PacketTypes: Abstract Specification of Network Protocol Messages

Peter J. McCann and Satish Chandra
Bell Laboratories
263 Shuman Blvd.
Naperville, IL 60566
{mccap, chandra}@research.bell-labs.com

Abstract

In writing networking code, one is often faced with the task of interpreting a raw buffer according to a standardized packet format. This is needed, for example, when monitoring network traffic for specific kinds of packets, or when unmarshaling an incoming packet for protocol processing. In such cases, a programmer typically writes C code that understands the grammar of a packet and that also performs any necessary byte-order and alignment adjustments. Because of the complexity of certain protocol formats, and because of the low-level of programming involved, writing such code is usually a cumbersome and error-prone process. Furthermore, code written in this style loses the domain-specific information, viz. the packet format, in its details, making it difficult to maintain.

We propose to use the idea of *types* to eliminate the need for writing such low-level code manually. Unfortunately, types in programming languages, such as C, are not well-suited for the purpose of describing packet formats. Therefore, we have designed PacketTypes, a small packet specification language that serves as a type system for packet formats. PacketTypes conveniently expresses features commonly found in protocol formats, including layering of protocols by encapsulation, variable-sized fields, and optional fields. A compiler for this language generates efficient code for type checking a packet, i.e., matching a packet against a type. In this paper, we describe the design, implementation, and some uses of this language.

1 Introduction

Networking software is difficult to construct. Networking code in a system must interface with bare hardware in a network device, and at the same time implement complicated real-time algorithms. Furthermore, both the low-level data manipulation and the emphasis on high performance have (barring few exceptions) limited the choice of an implementation language to C. As a result, development, testing, and deployment of new protocols is a slow and expensive process.

SIGCOMM'00, Stockholm, Sweden.


```
ip_fw_chk(struct iphdr *ip) {
  struct tcphdr *tcp =
    (struct tcphdr *)((__u32 *)ip + ip->ihl);
  int offset =
    ntohs(ip->frag_off) & IP_OFFSET;
  ...
  switch (ip->protocol) {
    case IPPROTO_TCP:
      if (!offset) {
         src_port = ntohs(tcp->source);
         dst_port = ntohs(tcp->dst);
         ...
```

Figure 1: Excerpt from the firewall code in Linux (v2.0.32).

Motivated by this problem, significant research effort has recently been put into systematic software architectures and languages suited for constructing networking software. These efforts usually address a specific aspect of the complexity of networking code. Some approaches try to better express the modular structure and composition of protocol layers (e.g., x-kernel [11], Foxnet [6]). Others try to better express the reactive control within each layer (e.g., Esterel [5]). Yet others emphasize the verification aspect (e.g., Promela++ [3]), or focus on an object-oriented implementation (e.g., Morpheus [1], Prolac [14]). These efforts demonstrate that an implementation methodology or language more suited to the task at hand can help build cleaner and more robust implementations and can do so without exacting performance penalties.

One aspect of the complexity of writing networking code arises from the fact that the wire format of a network packet is fixed by standards. Packet formats are independent of any given machine's architecture to allow interoperability, and generally pack data as tightly as possible to minimize the header overhead per unit of actual content. To interpret a buffer containing a "raw" network packet, a programmer must write low-level code that understands the packet format and that also performs any necessary byte-order and alignment adjustments as per host requirements. We illustrate such low-level code by means of an example.

Figure 1 shows an abstracted form of the firewall code in the Linux networking module (`net/ipv4`). The code first computes the starting address of an IP packet's payload into the variable `tcp`, by adding the size of the header (field `ihl`) to the start of the packet. It then computes the offset of the present IP fragment[1] into the variable `offset`; the macro `ntohs` converts a network byte order representation to the host byte order representation for a 16-bit quantity. If the protocol in the payload is TCP and offset is zero (first fragment), it extracts the source and destination port numbers from the TCP header. Later code (not shown) implements specific firewall policies based on this and other information.

The style of programming involved here is not complicated, but does have some undesirable properties. A programmer must explicitly encode the layout of an IP packet using pointer arithmetic and bit operations. He must encode the correlation between the `protocol` field's value of `IPPROTO_TCP` and the payload being a TCP packet. He must also remember to convert any multi-byte numeric quantities between the network byte order and the host byte order at all appropriate places.[2] While the TCP/IP suite does not have a complicated packet format, other protocols do, and writing code that interprets a packet belonging to one of those protocols can require a lot of cumbersome programming. For example, the Q.931 protocol [12] defines the format of ISDN signaling messages as rather complex hierarchical packets. A C implementation that parses a Q.931 message can run into thousands of lines of code, and it is easy to introduce coding errors in offsets, sizes, conditionals, or endianness.

The capability to interpret a network packet is key to many important networking applications, in addition to protocol processing. These applications include network monitoring, accounting, and security services. The only software architecture in the literature that addresses such applications is a packet filter. However, filter specifications essentially parse a packet in much the style of Figure 1, and are written in byte code (except for some higher-level expression languages available for TCP/IP). The need for a mechanism to build such applications *quickly* and *correctly*, even for complicated formats, has been our primary motivation.

At first blush, the concern for parsing a packet may not appear to be a problem at all—one might be able to simply overlay a raw buffer with an appropriately defined C `struct` or `union`, and read off the values from the fields of the structure. Because C supports bit fields in structures, this seems to be a plausible choice. However, the C type system is not adequate for describing packet formats for a number of reasons:

1. Protocol headers can contain fields whose sizes depend on the value of a previous field in the header. For example, the `options` field in an IP header can occupy between 0 to 40 bytes, depending on the value of a previous field `ihl` (header length). Variable-sized fields cannot be represented as static types.[3] Similarly, optional fields are not supported by C `structs`.

2. C types do not support the notion of layering of protocols in a clean way. One possibility is to define the payload of a packet as a `union` of all possible types of packets the payload could carry. A problem that immediately presents itself is how to know in advance all possible types of payload a given packet may be asked to carry. Thus, this solution is not easily extensible. Another problem with C `unions` is that one must still test the discriminating field and choose the interpretation consistently. For example, if we represent an IP payload as a `union` of TCP and UDP types, we must still make the right choice based on the value of the `protocol` field.

3. Finally, because of possible alignment and byte order mismatches between a host machine and the network format, applications may still need to do adjustments before using a numeric value as a primitive type.

```
case (#payload ip) of
  TCP {source, dst, ...} =>
    if ((#offset ip) = 0) then
      ...
```

Figure 2: An ML-style description. The syntax `#field var` is field selection from a record. The term on the left of `=>` is a pattern, here matching the datatype `TCP`.

What about data types in higher-level programming languages, such as Standard ML? In Standard ML, one might express the logic in Figure 1 using the code fragment shown in Figure 2. Unlike in Figure 1, the programmer does not need to explicitly interpret the wire format.

There are good reasons why protocol code is not written in this way. First, unmarshaling from the wire format to the high-level data type could be expensive, because ML's representation of data types is not as close to machine representation as that of C. Second, defining layered packets in an extensible way is still difficult. Although Foxnet [6] implements TCP/IP in Standard ML, packets are exposed as C-like byte arrays. The code uses low-level marshaling and unmarshaling into SML records to access fields that are needed for protocol processing.

In this paper we show that the idea of types can nevertheless be a useful one in the context of networking applications. To this end, we propose PacketTypes, a small packet specification language to describe packet formats. The philosophy behind this language is to supply an *external type system* for packet formats. While an external type system cannot be enforced by the compiler for a host language, we show that by its disciplined use, a programmer can avoid the problems mentioned earlier. PacketTypes has the following salient features:

- Packet descriptions are expressed as types.
- The fundamental operation on packets is checking their membership in a type.
- Layering of protocols is expressed as successive specialization on types.
- Refinement of types creates new types, a facility useful for packet classification.

