bitnest

Release 0.1.0

Quansight

Dec 02, 2021

CONTENTS:

CHAPTER

ONE

CONTRIBUTING

1.1 Architecture

At a high level bitnest is a compiler that generates parsers for binary data. Compilers can be represented in three sections frontend, transform, backend. In literature the transform step is usually referred to as term-rewriting.

1.1.1 Frontend

In the case of Bitnest the frontend is a python domain specific language (DSL). This was chosen because of the burden on creating new languages and that python is flexible enough to accurately represent the relationship between structs, fields, vectors, and unions. Great examples of these are in the models directory. Once a binary structure is represented in the model it is quickly converted into an internal AST done in bitnest/field.py. The internal AST structures is simply a nesting of python tuples. Why such a simple structure? This simple structure makes the later stages much easier to handle. And in my opinion where LISP shines. These nested tuple structures are wrapped in the Expression class which allows for the pythonic construction of the AST. Sometimes this class makes it easier to build a tree. For example suppose we have `1 + a`. In the internal AST this would be represented as

```
Expression((Symbol("add"), (Symbol("integer"), 1), (Symbol("variable"), "a")))
```

We can easily construct this using `bitnest.core` via `Integer(1) + Variable("a")`.

1.1.2 Transform/Analysis

Now that we have an internal AST we need to transform and analyze the AST. Often times it can be confusing when working with the internal AST since it is hard to keep track of the passes being made and where we are in the process. To help with this we have adopted the LLVM approach to working with the AST. We can think of each transformation as a logical pass over the tree. Within bitnest there are three types of passes.

The `analysis` pass will not modify the internal AST. It will analyze the AST and return information in python datastructures. See bitnest/analysis. An example is `inspect_datatypes.py` which returns the fields within each datatype, conditions for each datatype, and other useful information for understanding the AST.

The `transform` pass will modify the internal AST. The modifications can be simpler such as `(Symbol("add"), (Symbol("integer"), 1), (Symbol("add"), (Symbol("integer"), 2), (Symbol("integer"), 3)))` to `(Symbol("add"), (Symbol("integer"), 2), (Symbol("integer"), 3)`. The idea is to convert `(+ 1 (+ 2 3))` to `(+ 1 2 3)`. This is just one example of a transformation. This transformation is done in bitnest/transform/arithmetic_simplify.py.

Finally we have the `backend`. This pass takes the internal AST after several transformations and generates code that can then be executed independent of the compiler. The simplest example of this is the bitnest/backend/python.py. This

will convert for example `(Symbol("add"), (Symbol("integer"), 1), (Symbol("integer"), 2))` into `ast.BinOp(ast.Add(), 1, 2)` into `(1 + 2)`. Here bitnest lowers the internal AST into the python ast which then gets written to python source code.

It is common for multiple passes to be performed. Each pass through the tree is done in a specific order. Using the following labeling node `N`, left most child node `L`, right most child node `R`. See tree traversal currently pre-order (NL->R) and post-order (L->RN) are being used. In some cases both are used together (see bitnest/transform/realize_offsets.py which visit in the following pattern (NL->RN).

1.1.3 Backend

The backend as mentioned previously is responsible for generating executable code. The first example of this the python backend bitnest/backend/python.py. However, are being written for more efficient execution.

1.1.4 End-To-End Example

Take the following example of a MILSTD 1553 Packet

```
from models.simple import MILSTD_1553_Message

MILSTD_1553_Message.expression() \
    .transform("realize_datatypes") \
    .transform("realize_conditions") \
    .transform("realize_offsets") \
    .transform("parser_datatype") \
    .transform("arithmetic_simplify") \
    .backend("python")
```

Will generate the following code. Currently simple output but this will improve over time.

```
__datatype_mask = 0
if __bits[8:13] == 31:
    __datatype_mask = __datatype_mask | 1
if __bits[13:16] == 0:
    __datatype_mask = __datatype_mask | 2
```

1.2 Internal AST Nodes

Bitnest uses a LISP like internal representation of the abstract syntax tree (ast). In this section we will describe the structure of each of these nodes. At a high level there are three types of nodes. `Arithmetic` nodes deal with math operations on variables. `Field` nodes are data representations of the structure and relationship between structures. Finally `Programming` nodes are used near the end of the AST transformation to represent things that are necessary for representing programs to process the structures.

1.2.1 Arithmetic Nodes

- unary operation: invert `not`
- logical and `logical_and`, logical or `logical_or`
- bitwise and `bit_and`, bitwise or `bit_or`
- addition `add`, subtraction `sub`, multiply `mul`, floor divide `floordiv`, true divide `truediv` all support `N` arguments `(<op> <1> ... <N>)`
- modulus `mod`
- equal `eq`, not equal `ne`, less than `lt`, greater than `gt`, less than equal `le`, greater than equal `ge`
- integer `integer`
- float `float`
- enum `enum`

1.2.2 Field Nodes

- `field` is sequential bits not necessarily byte aligned. There are many Field subclasses such as `UnsignedInteger`, `SignedInteger`, `Bits`, etc. which is represented by a
- `struct` is an ordered collection of Fields, Union of Structs, and Vectors or Structs
- `union` is a set of `Struct`'s
- `vector` is `N` repeats of a given `Struct` represented as `(Symbol('vector'), <Struct> <length>)`
- `datatype` is a representation of a concrete structure (without `union` in the tree) important for reasoning about the size of a structure. Generated in the `realize_datatype` transform pass.

