A safe interface to sockets
Draft

John H. Reppy
AT&T Research
Murray Hill, NJ

May 28, 1996

1 Introduction

One of the arguments for using a higher-level language, such as Standard ML (SML), is the greater degree of static checking provided by the language's type system compared with more traditional systems programming languages. For example, all I/O descriptors in C are the same type (i.e., `int`), which can lead to mismatches between operations and the file descriptors that they are applied to. While the operating system will catch such type errors at run-time, it is much better to detect them statically at compile-time than to leave them to be caught during testing or after deployment in the field. Type checking is a weak form of program verification, but it has the key advantage that it is exhaustive and it scales to large systems.

When designing SML versions of standard systems APIs,[1] we should take advantage of the more powerful type system provided by the language, and provide as much static type security as possible. At the same time, we must be careful not to make the API too restrictive. It is important that the common patterns of usage found in C be allowed.

This paper describes the design of a type-safe sockets API for SML, which has been implemented as part of the *Standard ML of New Jersey* (SML/NJ) system. The interface exploits a non-standard use of polymorphism to provide flexible type safety. Before presenting the final design, we present two earlier attempts at designing a sockets API for SML: the first of these fails to provide adequate type safety, while the second lacks flexibility. It is hoped that this exposition will aid the design of other system programming APIs for SML. The interface is part of the *Standard ML Standard Library*, which is being developed as a collaboration between a number of SML implementations [GE96].

2 Sockets

Sockets are an abstraction for inter-process communication (IPC) that were introduced as part of the Berkeley version of UNIX in 1982. They have become a de facto standard for network communication, and are supported by most major operating systems (including PC

[1] *Application program interface.*

systems). Providing a complete API for sockets is an important requirement for a language to be useful for real-world system's programming.

In this section, we give a brief overview of Berkeley sockets, which should be sufficient for understanding the main points of the paper. For a more thorough treatment of sockets and network programming, there are many references on the subject: Stevens describes the use of sockets in UNIX systems [Ste90], while the socket programming interface for Microsoft Windows is described in [HTA+93] and [QS95].

The sockets API supports two styles of IPC over an abstraction of the underlying network protocol:[2] *stream* sockets provide *virtual circuits* between pairs of processes, and *datagram* sockets provide *connectionless* packet-based communication. In stream-based interaction, the server allocates a master socket that is used to accept connections from clients. The server then listens on the master socket for connection requests from clients; each request is allocated a new socket that the server uses to communicate with that particular client. As the name suggests, stream-based communication is done as a stream of bytes, not as discrete packets. Connectionless communication is more symmetric: messages are sent to a specific port at a specific address. While datagram sockets provide better performance, messages may be lost or received out of order, which requires additional programming by the client.[3]

A summary of the socket operations is given in Table 1; the second column notes which type of socket the operation expects (stream or datagram). In addition to these basic

Table 1: Summary of socket operations

Operation	**Socket type**	**Description**
`socket`	Both	*Create a socket*
`bind`	Both	*Bind a socket to a network address*
`listen`	Stream	*Enable a socket for listening*
`accept`	Stream	*Listen for client connections*
`connect`	Stream	*Connect to a server*
`recv`	Stream	*Read data*
`send`	Stream	*Write data*
`recvfrom`	Datagram	*Read a datagram*
`sendto`	Datagram	*Write a datagram*
`close`	Both	*Close a socket*
`shutdown`	Stream	*Shutdown a connection*

operations, the sockets API also supports a large collection of operations for controlling the behavior of sockets, but we do not discuss them here.

The standard C API for sockets provides little type security, and there are a number of "type" errors that can arise:

1. One of the most common errors is getting the byte-order (little-endian vs. big-endian) of the socket addresses wrong. When the network byte-order, which is defined to be

[2] All implementations of sockets support the TCP/IP and UDP protocols, but others are often supported.
[3] There are reliable datagram sockets, but the are not commonly supported.

big-endian, agrees with the machine byte-order these errors do not show up during testing, but will cause problems when the application is ported to a little-endian machine.

2. There are conflicts between operations on sockets and operations on other kinds of I/O descriptors. This is because all UNIX I/O descriptors are represented by the same type (`int`).

3. There are conflicts between operations on stream sockets and operations on datagram sockets.

4. There are conflicts between operations on a server's master socket and operations on connected sockets.

5. There are conflicts between operations that are specific to particular network protocols.

We show in the sequel that these errors can be prevented by the proper API design, without loss of programmer flexibility.