The role of PacketTypes is analogous to "yacc", in that it abstracts away the packet grammar into a separate specification language, and automatically creates recognizers for packets. PacketTypes can find application in a number of situations, such as the following:

[1] A long payload may be transmitted by IP as a series of IP fragments, each of which contains a segment of the original payload, and an indication of the offset of the segment in the original payload.

[2] To put this in perspective, in the Linux implementation (v2.0.32), endianness related macros `hton*` and `ntoh*` together appear about 300 times in the `net/ipv4`.

[3] By static types we mean data structures whose memory can be allocated at compile time. Pointer data structures can represent variable-sized objects, but cannot be allocated at compile time.

- *Specification of network monitoring code.* The task of writing network monitoring code can be automated—and perhaps more importantly, sped up—by writing the specifications in PacketTypes.
- *Packet classification.* Type definitions written using PacketTypes can also serve as packet filter specifications, and can be much simpler to write than byte codes.
- *Stub generation.* The PacketTypes compiler can automatically generate interface code between the wire representation and a host language's representation, eliminating the need for low-level programming.
- *Formal specification of packet formats.* Instead of English descriptions and ASCII graphics, *Request for Comments* documents can use PacketTypes to adopt a formal approach to describing formats. This could also provide a basis for formal verification.

We have implemented this language, and based on it, have built a number of applications, including a network monitoring system that works for Q.931 protocol messages. We have also measured the performance of these applications on simulated, yet realistic workloads. Based on our experiments, we believe that the packet specification language can be implemented efficiently enough and serve a number of uses in real systems.

We describe the packet specification language in Section 2, and a case study on its application to network monitoring in Section 3. We describe the implementation of this language in Section 4. Section 5 describes uses of our language in stub generation and packet classification, and also gives performance results. Section 7 compares PacketTypes to related work.

2 PacketTypes: A Packet Specification Language

In this section we describe a specification language for packet formats, based on the notion that the layout of fields within a packet, plus a collection of constraints on those fields, can be considered to be a *type*. The semantics of a type is a subset of the universe of binary strings. As we will see below, this specification technique is similar in many respects to regular languages, but it adds a system of constraints over attributes of terms which is lacking in most automata-theoretic models.

We start with one primitive type, `bit`, and add the ability to define new types using just a few operators. Definitions have the form `name := type`. For example,

```
byte := bit[8];
bytestring := byte[];
```

defines the term `byte` to be a sequence of exactly eight bits, and defines the term `bytestring` to be an arbitrarily long sequence of bytes. The empty square bracket operator `[]` is therefore analogous to the Kleene closure operator `*`, but with the addition of a constant argument, it constrains the repetition to have exactly the given number of occurrences.

We also allow grouping and sequencing of terms to form structures. For example, the specification of an Internet Protocol (IP) packet might begin as:

Attribute	*Meaning*
`value`	The natural number formed by concatenating all the bits of the field in network order.
`numbits`	The total number of bits occupied by the field.
`numbytes`	The total number of bytes occupied by the field.
`numelems`	The number of elements in an array-type field.
`alt`	For an alternative-type field, a collection of booleans indicating which alternative was chosen.

Table 1: Attributes and their meanings. See examples in the text for details.

```
nybble := bit[4];
short := bit[16];
long := bit[32];
ipaddress := byte[4];
ipoptions := bytestring;

IP_PDU := {
  nybble     version;
  nybble     ihl;
  byte       tos;
  short      totallength;
  short      identification;
  bit        morefrags;
  bit        dontfrag;
  bit        unused;
  bit        frag_off[13];
  byte       ttl;
  byte       protocol;
  short      cksum;
  ipaddress  src;
  ipaddress  dest;
  ipoptions  options;
  bytestring payload;
} ...
```

`IP_PDU` defines the fields of an Internet Protocol version 4 header [21]. This type imposes a structure on packets, but without additional constraints, it allows many sequences of bits that are not valid IP packets. The necessary constraints appear in a `where` clause following the sequence, as in:

```
IP_PDU := {
  ...
} where {
  version#value = 0x04;
  options#numbytes = ihl#value * 4 - 20;
  payload#numbytes = totallength#value - ihl#value * 4;
}
```

These constraints specify that the `version` field is set to 4 (for IP version 4), and give the number of bytes occupied by the `options` and `payload` fields. We use the syntax `field#attribute` to reference specific attributes of the given fields. A partial list of attributes and their meaning is given in Table 1. Note that the attributes `#numbits`, `#numbytes`, and `#numelems` can take on only natural numbers as values, and any packet for which the corresponding constraints associate a negative value with these attributes should be rejected as ill-formed.

A set of constraints in a `where` clause may follow any newly defined type. In the `IP_PDU` example, these constraints have been used to specify that a certain field will hold a certain constant value and to specify the length of variable-length fields. In the absence of constraints, the repetition operator `[]` is treated "greedily," meaning that any following data that could belong to the repetition is assumed to do so. Constraints on the number of repetitions may come from within the repeated member, because each element must match the given type; from the `where` section of a structure that contains the repetition, as in the example above; or from outside the containing structure in the form of constraints on the length of the data object in which the structure appears.

In addition to defining types by the `:=` operator, we also allow a construct known as *refinement*, which is represented by the `:>` operator. Refinement is used to add additional constraints to an already-specified type. For example, an Ethernet frame containing an IP packet might be specified as:

```
macaddr := bit[48];

Ethernet_PDU := {
  macaddr dest;
  macaddr src;
  short   type;
  bytestring payload;
}

IPinEthernet :> Ethernet_PDU where {
  type#value = 0x0800;
  overlay payload with IP_PDU;
}
```

`Ethernet_PDU` contains the specification of an Ethernet frame and `IPinEthernet` shows how to layer IP on it by constraining the `type` to have the value `0x0800` and *overlaying* the `IP_PDU` definition onto the Ethernet payload.

The `overlay...with` constraint allows us to merge two type specifications by embedding one within the other, as is done when one protocol is *encapsulated* within another. Overlay constraints introduce additional substructure to an already existing field. Essentially, the constraints of the `IP_PDU` type will become a part of the `payload` member of the Ethernet frame, and must be checked before a packet can match `IPinEthernet`. Because `IPinEthernet` is grounded in a link-layer frame that might come from a device driver, this would be an appropriate representation on which to base a network monitoring application.

Sometimes it is useful to specify additional constraints on an `overlay`ed type in a refinement, or to otherwise reference the fields of a sub-structure. For example, an `IPinEthernet` packet whose source address is 192.168.0.1 could be expressed as

```
My_IPinEthernet :> IPinEthernet where {
  payload.srcaddr#value = 192.168.0.1;
}
```

The notation `payload.srcaddr` is used to denote the IP source address of the packet. Here the dot notation is used to access the fields of an overlayed structure, but it may also be used in other contexts such as to reference the substructure of a definition that uses other definitions as member types. Note that if a field with substructure is overlayed, that structure is hidden by the new structure and the old fields are no longer accessible. This ensures that each dotted name will resolve to a unique member of the packet structure.

