swat engineering

# DAN – A DSL that targets Metal

The Netherlands Forensic Institute (NFI) is the national forensics institute of the Netherlands. Due to advance of digitalization in all layers of society, the role of the NFI has been quickly evolving in the past years. It has thus become increasingly necessary to be able to extract information from incomplete sources: either damaged physical devices, deliberately erased data, etc. In this context, file carving, that is, the systematic attempt to recognize data according to some format specifications, is fundamental.

The current file carving technology used in NFI is the "Metal"framework, that abstracts the complexities of parsing binary data, providing a Java API. To date, "Metal"has helped NFI developers to specify a number of data formats in a more user-friendly manner. There is, however, some opportunities for improvement. In particular, the way references are managed within the framework make dependencies difficult to reason about. Also, by using the API, developers create a layer of dynamically typed code on top of statically typed Java, which prevents the detection of type errors at compile time. To minimize these drawbacks, but at the same time reuse and learn from the effort put into "Metal", we propose DAN, a typed domain-specific language, sitting on top of “Metal” as a higher level layer of abstraction.

DAN (Domain-specific Language for Anonymization) is a domain-specific language for parsing binary data and was originally designed for anonymization of data. In DAN each token definition has its own type. Users can define new structured types that correspond to parsers, making the process of encoding new binary specifications less error-prone. To execute these specifications, DAN generates calls to the "Metal"API, reusing the knowledge encoded in that library. The present document is still a draft and explores the main features of the current design of DAN.

## 1 Execution model

A program consists of one or more modules. A module, thus, starts with a declaration of its name, followed by the modules it depends on:

```
module foo

import bar1
import bar2

...
```

The body of the module consists of a series of type definitions, that can be interpreted as parsers. A program starts being executed by parsing a stream of bytes using a token/type definition, such as:

```
struct Name{
    u8[] firstName[20]
    u8[] secondName[20]
}
```

`Name` defines a composed type and also specifies a parser. If we start the execution at `Name`, the result of that execution/parsing will be a data structure containing fields `firstName` and `secondName`. Consider another example of a type definition:

```
struct Info{
    Name name
    u8[] email[30]
}
```

`Info` defines another type, that depends on `Name`. If we start the execution at `Info`, the result of that execution will be a data structure containing fields `name` of the user-defined type `Name`, and field `email` of the primitive type `u8[]`.

# 2 Types

The main metaphor in DAN is that types correspond to parsers. However, for computation purposes, we also need plain types that are not associated to any parsing process. Therefore, we can distinguish non-token types such as `string`, `int`, `bool`, and token types such as `u8` (unsigned byte), `s8` (signed byte), `u16` (unsigned 16-bit word), `s128` (unsigned 128-bit word), etc. Both kinds of types can be aggregated in list types, specified as `T[]`, where `T` is a type, e.g.: `int[]`, `string[]`, `u8[]`. Notice that that the latter example is a list that is composed of token types, and therefore is also considered to be a token type.

Besides the aforementioned primitive types, user can define their own (token) types via `struct` or `choice` declarations.

In this section, we review in detail the available primitive types and how to define new user-defined types.

## 2.1 Primitive token types

Consider the following field definitions:

```
u8 oneUnsignedByte
s8 oneSignedByte
```

The first component of each definition is the type of the field, in these cases `u8` for an unsigned byte and `s8` for a signed byte. These definition can also be interpreted as parsers that will exactly parse 1 byte.

Now consider:

```
u16 twoUnignedBytes
u16 twoSignedBytes
```

They represent a 16-bit word (or a 2-byte word). The `u` and `s` types can be generalized to any multiple of 8, e.g. `u256` `s64`. An alternative way of defining sequential data is using lists:

```
u8[] twoBytes[2]
```

The first component of this definition is the type of the field, in this case, a list of unsigned bytes, followed by the name of the field and then, optionally, the size of the list. In this case, `twoBytes` will exactly parse 2 bytes. Alternatively, we can write:

```
u8[] aList
```

Field `aList` will contain a sequence of bytes of arbitrary length. (if this is parsed, it will consume all remaining bytes in the data stream)

Field definitions can also include conditions

```
u8 a ?(this != 65)
```

It defines field `a` of type `u8`. The value will only be assigned if at execution time, the parsed byte is not equal to 65, or, from another perspective, the parsed character is different from `A` (the character corresponding to 65). Notice that the special variable `this` in the condition refers to the field currently being defined.