1.2.3 Programming Nodes

- `quote` is to protect it's arg from evaluation (not used much in bitnest) but a valuable concept in lisp `(quote (...))`
- `list` defined a list of `N` elements (represented as the args) `(list <1> ... <N>)`
- `assign` meant for assignment statements e.g. `(assign (variable a) (integer 1))` which is equivalent to `a = 1`.
- `variable` which represents a given variable in the code that does not have a set value. `(variable "<name>")`.
- for convenience there is a `UniqueVariable()` constructor that is guaranteed to generate a unique variable name. Not used at the moment but a critical function needed for some AST transformation.
- `if` for representing conditional statements to execute `(if (eq 1
    1. (assign (variable a) (integer 10)))`which is equivalent to`if (1 == 1): a = 10`. Currently else not implemented but may be valuable.
- `statements` is a way of representing a list of statements. `(statement (assign (variable a) (integer 10)) (assign (variable a) (add (variable a) (integer 20))))` which is equivalent to `a = 10; a = a + 20`.
- `index` for indexing into a vector done via `(index (variable a) (integer 20) (integer 30))` which is equivalent to `a[20:30]`.

1.3 Python DSL

Suppose we have the simple structure. In python

```
class StructB(Struct):
    """All about StructB description"""

    name = "StructB"
    fields = [
        SignedInteger(name="FieldB", size=8, help="info about FieldB")
    ]

    conditions = [
        FieldReference('FieldB') == 0x10
    ]

class StructC(Struct):
    """All about StructB description"""

    name = "StructC"
    fields = [
        SignedInteger(name="FieldC", size=8, help="info about FieldC")
    ]

class StructA(Struct):
    """All about StructA description"""

    name = "StructA"
    fields = [
        Bits(name="FieldA", size=4, help="info about FieldA"),
        Union(StructB, StructC),
    ]
```

Get converted into the following lisp like structure. The structure can be thought of an the bitnest internal representation of a nested binary structure. Mentioned previously this list structure is wrapped in a `bitnest.core.Expression` object which allows for interacting with this structure in a more pythonic way.

```
StructA.expression()
```

```
(struct, 'StructA',
  (list,
    (field, 'bits', 'FieldA', None, (integer, 4), None, {'help': 'info about FieldA'}),
    (union,
      (struct, 'StructB',
        (list,
          (field, 'signed_integer', 'FieldB', None, (integer, 8), None, {'help': 'info␣
↪about FieldB'})),
        (list,
          (eq, (field_reference, 'FieldB', None), (integer, 16))),
        {'help': 'All about StructB description'}),
      (struct, 'StructC',
        (list,
          (field, 'signed_integer', 'FieldC', None, (integer, 8), None, {'help': 'info␣
↪about FieldC'})),
```

(continues on next page)

(continued from previous page)

```
        (list,),
        {'help': 'All about StructB description'}))),
  (list,),
  {'help': 'All about StructA description'})
```

1.4 Passes

Once the structure has been formed there are many transform, analysis, and backend passes that can be done.

1.4.1 Transform Pass

Realize DataTypes

This transformation plays an important role and is usually the first transform that takes place. Its job is to calculate all the complete datatypes that can exist from a nested set of structs along with uniquely identifying each field in the structure. This pass is a post order traversal of the AST. Each field encountered is numbered and since the traversal is deterministic the field id is deterministic as well.

The first point of calculating all the paths that result in unique datatypes deserves more discussion. Take for example the representative Struct below. Each path through the structure represents the construction of a datatype.

```
class StructB(Struct):
    """All about StructB description"""

    name = "StructB"
    fields = [
        UnsignedInteger(name="FieldB", size=8, help="info about FieldB")
        Vector(StructD, length=FieldReference('FieldB'))
    ]

    conditions = [
        FieldReference('FieldB') == 0x10
    ]

class StructD(Struct):
    """All about StructD description"""

    name = "StructD"
    fields = [
        SignedInteger(name="FieldD", size=8, help="info about FieldD")
    ]

class StructC(Struct):
    """All about StructC description"""

    name = "StructC"
    fields = [
        SignedInteger(name="FieldC", size=8, help="info about FieldC")
    ]
```

(continues on next page)

(continued from previous page)

```
class StructA(Struct):
    """All about StructA description"""

    name = "StructA"
    fields = [
        Bits(name="FieldA", size=4, help="info about FieldA"),
        Union(StructB, StructC),
    ]
```

