Quid2 Manual (Draft)

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

4th of November 2013

Contents

For more information and the most recent version of this specification check http://quid2.org.

What is Quid2?

Quid2[1] is a novel approach to the creation of open, evolvable, consistent and efficient distributed systems.

It aims to be as simple and pragmatic as possible but also expressive enough to represent and exchange any kind of information and implement any kind of distributed system.

It consists of:

[1] Quid2 is read as *quidquid*, a Latin word meaning *anything* or *whatever*. The name suggests that Quid2 is meant to be an open and universal system. Why a Latin name? Obviously because *quidquid Latine dictum sit altum videtur*.

- A flexible evaluation model to coordinate and exchange information across distributed agents.
- A simple yet expressive and precise language to define **globally unique** data types.
- An abstract network protocol plus a set of concrete implementations to communicate in different network environments (single process, Internet, Web clients).
- An efficient serialisation format to store and transfer values of potentially unlimited size.

Why Is Quid2 (Or A Close Relative) Needed?

Consider the following scenario: a user, connected to the Internet via a slow link, urgently needs to print a very big spreadsheet, appropriately named `BS.XSL`, in her office.

The office provides three services:

- a store where documents are held
- a converter that can transform a spreadsheet document to a PDF document
- a printer that can print PDF documents

The initial situation is the following:

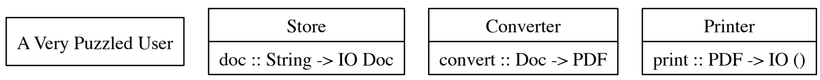

Figure 1: A Coordination Problem

A brief explanation about the diagram, the functionality provided by every service is neatly specified by a list of function signatures:

- Converter's `convert::Doc -> PDF` is a pure function that given a document (a `Doc`) will return a `PDF` document. Pure means that `convert` works just like a mathematical function: given a certain input it will always return the same output (`2+2` always equals `4`). A given `Doc` will always be converted to the same identical `PDF`.

- Store's `doc::String -> IO Doc` is an impure function that given a document name will return the corresponding file. We know that is impure because it doesn't return a `Doc` but an `IO Doc`. The IO means that this is an Input/Output function that gets its hands dirty by somehow interacting with the external world, in this case presumably a hard disk where documents are stored. If we apply the same function again at a later time we might get a different result as the document stored under the name "BS.XSL" in the disk might have been udpated.

- The Printer's `print::PDF -> IO ()` is an impure function that given a `PDF` file will print it and return an `()`. `()` is a nullary (empty) value, returned just to let us know that the printer has successfully completed its job. The function is obviously impure: if we try to print the same document again we might receive an "Out of Paper" error rather than a nice round `()`. The output of the function depends on the state of the world and as the world changes all the time (paper finishes, toner get stolen), the same result cannot be guaranteed.

Neither of these services in isolation can provide the service that the user needs so some coordination will be needed, so what is she to do?

Manual Coordination

The user might coordinate these services directly and manually: the following diagram shows how she might go about it, connecting to each service in turn and moving data back and forth[2].

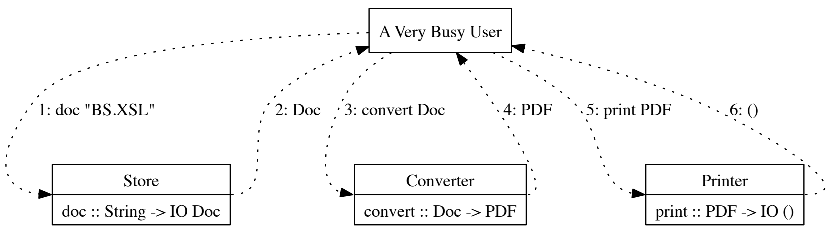

Figure 2: Manual Coordination

It works, but is cumbersome and not particularly efficient: what might be rather bulky documents are transferred multiple times on a slow connection.

[2] An arrow indicates that some information is being transferred from one agent to another. The first number in the arrow's label indicates the time at which the transfer takes place. A dotted line indicates a slow link, a full line a fast one.

A Custom Front End

There is another solution: maybe our user has been prevident and before leaving office she has created a custom front end, an agent that knows all about printing her BS document.

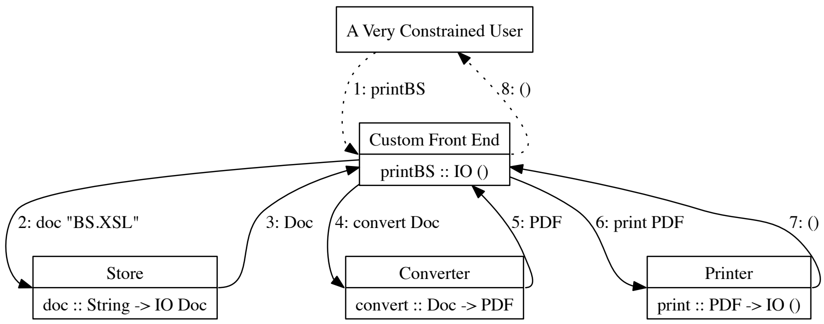

Figure 3: Custom Solution

It's a brilliant solution: all she needs to do is to get the process started, the custom front-end then takes care of all the coordination.

It's also efficient as most communication takes place on local high speed links.

But what if she wants to print a different document? Or if she has different kinds of documents to print? Or if occasionally some additional conversion or reformatting operation is required? How many different custom functions will she need[3]?

Distributed Evaluation

There is a better way. At the place of the rigid custom front end, she puts a flexible Coordinator that has the capability of evaluating simple expressions.

