Quid2 Model (First Draft)

Pasqualino ‘Titto’ Assini (tittoassini@gmail.com)

22$^{\text{nd}}$ of November 2015

Contents

Quid2 Model

Quid2 Model is a simple data declaration language.

It is self-described by the following model:

```
-- A type expression
data Type ref = TypeCon ref

               -- | Type application
               | TypeApp (Type ref) (Type ref)

-- Simple algebraic data type
data ADT name ref =
       ADT
         { declName          :: name
         , declNumParameters :: Natural
         , declCons          :: Maybe (ConTree ref)
         }

-- Constructors are disposed in an optimally balanced, right heavier tree
data ConTree ref =
  Con {
  -- | The constructor name must be unique in the data type
  constrName :: String

  ,constrFieldsTypes :: [Type ref]

  -- | If present, all fields must have a corresponding name
  ,constrFieldsNames :: Maybe (List String)
  }

  | ConTree (ConTree ref) (ConTree ref)

-- An Unicode string
data String = String (List Char)

-- An Unicode char
-- Defined as the corresponding unicode point (0..10FFFF hexadecimal)
data Char = Char Natural

-- Natural number (non-negative integer)
-- Defined as the concatenation of a list of Word7 (most significant Word7 first)
-- 0 is encoded as the empty list
data Natural = Natural (List Word7)

-- A, possibly empty, list
data List a = Nil
            | Cons a (List a)

-- An optional value
data Maybe a = Nothing
             | Just a

-- A 7 bits word (0..127 natural)
data Word7 = V0 | V1 | V2 | V3 | V4 | V5 | V6 | V7 | V8 | V9 | V10
           | V11 | V12 | V13 | V14 | V15 | V16 | V17 | V18 | V19 | V20
           | V21 | V22 | V23 | V24 | V25 | V26 | V27 | V28 | V29 | V30
           | V31 | V32 | V33 | V34 | V35 | V36 | V37 | V38 | V39 | V40
           | V41 | V42 | V43 | V44 | V45 | V46 | V47 | V48 | V49 | V50
           | V51 | V52 | V53 | V54 | V55 | V56 | V57 | V58 | V59 | V60
           | V61 | V62 | V63 | V64 | V65 | V66 | V67 | V68 | V69 | V70
           | V71 | V72 | V73 | V74 | V75 | V76 | V77 | V78 | V79 | V80
           | V81 | V82 | V83 | V84 | V85 | V86 | V87 | V88 | V89 | V90
           | V91 | V92 | V93 | V94 | V95 | V96 | V97 | V98 | V99 | V100
           | V101 | V102 | V103 | V104 | V105 | V106 | V107 | V108 | V109 | V110
           | V111 | V112 | V113 | V114 | V115 | V116 | V117 | V118 | V119 | V120
           | V121 | V122 | V123 | V124 | V125 | V126 | V127
```

Algebraic Datatype Definitions

Algebraic datatype declaration are used to introduce new datatypes and therefore new constructors.

Some examples follow.

A datatype that contains no values:

```
data Empty
```

A datatype with a single value (note that the datatype name can be the same as the name of one of its constructors):

```
data () = ()
```

A datatype with two values:

```
data Bool = False | True
```

A simple recursive datatype (a representation of the natural numbers):

```
data N = Z | S N

zero = Z

one = S Z

two = S (S Z)

three = S two
```

A parametric datatype:

```
data Maybe a = Nothing | Just a
```

A parametric datatype with two variables:

```
data Either a b = Left a | Right b
```

A parametric and recursive nested datatype (a list type):

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

Another list datatype, using symbols as constructor names:

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

Value declarations have the following syntax:

valueDecl = *name* `=` *value*

value = *id* | *value value* | (*value*)

Algebraic datatype declarations have the following syntax:

`data` *id* {*variable*}~0.. [`=` *constructor* {`|` *constructor*}]

constructor = *id* {*type*}

type = *id* | *variable* | *type type* | (*type*)

id = *name* | *symbol*

name = an identifier beginning with an uppercase letter

symbol = an identifier composed of non-alphanumeric characters

variable = an identifier beginning with a lowercase letter

Where:

- `data` , `=` , `|` ... are keywords
- | indicates an alternative between two elements
- {}$_{n..m}$ indicates an element repeated between n and m times
- [] indicates an optional element (a shorthand for {}$_{0..1}$)

Note: to avoid ambiguity with variables, we restrict data types and constructor names to start with an upper case letter, though this is good practice is not required by the model.

TOFIX: does not support constructor field names.

Why Algebraic?

As an algebraic datatypes is a sum of (named) products of types, their structure is similar to that of ordinary algebraic expressions.

Consider the following type:

```
data Either a b = Left a | Right b
```

How many values does it have?

`Either` contains all the `Left` values, that's to say all values of type `a`, plus all the `Right` values, that's to say all the values of type `b`.

We could say that:

`Either a b = a + b`

Now consider the type:

```
data Both a b = Left a | Right b | Both a b
```

It has all the values of `Either` plus the values added by the `Both` constructor.

How many values can be created using the `Both` constructor?

For every `a` value we can have any `b` value so the number of `Both` values is equal to the number of `a` values multiplied by the number of `b` values.

We could say that:

`Both a b = a + b + a * b`

Doesn't that look precisely like one of these little algebraic formula that we all studied at primary school?

Syntactically, the only difference is that in the datatype definition we:

- give an explicit name to every term
- write + as |
- don't explicitly write * (just as we usually do in algebra)

Applying these rules the algebraic formula:

`Both a b = a + b + a * b`

translates precisely back to our algebraic datatype definition:

```
data Both a b = Left a | Right b | Both a b
```