6-15-2004

PPL: a Packet Processing Language

Eric G. Krupski
Dartmouth College

PPL

a Packet Processing Language

Eric Krupski
15 June 2004
Dartmouth College Computer Science Technical Report TR2004-508

Table of Contents:

Abstract	2
Introduction	2
Details of PPL	7
C Representations of PPL Data	18
Analysis	19
Future Work	22
Conclusions	23
Appendix 1	25
Appendix 2	30

Abstract:

Any computing device or system that uses the internet needs to analyze and identify the contents of network packets. Code that does this is often written in C, but reading, identifying, and manipulating network packets in C requires writing tricky and tedious code. Previous work has offered specification languages for describing the format of network packets, which would allow packet type identification without the hassles of doing this task in C. For example, McCann and Chandra's Packet Types [3] system allows the programmer to define arbitrary packet types and generates C functions which match given data against a specified packet type. This paper will present a packet processing language named PPL, which extends McCann and Chandra's Packet Types to allow the programmer to not only describe arbitrary packet types, but also to control when and how a matching is attempted, with ML-style pattern matching. PPL is intended for multiple applications, such as intrusion detection systems, quick prototypes of new protocols, and IP de-multiplexing code.

Introduction:

Most networking software is written in C, because it is low-level enough to interact with hardware and be fast. Working with a packet requires identifying what it is, and extracting fields from the data. However, in many packets, not all fields are byte aligned. That is to say, the data of some fields may not start on the boundary between words. Additionally, it is not uncommon for data in sent and received to be byte-ordered differently than the data stored on the machine. For fields 2 bytes long, for example, the more significant byte may or may not come across the wire first, depending on the protocol. Extracting fields from a buffer requires adjusting for byte ordering and alignment. However, this is a tedious and error-prone task in C. The way the Linux system deals with this is to describe the layout of a packet with the "struct" construct of C and re-ordering bytes with a macro called "ntohs". Here is how the Linux system represents part of the IP packet using "struct":

```
struct ip_options {
  __u32             faddr;                          /* Saved first hop address */
  unsigned char     optlen;
  unsigned char srr;
  unsigned char rr;
  unsigned char ts;
  unsigned char is_setbyuser:1,                     /* Set by setsockopt?                 */
              is_data:1,                    /* Options in __data, rather than skb     */
              is_strictroute:1,             /* Strict source route                    */
              srr_is_hit:1,                 /* Packet destination addr was our one    */
              is_changed:1,                 /* IP checksum more not valid             */

              rr_needaddr:1,                /* Need to record addr of outgoing dev    */
              ts_needtime:1,                /* Need to record timestamp               */
              ts_needaddr:1;                /* Need to record addr of outgoing dev  */
  unsigned char router_alert;
  unsigned char __pad1;
  unsigned char __pad2;
  unsigned char __data[0];
};

#define optlength(opt) (sizeof(struct ip_options) + opt->optlen)
#endif

struct iphdr {
#if defined(__LITTLE_ENDIAN_BITFIELD)
        __u8    ihl:4,
                version:4;
#elif defined (__BIG_ENDIAN_BITFIELD)
        __u8    version:4,
                ihl:4;
#else
#error  "Please fix <asm/byteorder.h>"
#endif
        __u8    tos;
        __u16   tot_len;
        __u16   id;
        __u16   frag_off;
        __u8    ttl;
        __u8    protocol;
        __u16   check;
        __u32   saddr;
        __u32   daddr;
        /*The options start here. */
};
```

Sample IP code, from linux source file linux/ip.h (v2.4.20-8)

Here is sample code from the Linux kernel which checks IP packets, from the function ip_rcv:

```
if (iph->ihl < 5 || iph->version != 4)
        goto inhdr_error;

if (!pskb_may_pull(skb, iph->ihl*4))
        goto inhdr_error;

iph = skb->nh.iph;

if (ip_fast_csum((u8 *)iph, iph->ihl) != 0)
        goto inhdr_error;

{
        __u32 len = ntohs(iph->tot_len);
        if (skb->len < len || len < (iph->ihl<<2))
                goto inhdr_error;

        /* Our transport medium may have padded the buffer out. Now we know it
         * is IP we can trim to the true length of the frame.
         * Note this now means skb->len holds ntohs(iph->tot_len).
         */
        if (skb->len > len) {
                __pskb_trim(skb, len);
                if (skb->ip_summed == CHECKSUM_HW)
                        skb->ip_summed = CHECKSUM_NONE;
        }
}
```

Sample IP code, from linux source file net/ipv4/ip_input.c (v2.4.20-8)

Using C for writing networking code still has its limitations, though. In their paper "Packet types: abstract specification of network protocol messages" [3], McCann and Chandra make the case for why writing a language for describing packet formats might be better than writing all networking code in C. They argue that C is poor for describing packets formats because the size and structure of fields can be dictated by the values of previous fields. For example, an IP packet with protocol field set to 6 should have a TCP packet inside of it. The length of the nested TCP packet can be calculated either with the header fields of the IP packet or with the header fields of the TCP packet. For the packets to be well-formed, these calculations will have to agree. In C, there is no way to express this as one type that can be matched in one action.

McCann and Chandra, and Sekar, et. al. [5], have independently developed similar specification languages for describing network packets. In the following paragraphs, I will show both McCann and Chandra's Packet Types system, and the specification methods of Sekar, et. al. Since McCann and Chandra's system seems to offer more flexibility in describing individual packets and had greater influence over my project, I will describe the system of Sekar, et. al., first.

