and UDP packets. The resulting signature is shown in Figure 5.

To ensure that accesses to shared data are done correctly, `send` and `receive` now take a `connection` rather than an `address`. Values of type `connection` may or may not contain information equivalent to that carried by values of type `address`, so for example in simple situations the `connection` type could be the same type as `address`, `connect` could return the address it is given, and `send` and `receive` would be the same as for the first signature. In other cases `connect` and `disconnect` could be very complex in order to let `send` and `receive` be as simple and efficient as possible.

We want to modify the signature in Figure 5 to use upcalls to receive data, rather than explicit calls to a `receive` function. This is done by giving a *data handler* as a parameter to the connect call. The data handler is a function with the same signature as `send`, but which is passed in by the application or higher layer rather than supplied by this protocol level. When a protocol needs to deliver data for a connection, it simply calls the corresponding handler. A call to `disconnect` now not only returns any shared global data but also disables the data handler. The data handler is called by the protocol when data is available. The resulting signature is shown in Figure 6.

```
signature PROTOCOL = sig
  type address
  type data
  type connection
  val connect: address -> connection
  val send: connection * data -> unit
  val receive: connection -> data
  val disconnect: connection -> unit
end
```

Figure 5. `PROTOCOL` signature, version 2: `connect` and `disconnect`.

```
signature PROTOCOL = sig
  type address
  type data
  type connection
  val connect: address *
               {data_handler:
                    connection * data -> unit}
             -> connection
  val send: connection * data -> unit
  val disconnect: connection -> unit
end
```

Figure 6. PROTOCOL signature, version 3: Upcall.

We note at this point that the data handler, as is true for any upcall, is similar in concept to interrupt or signal handlers—the handler is called asynchronously when data becomes available, potentially interrupting the currently executing program. To support this asynchronous behavior, systems that provide upcalls have to rely on a thread mechanism to call the data handler asynchronously. The FoxNet's thread mechanism is described in Section 3.2. We also note that networking protocol implementations generally need to be multi-threaded anyway to support time-outs and retransmission [4], so providing multi-threaded support for upcalls does not significantly complicate the protocol implementation.

This signature is equivalent in many ways to the meta-protocol of the x-kernel, except that so far we have omitted any mention of mechanisms for opening connections passively, that is, in response to data received from a peer.

A further improvement can be obtained by noting that in normal operation, calls to connect and disconnect must be matched, that is, logically there must be one call to disconnect following every call to connect. Since function calls have the property that under normal circumstances[3] a return follows every call [12], connection and disconnection can be matched automatically by eliminating the function disconnect and passing to connect as an additional argument a *connection handler*, which is a function used to specify the lifetime of the connection. In other words, the connection is open exactly while the connection handler is executing. The intent is that the connection handler be called by the implementation of connect, and both it and the data handler can call send at any time during their execution. The data handler is disabled and any global resources are de-allocated once the connection handler returns and before the call to connect completes.

As shown in Figure 7, the call to connect now computes the value of type connection, installs the data handler, and calls the connection handler. Once the connection handler returns, connect disables the data handler, returns any global resources, and returns to the caller.

It can be seen that the connection type is needed (by the client of this interface) exclusively as an argument to send. Since this type is held abstract, there is nothing else the client can do with values of this type. Hence, the signature can be simplified by changing connection from an abstract type to a *type abbreviation*; a type abbreviation is a concrete type which is completely defined in a signature. In this case connection is a type for values with one element, and that element is a function named send with the given type. Since

```
signature PROTOCOL = sig
  type address
  type data
  type connection
  val connect:
     address *
     {connection_handler: connection -> unit,
      data_handler: connection * data -> unit}
    -> unit
  val send: connection * data -> unit
end
```

Figure 7. PROTOCOL signature, version 4: Matched open and close.

```
signature PROTOCOL = sig
  type address
  type data
  type connection = {send: data -> unit}
  val connect:
     address *
     {connection_handler: connection -> unit,
      data_handler: connection * data -> unit}
   -> unit
end
```

Figure 8. PROTOCOL signature, version 5: connect.

send is passed in to the connection handler and the data handler, there is no longer a need for a send function at top level in the signature, and the signature shown in Figure 8 shows connect as the only required top-level function.

We see in Figure 8 that send does not take a connection as an argument. In fact, send must be a specialized function that given a data item, somehow knows where to send it. The information provided to the connect call, and specifically at least the address of the remote host, has to be available to send in order for send to complete its task. Any implementation of send must therefore be a *higher-order function*, which allows pairing of executable code and specific values—in other words, building a *closure*. SML and other functional languages automatically and efficiently provide closures to implement such higher-order functions.

The signatures so far have been simplified for expository purposes. The full PROTOCOL signature is given in Appendix A. The major differences include the complete definition of handlers, listen to passively open connections, abort to prematurely close a connection, initialization and finalization of the protocol as a whole, extensibility of most concrete types, specification of operations on abstract types, and a set of standard exceptions for all protocols. All these differences are described in the remainder of this section, which covers detailed design decisions.

The actual definition of the connection type is shown in Figure 9. As well as send, a connection provides an abort function and an extension value. The send function is as

```
type connection = {send: outgoing -> unit,
                   abort: unit -> unit,
                   extension: connection_extension}
```

Figure 9. `Connection`.

described above except that the full signature permits the types of incoming and outgoing data to be different, and `send` transmits outgoing data. The function `abort` is called to immediately terminate the connection—`abort` is similar to `disconnect` in Figure 5 in that no data can be sent or received on this connection after the `abort` function is called. When `abort` is called, the `connect` call waits until the corresponding connection handler returns, then returns immediately. While `abort` has some of the disadvantages of the `open`/`close` mechanism shown in Figure 5, its occasional usefulness for terminating connections led us to the decision to include it.

The `extension` value is of a type held abstract in the `PROTOCOL` signature; this type may be further specified in ways that are appropriate to specific protocols, as explained in the previous section. This process is analogous to subtyping in object-oriented systems, with `PROTOCOL` analogous to the base type. We want the specific signatures to be able to extend this base type as needed. If these extensions require additional fields to be part of each connection, the connection extension can be used to define these specific fields. This was discussed in Section 2.2, using `NETWORK_PROTOCOL` as an example.

The actual signature for `connect` uses a type abbreviation called `handler`: a handler is a function from connection keys to a set of three handlers, a connection handler, a data handler, and a status handler (Figure 10). The connection key is a printable and comparable value that uniquely identifies the connection; it is provided to the handler so the handler may compute functions that are customized for the given connection. Connection handlers and data handlers were described above. The status handler is used by the protocol to communicate any necessary information that is not incoming data. For example, at least one `FoxNet` protocol implementation (TCP) uses the status handler to communicate that the peer has forcibly closed the connection.

As mentioned above, both the x-kernel and the `FoxNet` allow connections to be initiated by a peer. A call to `listen` specifies that such connections are now allowed. Like an `address`, a `pattern` describes one or more peers for communication. As in the case

```
type connection_key
type handler =
  connection_key
  -> {connection_handler: connection -> unit,
      data_handler: connection * incoming -> unit,
      status_handler: connection * status -> unit}
val connect: address * handler -> unit,
```

Figure 10. `handler` definition.

```
type listen = {stop: unit -> unit,
               extension: listen_extension}
val listen: pattern * handler * count -> listen,
```

Figure 11. `listen` definition.

```
type session =
  {connect: address * handler -> unit,
   listen: pattern * handler * count -> listen,
   extension: session_extension}
val session: setup * (session -> 'a) -> 'a
```

Figure 12. `session` definition.

of `incoming` and `outgoing` data, the two types have been separated to provide greater flexibility. The function `listen` also takes a handler which will be applied each time a connection is instantiated, and a value of type `count` which specifies how many connections will be accepted. For maximum flexibility, `listen` also returns a `stop` function which stops listening independently of the number of connections opened, and an extension value that is used in the same way as the connection extension (Figure 11).

Connections and the `connect` function are used to structure access to global data for communication with a single peer. In the same way, `session` is used to structure access to global data for an entire protocol (Figure 12). The basic idea is that a protocol as a whole may need to be initialized before first use and finalized after last use. We could provide functions `initialize` and `finalize` but since the two must be matched in the same way as `connect` and `disconnect`, we combine the two into the single function `session`. Like `connect`, `session` takes a handler called the *session handler*. The type of the session handler is `session -> 'a`. Whatever value the session handler returns at the end of the session will in turn be returned by `session`. Since `session` does nothing with that return value other than deliver it back to its own caller, `session` itself puts no constraints on the type, and the type is therefore polymorphic—any constraints are imposed by the caller of `session` or by the session handler. This polymorphic type is expressed in the signature as `'a`, pronounced "alpha".

Also, the session handler takes as parameters `connect`, `listen`, and an extension. Since `connect` and `listen` are given to the session handler, they no longer need to be defined at top level in the signature, and as a result `session` is the only function defined at top level.

Each of the abstract types that were used above—`setup`, `address`, `pattern`, `connection_key`, `incoming`, `outgoing`, `status`, and `count`—is in fact declared in a sub-structure (`SML` uses the keyword *structure* to mark a fully instantiated module, whereas a *functor* is a module that can be instantiated by providing the appropriate parameters). This allows a number of functions to be defined on the abstract type and encapsulated with the type definition in a separate module (Figure 13). For example the structures `Setup`, `Address`,

```
structure Setup: KEY
structure Address: KEY
structure Pattern: KEY
structure Connection_Key: KEY
structure Incoming: EXTERNAL
structure Outgoing: EXTERNAL
structure Status: PRINTABLE
structure Count: COUNT
```

Figure 13. PROTOCOL sub-structures.

```
signature KEY = sig
  type T
  val makestring: T -> string
  val equal: T * T -> bool
  val hash: T -> word
 end
```

Figure 14. KEY signature.

Pattern, and Connection_Key, defining the corresponding types, must all satisfy the signature KEY, shown in Figure 14.

The signature KEY is simple, defining an abstract type named T and three operations on this type. The type is named T so that references to the abstract type defined by the Address structure, for example, can simply be Address.T, a usage which is idiomatic in SML. The operations convert values of the given type to strings suitable for printing or to unsigned integral values suitable for hashing, or compare them for equality. These operations allow us to use values of type T as keys in tables, hence the name of the signature. For example, it is always possible to build a table of values indexed by the address of a peer, independently of the specific protocol which defines the address.

The signature PRINTABLE is like KEY but without equal or hash.

The signature EXTERNAL defines a number of operations on data aggregates, including copying bytes into and out of such an aggregate, appending aggregates, and allocating new aggregates.

The signature COUNT specifies that the count can be any one of an integer, the special value Unlimited, or a function which is called to determine whether future connections are allowed for a specific listen.

