# Flat (Draft)

Pasqualino 'Titto' Assini (tittoassini@gmail.com)

27th of May 2016

## Contents

What is Flat? . . . . . 1
Design Goals . . . . . 1
Design Non-Goals . . . . . 2
A Principled Data Model . . . . . 2
Word Alignment . . . . . 7
Normative Definitions . . . . . 8
About This Document . . . . . 9

### What is Flat?

Flat is a minimalist, principled and efficient binary data format suitable for data exchange and storage.

### Design Goals

The goal of a universal data format is to operate as a bridge across:

- different ways of modelling data (databases, object oriented, functional or imperative languages)
- processing systems (ranging from nanosystems to mainframes)
- time (long term storage and retrieval of data)

To be able to cross these barriers, it has to be based on the simplest possible abstractions and free from arbitrary and limiting decisions about predefined data structures or primitive types.

Briefly stated, it has to be discovered, not invented.

## Design Non-Goals

Some data formats embed semantic information (for example JSON states the names of the data type and of the fields being represented in every value, CBOR has type tags) or provide non essential features such as a versioning, redundancy, error-correction or compression.

The principled approach taken by Flat is that all these additional features can be provided much more cleanly, flexibly and effectively by additional conventions layered on top of the binary data format.

## A Principled Data Model

There are two concepts that necessary underlay any conceivable data modelling formalism:

- Choice: information, in its simplest form, is a choice among different alternatives; the basic unit of information, the bit, corresponds to the simplest possible choice, that between two different values.
- Sequencing or Aggregation: complex data types are built aggregating simpler data types.

From these two principles we can directly derive a simple but expressive data modelling framework.

Our starting point is the binary choice operator `|`.

Using it, a bit can be defined as (Figure 1):

```
Bit ≡ Zero | One
```

Meaning: ”the data type Bit is defined (≡) as either (|) a Zero or a One.”

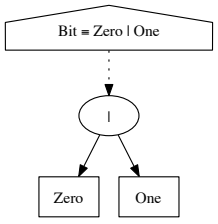


Figure 1: Bit

The binary choice operator | corresponds to a minimal binary tree.

Trees with more than one node can be used to represent choice across any number of values.

For the tree to be as compact as possible, we want it to be balanced.

The only arbitrary choice is if, in case of an odd number of choices, the tree should be left or right-size heavier.

If we opt for the second alternative [1], the traditional five Chinese directions can be represented as (Figure 2):

```
Direction ≡ (North | South) | (Center | (East | West))
```

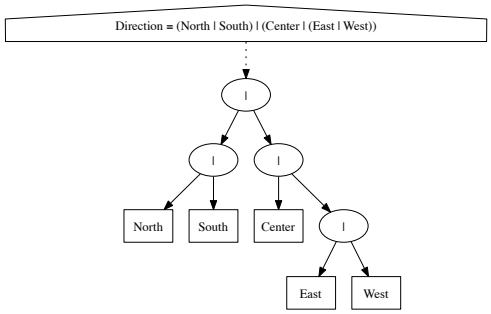


Figure 2: Direction

Note how the 5 values of `Direction` are split in groups of 2 and 3 (balanced and right-heavy) and the right group of 3 constructors is split in groups of 1 and 2 (again balanced and right-heavy).

Simple types that are usually introduced as primitives, such as the unsigned short integer, can be explicitly defined by stating their possible values:

```
Word8 = (..((Z | N1) | (N2 | N3)) .. | .. ((N252 | N253) | (N254 | N255))..)
```

So far values have consisted in unique labels (Zero, One, N255..). To create more complex data types, we allow the label (or constructor) to be followed by a (possibly empty) sequence of values.

As an example consider a data type that conveys the concept of an optional Bit:

```
-- An Optional Bit is either empty (None) or carries a value (Some Bit):
Optional Bit ≡ None | Some Bit
```

The constructor's fields can also be of the same type of the data type being defined, enabling the definition of recursive data types, such as a list of booleans (Figure 3):

```
-- A List Bool is either a 0-length list (Nil)
-- or a Cons value with two fields, the first pointing to a Bool
-- and the second recursively to the List Bool itself.
List Bool ≡ Nil | Cons Bool (List Bool)

-- A boolean is either False or True.
```

[1] This choice is arbitrary, but inconsequential, as the two options are completely equivalent.

```
Bool ≡ False | True`
```

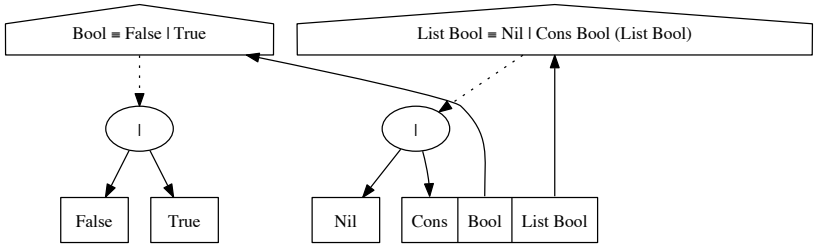


Figure 3: List of Booleans

Mutually recursive definitions are also possible (Figure 4):

```
-- A Forest is a list of Trees
Forest Bool ≡ Nil | Cons (Tree Bool) (Forest Bool)

-- A Tree is either empty or is a Bool followed by a Forest Bool
Tree Bool ≡ Empty | Node Bool (Forest Bool)
```