The intrusion detection system introduced by Sekar, et.al., in 1999 allows the programmer to add constraints to a packet type. Here is their definition of an IP packet [5]:

```
ETHER_IP = 0x0800
ip_hdr: ether_hdr with e_type=ETHER_IP  {
 {
   bit version[4]; /* ip version */
   bit ihl[4] ; /* header length */
   byte tos; /* type of service */
   unsigned short tot_len; /* total length */
   unsigned short id; /* Id for IP packet */
   bit flag[3]; /* Various flags */
   bit frag_of fset [ 13 ] ;
   byte t ime_tc i ire;
   byte protocol; /* high-level protocol */
   unsigned short checksum;
   unigned int saddr, daddr;
   /* Source and destintion IP addresses */
}
```

This system also relates packet types across protocol layers to each other. In the second line of the above example, the syntax "ip_hdr: ether_hdr" tells us that the fields to follow come after the fields of the "ether_hdr" packet type. "ip_hdr" is seen as extending "ether_hdr"; "ip_hdr" is a subtype of "ether_hdr" in that it inherits fields from "ether_hdr". The syntax "with e_type=ETHER_IP" means that the "e_type" field of the "ether_hdr" packet must be 0x0800. This system offers a more descriptive and robust way of defining packets than C offers. It allows the programmer to define their own packet types by both describing the field layout and optionally putting constraints on those fields.

McCann and Chandra's Packet Types system is quite similar to that of Sekar, et. al. The major semantic difference between the two is in how packet types are related. Here is their definition of an IP packet, taken from the 2000 paper "Packet types: abstract specification of network protocol messages":

```
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;
} …
```

In the system of McCann and Chandra, types are extended by adding constraints, and packets of lower protocols actually contain packets of higher protocols. Types are related across protocol layers through inclusion. Note the last field in the above example, "bytestring payload". This is refined to an actual packet type with syntax like the following:

```
IPwithTCP :> IP_PDU where {
        protocol = 6;
        overlay payload with TCP;
}
```

Hence, subtypes are created by extending the constraints of a parent.

The work of Chandra and McCann is also similar to that of Sekar, et. al., in that their systems emit packet matching functions in C. Aside from offering methods for matching data against a specific packet type and extracting fields from these types, the rest of the code would have to be written in C.

PPL is an attempt to develop these systems into a flexible language which can not only describe packets in terms of field layouts with constraints, but can also specify how and when to match data against these types and operate on instances of these types.

To illustrate these developments, here is a sample of PPL code which defines IP packets and a procedure. The procedure takes an IP packet and tests whether or not it contains a TCP or UDP packet:

```
packet uncheckedIP [signed int length] {
        unsigned int 4 big version;
        unsigned int 4 big header_length;
        unsigned int 8 big   TOS;
        unsigned int 16 big  datagram_length;
        unsigned int 16 big  identifier;
        unsigned int 3 big   flags;
        unsigned int 13 big  offset;
        unsigned int 8 big   TTL;
        unsigned int 8 big   protocol;
        signed int   16 big  header_checksum;
        unsigned int 32 big  source_ip_addr;
        unsigned int 32 big  dest_ip_addr;
        bytes [(header_length*4) - 20]     options;
        bytes [length - (header_length*4)]        data
}

packet IP extends uncheckedIP with {
        length == datagram_length;
        header_length >= 5
        }

import report_not_ip (bytes)->unsigned int
import report_bad_ip (IP)->unsigned int
import do_something_with_udp (bytes)->unsigned int
import do_something_with_tcp (bytes)->unsigned int

procedure ip_switch(IP x)->unsigned int =
        let ip_len = x.length in
        match x with {
           IP with {protocol == 17 }
              [ ip_len ]
              x_with_udp ->
                do_something_with_udp(x_with_udp->data)
        or IP with {protocol == 6 }
              [ ip_len ]
              x_with_tcp ->
                do_something_with_tcp(x_with_tcp->data)
        or _ -> report_bad_ip (x)
        }

procedure is_ip(bytes x)->unsigned int =
        match x with {
                IP [ x.length ] ip -> ip_switch(ip)
              or _ -> report_not_ip(x)
        }
```

In the next section, I will discuss the syntax and semantics of my packet processing language, and why each feature was developed.

Details of PPL, the packet processing language

This section contains details of the language in the form of a tutorial with snippets of sample code, grammar and syntax. Appendix A contains a full grammar and explanation of notational conventions.

Describing Packet Types and Types of Fields

The PPL type system started with the ideas of McCann and Chandra and Sekar, et. al. The simplest definition of packets is as a set of fields, where each field consists of a type and a name. The grammar for this is as follows[1]:

```
fields :=
        field
        | field ";" fields

simple_packet_description :=
        "packet" identifier "{" fields "}"
```

There are three classes of types which can be used as fields: basic primitive types, blocks of bytes, and packet types[2]. The basic primitive types are "unsigned int" and "signed int", and are twos complement and 32 bits in length[3]. A block of n bytes is n*8 bit long block of unspecified data.

These constructs are almost enough to describe the header of a UDP packet, as-is:

```
packet first_udp_header  {
    unsigned int            source_and_dst_port;
    unsigned int            packet_length_and_checksum
}
```

However, this system falls a little short, as the UDP packet has integer fields of 16 (not 32) bits. By adding modifiers which detail the length of a basic primitive type field, we get the following solution:

```
packet second_udp_header {
    unsigned int 16         sourceport;
    unsigned int 16         destport;
    unsigned int 16         packet_length;
    signed   int 16         checksum
}
```

The bit-length modifier is optional, and is only legal after a basic primitive type.