Here is the general algorithm. When you encounter:

- field node `(field ...)` this is a leaf node and results in one path.
- union node `(union [<s1>] ... [<s2>])` a union node results in the addition of each path in the union. Suppose `(union [5] [3] [2])` where each element of the union has 5, 3, 2 paths then it will result in 9 total paths. We simply merge each path.
- vector node `(vector [<s1>] length)` preserves the number of paths in `[<s1>]` thus if there were 4 paths then the application of vector would result in 4 paths.
- struct node is the last an most tricky one to handle. Suppose we have `(struct [<f1>] ... [<fn>])`. The number of paths is the cross product of each field's paths. Thus if there are `(struct [4] [5] [2] [1])` paths for each field the number of resulting paths would be `4 * 5 * 2 * 1 = 40` paths. In the case of python we use `itertools.product` to calculate all combinations.

Lets look at the concrete example above. A high level representation of this structure (not the actual lisp representation).

```
(struct StructA
  (field FieldA)
  (union
    (struct StructB
       (field FieldB)
       (vector
         (struct StructD
           (field FieldD))))
    (struct StructC
       (field FieldC))))
```

Lets walk through the tree is post order traversal (L->RN).

- `(field FieldA)` -> `[(field FieldA 0)]`
- `(field FieldB)` -> `[(field FieldB 1)]`
- `(field FieldD)` -> `[(field FieldD 2)]`
- `(struct StructD [(field FieldD 2)])` -> `[(struct StructD (field FieldD 2))]`
- `(vector [(struct StructD (field FieldD 2))])` -> `[(vector (struct StructD (field FieldD 2)))]`
- `(struct StructB [(field FieldB 1)] [(vector (struct StructD (field FieldD 2)))])` -> `[(struct StructB (field FieldB 1) (vector (struct StructD (field FieldD 2))))]`
- `(field FieldC)` -> `[(field FieldC 3)]`
- `(struct StructC [(field FieldC 3)])` -> `[(struct StructC (field FieldC 3))]`

- `(union [(struct StructB (field FieldB 1) (vector (struct StructD (field FieldD 2))))] [(struct StructC (field FieldC 3))])  ->  [(struct StructB (field FieldB 1) (vector (struct StructD (field FieldD 2)))), (struct StructC (field FieldC 3))]`
- `(struct StructA [(field FieldA 0)] [(struct StructB (field FieldB 1) (vector (struct StructD (field FieldD 2)))), (struct StructC (field FieldC 3))]) -> [(struct StructA (field FieldA 0) (struct StructB (field FieldB 1) (vector (struct StructD (field FieldD 2)))), (struct StructA (field FieldA 0) (struct StructC (field FieldC 3))]`

This shows the whole process and we see that it results in 2 datatypes:

- `(struct StructA (field FieldA 0) (struct StructB (field FieldB 1) (vector (struct StructD (field FieldD 2))))`
- `(struct StructA (field FieldA 0) (struct StructC (field FieldC 3))`

These structures are most importantly deterministic and we can calculate all the positions for fields and easily write a parser for these datatypes. The only slight difficulty is in handling vectors since this makes the location of datafields depend on the length etc.

This transformation then wraps these paths in `datatype` nodes. It is important to preserve the lisp like datastructure to make future transformation compose nicely. Importantly we can see from this that the growth of the number of datatypes is roughly proportional to the product of the lengths/cardinality of all unions (ends up being slightly less depending on nesting).

```
(list
  (datatype (struct StructA (field FieldA 0) (struct StructB (field FieldB 1) (vector␣
↪(struct StructD (field FieldD 2))))))
  (datatype (struct StructA (field FieldA 0) (struct StructC (field FieldC 3)))))
```

Realize Conditions

Once we have all the resulting datatypes we then need to link conditions with their corresponding fields. As mentioned in the previous translation each field has been assigned a field id (not shown in the example above).

Take for example the condition `FieldReference('FieldB') == 0x10` within the `StructB` struct. Within bitnest this has an equivalent form.

```
(eq (field_reference "FieldB") (integer 0x10))
```

Once this traversal is complete the field id is updated to reflect the actual id within the tree. This is a little trickier than just aimlessly searching for the field within the tree since there can be multiple occurences of a field within a tree. Thus we have to traverse the tree to find the field from the place that the condition was added.

```
(eq (field_reference "FieldB" 1) (integer 0x10))
```

Realize Offsets

Take the example above. While not included in this high level description we have the size (number of bits).

- FieldA (4 bits)
- FieldB (8 bits)
- FieldC (8 bits)
- FieldD (8 bits)

```
(list
  (datatype (struct StructA (field FieldA 0) (struct StructB (field FieldB 1) (vector␣
↪(struct StructD (field FieldD 2))))))
  (datatype (struct StructA (field FieldA 0) (struct StructC (field FieldC 3)))))
```

When we look at both of the datatypes we get the following structures:

- `[FieldA FieldB Vector(FieldD)]`
- `[FieldA FieldC]`

Also note that the Vector has length equal to the value of FieldB. From this we have all we need to calculate offsets from the beginning of the structure along with the total length. Lets start with the second one since the calculation is easier.

