Rust For Safer Protocol Development

Vivian Band, Stephen McQuistin, Dejice Jacob, Colin Perkins
University of Glasgow
Improving Protocol Standards
Improving Internet Standards

The Internet Engineering Task Force (IETF) standardises network protocols.
Improving Internet Standards

Packets and other protocol data units (PDUs) are illustrated using ASCII art and text descriptions in drafts and standards documents.

However, these diagrams and descriptions can contain errors and ambiguities:
Improving Internet Standards

With only minor changes to ASCII diagram and text description syntax, we can create machine-readable standards documents (Augmented Packet Header Diagrams):
Improving Internet Standards

These machine-readable documents allow us to build a custom, typed model of protocol data, called Network Packet Representation.

This typed model is language-agnostic, so can be used to create parser libraries for any programming language.
Improving Internet Standards

My internship work focused on the automatic generation of Rust parser libraries from our custom protocol type system.

We wanted our generated parsers to have strict typing and safety guarantees.
Typing for Network Protocols
Typing for Network Protocols

Before we can start building parsers, we need to know which building blocks we have.

There are several distinct protocol types we can draw from ASCII diagrams in Internet standards documents.
Typing for Network Protocols

Packet fields which contain raw data can be modelled as a bit string:
Typing for Network Protocols

Packet fields which could contain several instances of the same type can be modelled as an array:
Some packets contain fields which vary based on other values or which must meet a specific requirement.

We can call these constraint types:
Typing for Network Protocols

The content of some fields can depend on information from other packets or PDUs.

We would need context types to hold this information:
Typing for Network Protocols

A known, limited set of possible values can be modelled as an enum type:

3. TCP Options

A TCP Option is one of: a EOL Option, a NOOP Option, a Maximum Segment Size Option, a Window Scale Factor Option, a Timestamp Option, or a SACK Permitted Option.

An EOL Option is formatted as follows:

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
```

where:

Option Kind (Kind): 1 byte; Kind == 0. This option code indicates the end of the option list.

A NOOP Option is formatted as follows:
Typing for Network Protocols

Packets can be considered as Struct-like types containing a series of packet field types:
Typing for Network Protocols

Some protocols require functions to convert between dependent PDUs (eg. encryption and decryption in QUIC).

These can be specified as function types:

2. Header and Packet Protection

A Protected Packet is either a Protected Long Header Packet or a Protected Short Header Packet.

An Unprotected Packet is either a Long Header Packet or a Short Header Packet.

An Unprotected Packet is parsed from a Protected Packet using the remove_protection function. The remove_protection function is defined as:

```go
func remove_protection(from: Protected Packet) -> Unprotected Packet:
    remove header protection from protected_packet
    remove packet protection from protected_packet
    construct appropriate packet type
    return Unprotected Packet
```
Typing for Network Protocols

Types in our Network Packet Representation system:

- Bit string
- Array
- Struct
- Constraint
- Context
- Enum
- Function
Automatic Rust Parser Generation
Automatic Rust Parser Generation

How can we use these types to build Rust parser combinators?
Automatic Rust Parser Generation

```rust
#[derive(Debug, PartialEq, Eq)]
struct TcpHeaderSourcePort(u16);
```
Automatic Rust Parser Generation

#[derive(Debug, PartialEq, Eq)]

struct TcpHeaderSourcePort(u16);

pub fn parse_tcp_header_source_port<'a>(input: (&'a [u8], usize), context: &'a mut Context) -> (IResult<(&'a [u8], usize), TcpHeaderSourcePort>, &'a mut Context) {
    take(16 as usize)(input).map(|(i, o)| (i, TcpHeaderSourcePort(o))), context
}
Automatic Rust Parser Generation

```rust
#[derive(Debug, PartialEq, Eq)]
struct TcpHeaderSourcePort(u16);

pub fn parse_tcp_header_source_port<'a>(
    input: (&'a [u8], usize),
    context: &'a mut Context
) -> (IResult<(&'a [u8], usize), TcpHeaderSourcePort>, &'a mut Context) {
    (take(16 as usize)(input).map(|(i, o)| (i, TcpHeaderSourcePort(o))), context)
}
```

Incoming packet data, bit-level counter
Automatic Rust Parser Generation

#[derive(Debug, PartialEq, Eq)]
struct TcpHeaderSourcePort(u16);

pub fn parse_tcp_header_source_port<'a>(input: (&'a [u8], usize), context: &'a mut Context) -> (IResult<(&'a [u8], usize), TcpHeaderSourcePort>, &'a mut Context) {
    (take(16 as usize)(input).map(|(i, o)| (i, TcpHeaderSourcePort(o))), context)
}
Automatic Rust Parser Generation