[1] The grammar conventions I use are as follows. Anything in *italics* references another rule in the grammar and/or a regular expression, anything in "quotemarks" denotes a terminal symbol. Alternate rules are given on separate lines; if a rule is longer than a line long, it will be indented on the next line so as to differentiate it from an alternation.

[2] However, due to the structure of packets, both "on the wire" and in my language's runtime representation, recursive and circular types are simply not possible and are non-intuitive.

[3] In the implementation that accompanies this paper, these are simply translated into their C analogues, and so behave the same way in PPL as they do in C.

Another thing which PPL has thus far neglected is byte ordering. The UDP protocol specifies that the higher-order bytes of each field must come first. This is not the normal byte-ordering for data on x86 personal computers, and not every protocol uses this ordering (although most do). The way that PPL handles this is by allowing an "endianess modifier" describing the byte ordering to be placed on basic primitive type fields, as illustrated in the following:

```
packet udp_header {
    unsigned int 16 big    sourceport;
    unsigned int 16 big    destport;
    unsigned int 16 big    packet_length;
    signed   int 16 big    checksum
}
```

This modifier is optional, and the values it can have are "big", "little", and "native", the latter of which will use the native byte-ordering. If no modifier is given, native byte-ordering will be assumed.

These are all the language constructs needed for describing the UDP header, but more are needed to describe the UDP packet. Here is the description of the UDP packet with the PPL constructs defined thus far:

```
packet first_udp {
    unsigned int 16 big    sourceport;
    unsigned int 16 big    destport;
    unsigned int 16 big    packet_length;
    signed   int 16 big    checksum;
    bytes                  data
}
```

The field at the end is for the bytes of data which the UDP packet is transporting across the network. The problem with this is that we have no way to know the length of this data, which presents a type-safety issue. For PPL to be safe in any way, it will have to be able to calculate the length of all runtime data, including packets and blocks of bytes.

To refer to the example given, the UDP packet type will need to carry information about the length of its fields around with it. Consider the following description of a UDP packet :

```
packet uncheckedUDP [signed int length] {
    unsigned int 16 big    sourceport;
    unsigned int 16 big    destport;
    unsigned int 16 big    packet_length;
    signed   int 16 big    checksum;
    bytes [length - 8]     data
}
```

Here, the number of bytes in the block is calculated by evaluating the expression "length - 8". A variable environment for this expression consists of variables declared at the beginning of the packet layout and all of the preceding fields. The values of these variables are carried with the packet at runtime. These values are also not changed during the lifetime of a packet; they are given to the instance of the packet type when it is first constructed. For this reason, these variables were named "parameters" to the packet. Only data of basic primitive types can be parameters to a packet type. A packet type can take multiple parameters.

Parameters to a packet type are a feature which does not appear in McCann and Chandra's system nor in the work of Sekar, et. al. Since matchings can be conducted from within PPL, the length of a PPL packet type needs to be inferable or known at all times. In the systems of McCann and Chandra or Sekar, et. al., matching functions are called from C and hence can be passed the length of the data buffer as a parameter in C.

Instead of allowing user-defined parameters, PPL could mandate that the runtime representation of every packet carry the length of the data buffer as private data. This was not done because it seemed that there was little disadvantage in allowing user-defined parameters, but there were advantages in allowing the programmer to give the packet some extra data.

The following example demonstrates user-defined parameters:

```
packet IP_with_prot [signed int length, unsigned int prot] {
    unsigned int 4 big version;
    unsigned int 4 big header_length;
    unsigned int 8 big   TOS;
    unsigned int 16 big  datagram_length;
    unsigned int 16 big  identifier;
    unsigned int 3 big   flags;
    unsigned int 13 big  offset;
    unsigned int 8 big   TTL;
    unsigned int 8 big   protocol;
    signed int   16 big  header_checksum;
    unsigned int 32 big  source_ip_addr;
    unsigned int 32 big  dest_ip_addr;
    bytes [(header_length*4) - 20]     options;
    bytes [length - (header_length*4)]       data
} with { prot == protocol }
```

If a field of a packet is of a packet type that takes parameters, the field can take parameters using the same syntax that the "bytes" type does. Here is an illustration of this:

```
packet uncheckedIP_with_UDP  [signed int length] {
        unsigned int  4 big                                         version;
        unsigned int  4 big                                         header_length;
        unsigned int  8 big                                         TOS;
        unsigned int 16 big                                         datagram_length;
        unsigned int 16 big                                         identifier;
        unsigned int  3 big                                         flags;
        unsigned int 13 big                                         offset;
        unsigned int  8 big                                         TTL;
        unsigned int  8 big                                         protocol;
        signed     int 16 big                                       header_checksum;
        unsigned int 32 big                                         source_ip_addr;
        unsigned int 32 big                                         dest_ip_addr;
        bytes [(header_length*4) - 20]                              options;
       uncheckedUDP [length - (header_length*4)]                    udp
} with { protocol == 17 }
```

This example also shows how PPL relates packets across protocol layers: by including packets as fields of other packets. This example is the first variably-sized packet shown which has fields of all three classes of types. The runtime-size of a packet is simply the summation of the size of its fields: no padding is required or permitted.