`[FieldA FieldC]`:

- FieldA (size 4) (offset 0)
- fieldC (size 8) (offset 0 + 4)

With the total size being 12 bits for the second datatype. Now lets look at the first datatype.

`[FieldA FieldB Vector(FieldD)]`

- FieldA (size 4) (offset 0)
- FieldB (size 8) (offset 0 + 4)

Now the next part is trickier we know that the total length of the vector is equal to FieldB. We need to calculate the size of the contents of `Vector` to know the total size of the datatype. In this case the inner structure is only `FieldD` but this can be more complex and we need to recursively calculate the size. This means we can only know the size of the message at runtime. This is okay we have a symbolic formula to calculate the size. The size of FieldD is 8 bits. Thus the formula for offset for FieldD (assume we assign `i` to the vector index.

- FieldD (size 8) (offset 0 + 4 + 8 + (i * 8))

And the total size of the message is `0 + 4 + 8 + (FieldReference(FieldC) * 8)`. Since bitnest was designed to be symbolic this is a perfectly fine representation of the total length.

Arithmetic Simplify

This transformation is critical to for having easy to read expressions. Currently only addition operations are simplified. As additional simplification is needed this routine will be improved. Shows how expressions are simplified.

```
(+ 1 (+ a 3)) -> (+ 1 a 3) -> (+ a 4)
(+ 1 (+ 2 3)) -> (+ 1 2 3) -> (+ 6) -> 6
```

Parser Datatype

The parser datatype takes all of the information from the previous transformations to generate a program that can parse the datatype. All of the previous transformations were critical in getting enough information to write the program.

```
(list
  (datatype (struct StructA (field FieldA 0 4 0) (struct StructB (field FieldB 1 8 4)␣
↪(vector (struct StructD (field FieldD 2 8 (12 + i*8)))))))  # total size 12 +␣
↪FieldReference(FieldC) * 8
  (datatype (struct StructA (field FieldA 0 4 0) (struct StructC (field FieldC 3 8␣
↪4))))) # total size 12.
```

(continues on next page)

(continued from previous page)

Additionally we have a condition for StructB.

- `(eq (field_reference "FieldB" 1) (integer 0x10))`

In pseudo code we want to create a program that says the following:

```
if (FieldB == 0x10) and (length of message is 12 + FieldReference(FieldC) * 8):
   # it is datatype 1!
if (lenth of message is 12):
   # it is datatype 2!
```

This is exactly what this transformation generates. This is the power of having composible transformations! Note that this is a high level (not totally accurate portrayal of the programming AST). Notice that this AST that is emitted is language independent and can easily create efficient c and python code.

```
(statements
  (if (and (eq (field FieldB) 0x10) (len message (add 12 (mul (field_reference FieldC)␣
↪8))))
     (assign datatype 1)
  (if (len message 12)
     (assign datatype 2))))
```

1.4.2 Analysis Pass

The analysis passes do not return an `Expression` object. They do however ruturn python datastructures about the AST after running the analysis.

Inspect Datatype

Take the following example above once we have all the offsets etc. This analysis stage can be run earlier in the process it just requires `datatype` nodes to exist.

```
(list
  (datatype (struct StructA (field FieldA 0 4 0) (struct StructB (field FieldB 1 8 4)␣
↪(vector (struct StructD (field FieldD 2 8 (12 + i*8)))))))  # total size 12 +␣
↪FieldReference(FieldC) * 8
  (datatype (struct StructA (field FieldA 0 4 0) (struct StructC (field FieldC 3 8␣
↪4))))) # total size 12.
```

Several outputs come from the analysis for each datatype:

- fields
- conditions
- regions

`fields` is the list of fields that result from the datatype:

- `[FieldA FieldB Vector(FieldD)]`
- `[FieldA FieldC]`

`conditions` is the list of conditions for each datatype:

- `(eq (field_reference "FieldB" 1) (integer 0x10))`
- N/A

`regions` is so that we can understand the regions that a given struct/vector take.

- StructA (0-4 bits), StructB (4-12 bits), Vector (12 - 12 + FieldReference(FieldC) * 8), StructD (12 - 12 + FieldReference(FieldC) * 8)
- StructA (0-4 bits), StructC (4-12 bits)

1.4.3 Backend

Currently there is only a python backend and soon a c backend. But future ones should be easy enough to add.

Python

Take the following expression to see how the python backend works.

```
(eq (add (integer 1) (variable "a")) (integer 3))
```

We visit the tree in post order.