Finally, we allow the specification of alternative types using the *alternation* (`|=`) operator. This operator combines terms disjunctively, where a given bitstring matches the type if and only if it matches one of the constituent members. Earlier, in describing `IP_PDU`, we had defined `ipoptions` as a `bytestring`. We now provide a more precise definition of the `ipoptions` type:

```
ipoptions := {
  NonEndOption  neo[];
  EndOption     eo[];
  bytestring    padding;
} where {
  eo#numelems <= 1;
}
```

Here the options are specified as a sequence of `NonEndOption`s, followed by an optional `EndOption`, followed by padding. The `[]` operator along with the constraint `eo#numelems <= 1` effectively ensures that there will be zero or one `EndOption`s. We use this idiom to denote optional fields.

While the `EndOption` is just a single byte whose value is zero, the `NonEndOption`s are more interesting and make use of the `|=` construct:

```
NonEndOption |= {
  NoOperation   nop;
  Security      sec;
  LSRR          lsrr;
  SSRR          ssrr;
  RR            rr;
  StreamID      sid;
  Timestamp     tstamp;
}
```

The alternatives construct `|=` is syntactically similar to the definition construct `:=` in that it consists of a list of type, name pairs. The repetition operator can be used as before and there may be a `where` clause following the members with additional constraints. In contrast to `:=`, however, this example states that a `NonEndOption` may take on any one of the formats described by the members.

The `#alt` attribute may be used in constraints to detect which alternative was chosen. For example, the alternatives list above defines seven booleans expressions, `#alt @ nop` through `#alt @ tstamp`, each of which is true if and only if the member took on the given alternative. We use a separate `@` operator to compare the `#alt` attribute to one of the possible values in order to keep the space of such values distinct from other numeric or boolean values. In a `where` clause of the `ipoptions` structure, for instance, the boolean expression `neo[0]#alt @ tstamp` would be true if and only if the first alternative was a timestamp.

Clearly, the expressiveness of the language comes primarily from constraints. In general any kind of constraint, including relational operators `>`, `<` and boolean combinators may be used (see the appendix). However, for computability reasons, the language limits the kinds of constraints that it admits. Constraints that provide size information for variable-sized fields must be such that they reference attributes only of previously mentioned fields. An implementation should enforce this restriction.

Moreover, any given implementation may put further limitations in the kinds of constraints it supports; for example, it may require that lengths be given using very simple expressions such as the ones in `IP_PDU`. Limitations of our compiler are mentioned in Section 4.5.

3 Network Monitoring Case Study

In this section, we consider in detail a network monitoring example taken from a real-life situation: monitoring of signaling messages for ISDN calls in telecommunications switching equipment. The protocol under consideration is Q.931, the Layer-3 ISDN signaling, running over LAPD frames. The general format of a Q.931 message includes a single-byte protocol discriminator (8 for Q.931 messages), a call reference value to distinguish between different calls being managed over the same D channel, a message type, and various information elements (IEs) as required by the message type in question. Thus, the top-level description of a Q.931 packet is given as follows (the types `cref`, `mtype` and `infoelem` will be defined shortly):

```
Q931control := {
  byte     protdisc;
  cref     callref;
  mtype    messtype;
  infoelem elems[];
} where {
  protdisc#value = 0%0001000;
}
```

We wish to write a monitoring tool that taps Q.931 messages and prints out the fields of the message on a display device. We first describe the relevant portions of the format of a Q.931 packet, and alongside, show how PacketTypes supports the idioms that appear in it. We then compare the monitoring tool created by using PacketTypes to one written by hand in C to bring out the advantages of using a concise yet powerful description language.

The type `cref` contains one or more octets. Octets following the first octet in a `cref` denote an optional call reference value `crv`. The length of `crv` in octets is given in a portion of the first octet of `cref`, and could be 0 if `crv` is omitted. If `crv` is present, the most-significant bit of its first octet denotes a flag value. The packet types defined next capture this description.

```
cref := {
  nybble  reserved;
  nybble  length;
  crv     cr[];
} where {
  cr#numelems <= 1;
  cr#numbytes = length#value;
}
crv := {
  bit     flag;
  bit     cvalue[];
}
```

Here, the field `cr` is constrained to have zero or one occurrences. The constraint on the `numbytes` attribute of `cr` expresses the relation between the value of the `length` field and the number of octets in `cr`.

The type `mtype` is simple: one reserved bit followed by one seven-bit `MessageType`, which is an alternate list of bit patterns of seven bits each. In this list, instead of inventing names for fixed bit-patterns, we use the literals as types.

```
MessageType |= {
  0%0000000  Escape;
  0%0000001  Alerting;
  0%0000010  Proceeding;
  ...
}
```

The type `infoelem` is the most challenging part of this protocol. At the top level, an `infoelem` can be represented simply as follows:

```
infoelem |= {
  Type1SingleOctetInfoelem t1;
  Type2SingleOctetInfoelem t2;
  MultiOctetInfoelem       multi;
}
```

`infoelem` is defined to be one of three possible alternatives. The first two alternatives are simple, one-octet types. The third, multi-octet type is quite complex, and is described next. The first and second alternatives can be distinguished from the third by their first bit, which is always set, whereas it is always clear in the third; they differ between themselves in their next three bits. Each `MultiOctetInfoelem` describes its own length, which enables iteration over a variable-sized array of `infoelems`.[4]

`MultiOctetInfoelem` is the base for carrying IEs. It can carry a variety of IEs, such as Bearer Capability, Cause, Call State, etc. We focus only on Bearer Capability; other ones can be handled similarly. In PacketTypes, `MultiOctetInfoelem` is an alternatives list containing the various IEs:

```
MultiOctetInfoelem |= {
  BearerCapability bc;
  ...
}
```

Each IE is a refinement of the following "base" type:

```
MultiOctetInfoelemBase := {
  0%0 zero;
  LongElemIDcs0 elemid;
  byte length;
  bytestring payload;
} where {
  payload#numbytes = length#value;
}
```

`LongElemIDcs0` is another enumerated type expressed as a list of alternatives, each being a seven-bit pattern.[5]

IEs can be quite complicated. An ASCII picture of the Bearer Capability IE is shown in Figure 3. Its top level definition in our language is the following:

```
BearerCapability :> MultiOctetInfoelemBase
where {
  elemid#alt @ BearerCapabilityID;
  overlay payload with {
    BC_group3   g3;
    BC_group4   g4;
    BC_group5   g5[];
    BC_group6   g6[];
    BC_group7   g7[];
  } where {
     g5#numelems <= 1;
```

[4] The iteration continues until no more `infoelems` can be matched.

[5] The suffix `cs0` corresponds to codeset 0. Codesets can be changed on the fly to get a different set of IEs in place. We do not support codesets at present (see Section 6).

```
      g6#numelems <= 1;
      g7#numelems <= 1;
    }
  }
```

It is natural to describe the fields g3 to g7 of Bearer Capability in terms of *octet groups*. An octet group is a sequence of *non-final* octets followed by a *final* octet. A non-final octet has its most significant bit clear, whereas a final octet has it set.

```
octetgroup := {
  nonfinaloctet nfo[];
  finaloctet    fo;
}
nonfinaloctet := byte where {
  [0]#value = 0;
}
finaloctet := byte where {
  [0]#value = 1;
}
```

The syntax `[0]#value` denotes the first element of the type `byte`, in other words, the zero'th bit.

Each of the types BC_group3 through BC_group7 can now be defined by overlaying an `octetgroup` with appropriate decompositions. We do not describe these further, as they are quite routine.