Expressions

The "with { protocol == 17 }" construct at the end of the above definition is a constraint on the value of a "protocol". A constraint is an expression which has a Boolean type. For data to match a given packet type, all of the constraints must evaluate to true. Expressions have been thus far been used in constraints and in passing parameters to fields of packet types and of the "bytes" type. Also, as PPL is an attempt to improve on previous work by adding the ability to operate on packets, it needs a means of operating on the data. Before discussing constraints in more depth, expressions need to be explained.

PPL offers many of the same operators that C does, and most are employed in the same way. The precedence chart lists all of the operators available in PPL from highest to lowest:

Operator	Operator Name
"(...)" "." "->"	parenthesis/procedure call, member access, field access
"*" "/" "%"	multiplication, division, modulus
"+" "-"	addition, subtraction
">" ">=" "<" "<="	greater-than, greater-than or equal-to, less-than, less-than or equal-to,
"==" "!="	equal-to, not equal-to
"&&"	logical-and
"||"	logical-or

The expressions which use the operators at the top of the chart are called the expression leaves. These operators are member accesses (".") , field accesses ("->"), procedure calls, variables, and parenthesized expressions. The grammar for them, in order, is:

```
expression_leaf :=
        expression_leaf "." identifier
        expression_leaf "->" identifier
        identifier "(" expressions ")"
        identifier
        ["0"-"9"]*
        "(" expression ")"
```

where *expressions* is a comma separated list of *expression*s The field access operation returns the fields of a packet. The member access operation gets the values of the parameters of a packet. The next two levels of operators are called the arithmetic operators, the next two the relational operators, and the final two the Boolean operators. These operators all work in the intuitive way.

Constraints and Extending Packet Types:

So far, we have seen how PPL relates packet types across protocol layers and how constraints can be added to packet types. This section will show how PPL allows programmers to create related packet types (perhaps of the same protocol) by refining them with additional constraints.

The constraints at the end of the packet type are a list of Boolean-typed expressions that have the parameters and fields of a packet as their variable environment. For a packet to match a packet type, all of the constraints need to be evaluated as true.

We have already talked about defining packets in terms of a field layout and optional constraints. An alternate means of defining packet types is to add constraints to an already defined type. Consider this re-working of a previous example:

```
packet uncheckedIP [signed int length] {
        unsigned int  4 big                                     version;
        unsigned int  4 big                                     header_length;
        unsigned int  8 big                                     TOS;
        unsigned int 16 big                                     datagram_length;
        unsigned int 16 big                                     identifier;
        unsigned int  3 big                                     flags;
        unsigned int 13 big                                     offset;
        unsigned int  8 big                                     TTL;
        unsigned int  8 big                                     protocol;
        signed     int 16 big                                   header_checksum;
        unsigned int 32 big                                     source_ip_addr;
        unsigned int 32 big                                     dest_ip_addr;
        bytes [(header_length*4) - 20]                          options;
        bytes [length - (header_length*4)]                      data
}

packet IP extends uncheckedIP with {
        length == datagram_length;
        header_length >= 5
        }

packet IP_with_UDP extends IP with { protocol == 17 }
```

Note how these types describe packets of the same protocol, and how they relate to each other. In this, the packet type "IP" inherits its field layout and parameters from "uncheckedIP". It also inherits constraints, but also adds two new ones. Similarly, "IP_with_UDP" inherits from fields and constraints from "IP" and extends it with one new constraint. Packet types that extended previously defined types are automatically subtypes of the packet types which they extend. Hence, "IP" is a subtype of "uncheckedIP", and "IP_with_UDP" is a subtype of both. This is the only way in which subtypes of packet types can be made. As is typical, the subtype relation is transitive, and an instance of a subtype can be used anywhere the parent type was expected. Packets which match the type "IP_with_UDP" will be of a specific subset of all IP packets: those which ought to contain UDP packets as their payload.

Now that all of the aspects of PPL's type system have been described, here is the grammar for it:

```
basic_primitive_type :=
        "signed" "int"
        "unsigned" "int"

type :=
        basic_primitive_type
        "bytes"
        identifier

variable_decl  :=
        type identifier

variable_decls  :=
        variable_decl
        variable_decl  "," variable_decls

packet_extension :=
        identifier "with" "{" constraints "}"

packet_layout :=
        "{" fields "}"
         "{" fields "}" "with" "{" constraints "}"

packet_type_defn :=
        "packet" identifier packet_layout
        "packet" identifier "[" variable_decls "]"   packet_layout
        "packet" identifier "extends" packet_extension
```

Matchings : casting data to types

PPL's constructs for describing and defining packets are quite similar to the system employed by McCann and Chandra's PacketTypes. Where PPL is a marked development from their system is in the ability to specify how and when to match and cast data to packet types, and what to do upon success or failure. The PPL construct for this is called the matching construct.

A matching is a matching expression and a series of cases. Each matching case consists of a variable declaration (of a type and variable name) and a statement. A matching case is said to "match" if the data buffer is long enough to be an instance of the type of the declaration and (if the given type is a packet type) if all constraints evaluate to true. Each matching case is attempted, in order, until one case matches. When a case matches, the data in the expression is bound to the variable and the statement is executed. Then, the matching is exited; no more matching cases are attempted. If the case fails, then the next matching case is attempt. At the end of the list of matching cases, there is a terminal catch-all matching case which everything matches. It is similar to the "default" case in the C "switch" construct.