- `(integer 1)` -> `ast.Constant(1)`
- `(variable "a")` -> `ast.Name("a")`
- `(add ast.Constant(1) ast.Name("a"))` -> `ast.BinOp(ast.Add() ast.Constant(1) ast.Name("a"))`
- `(integer 3)` -> `ast.Constant(3)`
- `(eq ast.BinOp(ast.Add() ast.Constant(1) ast.Name("a")) ast.Constant(3))` -> `ast.Compare(ast.BinOp(ast.Add() ast.Constant(1) ast.Name("a")), ops=[ast.Eq()], comparators=[ast.Constant(3)])`

Now that we have the ast representation in python `ast.Compare(ast.BinOp(ast.Add() ast.Constant(1) ast.Name("a")), ops=[ast.Eq()], comparators=[ast.Constant(3)])` we can `astor.tosource(...)` on the ast and get the source code.

```
((1 + a) == 3)
```

CHAPTER

TWO

DESIGN DOCUMENT

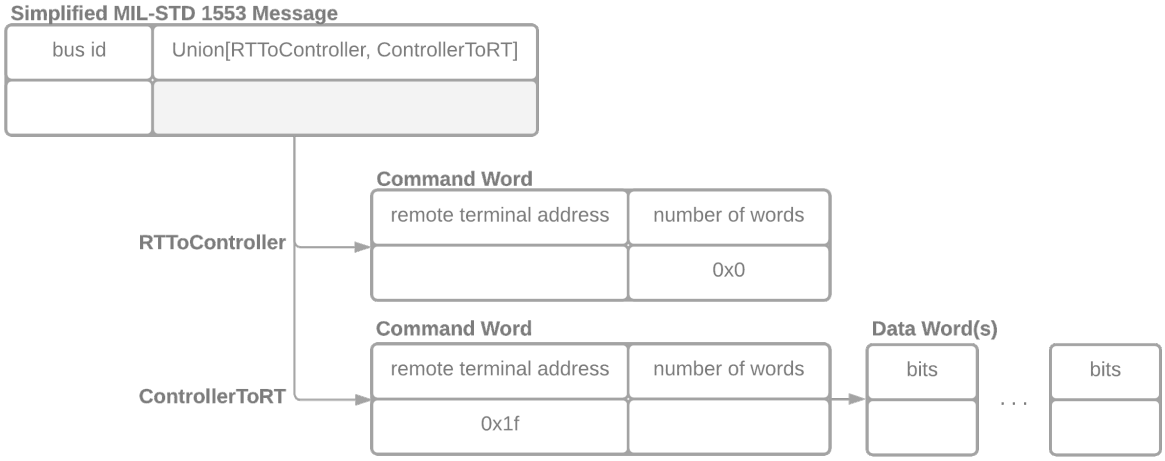
2.1 Conditional Nested Data Structures

This is the design document for as I am calling it not "bitnest" for "nested bits". The aim to explain the motivation for this package and how it is a needed piece in efficiently parsing binary packets of data. We hope to address why this is being built as well and how this is going to be built.

2.2 Background

2.2.1 1553 Motivating Example

This work was motivated for client work that we were doing on parsing the chapter 10 specification. A detailed document on the specification can be found on the irig106 site however still leaves some ambiguity. Chapter 10 is a complex specification with many nested parts see the project atac/libirig106 for a detailed list of all the unique packet types. The key here is that these protocols are well defined. In the image bellow we show a fake but representative specification of a 1553 message (a type of chapter 10 packet) that also tries to detail some of the challenges.



simplified 1553 packet

You will see that a 1553 message is composed of a header that consists of a `bus id` that is an unsigned integer of 8 bits followed by either an `RTToController` or `ControllerToRT` frame. In order to determine which one it is we have to check the `remote terminal address` (5 bit unsigned integer) which both have. If the `remote terminal address` is `0x1f` or 31 it is a `ControllerToRT` message otherwise it is an `RTToController` message. Additionally

if it is an `RTToController` frame the `number of words` must be equal to zero. If it is a `ControllerToRT` message then the frame is followed by `N` 16 bit `DataWords` where `N` is equal to the `number of words`. To summarize there are two packets "datatypes" that are described by this specification.

Simplified MIL-STD 1553 Message[RTToController[CommandWord]]

bus id	remote terminal address	number of words
00000011	01101	000
3	13	0

Simplified MIL-STD 1553 Message[ControllerToRT[CommandWord, *DataWord]]

bus id	remote terminal address	number of words	bits		bits
00000001	11111	010	01010111	. . .	10101101
1	31	2	87		173

conditional

nested structures

The first datatype is `1553 Message[bus id, RTToController[CommandWord[remote terminal address, number of words]]]` that maps out to a record of `bus id`, `remote terminal address`, and `number of words`. Notice how this datatype has a well defined static length `8 bits + 5 bits + 3 bits` and has three conditions associated with it `remote_terminal_address != 31`, `number of words == 0`, and the length of the packet must be 16 bits.