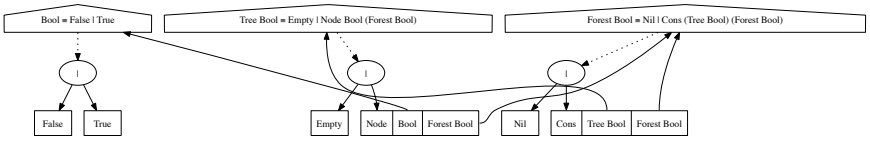


Figure 4: Forest and Tree of Booleans

As well as definitionally infinite data types (Figure 5):

```
Stream Bool ≡ Stream Bool (Stream Bool)
```

The building of compound values from simpler ones can be seen as the application of a binary sequencing or coupling operator `(,)`.

For example, the value `Cons Bool (List Bool)` is equivalent to the labelled binary tuple `Cons (Bool,List Bool)`.

Values with any number of fields can be built by nesting up binary tuples as in `(A,(B,(C,D)))`.

In conclusion, using two simple operators, a binary choice operator `|` and an (implicit) sequence operator `(,)` we have built a powerful data modelling formalism that can be used as a rosetta stone across any data processing system.

This formalism is:

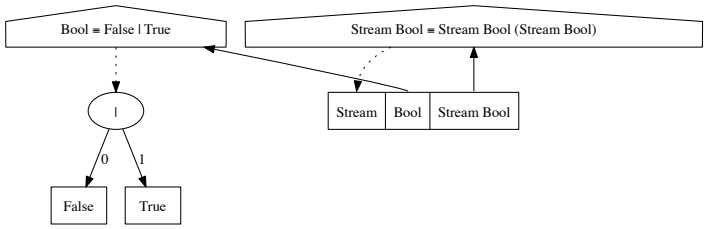


Figure 5: Stream of Booleans

- Principled: does not rely on an arbitrary selection of primitive types or data structures selected according to a designer's whim.
- Explicit: the structure of every data type, from a simple bit to the most complex data model, can be traced down to a combination of choice and sequence operators.
- Expressive: it captures the essential principles of data modelling and can represent any kind of data structure (enumerations, finite types, recursive and mutually recursive types, infinite types)

In a more developed form, this formalism is known as the algebraic data type. Because of its simplicity and flexibility, it has been adapted natively by an increasing number of programming languages.

Flat does not requires the full expressive power of algebraic types. An algebraic data type definition includes three aspects:

- syntactical: the 'shape' of a data type, its basic structure as a composition of | and (,) operators and references to other data types
- semantic: the 'meaning' of the data type, suggested by the names of the data type and its constructors
- type-theoretic: in a programming language, data types are defined in the context of a type system that can impose constrains on the set of acceptable values and use parametricity and other mechanisms to make definitions generic and composable.

Flat only captures the syntax of a data type, anything else is the job of higher level layers.

It is straightforward to derive the canonical syntactical structure of a data type:

- the data type name is ignored
- The left and right branches of the choice | operator are respectively marked with 0 and 1.
- the constructors' names are substituted by the nested sequence of bits obtained by joining left-to-right the markers on the path from the data type to the constructor, followed by the sequences corresponding to the values of the constructors' fields.

Figure 6 shows the canonical syntactical structures of the simple data types discussed so far.

The Unit type on the left, is the single valued type:

```
Unit ≡ Unit
```

The only value of Unit does not carry any additional information with respect to its type, there is no choice, its canonical name is therefore the empty sequence `[]`.

`Bool` and `Bit`, as any other two valued type, have their constructors assigned the canonical names of `[0]` and `[1]` and similarly for any other simple type.

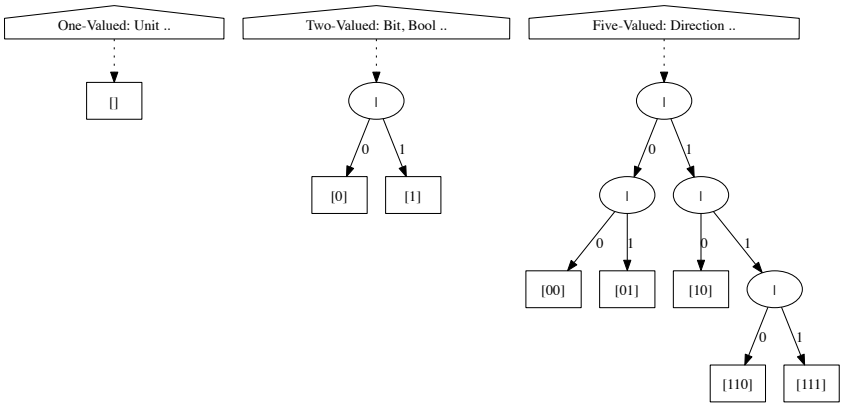


Figure 6: Simple Canonical Types

Compound data types work in the same way, with fields' sequences nested into the constructors' ones (Figure 7).