She can now send an expression that, when executed by the Coordinator, will do exactly what she requires:

```
Store.doc "BS.XSL" >>= Printer.print . Converter.convert
```

The Coordinator implements just two additional, but very powerful, higher order[4] functions:

[3]The idea that one might design a system as an unwieldy set of *ad-hoc* functions sounds absolutely preposterous till one realises that this is exactly how 99% of Web sites are made.

[4]A higher order function is a function that takes as inputs and/or returns other functions.

- `(>>=)::IO a -> (a -> IO b) -> IO b` is a sequencing operation used to bind together different IO operations [5]. Given an IO operation that returns some value (`IO a`), it will execute it and then feed its result to the next IO operation (`a -> IO b`). In the example, it is used to feed the `Doc` returned by `doc::String -> IO Doc` to the print operation that follows.
- `(.)::(b -> c) -> (a -> b) -> (a -> c)` is functional composition: given two compatible functions it will return their combination. In the example, the `Doc` returned by `doc` cannot be printed directly, so we compose `print::PDF -> IO ()` with `convert::Doc -> PDF` obtaining just what we need: a print function that given a `Doc` will print it (a `Doc -> IO ()` function).

Note that the `Coordinator` is totally generic, it can be used to orchestrate all kind of activities and calculations across any number of different services. It's simple but totally flexible[6].

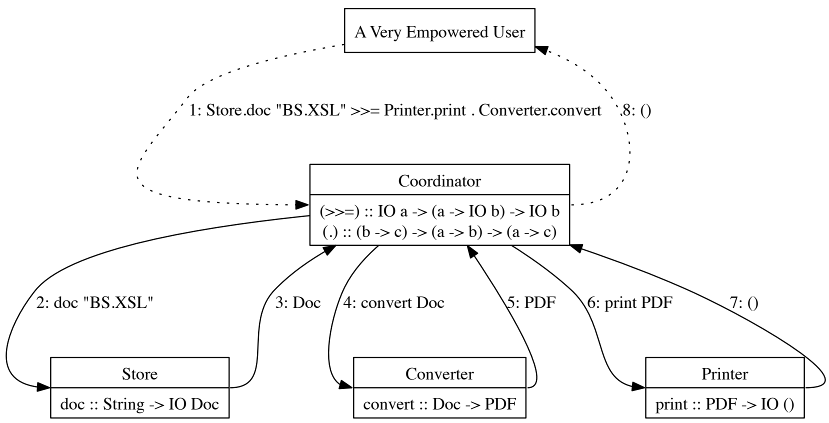

Figure 4: Distributed Evaluation

Extensible Distributed Evaluation

Obviously she has lost the extreme simplicity of the custom solution: repetitive operations now have to be entered as, possibly verbose, expression. This can be

[5] Technically, it is a monadic bind restricted to IO.

[6] An intelligent coordinator can also provide some automatic optimisations. As we mentioned, `convert` is a pure function. It's also probably expensive in terms of time and network bandwidth. An intelligent coordinator would cache the result of this operation and reuse it if the same conversion is requested again, so avoiding to contact the `Converter` at all.

easily fixed by adding to the Coordinator the capability of accepting definitions of new functions[7] ending up with a system that is both flexible and concise.

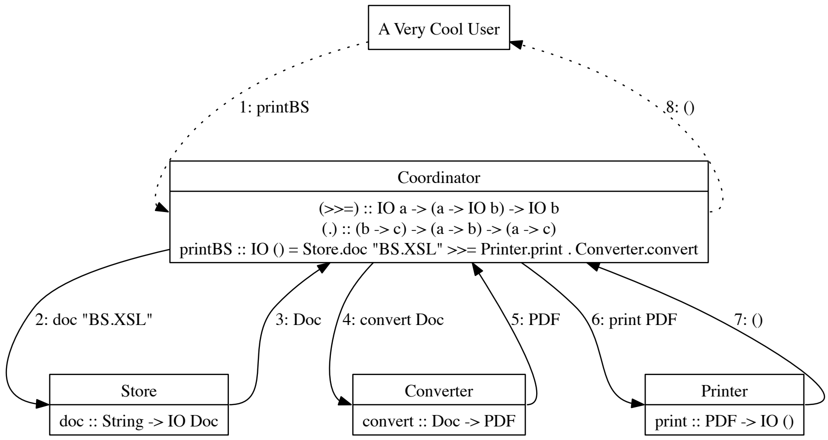

Figure 5: Extensible Distributed Evaluation

Quid2 provides precisely the tools required to build this kind of simple but flexible and scalable solutions.

Language

To both explain and specify the language at the core of Quid2 we:

- start by quickly introducing some basic concepts
- proceed to an informal exposition of the Quid2 language structure
- provide a formal specification of the language abstract structure in the language itself

Note that, as Quid2 is self-defined, we sometime need to use concepts before they are fully explained.

It might be advisable to start by quickly browsing the contents of this chapter, skipping what is not immediately understandable and coming back to it later.

Hopefully it will all make sense in the end :-)

[7] As many other aspects of this example application, this is an oversimplification. Definitions will usually be stored in separate repositories so that they can be easily shared.

Basic Concepts

The Quid2 global space is a distributed set of definitions (or declarations, we will use the two terms interchangeably) of functions and datatypes.

Quid2 is a typed language with a simple but expressive polymorphic type system and a hierarchical type hierarchy (composed by values, types, kinds, sorts and higher sorts).

A function declaration binds a qualified name to an expression.

An expression denotes a value and has a static type[8].

Evaluating an expression consists in reducing an expression to the value it denotes.