We can now specify a monitor for the wire format by refining a LAPD frame (definition is not shown, but assume it contains a `payload` field):

```
  Q931inLAPD :> LAPDframe where {
    overlay payload with Q931control;
  }
```

In addition to being a machine readable encoding of Figure 3, the types described above can be used to *automatically* construct monitoring code. The compiler generates print statements after each unit (or type) is matched. An important issue is the output format: the default print out may not be what is desired in a given monitoring system. One possibility is to specify a *printf*-like template string that defines the output format, and interpret it with the data values extracted from the packet. Another possibility is to generate a skeletal data-structure traversal routine in C and let the user insert print (or other) statements as needed.

By contrast, writing C code by hand to recognize and parse a Q.931 packet is tedious due to the large number of conditionals and bitwise operations needed to capture the description. One implementation of (roughly[6]) this functionality takes well over 10,000 lines of C. It is also harder to modify as specifications evolve (for example, a version of this protocol augments octet 3 of Figure 3 with an optional 3a octet.) By contrast, a PacketTypes version of a subset of the same specification takes far fewer lines—by about a factor of four—and contains far more readable text than the corresponding C code.

We have produced a working implementation of a monitor for Q.931 over LAPD (although we have implemented only a few information elements), and also for IP packets over Ethernet. The latter includes complete parsing capability for IP options. We are investigating the possibility of constructing monitors for ASN.1 messages, by expressing ASN.1 encodings in our language.

[6]For example, the implementation does support multiple codesets, which we do not currently support.

```
|  8  |  7  |  6  |  5  |  4  |  3  |  2  |  1  | Octet
|-----------------------------------------------|
|                                               |
|   Bearer Capability/Low Layer Compatibility   |   1
|-----------------------------------------------|
|                                               |
|                    Length                     |   2
|-----------------------------------------------|
|   1 |   Coding  |                             |
|     |  Standard | information transfer cap.   |   3
|-----------------------------------------------|
|     |  transfer |                             |
| 0/1 |    mode   | information transfer rate   |   4
|-----------------------------------------------|
|     |                 |          |            |
| 0/1 |    structure    |configura'n|establish't|   4a
|-----------------------------------------------|
|     |           | information transfer rate   |
|  1  | symmetry  | destination -> origination  |   4b
|-----------------------------------------------|
|     |  0     1  |                             |
| 0/1 |  layer 1  |  user info layer 1 protocol |   5
|-----------------------------------------------|
|     |sync |Negot|                             |
| 0/1 |async|posbl|       user rate             |   5a
|-----------------------------------------------|
|     |intermediat| NIC | NIC |flow |flow |     |
| 0/1 |    rate   |on Tx|on Rx|on Tx|on Rx|  0  |   5b+
|-----------------------------------------------|
|     |headr|multi|     |LL ID|assgr|inbnd|     |
| 0/1 |no hd|frame|mode |negot|assge|negot|  0  |   5b*
|-----------------------------------------------|
|     | number of | number of |                 |
| 0/1 | stop bits | data bits |      parity     |   5c
|-----------------------------------------------|
|     |duplx|                                   |
|  1  |mode |           modem type              |   5d
|-----------------------------------------------|
|     |  1     0  |                             |
|  1  |  layer 2  |  user info layer 2 protocol |   6
|-----------------------------------------------|
|     |  1     1  |                             |
|  1  |  layer 3  |  user info layer 3 protocol |   7
 -----------------------------------------------

5b+ for V.110
5b* for V.120
```

Figure 3: Bearer Capability Information Element

4 Implementation

We have implemented a compiler for PacketTypes. The compiler works in four steps. In the first step, it takes as input a list of PacketTypes type definitions and produces a parse tree consisting of a node for each term in the input. In the second step, it performs semantic processing to resolve fieldnames and propagate constants, so that structures of fixed size can be recognized. In the third step, it produces an elaborated intermediate representation consisting of one node for each usage of a definition. Finally, the compiler generates code from the elaborated intermediate representation. Construction of the parse tree is routine and is not discussed further. The subsequent phases are described next. We also discuss efficiency considerations in the matching code, and limitations of our current approach.

4.1 Semantic Processing

In this step, the compiler performs a bottom-up "size propagation" so that structures of fixed size are annotated with their size. A structure has a fixed size if:

- it is a bit;
- it is a fixed size array of bits;
- it is a binary string literal;
- it is a definition list (`:=`) consisting only of fixed size members; or
- it is an alternatives list (`|=`) consisting only of fixed size members where all members have the same size.

A refinement has the same size as the structure it is refining. If the size of a structure cannot be determined at this time, it must be computed at run time; the compiler generates code for this computation as described later.

The compiler also resolves fieldnames during this step. At each occurrence of a fieldname in the constraints, it inserts a pointer to the member to which it refers. Fieldnames can also be "dotted", i.e., contain references to fields of substructures, e.g., `payload.srcaddr`. For each dotted fieldname, a search is performed backwards from the point at which it appears in the following manner:

- A backwards search is performed to find a constraint that `overlay`s any fieldname which is a prefix of the target fieldname. If such an overlay is found, the overlaid structure or definition is searched for the rest of the fieldname.
- If no such `overlay` constraint is found, and this is a refinement of another type, the other type is searched for the fieldname as in the previous step.
- If this type is a definition or alternatives list, the member names are checked to see if any match the first term of the fieldname. If one is found, its definition is checked for the subsequent terms of the fieldname.
- An empty fieldname matches the entire structure in which it appears.

This algorithm implies that once a field is overlaid, its internal structure is no longer accessible. Only the newly overlaid fields may be accessed from that point on. It also shows that the order in which `overlay` constraints appear is significant.

4.2 Elaboration

In this phase, definitions are expanded so that each usage instance of a defined type is given its own data structure node. The later phases of the compiler use this data structure to store information specific to this instance of the type; for example, the offset of a particular type `t` will typically be different for each instance of `t`. For cases in which definitions are used in array types, only one node is inserted for each array instead of potentially unbounded replication. This has important implications for the code generation phase. Also, this phase identifies *structural constraints*, which are constraints that impose limits on the size of variable-lengths fields, and *demultiplexing constraints*, which are constraints comparing the value of a key field (see Section 4.4) to a constant.

4.3 Code Generation

The primary functionality of the generated code is to check if a packet belongs to a specified type. Therefore, for each definition or alternatives list, one routine is generated that checks for that type as well as any refinements of that type. Usually only one of these routines corresponding to a given link-layer or device-specific form will be of interest to the user, such as the `Ethernet_PDU` from Section 2 or the `LAPD-frame` from Section 3. This routine will search for the most refined type that matches a given packet, or stop after finding a single match for which the user has indicated interest.

During code generation, first each of the nodes in the elaborated intermediate representation is allocated named locations, henceforth referred to as registers, to hold the non-constant (or, run-time) information such as lengths and offsets of fields. The code also uses a set of scratch registers as needed. These registers are later mapped on to variables in the generated code. Next, the various language constructs are translated, as described below.

Definition List. For a definition list (`:=`), the compiler generates code that checks the constraints of each member of the type, and then checks the constraints of the `where` clause. Each member is checked recursively in the same manner. Note that some members may be variable-sized arrays and that each element of such an array may itself be of variable size. This necessitates an iteration strategy that first checks for any structural constraints on the number of elements or the number of bits that is occupied by the member, and then checks each element in turn to be sure that its constraints are satisfied. Iteration must stop when the member runs out of space, either by violating its structural constraint or by failing to leave enough room for all of the subsequent, constant-size members. Failure of an element to satisfy its constraints also terminates the array iteration.