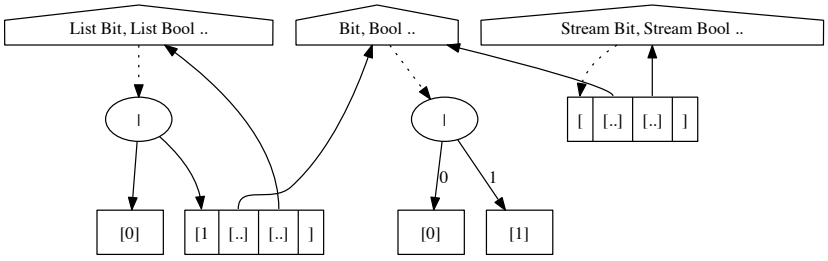


Figure 7: Compound Canonical Types

For example, the value `Cons False (Cons True Nil)` maps to `[1,[0],[1,[1],0]]`.

Note how the `Stream` constructor, being single, maps to an empty sequence so that a Stream of True values maps to `[[1],[[1],[[1]..]]]`.

The final binary encoding is obtained by flattening the canonical names (Tables 1-3).

Table 1: Unit values and codes

| Value | Code |
|---|---|
| Unit | <> |

Table 2: Direction values and codes

| Value | Code |
|---|---|
| North | <00> |
| South | <01> |
| Center | <10> |
| East | <110> |
| West | <111> |

Table 3: List values and codes

| Value | Code |
|---|---|
| Nil | <0> |
| Cons True Nil | <110> |
| Cons False (Cons True Nil) | <10110> |

The Flat encoding is:

- Optimal: takes as little space as possible, assuming that all encoded sequences are equi-probable.
- Complete: there are no erroneous codes, if the decoder asks for one more bit it can always interpret it.

Conveniently, in the common case of enumerations with a number of values equal to a power of two (e.g. `Word8`), the encoding is equivalent to the usual unsigned encoding.

## Word Alignment

Flat is a bit-oriented binary format.

This is a direct consequence of its principled stance, and has two advantages:

- removes any arbitrary dependency on architecture-specific word size

- provides the finest possible control on the binary encoding

When data is stored or transferred, however, we need to take account of the fact that information systems are word oriented.

It is therefore necessary, for functional and performance reasons, to byte or word-align data.

Following Flat general approach, this can be done simply and explicitly by embedding the value into an appropriate wrapper.

To store or transfer a value to another system, it should be post-aligned so that it fills into the required word size:

```
-- A PostAligned value ("a" is a parameter standing for any possible type)
PostAligned a ≡ PreAligned a Filler

-- A meaningless sequence of 0 bits terminated with a 1 bit.
Filler ≡ FillerBit Filler | FillerEnd
```

An intelligent encoder, aware of the word size required by the destination system, can modify the size of the Filler to align the data.

For example, the boolean value `True` embedded into a `PostAligned Bool` when sent to a byte-oriented device would be encoded as `<10000001>` where `<1>` is the encoding of 'True' and `<0000001>` is the Filler.

Similarly, to optimise the transfer of values that encode large binary sequences, it is useful to pre-align them using `PreAligned`:

```
-- A PreAligned value ("a" is a parameter standing for any possible type)
PreAligned a ≡ PreAligned Filler a
```

A smart decoder will take advantage of the fact that data is aligned, using fast byte-aligned operations to read in the data.

Naive decoders will still work correctly as `PreAligned`,`PostAligned` and `Filler` are just normal data types subject to the usual Flat encoding.

## Normative Definitions

A binary data format is an isomorphism between data values and binary sequences.

Every value has a data type.

A data type consists in a balanced, right-heavy, binary tree whose leaves are values and whose internal nodes have a left and right arrow, respectively marked with a `0` and a `1` bit.

A value is a (possibly empty) sequence of fields each of which points to a data type.

The binary sequence corresponding to a value is obtained by joining left-to-right the markers on the path from the root of the data type to the value, followed by the sequences corresponding to the values in the values' fields.

## About This Document

The only normative part of this spec is Normative Definitions, the rest is narrative.

Table 4: Document Metadata.

| | |
|---|---|
| Status | Draft |
| Reference URL | http://quid2.org/docs/Flat.pdf |
| License | GPLv3 (http://www.gnu.org/licenses/gpl-3.0.en.html) |
| First Published | 2016-05-27 |
| Last Revision | 2016-05-27 |
| Copyright © | Pasqualino ”Titto” Assini (tittoassini@gmail.com) |