Finally, each protocol defines a standard set of exceptions (Figure 15). Most of these exceptions are independent of any types defined in the protocol, and are declared by a substructure. The only exception declared at top level is Already_Open (Figure 16), which carries as a value the key of the connection which cannot be opened because it already is open. Since the type of this key is defined in the top level, it would be awkward to define Already_Open in the sub-structure, and it is defined at top level.

We need to mention an older release of the FoxNet [4], which had a different PROTOCOL signature. The older signature was more conventional in only having functions at top level,

```
signature PROTOCOL_EXCEPTIONS = sig
  exception Session of string
  exception Listen of string
  exception Connection of string
  exception Send of string
  exception Receive of string
  val makestring: exn -> string option
 end
```

Figure 15. PROTOCOL_EXCEPTIONS signature.

```
structure X: PROTOCOL_EXCEPTIONS
exception Already_Open of Connection_Key.T
```

Figure 16. Exceptions.

whereas the new PROTOCOL signature takes advantage of the ability we have in SML to return functions as results.

2.4. *Protocol and stack implementation*

Each layer of the FoxNet corresponds to a protocol implementation, and each protocol implementation is parametrized by a lower-layer protocol. For example, as shown in Figure 17, the IP protocol takes as parameter a protocol satisfying the ARP_PROTOCOL signature and itself satisfies the IP_PROTOCOL signature. It is worth remembering that—since IP_PROTOCOL is derived from NETWORK_PROTOCOL which is derived from PROTOCOL—any protocol which satisfies the IP_PROTOCOL signature also automatically satisfies both NETWORK_PROTOCOL and PROTOCOL.

Within the implementation of IP, the structure Arp provides all the functions of the next lower protocol in the stack. No actual implementation of Arp is needed until the IP implementation is instantiated. At instantiation time, the only requirement will be that the lower protocol satisfy the ARP_PROTOCOL signature.

The parameters to the implementation of the TCP protocol are almost identical, except that the lower protocol (which here is named Lower rather than Arp) is constrained by the NETWORK_PROTOCOL signature (Figure 18).

```
functor Ip_Protocol
   (structure Arp: ARP_PROTOCOL): IP_PROTOCOL =
 struct
  (* actual implementation, not shown here *)
 end
```

Figure 17. IP protocol implementation header.

```
functor Tcp_Protocol
   (structure Lower: NETWORK_PROTOCOL): TCP_PROTOCOL =
 struct
  (* actual implementation, not shown here *)
 end
```

Figure 18. TCP protocol implementation header.

```
structure Dev = Dev_Protocol ();
structure Eth = Eth_Protocol (structure Dev = Dev);
structure Arp = Arp_Protocol (structure Eth = Eth);
structure Ip =
  Ip_Protocol (structure Arp = Arp);
structure Tcp =
  Tcp_Protocol (structure Lower = Ip);
structure Udp =
  Udp_Protocol (structure Lower = Ip);
```

Figure 19. TCP/IP protocol stack construction.

Once all the protocol implementations have been written and compiled, building a stack is a matter of composing individual protocols. In Figure 19 we assemble a standard TCP/IP stack given protocol implementations for IP, TCP, UDP, ARP, and Ethernet, and a "device" protocol which allows communication with the raw device. Each of these implementations except for the device is parametrized by a lower protocol.

When the statements in Figure 19 are given to an SML compiler, the compiler automatically checks that each of the argument structures satisfies the parameter signature specified in the definition. For example, the compiler checks that the parameter to the `Ip_Protocol` satisfies the `ARP_PROTOCOL` signature, and the parameters to TCP and UDP satisfy the `NETWORK_PROTOCOL` signature. The latter is verified by the compiler since at the time `Ip_Protocol` was compiled the compiler checked that `Ip_Protocol` satisfied the `IP_PROTOCOL` signature, and as described above any module satisfying `IP_PROTOCOL` automatically satisfies `NETWORK_PROTOCOL`.

After compilation of the stack construction shown in Figure 19, the stack is ready to use. The entire meta-protocol specification, protocol implementation, and stack composition is done in SML, without a need for external languages to specify how the pieces fit together. This is in part because the SML *module language*, i.e. signatures, functors, and structures, was used to structure the program. We have seen in the examples above that SML signatures are used for interface definitions, and that signatures can be structured hierarchically. SML structures are collections of types, values, and functions, and may or may not satisfy a given signature. Functors are essentially parametrized structures, and in fact yield structures when instantiated on the appropriate parameters.

Figure 20 shows the construction of a non-standard stack. In this stack, TCP is layered almost directly over Ethernet. The module given by the `Pseudo_Ip_Protocol` functor

```
structure Dev = Dev_Protocol ();
structure Eth = Eth_Protocol (structure Dev = Dev);
structure Non_Ip =
  Pseudo_Ip_Protocol (structure Eth = Eth);
structure Tcp =
  Tcp_Protocol (structure Lower = Non_Ip);
```

Figure 20. Non-standard protocol stack construction.

is a very simple protocol which satisfies the NETWORK_PROTOCOL signature. This protocol records the length of outgoing segments in a two-byte header, which is needed since Ethernet will extend any segments with a payload shorter than 46 bytes and since TCP does not record segment length. Unlike IP, this protocol is unable to reach a host that is not on the same local net, so this protocol stack can be used to communicate reliably within a single local area network.

2.5. Access control

One of the drawbacks of using conventional languages such as C or even C++ is the lack of protection against misbehaving parts of the program—the global nature of particular types of errors, for example pointer errors. One advantage of using a modular language such as SML is the controlled access to other parts of the program. For example, in the absence of errors in the compiler, in the FoxNet it is impossible for a protocol to have access to the functions of any protocol other than the one it is directly layered on top of. And no module can have access to the internal variables and functions of another module. Although applications run in the same address space as the protocol stack, this language-enforced modularity at least guarantees the correct behavior of each module. All locks, for example, are local to each module, and cannot be misused by code outside the module.

Given these statements, it is interesting to see whether the FoxNet is in some sense secure, that is, protected against malicious misuse, both from the network and from the application programs.

In brief, there are very few guarantees about the behavior of a FoxNet protocol stack as a whole. Even if there were no programming errors,[4] and even though the FoxNet does not suffer from the buffer overflow problems occasionally present in C programs, the non-preemptive scheduler, for example, can be hijacked by an application program that never yields control.[5] In addition, since an application is able to build arbitrary protocol stacks, it is relatively easy to build nonsense stacks or stacks that will not function correctly. The Connection functor, described in the next section, is carefully coded against accidental misuse, so there are as few assumptions as possible about whether applications or higher-level protocols are using the API correctly. Careful coding, combined with the safety offered by the SML module system, is very helpful in protecting against errors, but does not guarantee protection against intentional misuse.

Many operating systems define different capabilities for different “users”, with some users allowed to (having the capability to) perform certain operations, and others restricted

from performing these operations. For example, in Unix port numbers 0-1023 can only be used by programs running under the system administrator account. Since the focus of the `FoxNet` is networking rather than security, we did not address these issues, and any application is allowed to, for example, use any TCP or UDP port.

3. Basic mechanisms

This section describes a number of mechanisms that have been used in the implementation of the `FoxNet`. In particular, it describes the connection functor, the `FoxNet` implementation of coroutines, the word array data structure, and ways to marshal and unmarshal data in the `FoxNet`.

3.1. The connection functor

The protocol signature imposes certain requirements on any protocol that satisfies it. Specifically, the required functionality includes the following.

1. Session management. This is responsible for maintaining any state that is associated with a protocol as a whole, for example the routing tables of an IP protocol.
2. Connection management. This is responsible for opening and closing connections, in response to requests from both the application program (higher-level protocols) or the network (lower-level protocols).
3. Data dispatch. This is the process of delivering incoming data to the appropriate higher-level protocol. Sometimes there is only one higher-level protocol, and then this process is simple, and sometimes there are many, so we must determine which protocol the data is for.
4. Passive connection management. This allows a higher-level protocol to listen for incoming connection requests.

We want to distinguish two types of protocols: multiplexing and non-multiplexing. A non-multiplexing *upper* protocol P layered on top of a *lower* protocol Q uses a separate connection in Q for each connection in P. In contrast, a multiplexing protocol P can use a single connection in Q to support multiple connections for P. The distinction is significant: a non-multiplexing protocol generally has a much simpler implementation than an equivalent multiplexing protocol. Pseudocode for the two is shown in Figure 21, where some of the complexity of the multiplexing protocol is hidden in the calls to `identify_connection`, `receiver_table`, and in the pseudocode to try to open a new connection. A non-multiplexing protocol can often implement each function directly in terms of the corresponding functions in the lower protocol. Stateless non-multiplexing protocols can have stateless implementations. For example, a trivial protocol that simply adds a checksum to outgoing packets and verifies it for incoming packets need not maintain any state or connection information.

In contrast, a multiplexing protocol has to co-ordinate the sharing of lower connections among the connections supported by the protocol. This sharing necessarily involves global state. Since a lower connection must remain active as long as any one connection using

```
fun nonmultiplex_upcall next_higher packet =
  let val new_packet = local_processing packet
  in next_higher new_packet
  end
fun multiplex_upcall packet =
  connection_id = identify_connection (packet);
  if in_table (connection_id, connection_table) then
    let val next_higher = receiver_table connection_id
        val new_packet = local_processing packet
    in next_higher new_packet
    end
  else
    try to open a connection, or discard the packet
```

Figure 21. Pseudocode for non-multiplexing and multiplexing upcalls.

it is active, and since a connection is active only until its connection handler completes, lower connections must be opened within a thread other than the thread(s) that open the upper connection(s). Hence the implementation of any multiplexing protocol must be able to correctly synchronize multiple threads of control accessing the shared state.

The substantial complexity of a correct implementation of a multiplexing protocol has motivated our design of a single generic module to implement exactly once the session management, the active and passive connection management, and the data dispatch for multiplexing protocols. This implementation is shared among all the protocols that need it. This module is called the `Connection` functor. The module is a functor because the `SML` functor mechanism is used to parametrize the implementation over:

- the types and structures of the protocol being implemented (for example, `Address`),
- the type of the state that must be maintained for each protocol,
- functions to do protocol-dependent processing (such as converting protocol addresses to lower-protocol addresses), and
- other utilities and debugging information.

The functor header for the `Connection` functor is shown in Appendix B. This functor header lists all the types, structures, and functions that the functor takes as parameters; it is readily apparent that the parameters fall neatly under these four groups.

We now describe in some detail the implementation of two of the four major functions of the `Connection` functor: data dispatch and connection management.