3 A first attempt

The simplest approach to an SML sockets API is a direct translation of the C API, but this does not address any of the problems mentioned above. A better approach is to introduce abstract types for the various types in the API, such as sockets and addresses. This interface is given in Figure 1. By making socket addresses into an abstract type (`sock_addr`), we

```
type addr_family
type sock_type
type sock
type sock_addr

type in_flags = {peek : bool, oob : bool}
type out_flags = {don't_route : bool, oob : bool}

val socket   : (addr_family * sock_type) -> sock
val bind     : (sock * sock_addr) -> unit
val listen   : (sock * int) -> unit
val accept   : sock -> (sock * sock_addr)
val connect  : (sock * sock_addr) -> unit
val recv     : (sock * in_flags) -> vector
val send     : (sock * vector * out_flags) -> int
val recvFrom : (sock * in_flags) -> (vector * sock_addr)
val sendTo   : (sock * sock_addr * vector * out_flags) -> int
val close    : sock -> unit
val shutdown : (sock * shutdown_mode) -> unit
```

Figure 1: A monomorphic sockets interface

protect the programmer from byteorder errors and promote portability. This interface also ensures that sockets are not confused with other types of I/O descriptors.

While this interface provides more type security than found in the C API, it is quite easy to misuse the operations. For example, the `connect` operation is only meaningful on `STREAM` sockets, but there is nothing in the interfaces to prevent a program from attempting a `connect` on a `DGRAM` socket. Furthermore, socket addresses have different formats depending on their domain, but our interface does not place any constraints on them. While these errors are detected at run-time by the operating system, it goes against the spirit of ML to rely on dynamic type checking.

4 Improving the safety of the interface

One standard approach that we can use to improve the type security of the sockets API is to introduce different types for the different kinds of sockets and socket addresses. We can use the SML module system to organize the API into a collection of modules with common interfaces, but different types. Figure 2 gives such an interface. The structure `InSock` defines internet domain sockets, while UNIX domain sockets are defined by `UnSock`. The four socket types (`InSock.DGram.sock`, etc.) are different, which ensures type safety. In the case of addresses, each top-level structure (`InSock` and `UnSock`) constrains the `sock_addr` types in its substructures to be the same. The address family and socket type arguments to the socket creation functions are implicit in the containing module.

While this approach provides sufficient type security, it has several disadvantages. First, there are many different instances of the same operation (e.g., two different `bind` operations per supported domain). This flys in the face of one of the design goals of Berkeley sockets: namely, providing a common interface to different underlying network transport layers [LMKQ89]. Furthermore, this design loses some necessary flexibilty. For example, some applications, such as the X Window System Protocol [Nye90], are based on a particular socket type (e.g., stream or datagram), but support multiple domains (e.g., both UNIX and internet). Because the socket I/O operations are specific to a particular socket type, a given application of an I/O operation is restricted to sockets of a specific type and domain. Writing code that could dynamically support multiple domains would require introducing new abstraction layers, and would cause significant inconvenience for programmers.

It is possible to avoid this problem by introducing additional types and coercions between the various types, but this would greatly increase the size of an already large API, and would be cumbersome to program with. In the next section, we present a different approach, which is based on a technique we call *constrained polymorphism*.

5 A safe and flexible interface

What we need is a restricted kind of polymorphism. Suppose that we had the types:

```
type in_addr          (* INet domain addresses *)
type in_strm_sock     (* INet domain stream sockets (TCP) *)
type in_dgram_sock    (* INet domain datagram sockets (UDP) *)
type un_addr          (* Unix domain addresses *)
type un_strm_sock     (* Unix domain stream sockets *)
type un_dgram_sock    (* Unix domain datagram sockets *)
```

```
signature SOCKET_BASE =
  sig
    type sock
    type sock_addr
    type in_flags = {peek : bool, oob : bool}
    type out_flags = {don't_route : bool, oob : bool}
    val socket : unit -> sock
    val bind   : (sock * sock_addr) -> unit
    val close  : sock -> unit
  end

signature DGRAM_SOCKET =
  sig
    include SOCKET_BASE
    val recvFrom : (sock * in_flags) -> (vector * sock_addr)
    val sendTo   : (sock * sock_addr * vector * out_flags) -> int
  end

signature STREAM_SOCKET =
  sig
    include SOCKET_BASE
    val connect  : sock * sock_addr -> unit
    val listen   : (sock * int) -> unit
    val accept   : sock -> (sock * sock_addr)
    val shutdown : (sock * shutdown_mode) -> unit
    val recv     : (sock * in_flags) -> vector
    val send     : (sock * vector * out_flags) -> int
  end

structure InSock : sig
    structure DGram : DGRAM_SOCKET
    structure Strm : STREAM_SOCKET
    sharing type DGram.sock_addr = Strm.sock_addr
    val addr : (inet_addr * port) -> DGram.sock_addr
  end = ...

structure UnSock : sig
    structure DGram : sig
        include DGRAM_SOCKET
        val sockPair : unit -> (sock * sock)
      end
    structure Strm : sig
        include STREAM_SOCKET
        val sockPair : unit -> (sock * sock)
      end
    sharing type DGram.sock_addr = Strm.sock_addr
    val addr : string -> DGram.sock_addr
  end = ...
```