Because each usage of a definition is given a fixed amount of storage space, all elements of an array share the same set of registers and only the last element touched is saved at any given moment. If values of another element are needed, they must be recomputed. Also, iteration over an array uses only local information. If an array uses up too much space and a later, variable-sized member is left without enough space, we do not attempt backtrack and change the number of elements occupied by the first array. We believe this is an appropriate limitation for most real-world applications, because most protocols do not require implementations to perform backtracking during parsing.

Alternatives List. In the code generated for an alternatives list (|=), each member is checked in turn as before, but processing stops at the first member which is found to match. Also, instead of being cumulative, bit offsets are restarted from zero for each member. The constraints in the `where` section are checked as before. Note that for alternative lists the `#alt` attribute is available to see which alternative was matched. If a fieldname of a constraint happens to reference an alternative that was not selected, then the entire boolean expression in which it appears is taken to be false.

Refinement. Refinements of a type (`:>`) are checked immediately after the base type has been checked. Each refinement of a type is checked in turn: the new constraints introduced by the refinement are simply evaluated to see if they hold. If a match is found, further refinements of the matching type are checked before examining any sibling types. Thus, this approach seeks to find a most-refined match, although other policies can also be implemented. If the type definitions connected by `:>` are arranged as a tree, in which `a :> b` causes node `a` to be a child of node `b` in the tree, then our matching strategy can be seen as an preorder traversal of this tree.

Constraints. As mentioned earlier, structural constraints provide information about variable-length fields. The compiler uses registers to hold values of attributes computed at run time. A structural constraint generates code that reads the necessary attributes from previously matched fields—these attributes are held in registers—and either evaluates a boolean, or assigns a value to an attribute of some field. Non-structural constraints generate code that evaluates a boolean condition on `#value` or `#alt` attributes of fields. If any boolean condition evaluates to false, matching for the corresponding type fails and the execution attempts to match the packet against the next type in its traversal. Note that overlay constraints have no run-time significance by themselves, except that they can contribute additional structural and non-structural constraints.

The matching code that we generate can optionally perform rigorous bounds checking to ensure that each structure fits within the memory it is given. Such code is often missing from hand-coded implementations of networking protocols. Along with matching code, we can optionally generate code to print out the values of fields as they are encountered. As shown in Section 3, this feature makes it easy to create network monitoring tools.

4.4 Efficiency Considerations

An optimization implicit in our matching scheme is to match common prefixes of composite types only once. For example, if there are two refinements of `IP_PDU`, say `TCPinIP` and `UDPinIP`, at run-time, the matching process matches the `IP_PDU` part only once, and then tries the additional constraints of `TCPinIP` and `UDPinIP` in sequence. This is a rather important optimization in the packet filter literature.

However, matching against a list of refinements could be slow if the implementation were to iterate through a large set of "sibling" types until a match was found. In many cases, these sibling types differ only in the value of one member or a set of members, which acts as a demultiplexing key. A similar situation also arises for alternatives lists, if most of the alternatives are either literal bit-strings or refinements of a common type with a demultiplexing key. Our compiler generates code to perform an optimized traversal in these cases. Instead of a sequential search, the compiler generates

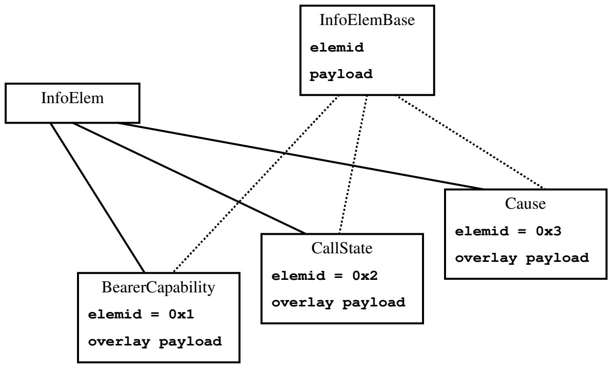

Figure 4: Optimization of traversal of sibling types. The bold edges lead to types included in the alternatives list (|=) of `InfoElem`. The dotted edges lead to types derived from `InfoElemBase` using refinement (`:>`).

(when possible) a switch statement indexed by the demultiplexing key to proceed directly to the code to match a specific refinement or alternative.

We illustrate this idea by an example. Figure 4 shows three types, `BearerCapability`, `CallState` and `Cause`, that form a part of the alternatives list of `InfoElem`. The three types also are refinements of `InfoElemBase`, and can be discriminated by the value of the `elemid` member. When matching a packet against type `InfoElem`, the base type can be checked just once. If the base type matches, the value of `elemid` can be used to determine which particular type to examine further.[7] A similar scheme could be followed to match a packet against a type "tree" rooted at `InfoElemBase`. If the local constraints of `InfoElemBase` match, we attempt to find a more specific match. Again, the value of `elemid` is used to directly go to the code that matches the particular type, bypassing a preorder traversal.

The compiler uses demultiplexing constraints to identify opportunities for this optimization. For now, only equality comparisons and `alt @` constraints are supported as demultiplexing constraints. This is because the implementation strategy is to generate C `switch` statements, which are efficiently supported by a native C compiler. If there is more than one field used as the key in the demultiplexing constraints, a nested `switch` statement is generated. Efficient handling of ranges in demultiplexing constraints would require the implementation of scalable range matching algorithms [7]. Also, handling of dynamic insertion and deletion of endpoints in a general and efficient way would require dynamic code generation to produce code equivalent to a C `switch`. We present a more limited scheme for dynamic insertion and deletion of demultiplexing keys in Section 5.3.

Another important consideration is how to obtain fast code for member value references. In principle, we could generate code that can extract any number of bits from any offset in the packet. However, such code is likely to run slowly, because machine instruction sets support efficient loads only

[7] In general, `InfoElem` could have other alternatives unconnected with the three types. In that case, if none of the three types match, the others alternatives are then tried in order.

from aligned addressed, and only of certain fixed numbers of bits (typically, 8, 16, and 32.). Fortunately, in practice most protocols allocate fields in a way that permits us to perform member value references efficiently. Our compiler makes certain assumptions to be able to generate efficient code for value references. It assumes that any `#value` that needs to be extracted from a packet will be of a fixed, constant size less than 32 bits, and that the value will not straddle an alignment boundary greater than this size. That is, fields having 32 bits or less will be completely contained within word-aligned four bytes, fields having 16 bits or less will be completely contained within a short-aligned two bytes, and so on. The compiler also optionally assumes that overlays are applied only to byte-aligned fields to avoid book-keeping for fractional octets.

4.5 Implementation Choices and Limitations

For simplicity, our compiler currently supports only limited forms of structural and demultiplexing constraints. Structural constraints must currently be written with a single attribute (`#numelems`, `#numbits`, or `#numbytes`) of a variable-length field on the left hand side, and an operator `=`, `<`, or `<=`, followed by an integer expression on the right-hand side. Currently, the compiler supports only demultiplexing constraints that are of the form *name*`#value = `*constant* or *name*`#alt @ `*constant*.

Assumptions on alignment of members could be relaxed while maintaining the same level of performance for some special cases with the use of an alignment inference engine that could be easily constructed, given our intermediate representations and an indication of the expected alignment of the beginning of each packet. Members for which no alignment or size could be inferred would need to use more general, less efficient code.

While we have chosen to produce C code from our intermediate bytecode, this is not the only possible choice. We took this path to take advantage of the optimization done by the native C compiler, and because compilation overhead is currently not a concern in the primary application we are interested in, viz., network monitoring. We could also have produced specifications for other packet filters or produced executable code directly, as is done using dynamic code generation in DPF [10]. Because each packet type uses a fixed amount of space during matching, hardware compilation also appears to be feasible.