3.1.1. The data handler for lower connections. The part of the `Connection` functor that is most interesting is the data delivery algorithm for received data. The data handler passed to the lower protocol is the same function for all lower connections, each time specialized using different connection information. This specialization, or partial application, is called *currying*, and is one of the ways that functional languages support higher-order functions. An example of currying is shown in Figure 22.

```
fun generic_handler connection data =
     ...
val specific_connection = ...
val specialized_handler: data -> unit =
      generic_handler specific_connection
```

Figure 22. Example of a higher-order function used to create a specialized function. The generic handler is curried, that is, applied to only one of its two arguments, yielding a single-argument specific function.

```
val identify:
 (Lower.Connection_Key.T * protocol_state)
 -> Lower.Incoming.T -> Connection_Key.T list
```

Figure 23. Signature for the function `identify`.

A data handler specific to a connection is created (instantiated) by currying the handler on some of its arguments to create a closure. This closure is then passed to the lower protocol as the data handler to be used specifically for this connection. The arguments on which the data handler is curried are the lower connection key and a set of closures. These closures are created by currying other functions, for example the functor parameter `identify`, on some of *their* arguments. The signature for `identify` is shown in Figure 23.

The function `identify` is called by the data handler to select one or more appropriate connections for an incoming packet.

The first argument to `identify` is the one over which `identify` is curried. In theory, the fact that `identify` is applied to some of its arguments once and the resulting curried function is applied many times allows some computation to be performed only once and the resulting function to execute more efficiently. In practice the last argument (the packet) is needed in order to do any of the computation, so it has not been possible to achieve substantial optimization via currying the `identify` function.

The curried data handler is called by the lower protocol and takes two arguments: a (lower protocol) connection, including a specialized send function to send on the lower connection, and a unit of data, a packet. The handler first applies `identify` to this packet and receives back a list of connection keys identifying potential connections that this packet could be for. If the list is empty, the packet is discarded. If the list holds one or more connection keys, the first one is used as an index into a table listing all connections supported by this protocol. One table is maintained for each lower connection, and the lower connection argument to the data handler is used to identify this table.

A connection can be in one of three states: active, pending, and inactive.

Active connections can send and receive data. If the connection is active, the data handler for the connection is identified and the data is delivered.

Pending connections are connections that have been created, but are not yet ready to receive data. If the connection is pending, the data is queued in the per-connection queue.

Inactive connections are connections that are not in a state to send or receive data. If the connection is inactive, the algorithm first checks to see whether the key matches one

of the pending passive connection requests ("listen" operations). If so, the connection is instantiated and the packet queued for delivery. Otherwise, the remainder of the list returned by `identify` is examined for other possible keys.

Systems programs must often handle many different cases, but in order to be efficient, must carefully optimize the common case. For receiving data, the common case is that the connection is already established, i.e., active. An optimization we made to the common case is to avoid locking shared data. The remainder of this section explains why locking is not necessary in this implementation.

If the connection is active, the algorithm reads shared state (the connection table) but does not modify it. Since the `FoxNet` uses non-preemptive co-routines for multi-threading (see Section 3.2) and the control path for active connections does not yield control, in the common case data can be delivered without explicit locking and synchronization. This is true if the call to `identify` returns without yielding control to other threads, since the call to `identify` is part of the common case. Note that `identify` inspects the data received and returns a list of keys that can be used to find the connection to which this packet can be delivered, if any—`identify` is generally stateless, that is, has no knowledge of which connections have been established. The function `identify` is passed as a parameter to the functor, so the `Connection` functor has no information about whether or not `identify` will yield control to other threads before completing. In case `identify` does not yield, the lack of locking does not cause any problem. In the `FoxNet` protocols, none of the `identify` functions yields control.

In case `identify` does yield control to other coroutines, we must verify that the `Connection` functor still functions correctly. The only consequence in the current implementation is that packets might be delivered out of order. We consider this consequence acceptable. Unacceptable behavior would be for global or connection state data to become corrupted, or to have packets queued forever causing a memory leak. We know that the state will not be corrupted because:

- in the common case, the data handler does not modify the global state.
- in all other cases (e.g., before instantiating a pending passive connection request), the data handler synchronizes (locks) before modifying the shared global state, so even if `identify` suspends, the locks will allow for synchronized access.

In the common case, packets will be delivered as soon as the call to `identify` completes, so there is no risk of memory leaks. In the other cases, again, explicit synchronization takes care of avoiding possible memory leaks due to unexpected state changes while processing.

3.1.2. Connection management. The most complex task for the `Connection` functor is the connection management. This is due to a number of factors:

- Connections can be opened actively by the higher-layer protocol, or passively as a result of receiving data.
- A lower connection is opened by a thread other than the thread that requests the opening of the upper connection. This requires synchronization, since the upper connection must be placed in the "pending" state until the lower connection is open.

– A new connection may need to use a pre-existing lower connection or may have to open a new lower connection.
– When a session is closed, all connections opened within that session must also be closed.
– Any exceptions raised by a connection handler, data handler, or status handler must be handled correctly.
– As soon as a connection handler exits, no more data may be delivered to the corresponding data handler.

These requirements make the implementation of this part of the `Connection` functor challenging, and in fact multiple bugs due to insufficient or incorrect synchronization were found in the connection management code during development. As is often the case with synchronization problems, finding and reproducing any given bug was often a much more demanding task than fixing it. SML does provide protection against some kinds of type errors, pointer errors, and memory misuse, but provides no special protection against synchronization errors. In this case, since all our protocols used the same code, we had more chances to observe the bugs and, once the problem was fixed, it was fixed for all our code—a strong argument for a modular implementation, which was made possible by the amount of parametrization permitted by SML.

3.2. Coroutines

The FoxNet uses a non-preemptive multithreading (coroutine) package written for the most part in SML (a system call is used to obtain the system time, everything else is in SML). Following Wand [29], we use continuations to implement coroutines. Continuations are implemented very efficiently in SML/NJ [1, 2], with a typical continuation transfer of control taking only as much time as calling a function.

Figure 24 gives the highlights of the coroutine signature used in the FoxNet. A call to `fork` executes a given function as a separate thread. Calling `exit` terminates the current thread and starts execution of some other thread, or raises `No_Ready_Thread` if there are no threads left to execute. In the current implementation `fork` suspends the child (forked)

```
signature COROUTINE = sig
  val reset: unit -> unit
  exception No_Ready_Thread
  val fork: (unit -> unit) -> unit
  val exit: unit -> 'a
  exception No_Such_Suspension
  type 'a suspension
  val suspend: ('a suspension -> 'b) -> 'a
  val resume: 'a suspension * 'a -> unit
  val sleep: int -> unit
 end
```

Figure 24. FoxNet COROUTINE signature.

thread and continues execution of the parent thread, but this is not explicitly specified by the signature, so it is good programming practice not to rely on this property.

A `suspension` is an abstraction of a single coroutine (thread). A running coroutine may suspend itself; the argument to `suspend` is a function which is applied to the suspension of the running coroutine. This function typically either calls `resume` immediately, or stores the suspension into some global state for future use. The two functions `suspend` and `resume` together allow the implementation of both a `yield` function and of subsidiary schedulers. The function `yield` can be implemented by `suspend (fn s => resume (s, ()))`. This suspends the thread and immediately requeues it for execution, but all other threads queued for execution get a chance to run before this thread is executed again. A subsidiary scheduler could define its own "suspend" by calling this suspend routine, storing the suspension as needed, and perhaps resuming other suspensions. `Resume` is like `fork` in that the calling coroutine continues execution, and the resumed coroutine is queued for execution. Again, it is bad practice for users of the coroutine package to rely on these implementation details.

Finally, `sleep` sleeps at least the specified number of milliseconds, then gets queued for execution again. Milliseconds were used because the granularity is sufficiently fine and the range sufficiently large for every use we have encountered in networking protocols.

As mentioned above, the use of continuations to implement coroutines is well-established. There is however one subtlety in the implementation of coroutines which some of the authors have published elsewhere [6], namely the interaction between exceptions and continuations.[6] We will briefly describe this problem as follows. For purposes of illustration, consider the following implementation for `fork`:

```
fun fork f =
     if callcc (fn c => (enqueue c; false))
     then (f (); exit ())
     else ()
```

In this implementation, `fork` obtains the current continuation, puts it on the queue, and then immediately returns, so the calling thread continues immediately. Once the queued continuation reaches the front of the queue, the scheduler will throw the value `true` to this continuation, so that the `then` branch of the conditional will be executed and the argument to `fork` (i.e., the child code) will be executed. After its execution is complete, `exit` is called to schedule the next available thread.

Now consider what would happen if the function `f` raises an exception. The call to `fork` would have "returned" twice, once immediately after forking, and once with the exception. This means the continuation of `fork` would be executed more than once, which could be very confusing. The intuitive thing to do is to wrap the call to `f` in an exception handler:

```
fun fork f =
     if callcc (fn c => (enqueue c; false))
     then (f () handle _ => (); exit ())
     else ()
```

This will correctly terminate any thread that ends in an exception and immediately schedule the next available thread. Let us assume for the time being that `exit` goes into an infinite loop if no thread is available, and hence never returns (this will be relaxed later).

Now consider what happens if the thread explicitly calls `exit`. The call to `exit` will schedule new threads and at any rate never return. However, the compiler and runtime system are unaware of this,[7] and hence the garbage collector will be unable to determine that the stack of the thread can now be reclaimed. If there is a constant number of threads in the system but at least some of them call `exit` explicitly (after forking other threads to keep the total number constant) then the total amount of space in the system will grow without bound, since the garbage collector will fail to collect the continuations of all the terminated threads. In this case, the exception handler in the scheduler has caused a memory leak. We observed this memory leak in actual trials with earlier versions of the `FoxNet`.

There are several possible solutions to this problem, none of them completely satisfactory. One solution is to allow `fork` to return should the forked thread raise an exception. This places responsibility for handling thread exceptions on the programmer of the thread, but poses the same storage leak risks should the programmer do the "natural" thing and try to handle any exceptions the thread might raise.

A better solution is to have the scheduler (the function `exit`) be a continuation rather than a function, as follows:

```
fun fork f =
     if callcc (fn c => (enqueue c; false))
     then (f () handle _ => ();
           throw exit_continuation ())
     else ()
```

Since there is no regular continuation (i.e., no return) from a call to `throw`, and even no exception continuation, the compiler and runtime system know that the continuation can be discarded and garbage collected. This is the algorithm currently used within the `FoxNet` coroutine package.

This definition of `fork` requires that we establish a continuation, called `exit_continuation`, to which we can transfer control. A continuation is the control context of the running program at the point at which `callcc` is invoked. In contrast, the `exit_continuation` is logically a continuation that is separate from and independent of the running program. The `exit_continuation` can almost be thought of as a *constant continuation*. However, since all continuations in `SML/NJ` are obtained from `callcc`, there are a number of choices for selecting the continuation to be used as the `exit_continuation`:

- Obtain a continuation at functor instantiation time. This is elegant, but the continuation will have references to the control context of the compiler, which may be large and take up unnecessary unclaimable storage.
- Obtain a continuation the first time `fork` or `exit` is called. This requires a conditional in `fork` or `exit` (to see if the continuation exists), and slows down the scheduler.
- Use a separate function (such as the function `doit` in CML [21]) to bracket the entire use of threads, and use its continuation.