The second datatype is a bit more complex `1553 Message[bus id, RTToController[CommandWord[remote terminal address, number of words], *DataWord[bits]]]`. Where `*` indicates zero or more of the given datatype (a vector) and `*` is equal to the `number of words` field. Similar to datatype 1 the length is well defined though symbol `8 bits + 5 bits + 3 bits + (16 bits) * number of words`. We also have two associated conditions: `remote terminal address` must equal 31 and the length of the packet must match the symbolic formula above.

2.2.2 Data Type Complications

While not shown in this example above additional complications show up in the fields seen within the chapter 10 data words section. First off we realize the fact that we are dealing with non byte-aligned fields of non standard lengths.

Non byte aligned makes it much more difficult to access the value of a given fields due to complex masking and shifting operations.

Non standard lengths pose a more complex problem however. Lets consider the case of a 6 bit two's compliment signed integer. Ignoring edianess. In this standard the left most bit designates the sign of the integer: 1 negative and 0 positive. This alone brings the problem of translating this into a standard size that can be operated on take for example `100111`. If we would like to operate on this as an 1 byte signed integer we need to move the left `1` left two places to `1XX00111`. But what to do with the `X`'s? Well in the two's compliment representation we will pad with `0` if it is positive and `1` if it is negative with our final result being `11100111`. This problem becomes even more complicated when you consider the IEE-753 floating point specification. How do you handle different size exponent and decimal bits.

At the end of the day once a given packet has been parsed these data types must be accessible via byte aligned fields

to allow for efficient computation. However this step should be deferred as long as possible since "realizing/creating" these values requires moving data. This step should not be required for matching a given packet with a specific data type.

2.2.3 Similar Problems and Approaches

Our use case is not unique. There are many binary protocols that are well defined similar to the chapter 10 specification. Many I would say are simpler protocols.

- Ethernet, IP, TCP/UDP, HTTP (of course with many more branches for each layer) is the dominant use case
- pyatv decodes a well defined binary protocol for audio and video
- etherium protocol brownie, raiden
- scancode scans binary files to determine the given file type and information about the file
- nmigen interacting with emulated hardware devices

Complex well defined binary protocols are everywhere where information has to travel over the wire where bandwidth is at a premium. There are toolkits designed for working with some of these problems primarily in the networking space. We would like to highlight some of them:

Scapy

Scapy is a python toolkit to crafting up and reading network packets. In addition these have a built in high level language for describing new frames. I'd argue this is a high level description of a frame within a packet.

```
class Disney(Packet):
    name = "DisneyPacket "

    fields_desc=[
        ShortField("mickey",5),
        XByteField("minnie",3) ,
        IntEnumField("donald", 1, { 1: "happy", 2: "cool" , 3: "angry" })
    ]
```

Lets say the `Disney` packet is at the application layer in the network stack. To me the missing piece here is given a full ethernet packet how do I efficiently parse it and return whether it is a Disney packet or HTTP packet? ScaPy doesn't seem to have an efficient backend for parsing these custom packet types and does not have the logic to inspect the packet to determine if it is an HTTP packet (e.g. the first 4 bytes are `HTTP`) or if it is a Disney packet. The hope is that our proposed solution could be a backend for `Scapy` to efficiently parse network packets. Additionally currently scapy translates this problem into a python `struct` within the stdlib. For example the following would translate to.

```
import struct

packet = ...
struct.unpack('h3BB', packet)
```

However here see that we are conflating identifying the packet type and parsing the packet type. Also here they are working with bytes and not bits.

P4 Programming Language

Taken from wikipedia "P4 is a programming language for controlling packet forwarding planes in networking devices, such as routers and switches". It is a domain specific language for describing network packets. This tool is specialized for the efficient parsing of network packets and based on a filtering specification certain packets are rejected and accepted. See here for a list of examples of the language. Below is a partial code example. Looking at the specification we can already see limitations on the data types that you can work with (e.g. no floating point support and non-signed two compliment integers). However it does support non-byte aligned bits. It would definitely be worth investigating how P4 works.

```
header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16>   etherType;
}

header ipv4_t {
    bit<4>    version;
    bit<4>    ihl;
    bit<8>    diffserv;
    bit<16>   totalLen;
    bit<16>   identification;
    bit<3>    flags;
    bit<13>   fragOffset;
    bit<8>    ttl;
    bit<8>    protocol;
    bit<16>   hdrChecksum;
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}

struct metadata {
    /* empty */
}

struct headers {
    ethernet_t   ethernet;
    ipv4_t       ipv4;
}

...
```