Currently, our compiler does not perform any optimizations on the generated bytecode. We could (and plan to) adopt an approach similar to that described for BPF+ [4]. This is desirable not only for dynamic code generation, where we do not have the benefit of a C optimizer, but also for some high-level transformations that a C compiler is not able to perform.

An unexpected benefit of generating C code is easy debugging, because we can use standard C debuggers to step through the generate code. Once we move to dynamically-generated object code, we will lose this facility. In fact, an option of generating C code is worth keeping around just for debugging the type definitions.

Our implementation strategy is geared towards batch compilation, in which all type definitions are presented to the compiler at once. In Section 5.3, we describe a (restricted) way in which we permit run-time addition of new types.

5 Applications and Performance

In this section we describe some sample scenarios in which PacketTypes can be used, and the performance of the generated code for those situations.

5.1 Network Monitoring Tool

We have already described an application of PacketTypes to rapid generation of network monitoring tools. The description outlined in Section 3 can be used to generate packet matching code that prints out the values encountered in each field. Because such a monitor is producing high-level output to a file or a screen, performance is not an important issue.

5.2 Stub Generator

In this section, we outline, by means of an example, how we can use PacketTypes to automatically obtain interface code between the wire representation and a host language's representation. We also examine the performance of such stubs (which were compiled using the native C compiler) with hand-written C stubs compiled using the same compiler (Sun Workshop 4.2, -xO2 option).

Reconsider the code from Figure 1. This code fragment (partially) unmarshals a TCP packet from an IP payload, and makes decisions based on some of the values found in the protocol data. In this respect, it resembles a packet filter, except it is interested in extracting some of the values in a packet rather than demultiplexing it. This code can be written as:

```
ip_fw_chk(struct iphdr *ip) {
  struct tcpview {
    short src_port;
    short dst_port;
  } tv;
  ...
  if (match((void *) ip,
            TCPinIP_firstfrag,
            TCPVIEW,
            (void *) &tv))
  ...
```

The function `match` takes four arguments, as shown in the figure. The first argument, `ip`, is simply a pointer to the packet buffer. The second argument, the type name `TCPinIP_firstfrag`, represents the encapsulation of a TCP datagram inside an IP packet that is a first fragment (IP `offset` is 0). Type names occur as integer constants in the C code. The third argument, a descriptor, specifies the set of field values that the programmer desires to extract (explained further shortly), and the final argument points to space where such extracted values must be stored. Note that the programmer did not have to write low-level C code to work with the wire representation. The style of this code is similar to the typeful style of Figure 2.

The `match` call entails two tasks. First, the packet is matched against the type `TCPinIP_firstfrag`. Second, the values desired by the programmer must be extracted and presented in the host language's representation. The mechanism we choose to do this is to use a *descriptor*. Each type name is associated with a few user-defined descriptors. A descriptor lists fields and the format in which the host language expects them. For example, use of the descriptor `TCPVIEW` implies that the src and dst fields of TCP_PDU should be used to populate struct tcpview in agreement with the host machine's alignment and byte order. The specification of the descriptor itself lists the field names that are desired, as well as the sizes in bits of the corresponding fields in the structure that hold the extracted values (e.g., we could ask for extracting a value that occupies 5 bits in the packet into a 8-bit char). For TCPVIEW, the view specification is given as follows:

```
TCPVIEW of TCPinIP_firstfrag
  payload.src D 16 0
  payload.dst D 16 32
```

This specification lists the fields to be extracted in sequence; for each field, it gives the field name, address (A) or data (D) to be copied, bit size and the bit offset in the target space.[8] Notice that the descriptor does not assume, but rather explicitly specifies the bit offsets in target memory where the extracted data is to be stored. This is because we did not want to generate code specific to the layout that the C compiler on any given machine produces for the target structure (struct tcpview in the example). The generated code corrects the endianness of multi-byte quantities, so on any given machine, the user sees the views populated in the correct host byte order. Code generation can be done in a portable fashion, without regard to the endianness of the host machine.

We can also automatically generate complete views—or parse trees—from a packet specification, producing C structs that contain fields for every member of the input specification. To accommodate variable-sized objects, these views necessarily make assumptions about pointers and memory management that might not hold in every usage situation, but they serve as convenient starting points when most or all fields need to be extracted.

We used our compiler to obtain match functions that returned a boolean value and populated a view structure on success. We generated stubs for the following three types:

1. Type1 matches any Ethernet frame that holds an IP or an ARP packet. The view extracts Ethernet's type field.

2. Type2 matches Ethernet frames that are IP packets from host 135.1.152.73 to host 135.1.152.66. (We specialize the type IPinEthernet described earlier.) The view extracts IP's src and dest addresses.

3. Type3 matches TCP segments between the above two hosts that have source port 1014 and destination port 513. (We specialize type TCPinIP, which, in turn, constrains the protocol field of IP_PDU to 6, and overlays its payload with TCP_PDU.) The view extracts IP's src and dest addresses, and also TCP's src and dst port numbers.

We ran each of these stubs for a large number of times on simulated data. We used three different packets for the Type1 (of which 2 match), three different packets for the Type2 (of which 1 matches) and four different packets for the Type3 (of which 2 match). Each packet was tried on the stub 10 million times. Our experiments were carried out on a 300 MHz UltraSparc-IIi machine with 128M memory; all timings were collected by calling getrusage. Since we are working with a small amount of simulated data, we believe that the working set fits in the cache and that the timings are fairly indicative of number of instructions executed.

The following table reports the execution time per call in nanoseconds for each of the stubs. The first two rows show the performance of the hand-written and automatically generated codes, respectively. The third row is explained further below.

	Type1	Type2	Type3
Hand-Written	91	105	158
Generated	128	110	189
Baseline	73	60	67

The generated code ran up to 40% slower than the hand-written code. We inspected the code to understand the reasons for this performance difference. In Type1, our compiler does not perform boolean short-circuit evaluation that the hand-written code does perform. When we applied this transformation (manually) to the generated code, it matched Type1 in 106 nanoseconds on an average, reducing the performance gap substantially. The remaining difference in times in each of the three types is due to different control sequences generated by the native C compiler for automatically generated C code and for the handwritten C code.[9] We do not currently know the reason for this different behavior; in the future we plan to explore transformations on bytecode that will result in better control constructs being generated in the resulting object code.

In many situations, for example in Figure 1, programmers write classification code inline. Therefore, it might be more appropriate to factor out the overhead of the function call, and of moving the data into a view structure in the timings. We measured this overhead by running a "Baseline" version of the experiment, in which no classification logic is run inside the functions. We see that the cost of the instructions to evaluate the logic is only a small part of the total cost of each stub call. Therefore, for performance-critical applications, a desirable optimization to perform would be to inline the match call and eliminate the loads and stores to the view structure.

5.3 Packet Classifier

We also implemented the core of a packet classifier using the technique for matching types described above. In a typical deployment, a packet classifier requires the capability of dynamically adding and removing types of interest. The implementation outlined in Section 5, however, works only on a static collection of types.

We have added a feature to our system that allows us to dynamically add and remove types, albeit in a restricted way. We make use of the observation that, typically, packet classifiers are interested in demultiplexing structurally similar packets to a large number of endpoints: that is, the packets are almost of the same type, with a few fields working as the demultiplexing key. For example, for TCP connections, the demultiplexing key is a four-tuple consisting of the source and destination IP addresses and the source and destination TCP ports. We call such types *parameterized* types.