The last suggestion is certainly the most elegant, since it does not require a distinct main thread. Unfortunately, the model does not work well with the way the `FoxNet` is commonly used, since every application would then need to call the bracketing function or be called within the bracketing function. The current version of the `FoxNet` uses the second option.

It should be noted that other `SML` threads systems we have looked at, including ML-threads [9] and CML [21] suffer from the space leak described here. We conjecture that the problem has not been detected because of the lack of long-running applications written using these systems. It must be emphasized that the problem is independent of whether the threads system is preemptive, and instead is due specifically to the fork call trying to do the "right thing" when the forked function raises an exception, in combination with `exit` being explicitly callable by threads. More details are available in a publication [6] co-authored by two of the authors of this paper.

3.3. Word arrays

Systems programming and network programming often have to be concerned with efficiently moving data around. In a protocol stack, data must be transferred across protocol layers, meanwhile adding protocol headers to outgoing data and removing headers from incoming data. For efficiency, all this must be done without copying the data. For networking, a given collection of data can variously be viewed as:

– an ordered sequence of bytes (for example for error checking)—a *byte array*
– an addressable "chunk of memory" (for buffering)—a *memory buffer*
– a collection of words (for efficient transmission to the device)—a *word array*
– an ordered collection of words of different sizes (for header decoding)—a *structured type*

In what follows, we generally use "byte array" to refer to the data type provided by `SML`—currently this is referred to as `Word8Array`—and "word array" to refer to the data type described in this section.

One effective and well-known abstraction for this issue of multiple views of the same data is provided by `C` pointers; this abstraction offers efficient word-wise and byte-wise random and sequential access to arbitrary data (using casts for structured data). The problem with `C` pointers is that they are unsafe, and in fact problems in `C` programs are often due to inadvertent misuse of pointers.

A safe language such as `SML` does not provide pointers. The implementation we use augments the language with the built-in type `Word8Array`, with operations `sub` and `update` which allow safe reading and writing of individual bytes in the array. This interface provides the basic functionality needed for data manipulation but is insufficient for fast access to data.

Fast access requires word-wise access to data and as few bounds checks as possible. When looping over an `SML/NJ` byte array, typically we have to have a conditional to check for the end of the array and a statement to access the data; this is equivalent to two bounds checks per byte or word accessed, one for the end of the array and the second to make sure the word accessed is within bounds. It is easy to reduce this to a single check if the `Subscript` exception is used to detect out-of-bounds accesses, but for short arrays the

```
val new: ('a -> (element * 'a) option) -> 'a -> T
val create: element * size -> T
val tabulate: (index -> element) * size -> T
val create_uninitialized: size -> T
```

Figure 25. Functions for array creation.

overhead of handling the exception might be significant, and if more than one array is being looped over, the out-of-bounds condition for the different arrays may need to be handled differently. Finally, the need for arithmetic when performing sequential access on arrays is generally error-prone and specifically allows fence-post (off-by-one) errors.

Because of these shortcomings, we have defined and use a new abstraction that provides all of the following:

– All accesses are safe, i.e., it is impossible to access beyond the bounds of the array.
– Automatic initialization of newly created arrays is optional and is supported through a variety of functions, shown in Figure 25.
– As with pointers, only one check is required in each iteration of a loop.
– Accesses are provided for word sizes of 8, 16, 32, 64, 128, and 256 bits.
– Little- and big-endian accesses are supported for all multi-byte sizes.

Figure 25 includes a function `create_uninitialized`. Whereas `SML` makes it impossible to create uninitialized values, this function allows the creation of "uninitialized" arrays, that is, arrays of bytes where each byte may have an arbitrary byte value. This could in theory be used to create non-deterministic programs (programs that behave differently when run on different systems), which may arguably be called "unsafe".[8] However, non-determinism is possible whenever a program has access to the time of day, and we do not believe that non-determinism necessarily leads to lack of safety. Since any byte value in memory is a valid byte value, creating an uninitialized byte array is quite safe in that it does not provide the program with any means of (accidentally or otherwise) subverting the run-time system. For this reason, and because it allows somewhat higher performance, we provide and use this function. No part of the `FoxNet` code makes use of this potential non-determinism.

Access to the elements of an array is mainly through the `next` and `update` functions. Typical styles of iteration over arrays, comparing conventional byte arrays to `Word_Arrays`, are shown in Figure 26.

The function `next` is modeled after the `SML` pattern matching access to lists, and `update` is a corresponding way of changing the contents of an array. Both functions return the "rest of the array", if there is any such, or the special value `NONE` if the end of the array has been reached.

Random access is provided by the `seek` function.

Arrays are of two different types, `T` which can only be read, and `U` which can only be written. These are two separate types since `next` and `update` have different invariants, corresponding to different ways of testing for the end of the array:

– `next` must first test for the end of the array, and return the next item only if the array is not finished.

```
(* with conventional arrays: *)
fun loop (a, i, result) =
  (* 1st call to Array.length, 1st comparison *)
  if i >= Array.length a then
    return result
  else
  (* bounds check: 2nd call to Array.length,
     2nd comparison *)
    loop (a, i + 1, something with Array.sub (a, i))
(* with word arrays: *)
fun loop (NONE, result) = result
  | loop (SOME (first, a), result) =
    (* bounds check: only one call to Array.length,
       only one comparison *)
    loop (Word_Array.next a, something with first)
```

Figure 26. Typical loop over an array, with conventional arrays and with word arrays.

– `update` guarantees it can store at least one element into the array, so it first stores the element (without checking), then returns a result appropriate to whether the remaining array is empty.

Why have different types for reading and writing? Consider having the same array type for both operations. For `next`, the same check that verifies that the array is non-empty also verifies that the operation succeeded, and also returns the new element. For `update`, the check is not functionally required, since `update` does not return a useful new element, so we could have a "silent failure". This is a common and often a severe problem with many C libraries, and it seems unnecessary to encourage such programming practices in SML. One alternative is to require two operations: a check before writing, and another operation to give us the "rest of the array". This is expensive. The only way we have found to have a single check per array element is to pay the price at the beginning of a loop, by converting the readable type to an updateable type which guarantees that an update operation will succeed. If the return value from `update` (the rest of the array) is not used, there is no need to check it. If the return value from `update` is needed, it will necessarily be checked to make sure it is non-empty, and the loop will terminate if it is empty.

The signatures for these functions are shown in Figure 27.

```
val next: T -> (element * T) option
val seek: T * index -> T

val update: U * element -> U option

val write: T -> U option
val read: U -> T
```

Figure 27. Functions for array creation.

The functions read and write convert between the two types.

The conversion of a U (write-only) array to a T (read-only array), using read, always succeeds, since a U always has at least one element and a T has zero or more elements. Converting a T to a U will succeed only if the array has at least one element. If the array is empty, write will return NONE instead of a new array.

With next or update, arrays can only grow smaller, not larger. That is, we can use these functions to hide part of the data and give to a function only a portion of the array, with the guarantee that the function will not be able to reach "outside" the portion it was given.

To support many different styles of access to word arrays, each of these functions is provided by a number of different structures providing all the different combinations of the following:

- Word size. All power-of-two sizes between 8 and 256 bits are supported.
- Aligned or unaligned access. On most architectures aligned access is faster than unaligned access, and aligned arrays provide such access without needing to check whether a pointer is aligned. Aligned arrays are of three types, aligned at one or the other of the two ends or at both ends. The SML type system guarantees that a function that requires alignment will only take aligned arrays as arguments.
- Forward or backward access. All arrays are accessible equally from the front or the back, except as restricted by alignment. Arrays that are only guaranteed to be aligned at one end only support access at the aligned end.
- Big-endian, little-endian, and native-endian access.

The Word_Array structure provides all of these sub-structures and functions to convert between different types of arrays. The top-level signature for Word_Array is shown in Appendix C.

One of the motivations for word arrays is efficiency. Unfortunately, as seen in Section 4, our implementation of word arrays fails to live up to its promise. Whereas the definition of word arrays would allow very efficient implementation if the arrays were implemented directly by the compiler, we have chosen not to modify the compiler.

The optimal implementation of word arrays would use two machine addresses, one to the first and one to the last element of the array. Checking that there is an element means comparing the two pointers. Accessing the element means dereferencing the appropriate pointer and incrementing it or decrementing it. Since these operations are extremely efficient on current hardware, such an implementation (which would require compiler modifications) is quite efficient. Some of the obstacles to implementing word arrays directly in the compiler include having to reserve two separate registers for each word array value, and modifying the garbage collector to preserve word arrays that have no pointer to the beginning, but may have pointers to the middle of the array. Garbage collectors exist that can handle this. This notably includes conservative garbage collectors for languages such as C. There is no reason why such a garbage collector has to be conservative, though the overhead would be greater than for the current SML/NJ garbage collector.

```
signature EXTERN = sig
  type T
  type extern_in
  type extern_out
  type cursor
  val size: T -> int
  exception Extern
  val marshal: (extern_out * T) -> (cursor -> cursor)
  val unmarshal: (extern_in * cursor) -> (T * cursor)
 end
```

Figure 28. EXTERN signature.

3.4. Marshaling and unmarshaling

Word arrays are used throughout the FoxNet to store incoming and outgoing data. Most protocol layers add a header to outgoing data, and strip a header from incoming data. The header must be written and read as a sequence of bytes, using a specified byte ordering for multi-byte fields. For example, big-endian ordering is standard for TCP/IP. Converting an internal representation to such a linearized representation is called marshaling. Reading a header from a linearized representation and constructing the equivalent internal representation is called unmarshaling. For uniformity and convenience, we have defined a generic signature, EXTERN (shown in Figure 28), satisfied by many marshaling and unmarshaling structures in the FoxNet.

Modules satisfying the signature EXTERN use T for the type of the internal representation, and extern_out and extern_in as the types of the linearized representations used when marshaling and unmarshaling respectively. The linearized representation is assumed to be indexable or addressable, and the type of such an index or address is the type cursor.

The marshal function is designed to be composed: partial application of the function to the external and to the internal representations yields a new function which accepts a cursor, linearizes the internal representation at the given index, and returns a new cursor for the next function to use. This use of marshal is shown in Figure 29.