Figure 2: The module-based interface

Then we want to constrain the type of **accept** to be one of:

```
val accept : in_strm_sock -> (in_strm_sock * in_addr)
val accept : un_strm_sock -> (un_strm_sock * un_addr)
```

Or more generally:

$$\forall(\alpha,\beta) \in \{(\texttt{in\_strm\_sock}, \texttt{in\_addr}), (\texttt{un\_strm\_sock}, \texttt{un\_addr})\}\ (\alpha \rightarrow (\alpha \times \beta))$$

The type of **accept** is constrained in two ways:

1. The address and socket it returns has the same domain as its argument.
2. The argument and result socket must both be stream sockets.

In general, we cannot specify such type constraints in SML, but there is a trick that we can use in this case. We make the socket type into a type generator with two arguments: one for the domain and one for the type of socket.

```
type ('a, 'b) sock
```

Likewise, we make the socket address type a type generator of one argument (the domain):

```
type 'a sock_addr
```

For a function such as **accept**, we can then enforce the first constraint by giving it the following type:

```
val accept : ('a, 'b) sock -> (('a, 'b) sock * 'a sock_addr)
```

This still leaves the problem of restricting **accept** to stream sockets. To enforce this constraint, we introduce new abstract types to be the instances of the type arguments to **sock** and **sock_addr**:

```
type dgram
type stream
```

We can then define the type of **accept** to be:

```
val accept : ('a, stream) sock -> (('a, stream) sock * 'a sock_addr)
```

The **stream** and **dgram** types are *void* types — they have no values, but merely serve as *witnesses* to enforce the various constraints in the signatures.[4] As shown below, we also define void types for the network domains. We then define multiple instances of the **socket** function; one for each combination of socket type and domain. Internally, these sockets are all the same type, but we use abstraction to make them appear distinct.

We can further improve the security of the interface by distinguishing between the *passive* socket used by a server to accept new connections and the *active* sockets used to communicate. We do this by parameterizing the **stream** type, and introducing two new void types:

[4]Since SML doesn't actually have void types, these are represented by unit types internally, but the values are not exported outside the implementation.

```
type 'a stream
type passive
type active
```

And we change the type of **accept** to:

```
val accept : ('a, passive stream) sock
               -> (('a, active stream) sock * 'a sock_addr)
```

With this refinement, we can now present the complete interface, which is given in Figure 3. Note that **accept** and **listen** are to restricted passive stream sockets, while **connect** re-

```
type ('a, 'b) sock
type 'a sock_addr

type dgram
type 'a stream
type passive
type active

val accept   : ('a, passive stream) sock
                 -> (('a, active stream) sock * 'a sock_addr)
val listen   : (('a, passive stream) sock * int) -> unit
val bind     : (('a, 'b) sock * 'a sock_addr) -> unit
val connect  : (('a, active stream) sock * 'a sock_addr) -> unit
val close    : ('a, 'b) sock -> unit
val recv     : (('a, active stream) sock * in_flags) -> vector
val send     : (('a, active stream) sock * vector * out_flags) -> int
val recvFrom : (('a, dgram) sock * in_flags) -> (vector * 'a sock_addr)
val sendTo   : (('a, dgram) sock * 'a sock_addr * vector * out_flags)
                 -> int
```

Figure 3: A polymorphic sockets interface

quires an active socket. Likewise, the stream I/O operations, **send** and **recv**, also require active stream sockets.

This use of parameterized type generators gives rise to something similar to a subtyping hierarchy. For example, `(ip, dgram) sock` might be viewed as a subtype of `(ip, 'b) sock`. Other mechanisms for supporting type hierarchies, such as *Haskell's type classes* [HAB+95], can also capture this relationship. Our mechanism goes further, however, since it also establishes constraints between different types (e.g., the domain of the socket and socket address in the **accept** operation).

```
structure IPSock : sig
    type ip

    val addr : (inet_addr * int) -> ip sock_addr

    val udpSocket : unit -> (ip, dgram) sock
    val tcpSocket : unit -> (ip, 'a stream) sock
  end = struct ... end
```

```
structure UnixSock : sig
    type unix

    val addr : string -> unix sock_addr

    val dgramSocket : unit -> (unix, dgram) sock
    val strmSocket : unit -> (unix, 'a stream) sock
  end = struct ... end
```

