正名 Zhèng Míng (Draft)

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

1st of May 2017

Contents

What is 正名 (Zhèng Míng)?

正名 (read as: Zhèng Míng) is a minimalistic, expressive and language independent data modelling language.

It provides a way to:

- indicate the semantic of a data type
- precisely define the syntactical structure of a data type
- precisely define the binary serialisation of the values of a data type
- deterministically calculate a compact and globally unique data type identifier

It can be used:

- for data exchange across different programming languages and software systems
- for long term data preservation

正名 literally means 'true or official name'. It also refers to the Confucian doctrine of the 'rectification of names'. Confucius thought that "*If names be not correct, language is not in accordance with the truth of things. If language be not in accordance with the truth of things, affairs cannot be carried on to success.*".

If there isn't a straightforward relationship between names and what they refer to, confusion arises and understanding and coordination becomes impossible. Getting the names right is just as important in 21st century information systems as it was in 6th century BC Chinese philosophy.

正名 provides a simple way to define concepts (as data types) and assign them "true names", that's to say globally unique and unambiguous identifiers.

Data Modelling

正名 defines a model of algebraic datatypes.

An algebraic type combines two key mechanisms to define data types:

- sum: choice among multiple variants
- product: sequencing of multiple values

An example of a sum type is:

```
Boolean ≡ False
        | True
```

This definition introduces a new data type `Boolean` with two possible values: `False` and `True` ('|' separates the different choices).

`False` and `True` are called constructors as they construct values.

The constructor names must obviously be unique for every data type.

A datatype without constructors and therefore no values is also admissable:

```
Void
```

An example of a product type is:

```
Tuple ≡ Tuple Bool Bool
```

This definition introduces a new data type `Tuple` whose values are all the possible combinations (the cartesian product) of two booleans, introduced by the constructor `Tuple`: `Tuple False False, Tuple False True, Tuple True False, Tuple True True`.

Note that the datatype name can be the same as the name of one of its constructors.

The fields of a constructor can, optionally, be assigned names:

```
FullName ≡ FullName {firstName::String,familyName::String}
```

This definition introduces a new data type `FullName` with a single constructor `FullName` with two fields of type `String`, called `firstName` and `familyName`. A possible value of this type would be `FullName {firstName="Leonardo",fullName="Da Vinci"}` or equivalently `FullName "Leonardo" "Da Vinci"`.

Datatypes can be recursive, as in this representation of the natural numbers:

```
Natural ≡ Zero | Succ Natural
```

The (infinite) valid values for this type are: `Zero, Succ Zero, Succ (Succ Zero), Succ (Succ (Succ Zero))` and so on.

Mutually recursive data types however, are not allowed, these definitions are invalid:

```
Forest ≡ Nil
       | Cons Tree Forest

Tree ≡ Empty
     | Node String Forest
```

Data types can take other types as parameters:

```
Maybe a ≡ Nothing
        | Just a
```

Parametric data types are abstract, they have no values.

Concrete data types are build by substituting the parameters with concrete types: `Maybe Bool`, `Maybe Natural` and so on.

The valid values of `Maybe Bool` are: `Nothing, Just False, Just True`.

A data type can have multiple parameters (up to a maximum of 255):

```
Either a b  ≡ Left a
            | Right b
```

The data type parameters do not necessarily appear in the right side of a data type definition.

For example in the following pointer type, we use the parameter to specify the type of the value we are pointing to: `Ptr Bool`, `Ptr String`.

```
Ptr a  ≡ Ptr Address
```

Data types and constructor names can be either literal (a unicode letter followed by a sequence of letters or numbers or the special character _) or symbolic (a sequence of unicode symbols).

An example of a parametric and recursive nested datatype (a list type) with literal names:

```
List a ≡ Nil               -- An empty list.
       | Cons a (List a)   -- A list: a value followed by another list.
```

An equivalent list datatype, using symbolic names:

```
[] a  ≡ []           -- An empty list.
      | : a ([] a)   -- A list: a value followed by another list.
```

Normative Definitions

正名 specifies:

- a set of valid algebraic data types definitions (ADTs)

- a set of valid types
- unique identifiers for every valid ADT
- the binary serialisation of the values of every valid type

Basic Concepts

Algebraic Data Type (ADT) A data type constructor with zero or more type parameters plus zero or more constructors.