2.2.4 Traditional Approach

The P4 language and Scapy show a "high level" of representing binary protocols without necessarily specifying "how" to parse the packets. This however is not the typical approach. In the case of ethernet packets the dominant library is libpcap and related wrappers around this tool. These are hand written parsers for the given protocols they are given to parse. Take the 1553 problem stated above. Here I'd like to show pseudo code of a solution to demonstrate the typical approach.

```
class CommandWord:
    length = 8
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, packet):
        self.packet = packet

    def is_rttocontroller_frame(self):
        return self.packet[0:5] != 0x1f and self.packet[5:8] == 0x0

    def is_controllertort_frame(command_word)
        return command_word[0:5] == 0x1f

    @property
    def number_of_words(self):
        return int(command_word[5:8])


class DataWord:
    length = 16


def parse(packet):
    header_length = 8 + CommandWord.length

    # can't parse command words unless correct length
    if len(packet) < header_length:
        return "packet not known"

    command_word = CommandWord(packet[8:16])

    if command_word.is_rttocontroller_frame():
        if len(packet) == header_length:
            return "datatype 1"
    elif command_word.is_controllertort_frame():
        if len(packet) == (header_length + DataWord.length * command_word.length_of_
↪words):
            return "datatype 2"
```

There certainly may be ways to simplify this code and write it better but one thing remains. The code has been expressed in this way so that it is easier to reason about. But what happends when you add an additional field to say the header. You now need to update the parser on header lengths etc. and there may be other new complex checks that you need to make. I believe that this is hard work that a compute could also generate from a high level specification of the data structures. The "checks" can be encoded as conditions that much be matched.

So to highlight some issues:

- writing this code efficiently requires low level programming language knowledge
- adding new fields can be tedious and non-intuitive how they change the parser
- the parser is not self documenting. there is no way to self document the grammar that this parser handles
- we not have one more parser for a specific binary protocol
- testing requires example data from the protocol which for some applications where the packets are classified is difficult for collaboration with open source

2.3 Implementation

I would argue that is starting to look like a well defined grammar of sorts. In Chomsky's Hierarchy this feels almost like a regular expression. To me this is partially validated when I look at the P4 specification. In this issue and several other issues they describe P4 as a finite state machine and not turning complete. I'd even say this is a grammar that the code paths can be accounted for to create a highly optimized parser to parse the packets. We are proposing a high level declarative representation of the binary protocol. For example a representation of the 1553 Message shown above.

```
from bitnest.field import Struct, UnsignedInteger, Bits, Union, FieldRef, Vector


class CommandWord(Struct):
    fields = [
        UnsignedInteger("remote_terminal_address", 5),
        UnsignedInteger("number_of_words", 3),
    ]


class DataWord(Struct):
    fields = [
        Bits("data", 16),
    ]


class RTToController(Struct):
    name = "Remote Terminal to Controller"

    fields = [
        CommandWord,
        Vector(DataWord, length=FieldRef("CommandWord.number_of_words")),
    ]

    conditions = [
        (FieldRef("CommandWord.remote_terminal_address") == 0x1F)
    ]


class ControllerToRT(Struct):
    name = "Controller to Remote Terminal"

    fields = [
        CommandWord,
    ]

    conditions = [
        (FieldRef("CommandWord.number_of_words") == 0x0)
    ]


class MILSTD_1553_Message(Struct):
    """This is a mock specification for a MILSTD 1553 Message to be as
    simple as possible while still representative of the difficulty of
    handling specifications.
```

(continues on next page)

(continued from previous page)

```
    """

    name = "MIL-STD 1553 Mock Message"

    fields = [
        UnsignedInteger("bus_id", 8),
        Union([
            RTToController,
            ControllerToRT,
        ]),
    ]
```