```
#[derive(Debug, PartialEq, Eq)]
struct TcpHeaderSourcePort(u16);

pub fn parse_tcp_header_source_port<'a>(input: (&'a [u8], usize), context: &'a mut Context) -> (IResult<(&'a [u8], usize), TcpHeaderSourcePort>, &'a mut Context) {
    take(16 as usize)(input).map(|(i, o)| (i, TcpHeaderSourcePort(o))), context
}
```
# [derive(Debug, PartialEq, Eq)]

struct TcpHeaderSourcePort(u16);

pub fn parse_tcp_header_source_port<'a>(input: (&'a [u8], usize), context: &'a mut Context) -> (IResult<(&'a [u8], usize), TcpHeaderSourcePort>, &'a mut Context) {
    (take(16 as usize)(input).map(|(i, o)| (i, TcpHeaderSourcePort(o))), context)
}
Automatic Rust Parser Generation

#[derive(Debug, PartialEq, Eq)]

struct TcpHeaderSourcePort(u16);

pub fn parse_tcp_header_source_port<'a>(input: (&'a [u8], usize), context: &'a mut Context) -> (IResult<(&'a [u8], usize), TcpHeaderSourcePort>, &'a mut Context) {
    (take(16 as usize)(input).map(|(i, o)| (i, TcpHeaderSourcePort(o))), context)
}
#[derive(Debug, PartialEq, Eq)]

struct TcpHeaderSourcePort(u16);

pub fn parse_tcp_header_source_port<'a>(input: (&'a [u8], usize), context: &'a mut Context) -> (IResult<(&'a [u8], usize), TcpHeaderSourcePort>, &'a mut Context) {
    (take(16 as usize)(input).map(|(i, o)| (i, TcpHeaderSourcePort(o))), context)
}

Take 16 bits from the input data, assign to TcpHeaderSourcePort as a value
Parses a TCP Header

This header structure, and each of its respective fields, has its own data type which may contain sub-types.

We can perform a depth-first search to generate all the individual parsers required to build a parser combinator for this header structure.
Parsing a TCP Header
Parsing a TCP Header

TCP Header

Internet-Draft

TCP APHD Example

November 2020
Parsing a TCP Header
Parsing a TCP Header
Parsing a TCP Header
3. TCP Options

A TCP Option is one of: a EOL Option, a NOOP Option, a Maximum Segment Size Option, a Window Scale Factor Option, a Timestamp Option, or a SACK Permitted Option.

An EOL Option is formatted as follows:

```
0 0
1 0
2 0
3 0
```

where:
Option Kind (Kind): 1 byte; Kind == 0. This option code indicates the end of the option list.

A NOOP Option is formatted as follows:

```
0 1
1 1
```

where:
Option Kind (Kind): 1 byte; Kind == 1. This option code can be used between options, for example, to align the beginning of a subsequent option on a word boundary.
3. TCP Options

A TCP Option is one of: a EOL Option, a NOOP Option, a Maximum Segment Size Option, a Window Scale Factor Option, a Timestamp Option, or a SACK Permitted Option.

An EOL Option is formatted as follows:

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
```

where:

Option Kind (Kind): 1 byte; Kind == 0. This option code indicates the end of the option list.

A NOOP Option is formatted as follows:

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
```

where:

Option Kind (Kind): 1 byte; Kind == 1. This option code can be used between options, for example, to align the beginning of a subsequent option on a word boundary.
3. TCP Options

A TCP Option is one of: a EOL Option, a NOOP Option, a Maximum Segment Size Option, a Window Scale Factor Option, a Timestamp Option, or a SACK Permitted Option.

An EOL Option is formatted as follows:

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
```

where:

- **Option Kind (Kind):** 1 byte, Kind = 0. This option code indicates the end of the option list.