[8] The purpose of providing the A alternative is that for payload of packets, one wants to retrieve a pointer to it rather than a copy.

[9] The sequence of loads and stores generated in the two cases was the same.

The use of demultiplexing parameters is similar to the idea of identifying a set of demultiplexing constraints presented earlier, but that technique relied on the presence of similar constraints in a number of types given statically at compile time. Here we are interested in generating parameter sets even for types that have no derived types at compile time, so we rely on the use of user pragmas in the type specification to identify parameterized types and their demultiplexing keys. Refinements of parameterized types can only be of certain kinds—essentially only different values of the demultiplexing key. These refinements can be installed in the system only through a specific interface at run time. Furthermore, no additional refinements are permitted. The compiler generates functions, with formal parameters for the demultiplexing key, to install or remove refinements of parameterized types. For the case of TCP packets, prototypes of such functions are the following:

```
void insert_TCP(int IPsrc, int IPdst, int srcport,
                int dstport, void *endpoint);

void delete_TCP(int IPsrc, int IPdst, int srcport,
                int dstport);
```

The functions `insert` and `delete` allow the user to install and remove endpoints. Another function returns a previously installed endpoint corresponding to the demultiplexing key, or `NULL` if none is found. For TCP packets, it has the following prototype:

```
void *lookup_TCP(int IPsrc, int IPdst, int srcport,
                 int dstport);
```

Typically, `lookup` is called only from inside the compiler-generated matching code.

At run time, a hash table for each parameterized type maintains a map of the demultiplexing key and the endpoint inserted for that key. Thus, the interface functions mentioned above map directly to insert, delete, and lookup functions of a hash table. The hash table approach is essentially the same as adopted in previous packet filter work [24, 10].

The traversal optimization presented in Section 4.4 and the run-time technique presented here are more closely related than they first appear to be. In Section 4.4, we used a C switch statement, whenever a suitable demultiplexing key could be identified. A compiler would (in most cases) implement this switch statement by using a "jump" table of code addresses. If the cases of this switch statement were known only at run time, we could maintain a similar table ourselves, without compiler support. Furthermore, since the size or distribution of the cases are unknown, a hash table is a good choice for implementing this table. This is essentially the technique that we have adopted here.

In the remainder of this subsection, we describe the runtime performance of demultiplexing TCP sessions. Our experimental setup is the same as in the case of the stub generator. The workload that we measured is a set of four simulated TCP packets, each of which vary only the TCP destination port number in `Type3` above. For three of the four packets, we installed a filter that matches the respective packets. We also inserted endpoints for other types, but these types did not occur in the workload. However, the additional types influence the performance of hash table lookups, which is a more realistic scenario.

We timed a test run that simulates 10 million arrivals of a sequence of the four packets mentioned above. We performed this test with various numbers of filters installed, from 10 to 200 (of which, only 3 filters match). All runs were conducted by a single user level process. A realistic deployment of such a system would require the standard packet filter machinery of inserting and deleting filters from an operating system kernel, which was not our focus here.

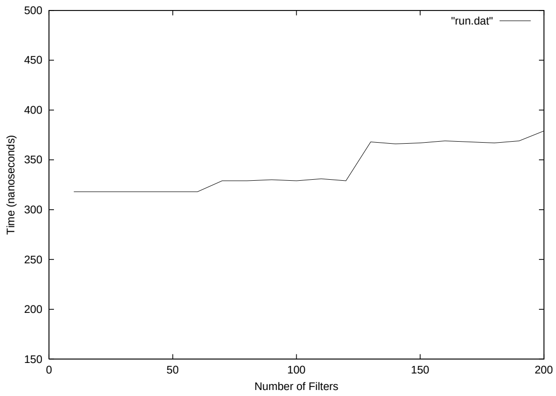

Figure 5: Execution Times of the Packet Filter

Figure 5 shows the time in nanoseconds needed to determine an endpoint (if any) for each packet. As is evident from the figure, the performance of our system scales very well upon increasing the number of installed filters. The steps in the figure denote the increase in the average lengths of the linked list for each hash table bin (this could be reduced by resizing the hash table periodically). Better scalability could be achieved with an implementation of more advanced lookup and insertion algorithms [7].

The absolute time that we obtain per packet, about 350 nanoseconds, is very low compared to the per-packet cost of running virtual-machine-based packet classifiers (e.g., MPF, which costs several microseconds per packet).[10] In our case, we run compiled C code to perform the match, so there is no interpretation overhead. On the other hand, our system would incur a higher expense when installing completely new parameterized types in the system, because we need to rerun the packet specification compiler and the C compiler (unless dynamic code generation is used). In a virtual-machine-based classifier, new filters can simply be downloaded in the form of bytecode into the kernel with a system call.

6 Limitations of PacketTypes

We have found PacketTypes quite adequate to express packet formats for typical ISO layer 3 and layer 4 protocols. However, sometimes there are consistency conditions on a packet that are at a semantic level beyond the scope of PacketTypes. One example is checking the value of a checksum field for whether it indeed corresponds to a standards-based checksum value for the packet. Another example is being able to respect ordering or uniqueness conditions in options, such as a hypothetical condition that there

[10] If optional bounds checking on fields and overlayed structures are enabled, the times go up by about 50%, to approximately 500 nanoseconds per packet.

is at most one appearance of the timestamp option in an IP packet. This situation is analogous to that in compilers: the *yacc* specification of a grammar cannot check for certain semantic conditions, which must be checked explicitly in a subsequent phase. In future work we will investigate adding universal and existential quantifiers to support more general conditions over array types.

Link-layer protocols sometimes have additional characteristics, such as character escaping in PPP [22], that are not considered in PacketTypes. Application-layer protocols are often ASCII-text based, and PacketTypes does not offer specific support to handle them. Another problem in expressing higher-layer formats all the way to link layer (using `:>` and `overlay`) is that we may need to convert from a datagram-oriented view of packets to a stream-oriented view, or we may need to re-assemble fragments of a lower-layer packet. These capabilities are beyond the scope of PacketTypes.

In certain protocols, the grammar of packets can essentially change dynamically depending on previously seen data. An example is the use of different sets of "information elements" that may change as a packet is interpreted. Similarly, ASN.1 packets can signal different encoding schemes to be used for the remainder of a packet, depending on values of certain fields. One approach is to keep a limited amount of state at run time, and identify certain type definitions as being applicable only in a certain state.

Another limitation of the language is that, while the user need not be aware of the endianness of his own machine, all specifications (in particular, the interpretation we have assigned to `#value`) assume that integers are represented as unsigned quantities in network byte order. This could easily be changed to any fixed byte ordering, but the expression of protocols with variable byte orders, such as CORBA's GIOP [19], which allows the byte order to change even within a single message, would require some mechanism for expressing this such as a new `#byteorder` attribute. Similarly, representation of signed quantities could be accommodated by perhaps adding `#svalue` attributes to stand for signed interpretation of values.

Finally, PacketTypes does not have any primitives to express error conditions; either a packet matches or it fails. Sometimes it is desirable for a packet specification to tolerate certain kinds of errors. One approach might be to indicate the closest or "deepest" match and then indicate which constraint failed to hold. We intend to add such features in the future.

7 Related Work

CSN.1. PacketTypes is most closely related to the CSN.1 language [18], developed within the European Telecommunications Standards Institute. Similarly to PacketTypes, CSN.1 specifies bit-strings that correspond to packet formats by using regular expression and grammar constructs. It also allows constructs similar to `length` and `value` attributes of PacketTypes. CSN.1 provides some desirable features lacking in PacketTypes; for example, it directly supports error handling by specifying the error case bit-strings along with the regular case bit-strings.