A first attempt at defining the syntax for a matching statement would yield:

```
matching_case  :=
        variable_decl  "->"  statement

matching_cases  :=
        ""
        matching_case "or" matching_cases

matching :=
"match" expression "with" matching_cases "or"  "_" "->" statement
```

The variable declarations in the matching case of the above syntax present the same problems with that packet fields did. That is, if we are trying to match against packet type, how would we pass the parameters so that the required length can be calculated? Or, if we are trying to match against a byte of blocks, how would we express the number of bytes we expect to see? Also, could it be possible to match against a basic primitive type which has a size other than 32 bits and a different byte ordering than the machine?

The same solution of parameters, bit length modifiers, and endianness modifiers will work here. Since this is the second time this construct has been used, it will be called a *modified_variable_decl* and the grammar will be given:

```
identifier_opt :=
        identifier
        "_"

endianness_opt :=
        "big"
        "little"
        "native"
        ""
bit_length_opt  :=
        [0-9][0-9]
        ""
packet_type_with_data  :=
        identifier  "[" expressions "]"
        identifier
        packet_extension  "[" expressions "]"
        packet_extension

modified_variable_decl  :=
        basic_primitive_type  bit_length_opt endianness_opt   identifier_opt
        "bytes"  "[" expression "]"  identifier_opt
        packet_type_with_data   identifier_opt

matching_case  :=
        modified_variable_decl  "->"  statement
```

In *identifier_opt*, if a "_" is used instead of a variable *identifier*, no variable will be bound to the data upon matching though the statement will still be executed. In the *packet_type_with_data* rule, "[" *expressions* "]" is the list of values passed to the packet as parameters to the packet type, and are listed in the same order they are declared in the packet type. There need to be exactly as many expressions as there are parameters, and the value of each expression must be the same type or a subtype of the parameter to the packet type it correlates with. *modified_variable_decl* is the rule which actually appears in the syntax of packet fields.

Putting it all together: Procedures and statements

This section presents the rest of PPL by showing how matchings, packet types, and expressions can be glued together into statements and procedures which can be invoked from C or run on their own. This is not something that the respective systems of McCann and Chandra, or Sekar, et. al., are designed to do, and this is the main contribution of PPL to these systems.

There are two kinds of procedure in PPL, user-defined procedures, and declarations of C functions which will be linked into the executable. These are called "imported" procedures.

The grammar for user-defined procedures is:

procedure :=
"procedure" *identifier* "(" *variable_decls* ")" "->" *type* "=" *statement*

Note the "->" *type* part of the syntax. This is the return type of the procedure. For the procedure to typecheck, the statement requires a value in each possible path of execution, and the type of each possible value will have to be a subtype of the return type.

There are 4 statements in the definition of PPL: matching statements, let-bindings, statement lists, and statements of expressions.

statement :=
m*atching*
let_binding
statement_list
expression

The syntax of a matching has already been given, and statement of an expression is simply an expression.

Let-bindings have been introduced into the language so that that programmer can bind expressions to variables at their own convenience. The grammar of a let binding is:

let_binding :=
"let" *identifier* "=" *expression* "in" *statement*

Lists of statements have been introduced as statements to allow a series of events to happen in place of one. The grammar of a statement list is:

```
statement_list :=
        "{" statements "}"
```

where statements is a semi-colon separated list of *statement*s. Note that if the list is empty, this statement produces no executable code.

Determining if a statement has a value in each possible execution path, and what that value is, is done as follows:

- a matching statement has a value if the statement of each matching case has a value,
- a let-binding has a value if the embedded statement has a value,
- a statement list has a value if the list is non-empty, and has the value of the last statement in the list,
- the value of a statement of an expression is exactly what that expression evaluates to.

The grammar of imports is:

```
"import" identifier "(" types ")" "->" type
```

where *types* is a comma separated list of *type*'s. Here is an example of user-defined procedure and imported procedures:

```
import report_not_ip (bytes)->unsigned int
import report_bad_ip (IP)->unsigned int
import report_bad_ip_6 (IP)->unsigned int
import report_bad_ip_17 (IP)->unsigned int
import report_udp (UDP)->unsigned int
import report_unchecked_udp (uncheckedUDP)->unsigned int
import report_tcp (TCP)->unsigned int
import report_unchecked_tcp (uncheckedTCP)->unsigned int

procedure ip_switch(IP x)->unsigned int =
    let ip_len = x.length in
    let ip_hdr_len = x->header_length in
    match x with {
      IP with {protocol == 17 }
        [ ip_len ]
        x_with_udp ->
          match x_with_udp->data with {
            UDP [ ip_len - ip_hdr_len*4 ] udp ->
                report_udp(udp)
          or uncheckedUDP [ ip_len - ip_hdr_len*4 ] udp ->
                report_unchecked_udp(udp)
          or _ -> report_bad_ip_17 (x_with_udp)
          }
    or IP with {protocol == 6 }
        [ ip_len ]
        x_with_tcp ->
          match x_with_tcp->data with {
            TCP [ip_len - ip_hdr_len*4 ] tcp ->
                report_tcp(tcp)
          or uncheckedTCP [ip_len - ip_hdr_len*4 ] tcp ->
                report_unchecked_tcp(tcp)
          or _ -> report_bad_ip_6 (x_with_tcp)            }
    or _ -> report_bad_ip (x)
    }
```

C Representations of PPL Data

The PacketTypes of McCann and Chandra generate code which is meant to be used in programs written in C. My PPL compiler emits C code, but PPL programs can be run with minimal C code, which would be needed to do things like open and close files and read from standard input. This ability to interoperate with C increases the power and utility of PPL; C functions can be called from PPL, and PPL functions can be called from C.