A NOOP Option is formatted as follows:

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
```

where:

- **Option Kind (Kind):** 1 byte, Kind = 1. This option code can be used between options, for example, to align the beginning of a subsequent option on a word boundary.
3. TCP Options

A TCP Option is one of: a EOL Option, a NOOP Option, a Maximum Segment Size Option, a Window Scale Factor Option, a Timestamp Option, or a SACK Permitted Option.

An EOL Option is formatted as follows:

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-------------------------------+-------------------------------+
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
```

where:

Option Kind (Kind): 1 byte; Kind == 0. This option code indicates the end of the option list.

A NOOP Option is formatted as follows:

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
```

where:

Option Kind (Kind): 1 byte; Kind == 1. This option code can be used between options, for example, to align the beginning of a subsequent option on a word boundary.
Parsing a TCP Header
Parsing a TCP Header
Parsing a TCP Header

Stage 1: Machine-readable protocol document

Stage 2: Network Packet Representation tree

Stage 3: Rust parser code
`crate` `derive(Debug)`

```rust
pub struct TcpHeader {
    pub source_port: TcpHeaderSourcePort,
    pub destination_port: TcpHeaderDestinationPort,
    pub sequence_number: TcpHeaderSequenceNumber,
    pub acknowledgment_number: TcpHeaderAcknowledgmentNumber,
    pub data_offset: TcpHeaderDataOffset,
    pub reserved: TcpHeaderReserved,
    pub cwr: TcpHeaderCwr,
    pub ece: TcpHeaderEce,
    pub urg: TcpHeaderUrg,
    pub ack: TcpHeaderAck,
    pub psh: TcpHeaderPsh,
    pub rst: TcpHeaderRst,
    pub syn: TcpHeaderSyn,
    pub fin: TcpHeaderFin,
    pub window_size: TcpHeaderWindowSize,
    pub checksum: TcpHeaderChecksum,
    pub urgent_pointer: TcpHeaderUrgentPointer,
    pub options: Option<TcpHeaderOptions>,
    pub payload: TcpHeaderPayload,
}
```
Generated TCP Header Parser

```rust
pub fn parse_tcp_header<'a>(mut input: &'a [u8], mut context: &'a mut Context) -> (Result<(&'a [u8], usize), &'a mut Context>) {
    let source_port = match parse_tcp_header_source_port(input, context) {
        (Ok((i, c)), o) => {
            input = i;
            context = c;
        }
        (Err(e), o) => return (Err(e), o),
    };

    let destination_port = match parse_tcp_header_destination_port(input, context) {
        (Ok((i, c)), o) => {
            input = i;
            context = c;
        }
        (Err(e), o) => return (Err(e), o),
    };

    let sequence_number = match parse_tcp_header_sequence_number(input, context) {
        (Ok((i, c)), o) => {
            input = i;
            context = c;
        }
        (Err(e), o) => return (Err(e), o),
    };

    let acknowledgment_number = match parse_tcp_header_acks_number(input, context) {
        (Ok((i, c)), o) => {
            input = i;
            context = c;
        }
        (Err(e), o) => return (Err(e), o),
    };
}
```
Conclusions
Conclusions

We already use a depth-first search to assign types to protocol data using our custom typing system.

Nom uses the same process to build parser combinators.

Matching the two makes sense!
Conclusions

Automatic parser generation minimises the chance of mistakes in complex parsers which would be difficult to write manually.

Our custom typed protocol data, along with Rust’s strong typing guarantees, makes it much more likely that we’ll catch errors early in protocol development.
Next Steps
Next Steps

Wider adoption of automatic parser generation within the IETF.

Implement functions to allow processing of encrypted headers in protocols like QUIC.

Automated testing for protocols themselves in practice - we can find errors now, but how can they be used to automatically correct errors?

Support more output languages.
Rust for Safer Protocol Development

Resources for our automatic parser generation project can be found at these locations:

https://github.com/glasgow-ipl/ips-protodesc-code/

http://dx.doi.org/10.5525/gla.researchdata.1083

This work is funded by the UK Engineering and Physical Sciences Research Council, under grant EP/R04144X/1
Rust For Safer Protocol Development

Vivian Band, Stephen McQuistin, Dejice Jacob, Colin Perkins
University of Glasgow