*TODO: should we discuss the parsing semantics of a failed condition here?*

## 2.2 Struct types

We know than we can define structured data that enrich the set of available types with user-defined ones:

```
struct Simple{
    u8 first
    u16 second
}
```

This definition introduces the user-defined type `Simple`, that corresponds to a composed type that contains fields first and second. The double notion of type/parsing is more evident in the case of structs, in the sense that the fields' order matters. When used as a specification for parsing, `Simple` will first parse a byte that will be assigned to field `first`, and then two consecutive bytes that will be assigned to `second`.

## 2.3 Primitive non-token types

Not every type is parseable. Non-token types allow us to represent common primitive values that are not attached to a particular byte representation, such as strings, integers or booleans. As they do not define parsers, fields of a non-token type need to be initialized at the moment of declaration, for example:

```
struct Simple{
    int myOne = 1
}
```

A non-token type can be used to compute derived attributes inside an user-defined token type:

```
struct DerivedExample{
    u8[] token1[3]
    int offset1 = token1.offset
    u8[] token2
    int length2 =  token2.length
}
```

**Important:** all list types have an implicit `length` field, and all token types, an implicit `offset` field.

For readability purposes, it is sometimes preferable to separate the token fields from the derived ones, as the token ones "define"the parser. To do so, a dependency graph is calculated and therefore programmers can arrange the derived fields in any order. Coming back to the previous example, the following struct definitions are two valid alternative encodings:

```
struct DerivedExample2{
    u8[] token1[3]
    u8[] token2

    int offset1 = token1.offset
    int length2 = token2.length
}

struct DerivedExample3{
```

```
    int offset1 = token1.offset
    int length2 = token2.length

    u8[] token1[3]
    u8[] token2
}
```

## 2.4 Choice types

We can also define choice types:

```
struct A{
    u8 content ? (this == "A")
}

struct B{
    u8 content ? (this == "B")
}

choice AB{
    A
    B
}
```

Choice types imply backtracking. In this case, an attempt to parse the input with type `A` will be made first. If this parsing does not succeed, then `B` will be tried, in strict declaration order. In other words, the disambiguation is driven by the parsing process. Note that the fields in a choice correspond to alternatives, therefore, they are not named.

Notice too that despite both alternatives featuring a field `content` of type `u8`, this field is not accessible if we have a reference to a value of type `AB`. In order to do this, we need to use *abstract fields*. An abstract field is a field declared inside a choice, whose implementation is abstract and must be defined in *each one of the alternatives*. Let us redefine `AB` using this concept:

```
choice AB2{
    abstract u8 content
    A
    B
}
```

Since both alternatives have a field called content of type `u8`, this definition is valid. Thus, `AB2` exposes a field content that will hold the proper value, depending on which alternative is going to be taken.

## 2.5 Implicit coercions

In the previous example, notice the equality check performed in the `content` declarations . We are comparing `this` (which refers to the field `content` of type `u8` with string literals ("Aänd "B"), of type `string`. This comparison works because there is an implicit cohercion from the type `u8` to the type `string`. Primitive token types, such as `u` types, `s` types, or list types containing them, can be coerced to primitive non-token type, such as `string` or `int` by using some implicit encoding conventions. To alter these coventions, we can use "meta properties", explained below.

## 2.6 Constructors of user-defined types

We can modularize programs by having parametric user-defined types:

```
struct MustBeOfCertainAge(int ageLimit){
    u8[] name[20]
    u8 age ? (this == ageLimit)
}
```

In this case, `MustBeOfCertainAge` acts as a constructor receiving one integer argument.