Type The application of a data type constructor to the number of type parameters specified in its data type definition.

Value The application of a constructor to the number of parameters specified in its data type definition.

For example, the ADT:

```
Maybe a ≡ Nothing
        | Just a
```

defines:

- a data type constructor `Maybe` that takes one type parameter
- two constructors `Nothing` and `Just` that take zero and one parameter respectively

Given these additional definitions:

```
Bool ≡ False
     | True
```

```
Void
```

A type is constructed by applying `Maybe` to exactly one type parameter: `Maybe Bool`, `Maybe (Maybe Bool)` and so on.

A value is constructed by applying `Nothing` and `Just` to their specified number of parameters.

The values of type `Maybe Bool` are: `Nothing, Just False, Just True`.

The only value of type `Maybe Void` is `Nothing`.

Formal Definitions

正名 formally specifies its concepts in three ways:

- syntactically, by the syntactical definitions that compose the canonical model
- semantically, by the names of the ADTs that indicate (hopefully in a self-explaining way) how their values should be interpreted, in particular:

Table 1: Semantic Constrains.

Type	Meaning
Char	A Unicode character
LeastSignificantFirst a	The parts that compose value `a` are ordered with the least significant (arithmetically) part first
MostSignificantFirst a	The parts that compose value `a` are ordered with the most significant (arithmetically) part first
SHAKE128_48	The first 48 bits of a SHAKE128 hash
UnicodeLetter	A Unicode character whose General Category is `Letter`
UnicodeLetterOrNumberOrLine	A Unicode character that is either the _ (underscore) character or whose General Category is `Letter` or `Number`
UnicodeNumber	A Unicode character whose General Category is `Number`
UnicodeSymbol	A Unicode character whose General Category is `Symbol`

- by convention, by stating the following additional constrains:
  - the constructor names in an algebraic data type must be all distinct
  - types passed as parameters to a data type constructor must be valid, that's to say applied to the correct number of valid type parameters

Algebraic Data Types

ADTs are represented by values of type: `ADT Identifier Identifier (ADTRef AbsRef)`
.

Valid ADTs must satisfy all the syntactical, semantic and conventional constrains previously stated.

正名 has no primitive types, all types are defined explicitly.

Types

Types are represented by values of type: Type AbsRef.

For example, the type `Maybe Char` is represented as:

TypeApp

(TypeCon (AbsRef (SHAKE128_48 218 104 54 119 143 212)))

(TypeCon (AbsRef (SHAKE128_48 6 109 181 42 241 69)))

Unique Identifiers

正名 defines (practically) unique identifier for ADTs and consequently for types.

Valid ADT's identifiers are values of type: AbsRef.

The identifier consists in the SHAKE128(M, 48) (6 bytes long) hash of the shortest byte-aligned Flat encoding of the value of type `PostAligned (ADT Identifier Identifier (ADTRef AbsRef))` where the value of type `ADT Identifier Identifier (ADTRef AbsRef)` corresponds to the ADT.

For display purposes, the identifier can be compactly represented as the letter 'K' followed by the hexadecimal representation of the hash.

For example, the identifier for the ADT:

```
Maybe a ≡ Nothing
        | Just a
```

is AbsRef (SHAKE128_48 218 104 54 119 143 212).

This can also be compactly represented as `Kda6836778fd4` .

Serialisation of Values

Value are serialised according to the mapping of ADTs to Flat data graphs specified in the Flat specification.