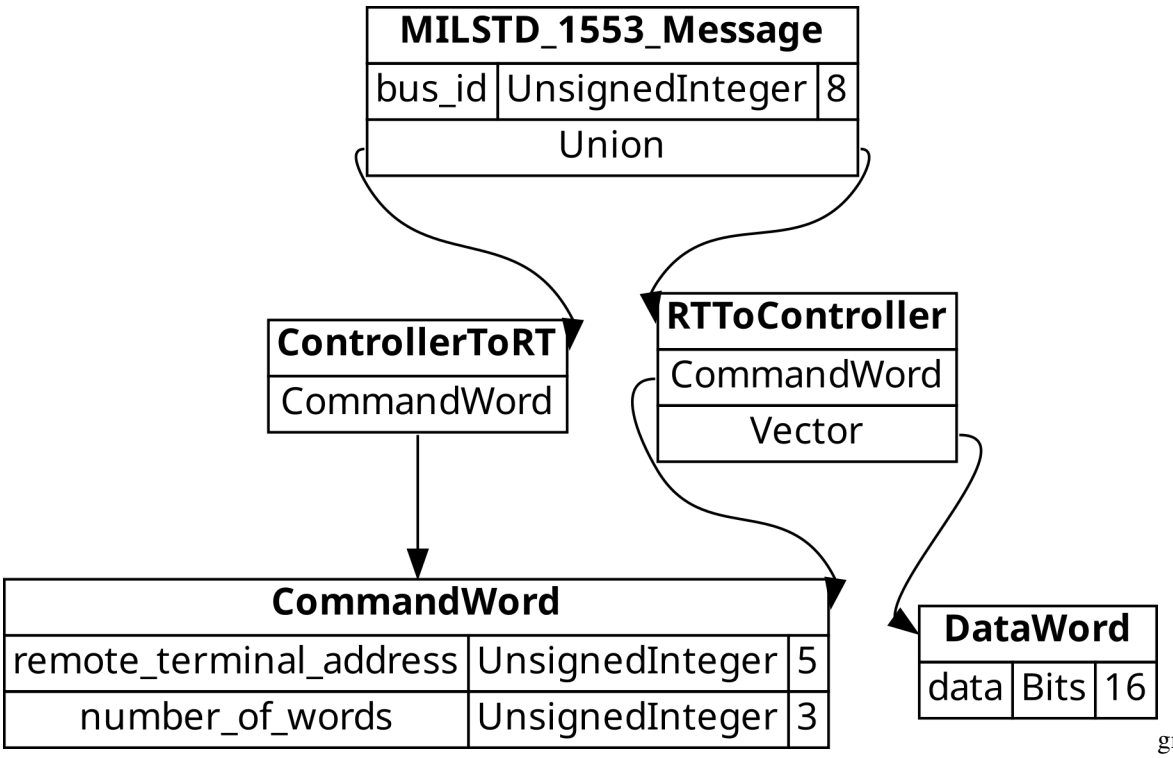
Here we describe a set of data structures which reference each other and data types within the `fields` attribute. Additionally there are a set of `conditions` that must be met in order for the given structure to be a valid path in determining the packet type. Said in another way if the conditions are not met for a given class e.g. `ControllerToRT` we know that the packet does not contain the path through the `ControllerToRT` and it is either an unknown packet datatype or a `ControllerToRT` packet. Using this high level description of the structures we can self document the protocol. This is also where the name for this work came from since it is a nested set of structures that satisfy given conditions. Bitnest already supports generating a graph visualization of the specification and markdown document of the specification shown below.

```
# MILSTD_1553_Message

This is a mock specification for a MILSTD 1553 Message to be as
simple as possible while still representative of the difficulty of
handling specifications.

## Structure

| name   | data type                                                                                  |␣
↪number of bits | description |
|--------|--------------------------------------------------------------------------------------------|-
↪---------------|-------------|
| bus_id | UnsignedInteger                                                                            |␣
↪8              |             |
|        | Union[[RTToController](#RTToController), [ControllerToRT](#ControllerToRT)] |␣
↪               |             |

# RTToController
## Structure

| name | data type                    | number of bits | description           |
|------|------------------------------|----------------|-----------------------|
|      | [CommandWord](#CommandWord) |                |                       |
|      | Vector                       |                | [DataWord](#DataWord) |

## Conditions

- 'CommandWord.remote_terminal_address' == 31

# CommandWord
```

(continues on next page)

(continued from previous page)

```
## Structure

| name                    | data type       | number of bits | description |
|-------------------------|-----------------|----------------|-------------|
| remote_terminal_address | UnsignedInteger | 5              |             |
| number_of_words         | UnsignedInteger | 3              |             |

# DataWord
## Structure

| name | data type | number of bits | description |
|------|-----------|----------------|-------------|
| data | Bits      | 16             |             |

# ControllerToRT
## Structure

| name | data type                    | number of bits | description |
|------|------------------------------|----------------|-------------|
|      | [CommandWord](#CommandWord) |                |             |

## Conditions

- 'CommandWord.number_of_words' == 0

# CommandWord

## Structure

| name                    | data type       | number of bits | description |
|-------------------------|-----------------|----------------|-------------|
| remote_terminal_address | UnsignedInteger | 5              |             |
| number_of_words         | UnsignedInteger | 3              |             |
```



graphviz output

While it is nice to be able to create documentation and visualize the specification ultimately it comes down to parsing performance. `bitnest` will act as a compiler of this high level domain specific python representation of nested conditional data structures and generate a parser.

In general the parser implementation should not matter. All that matters is that a performant program is produced. The easiest target that we see at this moment is numba which will JIT (just in time) produce an llvm compiled code to parse a given packet.

The pseudo code could go as follows. Since this is pseudo code of course the actual indexing and comparison of bits will be done via bit masks etc. but I'll leave that as an implementation detail. Additionally since we know the packets that we are parsing we can optimize these condition evaluations as to minimize the number of evaluations and provide the shortest path for the most common packets. Similar to PGO in gcc though in our case we would analyze the data to inform our compiler on the optimal code paths.

```
def parser(packet):
    if len(packet) == 16 and packet[8:11] != 0x1f and packet[11:16] == 0:
        return "datatype 1"
    elif len(packet) >= 16 and len(16 + packet[11:16] * 16) == len(packet) and␣
↪packet[8:11] == 0x1f:
        return "datatype 2"
    else:
        return "I have not clue what packet type this is!"
```

Would a human ever write this code? No! It is not maintainable and would be extremely prone to bugs. We are proposing a way to describe a given binary protocol at a high level and then implement a compiler to produce a parser of binary packets. Much like a regular expression is compiled into a parser for arbitrary strings of bytes.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search