In order to build parsers conforming to this definition we need to parameterize the construction of values of this type. Consider, for instance, this field declaration:

```
MustBeOfCertainAge person(18)
```

This field definition instantiate the `MustBeOfCertainAge` struct definition for the concrete argument `18`.

## 2.7 Anonymous fields

Sometimes, a field is needed just for parsing purposes but it does not really need to have a named assigned. In that case, we can use the field name `_`, which means to ignore that field when accessing the fields of a value of such type:

```
struct Ignoring{
       u8[] name[20]
       u8 _ ? (this == ' ') // there must be a space after the name, but we do not name it
       u16 age
       u8[] _[4] ? (this == '     ') // there must be four spaces after the age, but we do not name it
   }
```

## 2.8 Inline user-defined types

In the choice example, for defining types `AB` and `AB2` we also needed to define types `A` and `B`, exclusively in order to define the alternatives of the choice. In these situations, it might be more convenient to inline the struct definitions, as shown here:

```
choice AB3{
    abstract u8 content
    struct{
        u8 content ? (u8 == "A")
    }
    struct{
        u8 content ? (u8 == "B")
    }
}
```

The same applies for a struct definition, for example:

```
struct Info2{
    struct{
        u8[] firstName[20]
        u8[] secondName[20]
    } name
    u8[] email[30]
```

```
}
```

In order to access the `firstName` field of the inner struct, assuming that we have a variable `i` of type `Info2`, we need to write `anInfo2.name.firstName`.

# 3 Parsing

Sometimes it is necessary to write generic code that is independent of one specific format specification. In those cases, we want to have parameterized parsing (`tie` in metal). For this, we can use the `parse` expression, that parses a given list of bytes according to a token type specified as a type parameter. We have not discussed type parameters in this document as this is a feature we will add in a next iteration. Consider this example, illustrating the use of `parse`:

```
struct Digit{
    u8 a ?(this >=0 || this <=9)
}

struct TwoXs<X>{
    u8[] first
    u8[] second
    X firstX = parse<X>(first)
    X secondX =  parse<X>(second)
}

struct Simple{
    TwoXs<Digit> as
    X oneX = as.firstX
}
```

# 4 Functions

We provide a simple mechanism for reuse through functions. A function allows to abstract over common pattern of computations. Consider the standard squaring example:

```
int square(int x) = x * x
```

Having added this definition, we can now reuse this behavior when defining a type:

```
struct Foo{
    u8 bar
    int x2 = square(bar)
}
```

If more sophisticated behavior is required, we can declare "function bridges", that is, declare the signature of a function annotated with a Java implementation:

```
@(org.foo.VeryComplexEncoding)
int veryComplexEncoding(int x)

struct Foo2{
    u8 bar
    int x = veryComplexEncoding(bar)
```

```
}
```

# 5 Meta properties

Properties that have to do with how to parse the token types are represented as annotations at the declaration level:

```
struct Block@(encoding=LittleIndian){
    u32 content ?(this == 0xDEADBEEF )
    u8[] content[4] ?(this == 0xCAFEBEEF)
}
```

This means that Little Indian will be the encoding used for parsing content to produce a value of type `Block`.

Another meta-property is `offset`, that creates a new parsing pointer (disconnected from the context) and moves that parsing pointer to the specific (absolute) offset. If omitted, the default is to advance the normal pointer of the parent. Here an example where we start parsing from the 10th byte on:

```
struct Root {
    // parsing pointer is at position 0
    u8 oneByte
    // parsing pointer is at position 1
    u32 oneInt
    // parsing pointer is at position 5
    Block2 sim
    // parsing pointer is at position 5
    Block2 sim
    // parsing pointer is at position 5
    u8 oneByte
    // parsing pointer is at position 6
}

struct Block2@(offset=9){
    // parsing pointer is at position 9
    u8[] content[4]
    // parsing pointer is at position 13
}
```