For example, given the definitions:

```
List a ≡ Nil
       | Cons a (List a)

Bool ≡ False | True
```

The type `List Bool` is mapped to the Flat data flow graph:

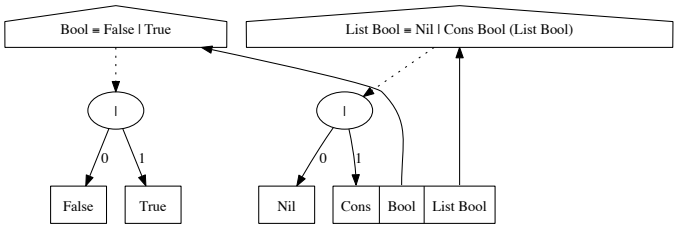

Figure 1: List of Booleans

So that, for example, the value `Cons True (Cons False Nil)` would be serialised as the binary sequence `11100`.

Canonical Data Model

The canonical model is also a meta-model, the model can be expressed in the model itself.

The complete definition of the canonical model is (every ADT is preceded by its unique identifier in compact format):

```
K3e8257255cbf:
 ADT a b c ≡ ADT {declName :: a,
                  declNumParameters :: Word8,
                  declCons :: Maybe (ConTree b c)}

K07b1b045ac3c:
 ADTRef a ≡   Var Word8
            | Rec
            | Ext a

K4bbd38587b9e:
 AbsRef ≡ AbsRef (SHAKE128_48 (ADT Identifier
                                   Identifier
                                   (ADTRef AbsRef)))

K066db52af145:
 Char ≡ Char Word32

K86653e040025:
 ConTree a b ≡   Con {constrName :: a,
                constrFields :: Either (List (Type b)) (List (Tuple2 a (Type b)))}
                 | ConTree (ConTree a b) (ConTree a b)

K6260e465ae74:
 Either a b ≡   Left a
              | Right b

Kae1dfeece189:
 Filler ≡   FillerBit Filler
          | FillerEnd

Kdc26e9d90047:
 Identifier ≡   Name UnicodeLetter
                     (List UnicodeLetterOrNumberOrLine)
              | Symbol (NonEmptyList UnicodeSymbol)

K20ffacc8f8c9:
 LeastSignificantFirst a ≡ LeastSignificantFirst a

Kb8cd13187198:
```

```
 List a ≡   Nil
          | Cons a (List a)

Kda6836778fd4:
 Maybe a ≡   Nothing
           | Just a

K74e2b3b89941:
 MostSignificantFirst a ≡ MostSignificantFirst a

Kbf2d1c86eb20:
 NonEmptyList a ≡   Elem a
                  | Cons a (NonEmptyList a)

Kab225802768e:
 PostAligned a ≡ PostAligned {postValue :: a, postFiller :: Filler}

K9f214799149b:
 SHAKE128_48 a ≡ SHAKE128_48 Word8 Word8 Word8 Word8 Word8 Word8

Ka5583bf3ad34:
 Tuple2 a b ≡ Tuple2 a b

K7028aa556ebc:
 Type a ≡   TypeCon a
          | TypeApp (Type a) (Type a)

K3878b3580fc5:
 UnicodeLetter ≡ UnicodeLetter Char

K33445520c45a:
 UnicodeLetterOrNumberOrLine ≡ UnicodeLetterOrNumberOrLine Char

K801030ef543c:
 UnicodeSymbol ≡ UnicodeSymbol Char

Kf92e8339908a:
 Word ≡ Word (LeastSignificantFirst (NonEmptyList (MostSignificantFirst Word7)))

K2412799c99f1:
 Word32 ≡ Word32 Word

Kf4c946334a7e:
 Word7 ≡   V0
         | V1
         | V2
```

```
         | V3
         | V4
...
         | V123
         | V124
         | V125
         | V126
         | V127


Kb1f46a49c8f8:
 Word8 ≡   V0
         | V1
         | V2
         | V3
         | V4
...
         | V251
         | V252
         | V253
         | V254
         | V255
```

About This Document

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

Table 2: Document Metadata.

Status	Draft
Reference URL	http://quid2.org/docs/ZhengMing.pdf
License	GPLv3
First Published	2017-05-01
Last Revision	2017-05-01
Copyright ©	Pasqualino ”Titto” Assini (tittoassini@gmail.com)