Could as in Coq, evaluation order be itself a parameter? Lazy/Eager ..? What would be the advantage?

A value is composed by a type and either:

- an array of bytes, built according to the canonical serialisation
- a function located at some network or local address
- an error

A datatype declaration defines an algebraic datatype (composed by one type constructor plus a set of value constructors).

A constructor is a function that creates a value (at some level of the type hierarchy).

Informal Syntax

Quid2 does not have a formally defined textual syntax[9].

However, for the purpose of this report, we will briefly introduce and use a convenient informal syntax.

Constructors

Constructors are denoted by identifiers beginning with an uppercase letter or symbols (identifiers composed of non-alphanumeric characters).

Constructors can be applied to one or more other values.

Some examples:

[8] A type that can be determined by a static analysis of the expression.

[9] It is not clear if such a surface syntax is needed at all, given that Quid2 definitions will probably be usually automatically derived from definitions in other languages. Quid2 abstract syntax is however very precisely defined in Quid2 itself.

```
False    -- boolean false

True     -- boolean true

-- a missing optional value
Nothing

-- a present optional value
-- Just is applied to True and together form a single value.
Just True

-- the nullary value
()
```

There is a special syntax for numbers, chars and strings constructors:

```
1      -- a positive integer

-5     -- a negative integer

3.14   -- a floating point

'a'    -- the character a

"a string" -- a unicode string
```

There is also a special syntax for two common data structures: lists (unlimited sequences of values of the same type) and tuples (fixed sequences of values of possibly different types):

```
-- a list of integers
[1,2,3]

-- a tuple composed by a char, a string and an integer
('a',"bc",22)
```

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.
```

Algebraic datatype declarations have the following syntax:

[`data`] *simpleType* [`=` *constructor* {`|` *constructor*}]

simpleType = *id* $\{variable\}_{0..256}$

constructor = *id* {*type*}

type = *id* | *variable* | *type type* | *type* `->` *type* | `(`*type*`,`*type*{`,` *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 $\{\}_{0..1}$)

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:

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

Primitive Types

We will assume a few primitive types, for which we do not need to provide an explicit datatype definition:

- Unit type: `()`
- Unicode characters: `Char`
- Unsigned integers: `Word8` (8 bits), `Word16`, `Word32` and `Word64`
- Signed integers: `Int8`, `Int16`, `Int32` and `Int64`
- Unlimited size signed Integers: `Integer`
- Floating point numbers: `Float16`, `Float32` and `Float64`

We also have the parametric types:

- the function type: `a -> b`
- lists: `[a]`
- a whole family of tuple types (of size >=2): `(a,b) (a,b,c) (a,b,c,d)` ...

We will also use `String` as a shorthand for `[Char]`.

We can imagine the primitive types defined as follows:

```
data () = ()

data Char = '\NUL' ...  | 'a' | 'b' ... | '\1114111'

data Word8 = 0 | 1 .. | 255

data Int8 = -128 | -127 .. | 127

data Integer = -infinite ... | +infinite

data Float32 = ... | 0.1 .. | 0.1E+1 ...

data (->) a b

data [] a = []
          | : a ([] a)

data (,,) a b = (,,) a b
data (,,,) a b c = (,,,) a b c
... infinite??? tuples
```

Finally the parametric IO type that indicates a value that is the result of an interaction with the external world.

You can think of an `IO a` as a pure function with an additional hidden parameter of type `World` that indicates the current state of the world[10]:

```
type IO a = World -> a
```

There are cases when we do not expect a value to be returned, these are marked by the special type `IOK`.