To make this interface independent of PPL implementation, some aspects of C representations of PPL data and types must be standardized. Here are the relevant implementation details of my PPL compiler.

For C functions and PPL procedures to be able to call each other, the programmer needs to understand how the PPL namespace maps into C. All PPL procedures will carry the same names in both languages. Data of the basic primitive types are represented in C as 32 bit int and unsigned int, as appropriate. A packet type named "a" will be called "struct a" in C, and will have a field for each parameter of the packet type, and also a final field of type "char called "__ppl_packet_data". The name of the "bytes" array is "struct bytes", and it is has a field of type "int" called "length", and a second field of type "char *" called "__ppl_array_data". PPL makes shallow copies of these structs, and hence does no heap allocation nor deallocation. In its current incarnation, PPL has no way of creating packets, and so whoever allocated the space for the data should also free it when it is done being used.

Note that if a PPL procedure taking a packet type is called from C, it is up to the C program to check the length of the data and constraints. To be typesafe, the C program ought instead to call a PPL procedure takes a "bytes" struct, and matching it to the desired packet type.

Analysis

Identifying packets is a pattern matching activity. This project and previous work on packet specification languages are attempts at making this task easier. McCann and Chandra recognize that ML-style pattern matching would be the most intuitive way expressing the logic of packet identification and network code. However, they are reluctant to use this as a solution. ML is too slow of a language for writing low-level networking code. For example, it would require stuffing packet data into ML data type formats.

PPL is an attempt to use ML-style pattern matching with a PacketTypes-style type system in a language which is close enough to C to preserve its efficiency. This is illustrated in the following example:

```
import report_not_ip (bytes)->unsigned int
import report_bad_ip (IP)->unsigned int
import do_something_with_udp (bytes)->unsigned int
import do_something_with_tcp (bytes)->unsigned int

procedure ip_switch(IP x)->unsigned int =
        let ip_len = x.length in
        let ip_hdr_len = x->header_length in
        match x with {
          IP with {protocol == 17 }
            [ ip_len ]
            x_with_udp ->
              do_something_with_udp(x_with_udp->data)
        or IP with {protocol == 6 }
            [ ip_len ]
            x_with_tcp ->
              do_something_with_tcp(x_with_tcp->data)
        or _ -> report_bad_ip (x)
        }
```

This PPL code produces the following C code:

```
/*Importing C Function report_not_ip*/
unsigned int report_not_ip (struct bytes);


/*Importing C Function report_bad_ip*/
unsigned int report_bad_ip (struct uncheckedIP);


/*Importing C Function do_something_with_udp*/
unsigned int do_something_with_udp (struct bytes);


/*Importing C Function do_something_with_tcp*/
unsigned int do_something_with_tcp (struct bytes);
```

```
/* Defining Procedure ip_switch*/
unsigned int
ip_switch(struct uncheckedIP x) {
 int ip_len = x.length;
 {
  unsigned int ip_hdr_len = __ppl_field_header_length_of_uncheckedIP
    (x);
  {
    struct uncheckedIP __ppl_matching_exp = x;
     int __ppl_matching_bitlength = __ppl_length_of_uncheckedIP(
       __ppl_matching_exp);
     unsigned char * __ppl_matching_data;
     __ppl_matching_data = __ppl_matching_exp.__ppl_packet_data;


     int __ppl_annon_3 = ip_len;
     if (__ppl_match___ppl_annon_1(__ppl_annon_3, __ppl_matching_data,
       __ppl_matching_bitlength)){
      struct uncheckedIP x_with_udp = __ppl_pack_uncheckedIP( __ppl_annon_3,
        __ppl_matching_data);
      return do_something_with_udp(__ppl_field_data_of_uncheckedIP
        (x_with_udp));
     }
       else {
      int __ppl_annon_4 = ip_len;
      if (__ppl_match___ppl_annon_2(__ppl_annon_4,
        __ppl_matching_data, __ppl_matching_bitlength
        )){
       struct uncheckedIP x_with_tcp = __ppl_pack_uncheckedIP( __ppl_annon_4,
         __ppl_matching_data);
       return do_something_with_tcp(__ppl_field_data_of_uncheckedIP
         (x_with_tcp));
      }
        else {
       return report_bad_ip(x);
      }
     }
    }
   }
  }
```

By offering an ML-inspired matching construct, we allow the programmer to specify when to try to match data against a packet type, which types to try in this matching, what to do if it fails, and what to do if it passes. Looking at the above example, it doesn't appear that the expense of this system over a straight C implementation would be as bad as McCann and Chandra might have feared in terms of memory or time. However, experiments would have to be run to test this for sure. Unfortunately, that was not something that could have been accomplished in the time allotted for this project.

Furthermore, the C translations of PPL datatypes do not require any more memory than the memory buffer containing the packet and the parameters to the packet the which the PPL programmer declares, and perhaps a temporary runtime stack for the functions defined for a PPL datatype. As these are needed for calculating the length of a packet, this extra data is a cheap price to pay for type safety.

PPL is a typesafe programming language built around describing and matching packet types. By implementing packet types and pattern matchings and using the "procedure" construct to bundle up PPL code in a way easily accessible from C, we abstract the programmer from low-level and error-prone C programming at little cost in efficiency and memory. This helps to reduce the amount of networking code that needs to be manually written in C, in hopes of speeding development time and reducing programming errors.