It would be nice to be able to compose unmarshal functions in the same way that we compose marshal functions. A marshal function curried on its first argument yields a function from cursors to cursors which can be further composed with other curried marshaling functions. The same is not possible with unmarshal. unmarshal needs to

```
fun marshal (array, self, peer, proto) =
     (Word16X.marshal (array, proto) o
      Word48X.marshal (array, self) o
      Word48X.marshal (array, peer))
```

Figure 29. Ethernet composition of marshal functions to produce a new marshaling function. Word16X and Word48X are local structures created by instantiating functors that provide 16-bit and 48-bit big-endian marshaling and unmarshaling. The result of calling marshal is a new function which takes as argument a cursor, and returns a new cursor corresponding to the position after the header has been marshaled.

return both a value of type T and a new cursor. We have been unable to find a simple mechanism in functional languages to express that a function should return two values, one to a function we are composed with and one to a different expression. Such a mechanism would somehow be equivalent to "currying" function return values (regular currying curries function parameters). We know of no mechanism to do this. Continuations can be used to return multiple values, but are unwieldy for the purpose, are hard to use, and require the use of global state. The simpler solution which we have adopted is to not allow unmarshal functions to be composed.

Sun XDR [24] (eXternal Data Representation) is a widely used language for describing data formats to be sent over communication channels, and specifically was designed for data to be sent using the RPC [25] protocol. Data format descriptions are compiled to a collection of marshaling and unmarshaling procedures [20] which are invoked to transform data from the "internal" representation, which is system- and compiler-dependent, to a standard "external" representation which can be exchanged among systems. Some differences between XDR and the marshaling system described here include:

- the XDR language is a data description language rather than a programming language. This means its expressiveness is limited specifically to describing data layout using a restricted set of formats.
- type-checking and safety in XDR is limited to what can be provided within the context of C programming, whereas for the FoxNet the compiler provides the usual strong safety guarantees provided for all SML programs.
- XDR requires an external "compiler" to convert it to C code, whereas in the FoxNet the same compiler that compiles the marshaling and unmarshaling functors also compiles the remainder of the code.

In addition to marshaling and unmarshaling basic types, XDR provides functions to marshal structured types. We have done this for the FoxNet as well: a functor takes, as parameters, structures to marshal specific component types, and results in a structure which will marshal structured types such as tuples or arrays whose components have the given types. This is shown for arrays in Figure 30.

The Array_Extern functor takes as parameters two structures, both of which satisfy the EXTERN signature: the first marshals integers, the second marshals array elements. The two

```
functor Array_Extern
  (structure Int: INT_EXTERN
   structure Element: EXTERN
     sharing type Element.extern_in = Int.extern_in
         and type Element.extern_out = Int.extern_out
         and type Element.cursor = Int.cursor
  ): EXTERN =
 struct
  type T = Element.T Array.array
...
```

Figure 30. Array marshaling functor.

```
structure Int_Extern =
  Int_Extern (type extern_in = in_type,
              type extern_out = out_type);
structure Int_Array_Extern =
  Array_Extern (structure Int = Int_Extern
                structure Element = Int_Extern);
structure Int_AArray_Extern =
  Array_Extern (structure Int = Int_Extern
                structure Element = Int_Array_Extern);
```

Figure 31. Using an array marshaling functor to marshal arrays of arrays.

```
functor Tuple2_Extern
  (structure Elt_1: EXTERN
   structure Elt_2: EXTERN
     sharing type Elt_1.extern_in = Elt_2.extern_in
         and type Elt_1.extern_out = Elt_2.extern_out
         and type Elt_1.cursor = Elt_2.cursor
  ): EXTERN =
 struct
  type T = Elt_1.T * Elt_2.T
...
```

Figure 32. Functor header for a two-element tuple marshaling functor.

structures must have the same types for `extern_in`, `extern_out`, and cursors: this is called *sharing* these types. The integer marshaling structure is used to encode the length of the array, and the element marshaling structure to encode each of the elements.

We can use this functor to instantiate a structure `S` that will marshal, for example, arrays of integers. `S` can then be used as the `Element` parameter in a new instantiation of this functor, giving us a new structure that can marshal arrays of arrays of integers. This example is shown in Figure 31.

We also have functors to marshal tuples. Figure 32 shows the header of a functor to marshal two-element tuples (pairs). As is the case with arrays, this functor is parametrized over the types of its elements and the types of the argument structures must share.

These building blocks can be composed to give marshaling and unmarshaling structures for almost any `SML` type (i.e., except function, datatype, and continuation types), including user-defined types. The compiler checks that the type `T` provided by the marshaling module matches the type to be marshaled or unmarshaled and hence prevents misuse of the marshaling and unmarshaling operations.

4. Performance measurement

One of the goals of the `Fox` project has always been to develop high-quality software that is competitive with production software. The high quality is reflected not only in the modularity and ease of maintenance, but should be seen in the overall performance

as well. One advantage of building standard software protocol stacks is that there is an abundance of competing software, with similar functionality, that we can compare against.

4.1. Code sizes

One frequent question about the `FoxNet` is how much code is in our stack compared to code in other stacks implementing the same protocol or almost the same protocols. The use of line counts to estimate code size or complexity is one with many pitfalls, especially when comparing programs with different structures, somewhat different purposes, possibly very different levels in the quality and completeness of the implementation, and different coding styles. There is also no widely accepted measure of either code complexity or code quality, and no known way to relate the two. In the absence of other easily computed metrics, we report in Table 1 our measurements of the lines of code in both the x-kernel (version 3.3.1) and the `FoxNet`.

We have used a script to count lines. For the `FoxNet` we counted the lines in `.sig` and `.fun` files, for the x-kernel the count includes the `.h` and `.c` files. Source code lines are non-blank lines that contain more than just a comment, comment lines include only comments. We have not counted the lines of code in the "glue" sections (`.str` files in the `FoxNet`, template files in the x-kernel).

In spite of the similarity in the goals and overall design of the two projects, it is hard to draw meaningful conclusions. The only number that stands out is the substantially larger size of the `FoxNet` IP protocol implementation, probably because the IP module in the `FoxNet` contains functionality which the x-kernel distributes among other protocol. Other protocol implementations are roughly comparable, which may suggest that both languages result in approximately the same code size for implementations of the same networking protocol. The overall project size for the x-kernel includes custom protocols that are not part of the standard TCP/IP stack, a much larger thread package than our scheduler, a substantial simulator, and porting packages to run the x-kernel over several different operating systems.

Table 1. Lines of code.

	`FoxNet`			x-kernel		
	Total	Code	Comments	Total	Code	Comments
project	75091	39370	16631	180222	110647	46382
`.sig/.h`	13907	3786	4993	44203	26756	11029
`.fun/.c`	61184	35584	11638	136019	83891	35353
Protocols	37798	20709	6903	31453	20352	6749
TCP	8191	4550	1542	5506	3207	1687
IP	9488	5866	1279	2969	1804	760
ARP	1491	809	207	1574	1046	306

4.2. Test environment

We have measured the performance of the FoxNet on two identical Intel Pentium II 266 MHz systems with 512 K of level-2 cache and 128 MB of memory. Both systems were running Linux 2.0.36 and were on an otherwise isolated, hub-connected 10 Mb/s ethernet connecting the two test systems and a third, mostly idle computer (the control computer). For the SML code, we compiled with SML/NJ version 110.9.1 (Flint version 1.41) dated October 19, 1998. In order to gain access to the network, we used the unsafe features of SML/NJ to access the Linux raw device. These extensions allow the FoxNet to access the Ethernet directly from a user process.

The tests are based on ttcp version 1.12, compiled with -O and with version 2.7.2.3 of gcc (the same gcc was used for all other C programs in this sections). We wrote a simple SML program that approximates the tests performed by ttcp,[9] in other words, measures the time to send a fixed amount of data over the FoxNet TCP. We note that most of the protocol stack for the standard ttcp is in the kernel, whereas for the FoxNet the entire protocol stack is in user space. This means that for ttcp a context switch occurs for every buffer sent by the application, for the FoxNet a context switch occurs for every packet sent by the lowest layer of the protocol stack.

4.3. Test results

Our first two tests measure basic network parameters: latency (for small packets) and bandwidth (when sending larger amounts of data).

We used ping to measure latency. Out of 100 packets, each 56 bytes long, sent using the Linux version of ping and echoed by the Linux on the other test machine, we had zero packet loss and 0.3 ms (3×10^{-4} seconds) minimum and average round-trip time, with 0.4 ms maximum. Using the FoxNet version of ping and using the FoxNet on the other machine to reply to the echo packets, we had zero packet loss and 1 ms (10^{-3} seconds) minimum, average, and maximum round-trip time. We note that the granularity of the clock in the FoxNet version of ping is 1 ms.

We tested throughput by sending 1 MB of data in one direction over TCP. The measurement shows the throughput measured when sending a single payload size of 1,048,576 bytes (2^{20} bytes). In each case we ran one test in each direction between the two machines, in each case getting results within 10% of the performance in the opposite direction. The average of each pair of tests is given in Table 2.

These measurements show that our goal, of producing systems software with performance comparable to that of carefully optimized production software, has been met.

Table 2. FoxNet throughput measurements. Throughput is in Mb/s.

Test	Linux native ttcp	FoxNet	FoxNet no checksum
Ethernet	7.1	6.5	6.4

In analyzing the performance, we wondered whether the checksum computation might require substantial time—this has been suggested by Derby [11]. From our measurements no such effect can be seen.[10] The difference in performance with checksums on or off was not significant. The system load was observed using the Unix/Linux `uptime` command during a larger bulk transfer, and there was no detectable difference between the two versions (with and without TCP checksums) of the `FoxNet`, though the load did appear to be higher than with `ttcp`.

The remainder of this section follows Derby [11] in analyzing in detail the costs of the TCP checksum computation.

4.4. Small functions

Derby has claimed [11] that the overhead of calling a function is a substantial and overly large component of the cost of executing small functions. This claim is based on a comparison of the performance of two equivalent functions, `checksum` and `inline_checksum` (reproduced, and corrected, in Figure 33. The code for the `fold` function called as `Word_Array.W32.Little.F.fold` is shown in Figure 34). The results of our measurements are shown in Table 3, and show that inlining does indeed produce some speedup.

```
fun check_one(new, accumulator) =
  Word32.+ (Word32.+ (Word32.>> (new, 0w16),
                      Word32.andb (new, 0wxffff)),
            accumulator)
fun checksum buffer =
  Word_Array.W32.Little.F.fold check_one 0w0 buffer
fun i_check_all(byteBuffer, first, last) =
  let
   fun loop(index, accumulator) =
     if index > last then accumulator
     else
      loop(index + 1,
           let val new =
                Pack32Little.subArr(byteBuffer, index)
           in Word32.+ (Word32.+
                        (Word32.>> (new, 0w16),
                         Word32.andb (new, max32)),
                        accumulator)
           end)
  in loop(first, 0w0)
  end
fun inline_checksum buffer =
   i_check_all(buffer, 0,
               (Word8Array.length buffer - 1) div 4)
```

Figure 33. Equivalent functions for computing the Internet checksum. The second function only takes about 85% of the time of the first.

Table 3. Performance change with inlined small function.

