A "Safe" Sockets API

John Reppy
AT&T Bell Laboratories
Sept 20, 1995
Background

The BSD sockets interface unifies a collection of different network protocols into a single socket type and address type.

```plaintext
type sock

type sock_addr

val socket : (addr_family x sock_type) -> sock

val accept : sock -> (sock x sock_addr)

val bind : (sock x sock_addr) -> unit

val connect : (sock x sock_addr) -> unit

val listen : (sock x int) -> unit

val close : sock -> unit
```
<table>
<thead>
<tr>
<th>Address Family</th>
<th>Socket Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>INET</td>
<td>UDP, Unix Dgram</td>
</tr>
<tr>
<td>UNIX</td>
<td>TCP, Unix Streams (Pipes)</td>
</tr>
</tbody>
</table>

Not all operations work on all kinds of sockets. Plus, the socket kind and network address must be consistent.

How can we use the type system to protect against mismatches so confusing different kinds of sockets?
First attempt: use the module system.

```ocaml
signature Sock =

sig
  type sock
  type sock_addr

  val socket : unit -> sock

  val connect : (sock * sock_addr) -> unit

end

structure TCPSock : sig

include Sock

val accept : sock -> (sock * sock_addr)

val listen : (sock * int) -> unit

end = struct ... end

structure UDPSock : sig

include sock

end = struct ... end

Problem: "polymorphic" uses of sockets not allowed.
```
A better solution:

```
<table>
<thead>
<tr>
<th>type (\alpha, \beta) sock</th>
</tr>
</thead>
<tbody>
<tr>
<td>type \alpha sock_addr</td>
</tr>
<tr>
<td>type \beta sock_addr</td>
</tr>
</tbody>
</table>
```

Introduce \textbf{void} types to constrain the polymorphism

```
type stream            \text{ socket types} \\
type dgram              \\
type ip                 \text{ address families} \\
type unix               \\

val accept : (stream, \alpha) sock \\
            \rightarrow ((\alpha) sock x \alpha sock_addr) \\

val listen : ((stream, \alpha) sock x \text{ mt}) \rightarrow \text{ unit} \\

val bind : ((\alpha, \beta) sock x \beta sock_addr) \rightarrow \text{ unit} \\
```
Socket creation:

structure IPSock : sig

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

val udpSocket : unit -> (dgram, ip) sock
val tcpSocket : unit -> (stream, ip) sock

end = struct ... end