5.1 Implementation

The implementation of this approach is quite simple. For the purpose of this discussion, let us assume that we have a structure `PrimSock` that implements the weakly typed interface of Figure 1. We define datatypes to represent the void types:

```
datatype 'a stream = STREAM
datatype dgram = DGRAM
datatype passive = PASSIVE
datatype active = ACTIVE
```

The representation of sockets and socket addresses is independent of these types:

```
datatype ('a, 'b) sock = SOCK of PrimSock.sock
datatype 'a sock_addr = SOCKADDR of PrimSock.sock_addr
```

Most of the socket operations are quite simple: they merely provide a translation from these types to the `PrimSock` types. For example:

```
fun accept (SOCK s) = let
      val (s', a) = PrimSock.accept s
      in
        (SOCK s', SOCKADDR a)
      end
```

Note that the inferred type of this function is unconstrained:

```
val accept : ('a, 'b) sock -> (('c, 'd) sock * 'e sock_addr)
```

The type gets constrained to the safer interface of Figure 3 by signature matching on the implementation structure.

One other aspect of the implementation deserves comment. There is an obvious tension between sharing the internal representation of sockets among the various implementation modules (e.g., `Sock`, `IPSock`, and `UnixSock`), and providing abstract interfaces. There are a number of ways to deal with this. One approach is to first define the modules without signatures (or with signatures that reveal the internal representation types), and then to rebind the structure names with the final signature constraints.[5] The compilation manager (CM) [Blu96] for the Standard ML of New Jersey system [AM91] provides a more elegant way of doing this. CM organizes source files into *source groups*, and provides a mechanism

[5] Traditional SML systems define a linear order on the processing of top-level declarations (including modules).

for limiting which modules are exported outside the group. Using this feature, we can collect the representation types in an internal module that is available to the other modules in the source group, but not outside the group.

6 Extending the interface

The main limitation of this approach is that it only works because the system is *closed*; there is no way to define new flavors of sockets. We can open up the system by introducing a generic socket structure for creating fully polymorphic sockets:

```
structure GenericSock : sig
    val socket : (Sock.addr_family * Sock.sock_type) -> ('a, 'b) Sock.sock
  end = struct ... end
```

Introducing such a structure allows the user to "spoof" arbitrary sockets, so the system is no longer statically type safe (although, errors will still be caught at run-time). This is a trade-off between complete type security and extensibility.

Fortunately, it is easy to distinguish the programs that are type secure from those that are not. Since the user must use `GenericSock` to circumvent the type restrictions, we know that a program that is free of references to `GenericSock` will be safe (this is analogous to the "`UNSAFE`" keyword in Modula-3).

7 Conclusions

[[**Limitations of this approach**]]

Acknowledgements

The designs presented here are part of an ongoing effort to develop new APIs for Standard ML [GE96]. The design in Section 3 was the first design proposed by the author. Dave Berry pointed out the lack of type security in the design, and suggested the approach of Section 4 as a counter proposal. This served as motivation for improving the first design, which lead to the final approach described.

References

[AM91] Appel, A. W. and D. B. MacQueen. Standard ML of New Jersey. In *Programming Language Implementation and Logic Programming*, vol. 528 of *Lecture Notes in Computer Science*, New York, N.Y., August 1991. Springer-Verlag, pp. 1–26.

[Blu96] Blume, M. *CM: A Compilation Manager for SML/NJ (User Manual)*, 1996. *Included in the SML/NJ distribution.*

[GE96] Gansner, E. R. and J. H. R. (Eds.) (eds.). *The Standard ML Basis Library Reference Manual. Unpublished draft*, 1996.

[HAB+95] Hammond, K., L. Augustsson, B. Boutel, W. Burton, J. Fairbairn, J. Fasel, A. Gordon, M. Guzman, J. Hughes, T. Johnsson, M. Jones, D. Kieburtz, R. Nikhil, W. Partain, J. Peterson, S. P. Jones, and P. Wadler. *Report on the Programming Language Haskell (Version 1.3)*, June 1995.

[HTA+93] Hall, M., M. Towfiq, G. Arnold, D. Treadwell, and H. Sanders. *Windows Sockets: An Open Interface for Network Programming under Microsoft Windows (Version 1.1)*, January 1993.

[LMKQ89] Leffler, S. J., M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison Wesley, Reading, Mass., 1989.

[Nye90] Nye, A. *X Protocol Reference Manual*, vol. 0. O'Reilly & Associates, Inc., 1990.

[QS95] Quinn, B. and D. Shute. *Windows Sockets Network Programming.* Addison-Wesley, Reading, Massachusetts, 1995.

[Ste90] Stevens, W. R. *UNIX Network Programming.* Prentice Hall, Englewood Cliffs, New Jersey, 1990.