	10^6 bytes	10^7 bytes
Modular version	178 ms	1.8 s
Inlined version	150 ms	1.6 s
Ratio	84%	89%

```
fun fold f init (A data, first, last) =
  let fun loop (index, value) =
        if index > last then value
        else
         loop (index + 1,
               f (Pack32Little.subArr (data, index),
                  value))
  in loop (first, init)
  end
```

Figure 34. The implementation of `Word_Array.W32.Little.F.fold`.

In the `FoxNet`, the need to call unknown functions arises from the desire to maintain modularity. The `Word_Array` data structure should be independent of the applications that use it, and therefore provides only a generic looping operation (`fold`) rather than a specific, optimized Internet checksum operation. `SML` supports this modularity by allowing us to pass a specific function to `fold` (in other cases we pass to `fold` functions that are partially instantiated, or curried). This modularity makes the code easier to maintain, but is somewhat less efficient than a mechanism which would provide direct inlining.

4.5. Foreign memory

Any systems program must, as part of its function, access memory structures either from hardware devices, or defined by programs written in other languages. This is a task at which `C` excels—the efficient pointer mechanism and the lack of bounds checks all contribute to this ability. Derby has performed tests to compare the performance of the `SML` inlined checksum computation and an equivalent `C` version to similar code that performs the same computation without accessing the contents of an array. The "no-access" code does the same arithmetic on the value of a global variable. Since the variable is global, neither the `C` nor the `SML` compilers should be able to optimize away the reference and the resulting computation, though we have not actually verified this.

The code for the `SML` version of this computation is shown in Figure 33. The code for the corresponding `C` computation is shown in Figure 35, and the equivalent no-access test code in Figure 36.

The performance of these codes is shown in Table 4.

The `Word_Array` structure (Section 3.3) was designed to be implemented as a built-in data structure—a more general, elegant, and efficient version of the currently standard

Table 4. Time (in milliseconds) for the 4 versions of the checksum code to checksum 10^6 (one million) bytes.

Language	Array access	No array access	Ratio
SML	150 ms	20 ms	13%
C	3.8 ms	2.3 ms	61%

```
unsigned int inlineChecksum (unsigned int * byteBuffer,
                             int first, int last){
  int index = first;
  unsigned int sum = 0;
  while (index <= last ){
    unsigned int w32 = byteBuffer[index];
    index += 1;
    sum += ( (w32>>16) + (w32&0xFFFF) );
  }
  return sum;
}
```

Figure 35. Reference C checksum code. This code is about 40 times faster than the equivalent SML code.

```
val dummy = ref 0
fun i_check_all (byteBuffer, first, last) =
  let fun loop (index, accumulator) =
       if index > last then accumulator
       else
        loop(index + 1,
             let val new = ! dummy
             in
              Word32.+(Word32.+(Word32.>> (new, 0w16),
                                Word32.andb (new, max32)),
                      accumulator)
             end)
  in loop(first, 0w0)
  end
unsigned int dummy = 0;
unsigned int inlineChecksum (unsigned int * byteBuffer,
                             int first, int last){
  int index = first;
  unsigned int sum = 0;
  while (index <= last ){
    unsigned int w32 = dummy;
    index += 1;
    sum += ( (w32>>16) + (w32&0xFFFF) );
  }
  return sum;
}
```

Figure 36. SML and C checksum code with array references removed. The SML code executes in about 15% of the time of the original. The C code executes in about 60% of the time of the original.

Word8Array. Word_Array could be more efficient than Word8Array if directly supported by the compiler, since Word_Array would only require a single bounds check per loop iteration instead of the two required when using Word8Array. Using only safe user code,[11] however, we have been unable to implement Word_Array as efficiently as we know it could be implemented.

4.6. Related performance measurements

Derby has reported in detail on the performance of the FoxNet. Unfortunately, we find that the report lacks some of the information that would be essential for an adequate understanding and repeatable testing of the data presented. We also find that some of the data itself strains our credulity under any reasonable assumptions about the missing context. In spite of repeated attempts, we have been unable to personally contact the author of the report. Where possible, the results described above have been obtained by following Derby's stated measurement methodology, but running the tests ourselves.

Some of our results differ significantly from those obtained by Derby, while others confirm his results. Specifically:

- we were unable to confirm Derby's claim that the checksum computation adds substantial overhead to TCP.
- we were able to confirm that the system load when running the FoxNet throughput test is substantially higher than when running ttcp.
- we were not able to confirm Derby's result that inlining the checksum computation produces a 62% speedup—our inlining only gave about a 10%–15% speedup. We did confirm that the SML function call is expensive compared to C.
- the cost we measured for memory access differed from Derby's, but again we were able to confirm that the current SML/NJ implementation of word access primitives will substantially affect the performance of any memory-intensive program.

The large numerical discrepancy between our measurements and those reported by Derby, especially for inlining functions, may simply be due to the different choices of architectures and compiler versions. The version of the compiler that we used has been designed to be more aggressive at inlining functions, even unknown functions, than the version used by Derby.

5. Evaluation

The FoxNet is a significant SML project. By one count, the sources include over 50,000 lines of code and comments, with contributions from at least half a dozen programmers over several years. The FoxNet is significant in other ways as well. By using an advanced functional language to implement code normally written in C, we have helped to advance the state of the art in language design, compiler implementation, and systems programming, contributing to or helping to inspire many other projects.

In this section, we take stock of the specific features of our system which are novel or unusual compared to pre-existing systems. We are especially interested in interactions between language features and the FoxNet implementation.

5.1. *Features of* SML *that support the* FoxNet *implementation*

In this section we describe the features of SML that were helpful in implementing the FoxNet: the modules language, strict typing, continuations, exceptions, and garbage collection.

5.1.1. Modules. As can be seen starting in Section 2.3, the FoxNet relies heavily on the SML modules language for structuring and organizing the code. The SML modules provide us with unparalleled flexibility in composing the code building blocks while preserving all the safety, error-checking, and compile-time consistency checking of the SML language. While subsets of these features are available for other languages,[12] as far as we are aware only the dialects of ML provide all of these.

Our focus on using the modules language made us wish for features that are not present in SML/NJ. One is the ability to generate specialized code for a functor at functor application time (when the parameters are supplied to the functor) instead of generic code produced at functor definition time (before the parameters are available to the functor). Currently, SML/NJ generates code for each functor when the functor is compiled. Internally, the result is similar to a polymorphic function, and the code generated is correspondingly generic. This strategy is effective when each functor is applied many times, since it yields substantial code savings. The alternative, which we are advocating here, is to not generate machine code until the functor is applied to its arguments. This happens at compile time, so code generation is straightforward. The advantage of generating the code at functor application time is that more types are known and therefore the code can be specialized more, and hence can be more efficient, than code generated at functor definition time. SML/NJ will not generate code at functor application time, in part because in extreme cases it can lead to code explosion. We conjecture that our performance (and most likely that of many other SML programs) might improve if we had such a feature. Perhaps what is needed is simply a mechanism to let the programmer specify for each functor whether code should be produced at functor compilation or functor application time.

5.1.2. Strict Typing. In general, the strict typing of SML worked in our favor. We came to take it almost for granted that even after making a pervasive change in a large number of modules, fixing the compilation errors shown by the compiler would give us a correctly running system. While it is remotely possible that this is due to the programmers' skills, the experience of these same programmers with large and complex C programs has been quite different, suggesting that the strict typing and safety of the language contributed substantially to coding productivity.

There is exactly one point at which the strict typing got in our way. A careful study of the PROTOCOL signature in Appendix A shows that both the connection_handler and the connect call return a value of type unit. In our original design, the connection handler would be polymorphic, returning an arbitrary value which is returned by the corresponding

call to connect. The corresponding types would be:

```
type 'a handler =
  Connection_Key.T
  -> connection_handler: connection -> 'a,
      ...
type session =
  connect: Address.T * 'a handler -> 'a,
   ...
```

Unfortunately, the SML type system is not sufficiently powerful to express this type.[13] The need is for "deep" or "nested" polymorphism, which SML does not support.

In all other cases, the SML type system was entirely adequate to the task, and coupled with type inference that is mostly automatic, we found it both convenient and helpful.

One only needs to consider the many type casts required in any systems program (and in most application programs) written in C to appreciate the significance of the FoxNet compiling entirely within the SML type system.

The lack of type casts required us to copy data when marshaling and unmarshaling headers. Since many C programs do this anyway, for example when putting headers and data into a standard byte order, it is not clear that our type-safe solution is necessarily any worse than the unsafe solutions that are used with other languages. As described in Section 4.5, in practice we had performance issues for word and byte accesses that were caused by the poor performance of memory access operations in the specific version of SML/NJ that we used.

5.1.3. Continuations and exceptions. Two features of SML that we used heavily were continuations and exceptions.

Continuations were used to implement co-operative multi-threading (multi-tasking). Our entire scheduler, except for obtaining the system time (needed to wake up sleeping threads), is written in SML. Continuations are not part of the standard SML language, and are provided as one of the SML/NJ extensions.

We note that the Hello project at the University of Hawai'i [14] ported the FoxNet to run on a bare machine. The system clock was maintained by SML code, so in Hello the entire scheduler was written in SML.

In the FoxNet, exceptions are used to report unusual conditions. Since SML and SML/NJ provide flawless specification and implementation of exceptions and exception handling, we were generally happy with our use of exceptions. Our only difficulty was sometimes identifying which particular piece of code raised a particular exception. This difficulty was compounded by the size and multi-threaded nature of the FoxNet. We are pleased to note that later versions of SML/NJ now report the site at which an unhandled exception (technically, an exception handled by the compiler's interactive front end) was raised. Unfortunately, this is of limited usefulness in a systems program that must continue to execute, and therefore must under every circumstance handle any and all exceptions generated in the code. Short of an exhaustive analysis of which pieces of code can generate what exceptions,[14] our recourse was the clumsy one of adopting a programming style whereby "error" uses of raise are

encapsulated with an identifying print statement. This leads to occasionally undesirable printing (and occasionally mystifying exceptions), again acceptable for a research project but unacceptable in a production environment. We emphasize that this problem only arises because the SML exception mechanism is extremely useful for systems programming.

One way of ameliorating this problem would be to provide a debugger for SML/NJ that could keep track of where exceptions are raised. The versions of SML/NJ we used in developing the FoxNet has no debugger, and since the stack frame format is different from the native stack frame format on the system, it is not possible to use a generic debugger to debug a program compiled with SML/NJ. Interestingly, the lack of debugger, while annoying, was rarely a major nuisance. Since we only used safe programming constructs, we never had pointer errors or undetected out-of-bounds accesses, never corrupted our stack or accidentally overwrote unrelated data structures. The consistency of our data being guaranteed by the SML implementation, print statements were generally sufficient to debug the FoxNet. Nonetheless, though we have shown that it is possible to debug SML programs without a debugger, we are happy to report that a more recent version of SML/NJ, version 110.29, has at least a rudimentary stack tracing utility.