# 6 Metal constructs mapped to DAN

## 6.1 Tokens

| Metal shorthand | Description | DAN |
|---|---|---|
| `def(name, size)` | name a field with a size in bytes | `u8[] name[size]` or `u32 name` |
| `nod(size)` | a ignored set of bytes | `u8[] _[size]` |
| `cho(name, list[Token])` | find the first token that parses successfully | `choice name{ ... }` |

| Metal shorthand | Description | DAN |
|---|---|---|
| `rep(name, Token)` | name a field that repeats a token until it fails to parse | `T[] name` |
| `repn(name, Token, size)` | name a field that repeats a token n times | `T[] name[n]` |
| `seq(name, list[Token])` | define a sequence of tokens | `struct name{ ... }` |
| `sub(name, startAt, token)` | parse a token at a specific offset | `struct@(offset=startAt) name { ... }` |
| `pre(name, Token, predicate)` | to be determined | |
| `post(name, Token, predicate)` | `token name ?(pedicate)` | |
| `whl(name, Token, predicate)` | to be determined | |
| `opt(name, Token)` | to be determined | |
| `token(name)` | get a reference to a token "type" instead of a value | `T.type` |
| `tie(name, Token, data)` | Run a token parser on the result of a data expression | `u8[] name = parse<Token>(data)` (`Token` is a type parameter representing the token type, while `data` corresponds to a list of bytes, having type `u8[]`) |
| `until(name, initialSize?, stepSize?, maxSize?, terminatorToken)` | unclear, the size params are all optional | |
| `when(name, Token, predicate)` | unsure | |

## 6.2 Expression

| Metal shorthand | Description | DAN |
|---|---|---|
| `add` | | `l + r` |
| `div` | | `l / r` |
| `mul` | | `l * r` |
| `sub` | | `l - r` |
| `mod` | | `l % r` |
| `neg` | | `-l` |
| `shl` | | `l << r` |
| `shr` | | `l >> r` |
| `and` | binary and | `l & r` |
| `or` | binary or | `l \| r` |
| `not` | binary (and boolean) not | `!l` |
| `and` | boolean and | `l && r` |

| Metal shorthand | Description | DAN |
|---|---|---|
| `or` | boolean or | `l \|\| r` |
| ‘eq* | comparision operators | default comparison operators |
| `con` | constants | literal syntax for arbitrary precision ints, strings, arrays |
| `len(l)` | the size of a token | `aToken.size` |
| `ref(name)` | create a list of all instances of something with that name | `x.name` (however, it’s not dynamic scoping anymore) |
| `first(l)` | first element of the list/array | `l[0]` |
| `last(l)` | last element of the list/array | `l[-1]` or `l[l.length - 1]` |
| `nth(l, i)` | nth element of the list/array | `l[i]` |
| `offset(reference name)` | get the absolute offset of something that has already been parsed | `n.offset` (without dynamic scoping aspect) |
| `cat(a, b)` | concat the bytes together of two parsed trees | `a + b` |
| `elvis(v, orElse)` | still need to design this | |
| `count(t)` | the size of a list | `aList.length` |
| `foldLeft(values, reducer, initial?)` | still need to be designed | |
| `foldRight(values, reducer, initial?)` | still need to be designed | |
| `fold(values, reducer, initial?)` | still need to be designed | `(T initial \| reducer \| x <- values)` (reducer is an expression that has in scope both the special variable `it` , representing the accumulator, and variable `x` representing the current element being processed; the result of this expression is of type `T`) |
| `mapLeft(values, mapper, left, rightExpand)` | still need to be designed | |
| `mapRight(values, mapper, leftExpand, right)` | still need to be designed | |
| `rev(values)` | reverse a list | still need to be designed, might be less needed since references aren’t lists anymore |
| `exp(base, count)` | repeat (expand) `base` `count` times | missing, maybe something with slices? |
| `bytes(x)` | get the bytes represented by something | automatic cohercion solves this |