PacketTypes differs from CSN.1 in significant ways. (a) PacketTypes provides a more general constraint language than CSN.1. For example, the CSN.1 core specification does not support relational operators on arithmetic quantities. (b) CSN.1 is geared primarily for describing formats and for generating conforming packets for testing. It does not have a notion of views for extracting data from a packet. (c) PacketTypes pays significant attention to efficiency of the generated matching code, and makes a number of assumptions to be able to do so. (d) PacketTypes tries to provide a user with a syntax similar to C `structs`, where as CSN.1's syntax is close to pure regular expression and grammar descriptions, which sometimes appears overly terse.

Packet Filters. Packet filtering systems also provide some way of specifying a packet format, which could be considered a type definition. Examples of packet filters include CSPF [17], BPF [16], BPF+ [4], MPF [24], DPF [10] and PathFinder [2]. Except for PathFinder and BPF+, filters are specified as low-level bytecode. Work by Jayaram and Cytron [13] used context-free grammars to specify filters in an attempt to gain composability and allow specification of variable-length fields. The generality of their approach forced them to use LR parsers to create recognizers for packets. Furthermore, since their language is a string of bits, semantic relationships between fields must be broken down to allowable strings of bits, which leads to hard-to-read specifications.

Data-structure Description Languages. The need for describing a data structure has arisen in many contexts in the past, but most importantly in distributed computation, such as remote procedure calls and component-based programming. The USC stub compiler [20] employs very precise descriptions of structure layouts, but lacks the ability to specify variable-length fields. Other format description languages such as ASN.1 [8] or XDR [23] allow for very flexible descriptions of structure, but do not provide control over layout adequate to be used for specification of arbitrary packet formats. Stub compiler languages such as CORBA IDL [19] or Flick [9] have similar limitations.

Other. The idea of creating subtypes by adding constraints on a type bears resemblance to the work by Liskov and Wing [15].

8 Conclusion

Interesting programming language concepts, such as types, often go unused in systems software, because of the deeply entrenched practice of writing low-level C code. This leads to code that is hard to write, maintain, and debug. This paper shows that even within this traditional coding practice, the idea of types has much to offer.

Acknowledgments

We thank the reviewers for their insightful comments on this work, and specially our shepherd for his guidance in preparing the final version of this paper. A previous version of this language was presented at the Workshop on Compiler Support for Systems Software, in May 1999.

References

[1] Mark B. Abbott and Larry L. Peterson. A language-based approach to protocol implementation. *ACM Transactions on Networking*, 1(1):4–19, February 1993.

[2] Mary L. Bailey, Burra Gopal, Michael A. Pagels, Larry L. Peterson, and Prasenjit Sarkar. PathFinder: A pattern-based packet classifier. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*. USENIX Association, November 1994.

[3] Anindya Basu, Mark Hayden, Greg Morrisett, and Thorsten von Eicken. A language-based approach to protocol construction. In *Proceedings of the ACM SIGPLAN Workshop on Domain Specific Languages (WDSL)*, Paris, France, January 1997.

[4] Andrew Begel, Steven McCanne, and Susan L. Graham. BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture. *Computer Communication Review*, 29(4):123–34, October 1999.

[5] Gérard Berry and Georges Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. Technical Report 842, Ecole Nationale Supérieure des Mines de Paris, 1988.

[6] Edoardo Biagioni, Robert Harper, and Peter Lee. Signatures for a network protocol stack: A systems application of Standard ML. In *Lisp and Functional Programming*, 1994.

[7] Milind M. Buddhikot, Subash Suri, and Marcel Waldvogel. Space decomposition techniques for fast layer-4 switching. In *Proceedings of the Sixth International Workshop on Protocols for High Speed Networks*, pages 25–41. Kluwer Academic Publishers, August 1999.

[8] CCITT. *Recommendation X.208: Specification of Abstract Syntax Notation One (ASN.1)*, 1988.

[9] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. Flick: A flexible, optimizing IDL compiler. In *Proceedings of PLDI '97*, pages 44–56, 1997.

[10] Dawson R. Engler and M. Frans Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *Proceedings of ACM SIGCOMM'96 Conference on Applications, Technologies, Architectures and Protocols for Computer Communication*, 1996.

[11] Norman C. Hutchinson and Larry L. Peterson. The *x*-Kernel: An architecture for implementing network protocols. *Transactions on Software Engineering*, 17(1):64–76, January 1991.

[12] International Telecommunication Union. *Recommendation Q.931 - ISDN user-network interface layer 3 specification for basic call control*, May 1998.

[13] Mahesh Jayaram and Ron K. Cytron. Efficient demultiplexing of network packets by automatic parsing. In *Proceedings of the Workshop on Compiler Support for System Software (WCSSS)*, 1996.

[14] Eddie Kohler, M. Frans Kaashoek, and David R. Montgomery. A readable TCP in the Prolac protocol language. *Computer Communication Review*, 29(4):3–13, October 1999.

[15] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(4), November 1994.

[16] Steven McCanne and Van Jacobsen. The BSD packet filter: A new architecture for user-level packet capture. In *1993 Winter USENIX*, pages 259–269, January 1993.

[17] Jeffrey C. Mogul, Richard F. Rashid, and Michael J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 39–51, November 1987.

[18] Michael Mouly. *CSN.1 Specification, Version 2*. Cell & Sys, 1998.

[19] Object Management Group. *CORBA/IIOP 2.2 Specification*, 1998.

[20] Sean W. O'Malley, Todd A. Proebsting, and Allen B. Montz. USC: A universal stub compiler. In *Proceedings of the SIGCOMM '94 Symposium*, October 1994.

[21] J. Postel. RFC 791: Internet Protocol, September 1981. Obsoletes RFC0760. See also STD0005. Status: STANDARD.

[22] W. Simpson. RFC 1662: PPP in HDLC-like framing, July 1994. See also STD0051 . Obsoletes RFC1549 . Status: STANDARD.

[23] R. Srinivasan. RFC 1832: XDR: External data representation standard, August 1995. Status: DRAFT STANDARD.

[24] Masanobu Yuhara, Brian N. Bershad, Chris Maeda, and J. Eliot B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *Proceedings of the 1994 Winter USENIX Conference*, pages 153–165, January 1994.

Appendix

definition-list:
  [*definition*]*

definition:
  id `:=` *structure*
  id `:>` *structure*
  id `|=` *structure*

structure:
  singleton-type `;`
  `{` [*member*]* `}`
  `{` [*member*]* `}` `where` *constraints*

singleton-type:
  id
  id `[` *integer-constant* `]`
  id `[]`

member:
  id id `;`
  id id `[` *integer-constant* `];`
  id id `[];`

constraints:
  `{` *constraint-list* `}`
  singleton-constraint

constraint-list:
  [*singleton-constraint*]*

singleton-constraint:
  boolexpr `;`

boolexpr:
  expr `=` *expr*
  expr `>` *expr*
  expr `<` *expr*
  expr `>=` *expr*
  expr `<=` *expr*
  expr `!=` *expr*
  expr `@` *id*
  boolexpr `||` *boolexpr*
  boolexpr `&&` *boolexpr*

expr:
  integer-constant
  id [[`.` *id*] | [`[` *expr* `]`]]* `#` *id*
  expr `+` *expr*
  expr `*` *expr*
  expr `/` *expr*
  expr `-` *expr*
  `(` *expr* `)`