Future Work

Although PPL is quite a flexible language, there are various additions that could make it more powerful.

Some ideas for expanding the language are:

- Making Boolean primitives ("true" and "false"), and allowing variables to be assigned to Boolean typed values.
- Adding a string type and operators both for typical string manipulation, for converting integers to strings, and for printing strings to standard output. This would make packet reporting easier, without deferring to C for string construction.
- Only allowing C programs to call PPL procedures which take basic primitive types or bytes of blocks. This would allow PPL to check if data is the correct size and matches all necessary constraints before allowing it to have a packet type.
- Adding preprocessing so as to include common packet type definitions from header files.
- PPL has many built-in facilities for packet reading. A useful addition to the language would be constructs for creating new packets, or modifying packets in place. If PPL had the ability to both read and write packets, I believe that a significant portion of a network stack could be written in PPL.
- Adding algebraic types, such as unions, would be an interesting endeavor, although a similar effect is already created by extending a type into various different subtypes.
- Linking a mathematical engine in with the typechecker could allow many complicated relations to be made between the packet types. It could perhaps prove, for example, that certain matching cases were simply unreachable, or that the sets of data that would match each case were mutually exclusive, allowing them to be tested in any order.
- Another idea would be to allow matchings to be labeled with an identifier, and treat both packet types and labeled matchings as first class objects. Then, we could add language constructs which would allow runtime-created matching cases to be added to or removed from labeled matchings. This would be akin to what other languages like Dylan have as dynamic dispatching to multi-methods. An

example application of this would be automatically updating router tables: the table would be stored as a labeled matching, rather than as a more typical data structure, and as more data comes in, more matching_cases can be constructed and added to the labeled matching.

Conclusions

PPL, though a small language, is extremely flexible and is capable of describing packets of various protocols, real or still on the drawing board. PPL is not limited to using only a subset of current protocols, nor is it limited to simply reporting the packets it identifies. PPL gives the programmer more power than other systems do by allowing the programmer to define how a matching should be attempted. PPL lowers the amount of network code that needs to be written in C while hence hopefully reducing programming errors and increases safety. I hope that PPL will be the basis of future projects which will upgrade PPL and use it in the creation real network appliances, such as self-updating routers, typesafe network stacks, and intrusion detection systems.

Bibliography ======

[1] Karthikeyan Bhargavan, Satish Chandra, and Peter J. McCann, "What Packets May Come: Automata for Network Monitoring," 2001.

[2] Yang Guang, "A Real-time Packet Filtering Module for Network Intrusion Detection System," Jul 1998.

[3] Satish Chandra, Peter J. McCann, "Packet Types: Abstract Specification of Network Protocol Messages," 1999

[4] R. Sekar, Y. Guang, S. Verma and T. Shanbhag, "A High-Performance Network Intrusion Detection System," ACM Symposium on Computer and Communication Security, 1999.

[5] R. Sekar, A. Gupta et al.,"Specification-based anomaly detection: a new approach for detecting network intrusions," ACM Computer and Communication Security Conference (CCS), 2002

[6] R. Sekar, R. Ramesh and I.V. Ramakrishnan, "Adaptive Pattern Matching," International Colloquium on Automata, Languages and Programming (ICALP), July 1992.

Appendix 1:
Grammar for PPL

The grammar conventions I use are as follows. Anything in *italics* references another rule in the grammar and/or a regular expression, anything in "quotemarks" denotes a terminal symbol. Alternate rules are given on separate lines; if a rule is longer than a line long, it will be indented on the next line so as to differentiate it from an alternation. An *identifier*, in PPL, must start with a letter and may have numbers and "_" after the first symbol. As a regular expression, it is

*[a-zA-Z][a-zA-Z0-9_]**

Although many languages allow it, PPL does not allow identifiers to start with "_". This is so that PPL compilers can emit private identifiers outside of the legal PPL namespace.

A PPL program is a series of components; *component* is the top level of the PPL grammar:

```
component :=
        import
        procedure
        typedef

import :=
        "import" identifier "(" types ")" "->" type


basic_primitive_type :=
        "signed" "int"
        "unsigned" "int"


type :=
        basic_primitive_type
        "bytes"
        identifier


variable_decl :=
        type identifier


variable_decls :=
        variable_decl
        variable_decl "," variable_decls


packet_type_defn :=
        "packet" identifier packet_layout
        "packet" identifier "[" variable_decls "]"   packet_layout
        "packet" identifier "extends" packet_extension


packet_layout :=
        "{" fields "}"
         "{" fields "}" "with" "{" constraints "}"
```

packet_extension :=
    identifier “with” “{“ *constraints* “}”

fields :=
    modified_variable_decl
    modified_variable_decl “;” *fields*

modified_variable_decl :=
    basic_primitive_type bit_length_opt endianness_opt identifier_opt
    “bytes” “[“ *expression* ”]” *identifier_opt*
    packet_type_with_data identifier_opt

identifier_opt :=
    identifier
    “_”

endianness_opt :=
    “big”
    “little”
    “native”
    “”

bit_length_opt :=
    [0-9][0-9]
    “”

packet_type_with_data :=
    identifier “[” *expressions* ”]”
    identifier
    packet_extension “[” *expressions* ”]”
    packet_extension