5.1.4. Garbage collector. A final feature we found consistently helpful was the garbage collector. Now that Java has swept the world, garbage collection is much more acceptable than it was in 1993, when we started implementing the FoxNet. Moreover, even now garbage collectors are regarded suspiciously for real-time and near-real-time programs. A network protocol stack is a near-real-time program: performance suffers if response is not within a predictable time. Since garbage collection can occur at any time, response time is not accurately predictable and the argument can be made that this will affect performance.

Again, while we are unable to prove that there is no such effect, our measurements do not show substantial performance penalty from using a garbage collector. One reason is that most of the pauses of the SML/NJ garbage collector are short, on the order of a few milliseconds. Another reason may be that even in a more conventional operating system, response time is not guaranteed—other, higher priority events and interrupts may cause unexpected delays.

If the costs of garbage collection are uncertain, the benefits are very clear. Relatively little programmer effort is spent managing memory, and complex sharing schemes where many different modules can refer to the same data are no harder than having a single module own each piece of data—the latter being almost necessary in writing reliable C programs.

5.2. *Limitations of* SML

5.2.1. Memory leaks and synchronization. It would be nice to believe that garbage collection would prevent memory leaks. Memory leaks occur when a long-running program allocates memory that is no longer used but never returned to the heap. Unfortunately, the garbage collector only succeeds in preventing *most* memory leaks.

One of our most interesting memory leaks [6] was caused by our use of continuations to implement threads. This is described in detail in Section 3.2.

As mentioned in Section 3.1.2, SML is similar to most programming languages in offering no special protection against synchronization errors.

5.2.2. 32-bit integers. Another feature that was missing from SML when we began the project was any kind of 32-bit integer type. SML/NJ has 31-bit integers, since one of the 32 bits of a word (on 32-bit architectures) allows the garbage collector to distinguish integers from pointers. This works for pointers since the low-order bits of a word pointer on a byte-addressable architecture are not significant and all SML/NJ pointers are word pointers. Since the Internet protocols often require 32-bit storage and 32-bit arithmetic, one of the features we added to SML/NJ was support for 32-bit signed and unsigned integers. These types have since been added to the SML standard, but in SML/NJ are still implemented relatively inefficiently: in memory, 32-bit integers are always represented by a pointer to a 32-bit word containing the integer. This boxed representation would not be necessary with a different garbage collector, such as some of the conservative garbage collectors available for C. Since the selection of a garbage collection algorithm is a complex process involving many tradeoffs in both space and time, it may well be that this is the best that can be done for SML, but certainly any improvement would improve the performance of SML/NJ for typical systems programs.

5.2.3. Byte arrays. As we mentioned in Section 3.3, the whole concept of byte arrays (or `Word8Arrays`, as they are currently referred to in SML) is somewhat primitive. The concept of Word Arrays is that of an elegant, safe, and efficient pointer. Byte arrays are inefficient because typically one range check must be done in the program to check whether we've reached the end of the array, and another in the (compiler generated) array access code to verify that the index is within bounds. The alternative of not explicitly checking for the end of the array in the program and instead handling the `Subscript` exception is clumsy and error prone, and requires the use of global reference variables to return results of an inner loop. Word Arrays as currently defined and implemented are overly complex and inefficient, but there is no reason why a compiler should not be able to provide an efficient, elegant, and safe implementation for safe pointers. Some of the requirements of such a system include:

- pattern matching for dereference, analogous to lists and streams, and returning a distinct value when the end of the array is reached
- allowing a pointer to the interior of an array
- efficient implementation
- if it is possible to specify a standard layout for SML structured types (tuples, records), allowing a pointer to the interior of such a type

We believe that at least the first three requirements are possible. An optimal implementation of word arrays is described at the end of Section 3.3.

5.3. *Limitations of the* FoxNet *implementation*

Our choice to implement the FoxNet as a user-space application program accessing data via raw device interfaces imposes overheads that are hard to quantify and measure, and makes it harder to argue that we are truly in a head-to-head comparison when we compare against

a kernel-resident implementation. The Hello project at the University of Hawai'i [14] has addressed this issue.

Our choice to use non-preemptive threads has simplified the implementation, but also imposed some burdens, especially on the application programmer—applications must be thread-aware and yield when they are not being productive.

Finally, considerably more work is needed before we can believe the FoxNet is as finely tuned as a production system, and be comfortable that the performance is as good as it can be within the limitations of the current (and constantly evolving) SML/NJ system.

5.4. *Related work*

The Ensemble project [15] has also developed communications software in ML. Similarities with the FoxNet include the use of the ML language, careful attention paid to structuring and efficiency, and dealing with multiple threads of control. Differences include using a different dialect of ML (CAML instead of SML), run-time protocol composition, and the focus of the project being more on protocol design than on protocol implementation—Ensemble did not attempt to implement standard protocols. It can be argued that at least some of the efficiency they claim comes from their unconventional coding of protocol headers, which was not an option in the FoxNet.

The Express project's goals are to explore the interaction between advanced programming languages, operating systems, and compilers, and develop the software technology to make advanced programming languages practical, useful tools for systems programming [27]. The project to date has focused more on enabling technology than on building actual systems. There has been a paper on new ways to express concurrency [26] and the project has participated in building the Flux OS kit [13]. All these activities are relevant to systems building, but none of them compare directly to the activity of building a running system and evaluating its performance.

The Spin project [3] has developed an operating system using another type-safe, strongly-typed language, Modula-3. The focus in Spin has been on building the operating system rather than the networking protocols, and the Spin OS is a stand-alone operating system. Some of the challenges faced in Spin, for example access to hardware resources from a safe language, were bypassed in building the FoxNet (which runs in user space), whereas other challenges, such as structuring the overall system, were emphasized a lot more in the FoxNet than in Spin. In spite of both being type-safe and strongly typed,[15] the languages differ substantially. Modula-3 is not a functional language, and both the module constructs and the compilation of Modula-3 are fairly conventional. In contrast, SML and specifically SML/NJ provide higher-order functions, a powerful module language, and continuation passing compilation. The latter results in fast continuation creation at the expense of more frequent garbage collections. All these language differences, and the Spin focus on having a running, stand-alone operating system, still lead the conclusion that it is possible and often profitable to use advanced languages for systems programming.

The Prolac project [16] has developed implementations of networking protocols, most notably TCP, in an advanced modular object-oriented language. The Prolac project is similar to the FoxNet in a number of respects, including the fact that the Prolac language is type safe.

The Prolac language differs from SML in being object-oriented, in being directly linkable with C (which allows the Prolac TCP to run in the Linux kernel), and perhaps in being better optimized for performance, most especially by inlining. The report mentions the FoxNet and states that the FoxNet is not built for protocol extensibility—which says more about the focus of the Prolac project than about the extensibility of the FoxNet. Since Prolac and Spin were initiated after the FoxNet, it is gratifying to think that the FoxNet may have contributed to inspiring these projects.

Marshaling and unmarshaling, described in Section 3.4, is similar to what in the systems world is often done by Sun XDR [24] (eXternal Data Representation), a widely used language for describing data formats to be sent over communication channels. Data format descriptions are compiled to collection of marshaling and unmarshaling procedures which are invoked to transform data from the “internal” representation, which is system- and compiler-dependent, to a standard “external” representation which can be exchanged among systems. Some differences between XDR and the marshaling system are described in Section 3.4.

Several optimizing compilers for XDR are available, including the Universal Stub Compiler [20].

Also notable are the optimizations introduced by the Tempo Partial Evaluator [28], which can automatically optimize the Sun RPC code. The marshaling and unmarshaling code is regular and suitable for automatic partial evaluation. The speedup from applying Tempo to this code was up to 3.7 times faster. The optimizations introduced by Tempo are not unlike the optimizations we tried to introduce by currying the marshaling function, but apparently considerably more effective, perhaps in part because the code produced by Tempo does not need to be as modular as the original source code.

6. Concluding remarks

6.1. Future work

The runtime system of SML/NJ is written in C. Of this runtime, the largest single component is undoubtedly the garbage collector. Having this garbage collector in C is currently necessary, as SML and SML/NJ provide no efficient mechanisms for manipulating memory in the ways required of a garbage collector.[16] Implementing the garbage collector in SML would not only substantially shrink the runtime, it would make it easier to provide the kind of modularity and invariants that a garbage collector needs as much as any other complex program.

Another promising avenue to explore is the further modularization of the protocol stack. The FoxNet essentially has one module for each of the protocols in the TCP/IP stack, for example, TCP, IP, ARP, DNS. There is some reason for believing that even these individual protocols need not be implemented monolithically, and can instead be broken up into smaller functional blocks. IP, for example, might have one section devoted to fragmentation and reassembly, a different sub-protocol devoted to header checksums, and so on. While both we and others have spent time looking at this issue, there is undoubtedly a more elegant and interesting solution still waiting to be developed.

Finally, preliminary work by Fu [14], in collaboration with the first author, has succeeded in porting the `FoxNet` to run directly on bare hardware, eliminating the requirement for a user space implementation and for communicating with the system to access the raw device. This also requires writing device drivers and dealing with true concurrency, but is a very interesting and exciting avenue of future work.

6.2. Conclusions

We set out to design and implement a networking system using an advanced programming language, using modularity and structure wherever possible, relying on strict typing, and rigorous adhering to safe programming practices. The resulting system is modular and composable, can be maintained fairly painlessly, and has the performance one would expect of a research system. We have demonstrated that `SML` is an adequate programming language for systems programming, and have listed a number of features that we believe would improve the language's overall usefulness. We have also showed how we used the features of this advanced programming languages to improve the implementation of such stock systems programs as schedulers and networking protocols.