```
type IOK = World -> ()
```

From the point of view of the caller, a function of type `IOK` performs exactly like an `IO ()`.

From the point of view of the callee, the function is received, its behaviour performed but nothing is returned back.

[10]Something to ponder upon on a rainy day: if we could capture the state of the world in a value, all impurity would disappear.

Type Judgments, type checking

Requirements: * simple to understand * simple to use * efficient and simple to implement (even better if typechecking could be subsumed by de/serialisation) * easy to map to a variety of existing languages (haskell, js, java...) * support for upward compatibility: built in numeric subtypes and, more importantly, data type subsets * compatible with absolute types * support for data types * support for data structures: list, tuples, maps. * expressive, open potential for future expansion ** support for type classes/interfaces ** support for first class modules ** support for first class patterns ** The concepts to unify: value, expression, type, kind, sort, pattern, class/interface, module. ** Pluggable? As is defined in the language it can also be changed.

We need to support multiple types for functions.

The most expressive, though the least constraining is a verbal description. It allows to express anything but cannot be mechanically verified.

In between, we have simple fully inferable type systems.

Then we have more sophisticated but maybe only checkable type systems.

General type and subtype theory: http://lucacardelli.name/Papers/OnUnderstanding.A4.pdf

Type system design options: * haskell like: terms : types : * | * -> * full implementation of typechecking algorythm, including typeclasses in ~/cache/haskell/language/thih-multiparam/ http://web.cecs.pdx.edu/~mpj/thih/thih.pdf

- pure subtype system best to represent numeric subtypes and, more importantly, data type subsets (used for upward compatibility). ?? 3.3 controvariance: can a function on t be applied to t1<t ? if not what is the whole point?

http://redwood.mza.com/~dhutchins/papers/popl10-hutchins.pdf

- pure type systems: Jan-Willem Roorda; Johan Jeuring. "Pure Type Systems for Functional Programming". Roorda's masters' thesis (linked from the cited page) also contains a general introduction to pure type systems.

Pure type system with single term level and data types /cache/haskell/language/Henk2000/ http://www.staff.science.uu.

- dependent system: single level with dependent types

Simple implementation of dep type system http://www.andres-loeh.de/LambdaPi/LambdaPi.pdf ~/cache/haskell/language/Agda-2.3.2.2/src/prototyping/termrep/lambdapi/

** Idris: practical haskell-like dependent type language

** lambda aleph dependent language with single level terms, (sub)types as multi-values: http://www.leafpetersen.com/leaf/publications/dtp2013/lambda-aleph-overview.pdf Formalisation of the Runtime http://arxiv.org/pdf/1307.5277v1.pdf

** Agda, total dependent language and theorem prover http://oxij.org/note/BrutalDepTypes/ http://www.cse.chalmers.se/ulfn/papers/afp08/tutorial.pdf http://www.cse.chalmers.se/peterd/papers/DependentTy

Can this be defined as a type function?

-- typeOf 0 = Word8

Adding to the set of values is also called lifting.

Coordination

See Concurrent bondi ~/cache/haskell/language/Concurrent Pattern Calculus .pdf

Type Classes

http://okmij.org/ftp/Computation/typeclass.html

using modules, 3.4.5 3.4.6 and 4.8.1 of file://localhost/Users/titto/cache/haskell/language/pure%20subtype

http://ropas.snu.ac.kr/~bruno/papers/TypeClasses.pdf

http://ipaper.googlecode.com/git-history/243b02cb56424d9e3931361122c5aa1c4bdcbbbd/Typeclasses/typ class-explorationi.pdf

Dynamic typeclasses: https://www.fpcomplete.com/user/thoughtpolice/using-reflection

Pattern matching

Scala allows types to be used as patterns in pattern matching expressions

The Values Hierarchy

Values are organised in a hierarchy, where higher levels organize the next lowest level.

At the lowest level we have the values, neatly separated in sets, each labelled by a type.

Types are themselves part of a set: that labelled by the catch-all kind *.

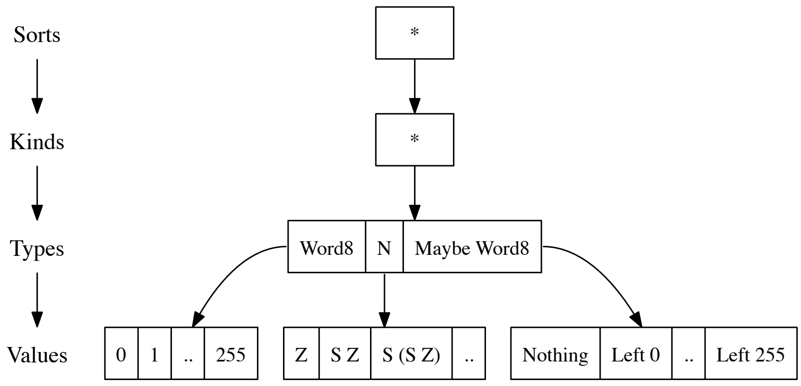

Figure 6: Values Hierarchy

Note the similarity in the relationship between kind and types and the one between types and values.

In fact, we can think of the * kind as an higher level datatype, whose constructors are the types themselves:

```
kind * = Word8
       | Maybe *
       | N
       ...

-- Or:

open kind * :: * -- There is an open kind * of sort *

Word8 :: *

Maybe :: * -> *

N :: *
```

The main difference with ordinary datatypes is that * is open, every time we define a new datatype it gets an additional constructor[11].

So values are really 0-level values while types are 1-level values, kinds are 2-level values, sorts are 3-level values and so on.

[11] Alternatively, you can imagine that it contains *ab initio* all possible datatypes.

Every level defines a different namespace so, for example, the `*` kind cannot be confused with the `*` sort.

Datatypes also have their own namespaces, same named constructors from different datatypes are always distinguishable.

Obviously, the hierarchy continues *ad infinitum*, but is not a very interesting kind of infinity given that, starting with the kind level, is `*` all the way up (things get a lot more interesting if we start promoting lower level datatypes to higher levels, more on this later).

Function Definitions

Functions are introduced by declarations of the form:

[*name* `::` *next-higher-level-expression*]

[`type` | `kind`] *name* `=` *expression*

The first line is an optional type signature, it indicates the type (or kind or sort) of the function. The declaration is optional when the type can be inferred by the expression itself.

The initial keyword (`type` , `kind` or absent) indicate the level in the type hierarchy at which the definition applies.

If absent, the definition is at the value level.

Some examples:

```
-- Value level definitions

-- No need for a type signature, this is clearly a string
speech = "to be or not to be"

pi :: Float32
pi = 3.1416

printDoc :: String -> IO ()
printDoc name = doc name >>= print . convert

-- Type level definitions
type String = [Char]

type E a = Either a Int
```

Qualified Names

It's convenient to organise our definitions in a hierarchical namespace.

We will specify the namespace with the syntax:

`module` *namespace* `where`

For example:

```
module Data where

data Bool = False | True
```

defines the type `Data.Bool` and the constructors `Data.False`, `Data.True`.