constraints :=
    expression
    expression “;” *constraints*

expression :=
    expression “||” *exp_and*
    exp_and

exp_and :=
    exp_and “&&” *exp_rel_eq*
    exp_rel_eq

```
exp_rel_eq  :=
            exp_rel_eq  “==”  exp_rel_ineq
            exp_rel_eq  “!=”  exp_rel_ineq
            exp_rel_ineq


exp_rel_ineq  :=
            exp_rel_ineq  “>=”  exp_factor
            exp_rel_ineq   “>”  exp_factor

            exp_rel_ineq  “<=”  exp_factor
            exp_rel_ineq  “<”  exp_factor
            exp_factor


exp_factor  :=
            exp_factor  “+”  exp_term
            exp_factor   “-”  exp_term
            exp_term


exp_term  :=
            exp_term  “*”  exp_leaf
            exp_term  “/”  exp_leaf
            exp_term  “%”  exp_leaf
            exp_leaf


expression_leaf  :=
            expression_leaf  “.” identifier
            expression_leaf  “->” identifier
            identifier  “(“ expressions “)”
            identifier
            [0-9]*
            “(“ expression “)”


procedure  :=
            “procedure” identifier “(“ variable_decls “)” “->” type “=” statement


statement  :=
            matching
            let_binding
            statement_list
            expression


let_binding  :=
            “let” identifier “=” expression “in” statement


statement_list :=
            “{“ statements “}”
```

matching_case :=
    modified_variable_decl "->" *statement*

matching_case :=
    variable_decl "->" *statement*

matching_cases :=
    ""
    matching_case "or" *matching_cases*

matching :=
    "match" *expression* "with" *matching_cases* "or" "_" "->" *statement*

Appendix 2:
“Well Formed” Rules

The text does not refer to this appendix, but it is included as a resource for some of the more technical details of the language definition.

Determining Well-Formedness of *modified_param_decl*s:

A *modified_param_decl* is well-formed if the following conditions hold:

- If the *modified_variable_decl* has the type "unsigned int" or "signed int", if it has a bit-length modifier, the given length is in the range 1-32.
- If the *modified_variable_decl* is a block of bytes, then the expression dictating the length is well-formed and has the type "unsigned int" or "signed int".
- If the *modified_variable_decl* has a packet type, then exactly as many values are being passed as parameter to the packet type as the type declares, and the type of each of these expressions is a subtype of the type of the associated parameter to the packet type. If the packet type is given by name, then the name must be previously bound to a packet type.
- If the *modified_variable_decl* has a packet type which is given as a packet extension, then the above condition must hold, and the packet extension must be well formed, as specified bellow.

Determining Well-Formedness of *packet_type_defn*s:

A packet type definition binds a new packet type to an identifier. A packet type definition is well-formed if the indentifier given is not yet bound to a packet type and the following conditions hold:

If a packet type is given as an extension, then it must be well-formed. That is:

- The *identifier* it is claiming to extend has already been bound to by a packet type.
- The constraint list is not empty, and each constraint is a well-formed, Boolean-typed expression which uses only the parameters and the fields of the packet type being extended as a variable environment.

If a packet type is given as a layout, then it must be well-formed. That is:

- The formal parameters are either signed or unsigned ints.
- The fields, as *modified_variable_decl*s, are well-formed and use only the parameters to the packet as a variable environment.
- All fields which are blocks of bytes or have a packet type are byte-aligned.
- All of the identifiers used for fields or parameters to the packet type are (mutually) unique.
- Each constraint is a well-formed, Boolean-typed expression which uses only the parameters and the fields of the packet type being extended as a variable environment.

Determining the Type and Well-Formedness of an expressions leaf:

If the expression leaf is a member access, then it is well-formed if the expression being accessed has a packet type which has a parameter with the same name as the identifier given. The resulting value has the type of the packet type parameter.

If the expression leaf is a field access, then it is well-formed if the expression given has a packet type which has a field with the same name as the identifier given. The resulting value has the type of the packet type field.

If the expression leaf is a procedure call, then it is well-formed if the identifier naming the procedure is already bound to a procedure (imported or user defined), and the exact same amount of values are supplied as the procedure is declared to take. Furthermore, the expressions given for the parameters all have types which are subtypes of the procedure's formal parameters. The resulting value has the return type of the procedure.

If the expression leaf is a variable, then it is well-formed if it is bound in the current environment. If the expression leaf is a constant, then it is well-formed if it is in the range 0 to (2^{32} - 1). If the expression leaf is a parenthesized expression, then it is well formed if the enclosed expression is.

Determining the Type and Well-Formedness of Statements and Procedure:

A statement is well-formed if the following conditions apply:

- If the statement is a "let binding", then it is well-formed if the embedded statement is well-formed and the expression that the variable is being assigned to is well-formed and the value is either a block of bytes, is a signed or unsigned int, or has a packet type.
- If the statement is a statement list, then it is well-formed if and only if each statement in the list is well-formed.
- If the statement is an expression, then it is well-formed in the expression is well-formed.
- If the statement is a matching, then it is well-formed if (1) the expression being matched is well-formed and is either a block of bytes, has a signed or unsigned int, or has a packet type, (2) each matching case consists of a well-formed "*modified_variable_decl*"and a well-formed "*statement*", and (3) the statement in the terminal "_" matching case is well-formed.

A user defined procedure is well-formed if the name of the procedure is not yet bound to any other procedure (user-defined or imported), every parameter has a unique name, the statement is well-formed, and any parameter type or return type that is given by name is previously bound to a packet type. Imported procedures are well-formed if and only if the name of the procedure is not yet bound to any other procedure (user-defined or imported) , and any parameter type or return type that is given by name is previously bound to a packet type.