Appendix

A. PROTOCOL signature

```
signature PROTOCOL =
 sig
  structure Setup: KEY
  structure Address: KEY
  structure Pattern: KEY
  structure Connection_Key: KEY
  structure Incoming: EXTERNAL
  structure Outgoing: EXTERNAL
  structure Status: PRINTABLE
  structure Count: COUNT
  structure X: PROTOCOL_EXCEPTIONS
  exception Already_Open of Connection_Key.T
  type connection_extension
  type listen_extension
  type session_extension
  type connection = {send: Outgoing.T -> unit,
                     abort: unit -> unit,
                     extension: connection_extension}
  type listen =
    {stop: unit -> unit, extension: listen_extension}
  type handler =
    Connection_Key.T
```

```
    -> {connection_handler: connection -> unit,
        data_handler: connection * Incoming.T -> unit,
        status_handler: connection * Status.T -> unit}
  type session =
    {connect: Address.T * handler -> unit,
     listen: Pattern.T * handler * Count.T -> listen,
     extension: session_extension}
  val session: Setup.T * (session -> 'a) -> 'a
 end
```

B. Parameters for the connection functor

```
functor Connection
   (structure Lower: PROTOCOL
    (* types and structures of the protocol that this
       instantiation implements *)
    structure Setup: KEY
    structure Address: KEY
    structure Pattern: KEY
    structure Connection_Key: KEY
    structure Incoming: EXTERNAL
    structure Outgoing: EXTERNAL
    structure Status: PRINTABLE
    structure Count: COUNT
    structure X: PROTOCOL_EXCEPTIONS

    (* the types of the state that must
       be maintained for this protocol. *)
    type connection_extension
    type listen_extension
    type session_extension
    type connection_state
    type protocol_state

    (* functions to do
       protocol-dependent processing *)
    val lower_setup: Setup.T -> Lower.Setup.T

    val init_proto:
          Setup.T * Lower.session
          * (Connection_Key.T * Status.T -> unit)
          -> (protocol_state * session_extension)

    val fin_proto: protocol_state -> unit
```

```
val resolve: protocol_state * Address.T
           -> Lower.Address.T option

val make_key:
   protocol_state * Address.T
   * Lower.Connection_Key.T
   * {conns: unit -> Connection_Key.T list,
      listens: unit
           -> (Pattern.T * listen_extension) list}
   -> Connection_Key.T

val map_pattern:
   protocol_state * Pattern.T
   * {conns: unit -> Connection_Key.T list,
      listens: unit -> (Pattern.T
                        * listen_extension) list}
   -> (listen_extension * Lower.Pattern.T) option

val match: protocol_state * Pattern.T
         * listen_extension * Connection_Key.T
         -> bool

val init_connection:
      protocol_state * Connection_Key.T
      * Lower.connection
      -> connection_state * connection_extension

val fin_connection: connection_state -> unit

val send: Connection_Key.T * connection_state
        -> Outgoing.T -> Lower.Outgoing.T list

val identify: Lower.Connection_Key.T
              * protocol_state
             -> Lower.Incoming.T
             -> Connection_Key.T list

val receive: Connection_Key.T * connection_state
           -> Lower.Incoming.T
           -> Incoming.T option

val undelivered: Lower.Connection_Key.T
                 * protocol_state
                -> (Lower.connection
```

```
                        * Lower.Incoming.T)
                    -> unit

    val lower_status: protocol_state
                     * Lower.Connection_Key.T
                     -> Lower.Status.T -> unit

    (* miscellaneous utilities and debugging *)
    structure B: FOX_BASIS
    val module_name: string
    val debug_level: int ref option): PROTOCOL =
  (* actual implementation, not shown here *)
end
```

C. WORD_ARRAY signature

```
signature WORD_ARRAY = sig
  type T
  structure W8  : BYTE_ACCESS_ARRAY
  structure W16 : BYTE_ACCESS_ARRAY
  structure W32 : BYTE_ACCESS_ARRAY
  structure W64 : BYTE_ACCESS_ARRAY
  structure W128: BYTE_ACCESS_ARRAY
  structure W256: BYTE_ACCESS_ARRAY
   sharing type W8.element   = Word8.word
       and type W16.element  = Word16.word
       and type W32.element  = Word32.word
       and type W64.element  = Word64.word
       and type W128.element = Word128.word
       and type W256.element = Word256.word
  val from8 : W8.T -> T
  val from16 : W16.T -> T
  val from32 : W32.T -> T
  val from64 : W64.T -> T
  val from128: W128.T -> T
  val from256: W256.T -> T
  val to8 : T -> W8.T
  val to16 : T -> W16.T
  val to32 : T -> W32.T
  val to64 : T -> W64.T
  val to128: T -> W128.T
  val to256: T -> W256.T
  val alignment_f: T -> Word.word
  val alignment_r: T -> Word.word
end
```

Acknowledgments

It is impossible to adequately acknowledge the contributions made to the Fox project by all of the following individuals: Perry Cheng, Herb Derby, Mootaz Elnozahy, Robby Findler, Guangrui Fu, Brian Milnes, J. Gregory Morrisett, Eliot Moss, George Necula, Chris Stone, David Tarditi, Daniel Wang. We also want to acknowledge the many other individuals not directly involved with the project, who nonetheless contributed to its success. This includes reviewers of this and other reports, who must remain anonymous.

Notes

1. An Internet *socket* is identified by TCP source and destination port number as well as by IP source and destination number. In order to identify the socket corresponding to an incoming packet, an implementation of TCP must be able to figure out the IP number of the sender of each data packet.
2. For example, TCP does not preserve segment boundaries, and UDP does not guarantee in-order transmission.
3. If a program uses explicit continuations or multiple threads, calls and returns may not be matched.
4. Most programs have programming errors.
5. Early versions of the Mac Operating System had no pre-emption and could also suffer from programs not yielding control.
6. Part of the reason for this oversight is undoubtedly that much of this work has been done in Scheme, which lacks an established exception mechanism. These remarks apply to the version of SML/NJ that we used, and may no longer be true, though as noted elsewhere, it is hard to see how to get around this problem in any reasonable compiler.
7. This is true for every implementation of SML that the authors are familiar with. It is even hard to imagine a compiler that could automatically deduce this.
8. There is also a security issue, since uninitialized arrays could contain data that should remain private. See also Section 2.5.
9. The entire SML code for our test, as well as the source to ttcp, is available from the author's web site [5].
10. TCP checksums are not optional, so this protocol is non-standard and was only used for performance testing. IP checksums were computed as usual.
11. To avoid "cheating" and to preserve the safety benefits of using SML, we have restricted the FoxNet to only use safe features of SML/NJ. This has been relaxed in one place in the Linux version of the FoxNet, where unsafe features are used to access the raw device.
12. Java, for example, provides safety and error- and consistency-checking, but not at compile time. At any rate, our project began before Java was available.
13. The compiler could be changed to accept this type, but a new type system and a new type theory would be needed to discriminate against other, unsafe types.
14. Java now supports specifications of which methods can generate what exceptions, but not automatically and very conservatively—programmers must manually specify all the exceptions a method can raise, and Java only checks the consistency of the specification.
15. Modula-3 provides optional unsafe modules.
16. There is also the challenge of garbage collecting the garbage collector itself. For a stop-and-copy garbage collector, for example, this could be done very efficiently by allocating a space specifically for the collector at the beginning of a run. At the end of the run, this entire space can be reclaimed.

References

1. Appel, A. *Compiling with Continuations*. Cambridge University Press, Cambridge, 1992.
2. Appel, A. and MacQueen, D. Standard ML of New Jersey. In *Third International Symposium on Programming Languages Implementation and Logic Programming*, J. Maluszynski and M. Wirsing (Eds.). New York, 1991, pp. 1–13.

3. Bershad, B., Chambers, C., Eggers, S., Maeda, C., McNamee, D., Pardyak, P., Savage, S., and Sirer, E. SPIN—An extensible microkernel for application-specific operating system services. In *SIGOPS 1994 European Workshop*. Dagstuhl, Germany, 1994.
4. Biagioni, E. A structured TCP in Standard ML. In *Proceedings, 1994 SIGCOMM Conference*, London, UK, 1994, pp. 36–45.
5. Biagioni, E., Cline, K., Haines, N., and Milnes, B. FoxNet performance test code. Available at http://www.ics.hawaii.edu/~esb/prof/foxtest200004.tar.gz.
6. Biagioni, E., Cline, K., Lee, P., Okasaki, C., and Stone, C. Safe-for-space threads in Standard ML. *Higher-Order and Symbolic Computation*, **11**(2) (1998).
7. Biagioni, E., Harper, R., Lee, P., and Milnes, B. Signatures for a network protocol stack—a systems application of Standard ML. In *1994 ACM Conference on Lisp and Functional Programming*, Orlando, FL, 1994.
8. Clark, D. The structuring of systems using upcalls. In *Proceedings of the 10th SOSP*, Orcas Island, Washington, 1985, pp. 171–180.
9. Cooper, E. and Morrisett, J.G. Adding threads to Standard ML. Technical Report CMU-CS-90-186, Carnegie Mellon University, 1990.
10. Day, J.D. and Zimmerman, H. The OSI reference model. In *Proceedings of the IEEE*, **71**(12), (1983) 1334–1340.
11. Derby, H. The performance of FoxNet 2.0. Technical Report CMU-CS-99-137, School of Computer Science, Carnegie Mellon University, 1999.
12. Dijkstra, E.W. Recursive programming. In *Numerische Mathematik*. 1960, pp. 312–318.
13. Ford, B., Back, G., Benson, G., Lepreau, J., Lin, A., and Shivers, O. The Flux OSKit: A substrate for kernel and language research. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP-16)*, Saint-Malo, France.
14. Fu, G. Design and implementation of an operating system in Standard ML. Master's Thesis, University of Hawai'i at Mānoa, 1999. Available at http://www.ics.hawaii.edu/~esb/prof/proj/hello/guangrui/thesis/.
15. Hayden, M. The Ensemble system. Ph.D. Thesis, Cornell University, 1998. Available at http://simon.cs.cornell.edu/Info/People/hayden/thesis.ps.
16. Kohler, E., Kaashoek, M.F., and Montgomery, D. A readable TCP in the Prolac protocol language. In *SIGCOMM*, 1999, pp. 3–13.
17. Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.
18. Milner, R., Tofte, M., Harper, R., and MacQueen, D. *The Definition of Standard ML(Revised)*. MIT Press, 1997.
19. O'Malley, S. and Peterson, L. A dynamic network architecture. *ACM Transactions on Computer Systems*, **10**(2) (1992).
20. O'Malley, S., Proebsting, T., and Montz, A.B. USC: A universal stub compiler. In *Proceedings, 1994 SIGCOMM Conference*, London (UK), 1994, pp. 295–306.
21. Reppy, J.H. CML: A higher-order concurrent language. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*.
22. RFC 0791 Internet protocol. Information Sciences Institute, USC, 1981.
23. RFC 0793 Transmission Control Protocol. Information Sciences Institute, USC, 1981.
24. RFC 1014 XDR: External data representation standard. Sun Microsystems, Inc. 1987.
25. RFC 1057 RPC: Remote procedure call protocol specification, Version 2. Sun Microsystems, Inc, 1988.
26. Shivers, O. Virtualisable threads. In *Proceedings of the Second ACM SIGPLAN Workshop on Continuations*, Jan 1997, Paris, France, O. Danvy (Ed.) vol. 2, pp. 1–15. Technical Report BRICS-NS-96-13, University of Aarhus.
27. Shivers, O. The Express Project, 2000. Available at http://www.ai.mit.edu/projects/express/.
28. Thibault, S., Consel, C., Lawall, J.L., Marlet, R., and Muller G. Static and dynamic program compilation by interpreter specialization. *Higher-Order and Symbolic Computation*, **13**(3) (2000).
29. Wand, M. Continuation-based multiprocessing. *Higher-Order and Symbolic Computation*, **12**(3) 1999, 285–299. Reprinted from the proceedings of the 1980 Lisp Conference.