```
module Math.Constant where
pi = 3.1416
```

defines the function `Math.Constant.pi`.

Located Names

Quid2 is just a coordination framework, the real functionality is provided at its borders by service providers.

Functions provided by service providers are denoted by a naming scheme expressed as an ordinary datatype.

For example, the services described in the initial example might be denoted by values of the following type:

```
data SimpleAddr =
  SimpleAddr
  String -- Agent where function is located: "User" | "Printer" ...
  String -- Function name: "print" | "convert" ...
```

Syntactically, we write these references as:

`<<` *reference* `>>`

A subset of the functions of our example could be expressed as:

```
module Quid2.Example where

print :: PDF -> IO ()
print = <<SimpleAddr "Printer" "print">>
```

```
convert :: Doc -> PDF
convert = <<SimpleAddr "Converter" "convert">>

data PDF = PDF String

data Doc = Doc String
```

Formal Definition

The abstract syntax of Quid2 is defined as follows:

```
-- This is what Quid2 is all about, evaluating expressions.
-- Evaluation reduces an expression
-- to the equivalent normal form value.
evaluation :: Expr n t -> NFValue n t

{- A typed value in normal form.
This is the form in which all data
is ultimately exchanged across Quid2 agents.
The presence of a Fun contructor
is exploited to allow the efficient
progressive transfer of data of
potentially unlimited size (more about this later).
-}
data NFValue n t =
  -- a function located at some address
  Fun n
  -- an array of bytes, built according to the canonical serialisation
  | Lit [Word8]
  -- an error
  | Err String

-- Absolute declarations, see next section for an explanation.
type AbsType = Type AbsName
type AbsDecl = DataDecl AbsName
type AbsDefinition = Definition AbsName

data InternalAbsName =
  -- Reference to a data type in the same recursive group.
  InternalName QualName
  -- A declaration in a group of mutually recursive definitions.
  | ExternalName AbsName

data AbsName =
```

```haskell
  -- A stand alone declaration
  AbsName (Ref (DataDecl InternalAbsName))

  -- A decl in a group of mutually recursive definitions.
  | AbsNamePart (Ref [DataDecl InternalAbsName]) Word8

-- A value expression
type Value n = Expr (ValueName n) AbsType

-- A type expression
type Type name = Expr (TypeName name) ()

-- Generic expression.
data Expr n t =
  App (Expr n t) (Expr n t)     -- functional application
  | Name n
  | Var Variable                -- a variable
  | Lambda Variable (Expr n t)  -- a lambda expression
  | Signature (Expr n t) t      -- an explicit type signature

type Variable = Word16

-- Value constructors
data ValueName n
    =
      Function n
    | Con AbsName Tag  -- data constructor
    -- Primitive constructors
    | Tuple Word16  -- (,,) (,,,) ...
    | ListCons      -- :
    | ListNil       -- []
    | Unit          -- ()
    | Char Char
    | Word8 Word8 | Word16 Word16 | Word32 Word32 | Word64 Word64
    | Int8 Int8 | Int16 Int16 | Int32 Int32 | Int64 Int64
    | Float32 Float | Float64 Double
    | Integer Integer

-- Type constructors
data TypeName name
     = TyDecl name
     | TypeFun           -- ^ function type
     | TypeTuple Word16  -- ^ tuple types: example: TypeTuple 3 == (,,)
     | TypeList          -- ^ list type
     | TypeIO            -- ^ IO type
     | TypeUnit
```

```haskell
      | TypeChar
      | TypeWord8 | TypeWord16 | TypeWord32 | TypeWord64
      | TypeInt8 | TypeInt16 | TypeInt32 | TypeInt64
      | TypeFloat32 | TypeFloat64
      | TypeInteger

-- Function definition
data Definition n t = Definition QualName (Expr n t)

-- Algebraic data declaration
data DataDecl name = DataDecl {
  -- qualified Name
  ddName::QualName

  -- number of parametric variables
  ,ddNumVars::Word8

   -- value constructors in order of definition
  ,ddCons:: [Cons name]
}

-- Constructor declaration
data Cons name = Cons Name Tag (Maybe [Name]) (Type name)

-- Position in data type declaration, 0-based.
type Tag = Word16

data Name
    = Id String        -- "a" "plus" "List"
    | Symbol String    -- "+" "()"

type ModuleName = String

-- Qualified Name
data QualName = QualName ModuleName Name

-- A transferable, compact, reference to an object.
data Ref a = Verb [Word8] -- Verbatim serialisation
           | Hash [Word8] -- SHA-256 hash
```

The Quid2 structural definitions are on purpose parametric, allowing naming conventions suitable for different network environments (in-process, local or global) to be used.

Absolute Datatypes and Functional Definitions

Quid2 can be used in many different ways:

- as a storage and interchange format
- to build distributed but local and self-contained systems, as the one we briefly discussed in the introduction
- to coordinate independently developed Internet services

To achieve the last objective, it is necessary that, progressively, service providers converge on common or compatible data definitions.

For example, to allow the coordination of different flight reservation system a common vocabulary of concepts such as flight, ticket, airport, departure and arrival time need to be established.

This is similar to the process by which natural languages evolve: new words are added all the time and those that are found useful are progressively adopted.

This is a diffuse, distributed process that cannot be imposed or hurried by force or decree but that can be made smoother by providing a common framework in which these concepts can be expressed.

To favour this process, Quid2 datatypes and functional definitions are denoted by globally unique identifiers that are deterministically derived from their structure.

In other terms, datatypes that are structurally identical, even when developed independently, will end up having the same global identifier and therefore functions that use them will be *ipso facto* composable.

Absolute datatypes references are build with the following algorithm:

- Split the type graph in sets of strongly connected types (sets of mutually recursive datatypes).
- For each set:
  - Transform the set to its canonical form:
    - Sort datatypes declarations lexicographically
    - For each datatype:
      - Normalise variables
      - Convert external references to absolute references
      - Serialise the dataset value and calculate a unique hash value

There are many ways in which structural equivalence can be defined.

In Quid2, two datatypes are considered equivalent only if:

- the fully qualified names of their type constructors are identical
- their normalised parametric parameters are identical
- their value constructors are also identical

Both standalone and mutually recursive sets of datatypes are supported.

Consider the two declarations:

```
data Maybe b = Nothing | Just b

data Maybe c = Nothing | Just c
```

Maybe does not explicitly refer to any other type.

So the only thing we need to do is to normalise the variable names transforming both of them to:

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

Their global id will therefore be the same.

In contrast the two declarations:

```
data Maybe b = Just b | Nothing

data Maybe c = Nothing | Just c
```

won't match as the constructor ordering differs.

A looser concept of structural equivalence is possible and might be useful in certain cases, it however increases the risk of mismatches (what would be called *false friends* in natural languages).

Functional definitions are similarly denoted by globally unique identifiers deterministically derived from their structure.

Protocol

In Quid2, communication takes place between endpoints:

- bound to a specific identity
- addressed via a local or global addressing scheme
- connected via uni-directional typed channels

Identity

Identity is defined as the provable control of an end point.

A datatype that captures some of the many possible identities:

```
data Identity = Anonymous
        | OpenID String
        | Email String
        | Facebook String
```

Addressing Schemes

As discussed previously, Quid2 doesn't impose a particular addressing system as different ones might be suitable for different network environments.

For Internet wide services, the use of the following datatype is however advised:

```
module Network where

-- An Internet endpoint address
data Route =
  -- A route that goes through an intermediary (e.g. a proxy or a router)
  Via
  Route -- care taker, gateway,
  Route -- final destination

  | Local String -- Local address

  | IP IPAddress -- Internet address

data IPAddress = IPAddress Host Port

-- -- Either an IP or symbolic domain: e.g. "127.0.0.1" or "example.org"
type Host = String

-- e.g. 8080
type Port = Word32
```

Example:

```
-- ring a bell at example.org, reached via quid2.org:8080 and local port 1234.
ringABell :: IO ()
ringABell = <<Via (IP "quid2.org" 8080)
            (Via (IP "example.org" 1234) (Local "bell"))>>
```

Channels

A typed channel can transfer an unlimited number of values of its type.

Contrary to usual network protocols, there is no need for extensive *in-protocol* negotiation of protocol parameters such as versions, format, compression, timeouts or quality of service.

Assuming that every agent has a bootstrap channel of known type to start its interaction with the Quid2 system, further channel references should be obtained by functions that take care of all these aspects, escaping the limits of hard wired network protocols.

Each channel has an associated identity (the identity of the peer with which we are communicating).

Channels come in all kind of flavours, some examples follow.

```haskell
-- A proxy used by web clients, on either HTTP or HTTPS
-- with optional compression.
data HttpChannel a framing = HttpChannel IP Secure framing

type Secure = Bool

-- Some examples of framings
data WebRPC

data NoCompression = NoCompression
data GZIPCompression = GZIPCompression
data SnappyCompression = SnappyCompression

-- A TCP channel

data TCPChannel a = TCPChannel IP

-- A Local, in-process channel
data LocalChannel a = LocalChannel Word64 -- channel identifier.

-- Some channels might be constrained to transfer only a certain type of values
-- This will only transfer strings coded according to Twitter's standards.
data TwitterChannel = TwitterChannel String -- channel identifier
```

Some example channels:

```haskell
-- A stream of IBM's market prices
ibmQuotes :: TCPChannel Float32
ibmQuotes = TCPChannel (IP "quotes.ibm.com" 1234)
```

```haskell
-- A channel to a local Quid2-aware printer
-- that is able to interpret simple expressions.
myPrinter :: TCPChannel (Expr String)
myPrinter = TCPChannel (IP "192.168.1.4" 4444)
```

Serialisation

Serialisation is the process by which a Quid2 value is encoded into a concrete binary representation, transferred over a network of stored locally, and then decoded to an equivalent value.

The stages of the serialisation process:

- Canonicalisation: data value is converted into a canonical form.
- Encoding: the data value is encoded into a byte stream.
- Framing: data is framed according to the rules of the underlaying communication protocol (e.g. if data is to be stored locally is written to a file, or if transferred via HTTP is framed as an HTTP request). Data compression is an optional sub-stage of framing.
- Unframing: data is unframed (and possibly uncompressed).
- Decoding

```haskell
module Network.Serialise where

-- A value of type `a`, encoded as specified by `encoding`.
data Encoded a encoding = Encoded [Word8]

-- The canonical encoding.
data CanonicalEncoding = CanonicalEncoding

primitive encode :: a -> Encoded a CanonicalEncoding
primitive decode :: Encoded a CanonicalEncoding -> Either String a
```

Requirements:

- Support for all definable data types (including function types).
- Support for expressions and infinite data structures.
- Support for undefined values and errors.

- Support for basic data structures (tuples, lists and maps).
- Support for unsigned and signed numeric types.
- Support for infinite precision integers.
- Upward compatibility for unsigned and signed numeric types: numeric types of lower precision can be decoded as higher precision types (for example: an Int8 can be read as an Int16, however a Word8 cannot be read in as an Int16!).
- Support for IEEE 754 floats.
- Support for infinite precision floats.
- Easy and fast to decode.
- Compact data representation (competitive with existing standard such as rfc7049).

Non requirements:

- Self-describing format: unnecessary as communication is expected to take place on typed channels(dynamically typed values can also be transferred using Typed values).
- Human readable (though is convenient to be able to read at least small unsigned integers)
- Encoding efficiency, not very important compared with decoding efficiency as:
  - For every encoding operation there are 1..N decoding operations.
  - Static values can often be encoded offline and the result cached.
  - The time needed to dynamically generated a value is likely to dominate the time taken to encode it.
- High compression: this is better provided by a separate compression stage.

Canonical Implementation

Encoding is byte-aligned: an encoding is a, possibly empty, sequence of bytes.

An encoded value is not self-describing, we need to know its type to decode it.

In this section, we will use the symbol -> to indicated encoding.

So 'a -> [97]' means that the characted 'a' is encoded as the bytes sequence '[97]'.

Coding of Data Types

As we have seen, data types are defined as a sum of constructors and every constructor is a named product of values.

Every constructor is uniquely identified by its tag (that correspond to its 1-based position in the data type declaration).

The tag has type `Word16` so a datatype can have up to 65535 constructors.

A value created by a given constructor is encoded as the concatenation of its constructor tag and of the encodings of all its components.

If the type has a single constructor, the tag is omitted.

For example, the single constructor type () is encoded as an empty byte sequence:

```
() -> []
```

Consider the parametric type:

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

It has two constructors: `Nothing` that has a tag of 1 and `Just` that has a tag of 2.

The encoding of a `Nothing` value is simply its tag, the encoding of a `Just a` is its tag followed by the encoding of the 'a'.

For example:

```
(Nothing::Maybe Char) -> [1]
```

and:

```
(Just 'z'::Maybe Char) -> [2,122]
```

Data types that have no constructors obviously cannot be serialized:

```haskell
data Void
```

Coding of Tuples

The encoding of a tuple is simply the concatenation of the encodings of its components.

This is consistent with the view that a tuple is just a predefined data type with a single constructor, as for example:

```
data (,,) a b c = (,,) a b c
```

As example, given that:

```
"abc" -> [4,97,98,99,1]
```

and:

```
(34::Word8) -> [34]
```

and:

```
'g' -> [103]
```

then:

```
("abc",34,'g') -> [4,97,98,99,1,34,103]
```

Tuples can of course be nested so:

```
('g',("abc",(34,'g'))) -> [103,4,97,98,99,1,34,103]
```

Coding of Lists

For the purpose of serialization, a list is considered equivalent to the data type:

```
data List a =  L0
             | L1 a        (List a)
             | L2 a a      (List a)
             | L3 a a a    (List a)
             | ...
             | L65534 a .. (List a)
```

For example:

```
([5,10,11]::[Word8]) -> [4,5,10,11,1]
```

and:

```
([11,22,33]::[Word8]) -> [4,11,22,33,1]
```

In other terms: lists are represented as a sequence of chunks of no more than 65535 elements each with the last chunk being of zero length.

Coding of Chars

Characters are encoded in UTF-8.

For example:

```
'a' -> [97]
```

and:

```
'\32654' -> [128,231,128,190,128,142]
```

Coding of Numeric Types

The primitive numeric types are:

- Fixed size, unsigned integers: `Word8` (8 bits), `Word16`, `Word32` and `Word64`
- Fixed size, signed integers: `Int8`, `Int16`, `Int32` and `Int64`
- Unlimited size, signed integers: `Integer`
- Fixed size, signed floating point numbers: `Float16`, `Float32` and `Float64`

From these all kinds of numbers can be easily derived.

{- * large floats (rationals, to represent precise decimal fractions like 1.1) - how to represent Infinity, -Infinity, or NaN ? (As there are many representations for NaN. If NaN is an allowed value, it must always be represented as 0xf97e00, from rfc7049).

for testing: check test values at end of http://tools.ietf.org/html/rfc7049#section-2.3 -}

Coding of Words

All unsigned integers (words) are encoded as varwords^[The varword encoding is a variation of the varint representation used in Google's Protocol Buffers.

A varint is a sequence of bytes where every byte except the last one has the most significant bit set (the msb signals that there are more bytes in the sequence).

Varword is basically the same but with all the signalling bits moved into an initial prefix.

This should have two (small) advantages:

- as the word is already in the correct sequence (no extraneous msbs), less operations are required to decode it
- in case of very long words, we end up with an initial prefix of 0xFF bytes that can be easily compressed away

The disadvantage is a more complex encoder.]

A varword is a sequence of bytes composed by:

- A prefix made of:
  - A sequence of 1 bits of length equal to the number of bytes to read in after the one containing the last bit of the prefix
  - A 0 bit
- The word as:
  - Zero or more padding 0 bits
  - The significant bits of the word in network order (most significant bit first) with the last bit of the word stored in the lsb of the last byte in the sequence

Some examples:

Number	Encoding (**prefix**)
0	**0**0000000
1	**0**0000001
127	**0**1111111
128	**10**000000 10000000
255	**10**000000 11111111

256	**10**000001 00000000
16383	**10**111111 11111111
16384	**110**00000 01000000 00000000
72057594037927935 |	**11111110** 11111111(*7)
72057594037927936 |	**11111111 0**1111111 11111111(*7)

Table 1: Varword Encoding of Unsigned Integers

BUG: there is no valid encoding for

72057594037927936 **11111111 0**0000001 0(*7) wrong as there are 7 after prefix* ***11111110*** *00000001 0(*7) wrong as there are 8 after prefix

Number	Encoding (**prefix**)
-2	**01**111110
-1	**01**111111
0	**00**000000
1	**00**000001
63	**00**111111
64	**100**00000 01111111
128	**100**00001 00000000

Table 2: Coding of Signed Integers

The encoder should use the minimum sequence of bytes necessary to code the number, the decoder however needs to be able to handle an unlimited number of leading zeros.

Note that this implies that equivalent unsigned numbers, even if of different types, are encoded in the same way: `3::Word8` is encoded exactly as `3::Word64` (provided obviously that the number fits in the given numeric type).

{- The coding is optimised for small unsigned integers, that are widely used, particularly as constructor tags, and works as follows:

- `Word`s have variable length representations
- In the first byte in the sequence:
  - the values between `0 and 256 minus the Word length in bytes` are used to code the corresponding integer
  - the other values indicate the number of bytes that follow where `number of bytes = 257 - value`
- `Word`s (unsigned integers) are written in big-endian (network) order.

Major types 0 and 1 are designed in such a way that they can be encoded in C from a signed integer without actually doing an if-then- else for positive/negative (Figure 2). This uses the fact that (-1-n), the transformation for major type 1, is the same as ~n (bitwise complement) in C unsigned arithmetic; ~n can then be expressed as (-1)^n for the negative case, while 0^n leaves n unchanged for non-negative. The sign of a number can be converted to -1 for negative and 0 for non-negative (0 or positive) by arithmetic- shifting the number by one bit less than the bit length of the number (for example, by 63 for 64-bit numbers).

void encode_sint(int64_t n) { uint64t ui = n >> 63; // extend sign to whole length mt = ui & 0x20; // extract major type ui ^= n; // complement negatives if (ui < 24) *p++ = mt + ui; else if (ui < 256) {* p++ = mt + 24; *p++ = ui; } else

0..127==2^7-1 0XXXXXXX 128..16383 10XXXXXX XXXXXXXX 16384.. 110XXXXX XXXXXXXX XXXXXXXX XXXXXXXX

Start with a compact count of bytes followed by the actual complement-2 number. This can be implemented efficiently as a jump table? No, it requires a number of left shifts.

Count: Prefix | Num Bytes Overhead 0 1 1/7 1.00 2 3/16 1.01 3 3/24 1.10 4 3/32 111.000 5
111.001 6 111.010 7 111.011 8 111.100 9 111.110 10 111.111.0000 11 ...

The encoding is composed by:

A Prefix composed by: * 0 if the last byte of the prefix also contains the first byte of the number | 1 otherwise * (num of bytes used by number-1) 1s * 0

The number in network order, with the first byte, if indicated, stored in the last byte of the prefix.

0 00.000000 1 00.000001 63 00.111111 64 110.00000 01000000 .. 127 110.00000 11111111 128 010.00001 00000000 4095 010.11111 00000000

-128 0.100000 -3 0.1111100 -2 0.1111101 -1 0.1111111

Plus: Or: The number is already in correct network order (8086/ARM is however LE order), no shifts required (but negative numbers need to have their sign extended).

Minus: An additional bit lost.

The encoding is composed by:

A Prefix composed by: * (num of bytes used by number-1) 1s if the number is >0 |0s otherwise * A number of 0s if the number is >0 |1s otherwise to fill the space up to the beginning of the number

- The number in network order, with the first byte, if it fits, stored in the last byte of the prefix.

-2 1111111.0 -1 11111111. 0 00000000. 1 0000000.1 127 0.1111111 128 1000000.1 00000000

00000000 1 1/8 1.0 2 1/8 11.0 3 1/8 111.0 4 1/8
1111.0 5 1/8 11111.0 6 1/8 111111.0 7 1/8 11111110 8 1/8 11111111 9 1/8

This also makes for a nice jump table. And it should be really easy to compress sequences of 1111111s so it also works for really long numbers so the real overhead is almost nothing.

Another alternative: Always optimise for the specific type, so Word8 is just a Word8, Word16 is .. a varint.

As soon as we know the number of bytes we can just copy them over and mask them to remove the prefix.

Or recursive numbytes we calculate how many bits are required to code the number. we write this as: num_bits ++ num if num_bits is <=2 recursively written

The first two bits mean: 00 = 6 bits 01 = 8+6 bits 10 = 2*8+6 bits 11 = read next 4 bits as number of bits 11.0000 = 3*

Or equivalently MSB encoding: higher bit is used as has-next-byte flag. -}

Coding of Signed Integers

All signed integers, including infinite precision `Integers`, are converted to unsigned integers using Zig-Zag encoding and then coded.

<-- * http://tools.ietf.org/html/rfc7049 * prefix with sign bit -- one extra byte per integers ++ (ok for infinite ints) unsigned starting with zero -1 minus the encoded unsigned integer. For example, the integer -500 would be 0b001_11001 (major type 1, additional information 25) followed by the two bytes 0x01f3, which is 499 in decimal * map -max..+max to 0..2*max by adding max: http://en.wikipedia.org/wiki/Offset-binary -- not ok for infinite ints

`Ints` are converted to their equivalent two's complement `Word` and then coded. -->

For example:

```
(-1::Int8) -> [1]

(3::Int8) -> [6]

(-2::Int16) -> [3]

(5::Int16) -> [10]

(-5::Int32) -> [9]

(11::Int32) -> [22]

(-17283923::Int64) -> [226,15,118,165]

(1567823::Int64) -> [224,47,216,158]

(-2::Integer) -> [3]

(5::Integer) -> [10]

(-17283923::Integer) -> [226,15,118,165]

(1567823::Integer) -> [224,47,216,158]

(829374657329284628329102323297256347::Integer) -> [255,255,1,63,118,167,198,59,48,174,17
```

Coding of Floats

Float16, Float32 and Float64 numbers are coded as standard big endian IEEE754 floats.

Coding of Large and Precise Floats

When Float64 precision is not enough or when there is a need of preserving decimal fractions such as 1.1 that cannot be represented exactly in binary floating point the `Rational` data type should be used:

```
module Math where

import Math.Integer

data Rational = Rational
                Integer -- Numerator
                Integer -- Denominator
```

Example: `1.1` is expressed as `Rational 11 10`.