UniFormal attempt

Lambert Meertens

Printed March 24, 2014

1 General

This document uses Haskell to describe the structure of data meant to be used with a UniFormal-conforming tool. This does not imply that the data actually 'lives' inside Haskell. The actual encoding may use XML or some similar formalism.

Haskell's laziness allows values of data types to be 'infinite', but tools will generally not be able to cope with infinite values, and therefore objects modelled with the following data types are always meant to be finite.

Some subtype restrictions are expressed in natural language.

This version is only concerned with the content of the data, not with the way it should preferentially be presented to human users. Content-wise, unary minus and factorial are both functions, but in applications one is preferentially displayed as a prefix operator, as in '-6', while the other is traditionally written as a postfix operator, as in '6!'. There are several ways to approach such issues, but for now this is left unaddressed.

2 Annotations

A UniFormal object can carry ‘annotations’, being additional information that is not supposed to change the formal meaning of the object but may have a tool-specific effect. A tool should ignore any annotations it does not understand.

```
data X = X [UFannotation]
data UFannotation = A {service :: UFtag, contents :: String}
type UFtag = String
```

(The letter ‘*X*’ is meant to suggest ‘extra information’.)

Each annotation carries a ‘service’ field, that is intended to identify the tool (or tool set) for which it is intended. A developer can freely choose their own tags for whatever services they offer, as long as they pick something that is unlikely to be the same as what someone else might pick. But a tool-specific service tag should not start with ‘*UF*’; these tags will be reserved for possible tool-independent annotations (e.g., date and author).

There may be a need for conceptually binary data in annotations, but such data should always be turned (e.g. by using Base64 encoding) into a string of printable characters – which are assumed to be Unicode characters.

3 Objects and terms

```
data UFobject
  = UFcomment {x :: X}
  | UFtermAsObject UFterm
```

```
data UFterm
  = UFconstant {tx :: X, name :: String, mtype :: UFmtype}
  | UFvariable {tx :: X, name :: String, mtype :: UFmtype}
```

```
    | UFliteral {tx :: X, style :: UFtag, denotation :: String, mtype :: UFmtype}
    | UFapplication {tx :: X, head :: UFterm, arguments :: [UFterm]}
    | UFabstraction {tx :: X, parameters :: UFparameters, body :: UFterm}
    | UFaggregate {tx :: X, aggregator :: UFtag, items :: [UFterm], mtype :: UFmtype}
    | UFquantor {tx :: X, quantor :: UFtag, mtype :: UFmtype}
    | UFcontextedTerm {tx :: X, tcontext :: UFcontext, body :: UFterm}


type UFmtype = Maybe UFtype
type UFtype = UFterm   -- denoting a type


data UFparameters
    = UFparameters {pax :: X, paravars :: [UFvarterm],
      domrestriction :: Maybe UFstatement}


type UFconterm = UFterm
   -- with the restriction that the constructor must be UFconstant
type UFvarterm = UFterm
   -- with the restriction that the constructor must be UFvariable
```

The optional type information supplied by *mtype* fields serves to allow for facilitating the resolution of forms of overloading if the necessary information is locally available. So as not to be unduly restrictive, the syntax for terms representing types is as permissive as that for terms representing values.

Not all values of type *UFterm* represent meaningful UniFormal terms; there are some non-context-free restrictions. For example, it is presumably meaningless to use a decimal literal denoting a conventional natural number as the *head* of an application (aka function call).

There will be only a few pre-defined constants; possibly only for generic equality, the type of truth values with its well-known inhabitants and the standard logical operators, and the most common mathematical functions. Others must be imported through libraries or be defined in a local context.

The style of a literal could be, e.g., `"UFbase10"` with *denotation* `"912559"`, or `"UFbase16"` with denotation `"DECAF"`. The format allows for tool-specific styles.

The *aggregator* of an aggregate could be, e.g., `"UFset"`, `"UFvector"`, or `"UFmatrix(rowwise)"`. In the latter case the *items* should be vectors representing the successive rows. The *mtype* may be needed in case there are zero items, in which case the type of the aggregate cannot be derived from the type of the items.

An abstraction corresponds to a lambda form; it introduces bound variables (called '*parameters*') having the *body* of the abstraction as their scope. This is not the only way bound variables with scopes can be introduced. Another way is through definitions and declarations. These can be given through an explicitly supplied *context*, or be implied. If used in a high-school algebra setting, a variable whose *name* is something like 'plus-operator' will likely be bound to a global implicit definition equating it with the usual concept of the addition of real numbers.

In UniFormal terms the bound-variable aspect of a quantification is not tied to the *quantor*; instead, to achieve the effect, a separate abstraction is used for the *body*. So, e.g., $\forall x.P(x)$ would be represented, as it were, by a term $\forall(\lambda x.P(x))$. The terms 'quantification' and '*quantor*' are perhaps not the best; also, e.g., the least-fixpoint operator μ is a possible '*quantor*'.

4 Contexts

type *UFcontextLayer* = [*UFdeclination*]
data *UFdeclination*
  = *UFdefinition* { *dx* :: *X*, *definiendum* :: *UFconterm*,
    definiens :: *UFterm* }
  | *UFcharacterization* { *dx* :: *X*, *definiendum* :: *UFconterm*,
    character :: *UFstatement*, *mequivalence* :: *Maybe UFequivalence* }
  | *UFdeclaration* { *dx* :: *X*, *declarandum* :: *UFvarterm*,
    dependsOn :: [*UFvarterm*], *mrestriction* :: *Maybe UFstatement* }
  | *UFassumption* { *dx* :: *X*, *axiom* :: *UFstatement* }

```
    | $\mathit{UFresource}\ \{\mathit{dx} :: X, \mathit{uri} :: \mathit{URI}\}$
data $\mathit{UFcontext}$
    = $\mathit{UFemptyContext}\ \{\mathit{cx} :: X\}$
    | $\mathit{UFcontextedContext}\ \{\mathit{cx} :: X, \mathit{clayer} :: \mathit{UFcontextLayer}\}$


type $\mathit{UFstatement} = \mathit{UFterm}$   -- denoting a truth value
type $\mathit{UFequivalence} = \mathit{UFterm}$   -- denoting an equivalence relation
type $\mathit{URI} = \mathit{String}$   -- denoting a uniform resource identifier
```

A definition like $f(x) = g(x, h(x))$ is represented as $f = \lambda\, x.\ g(x, h(x))$. If the tool allows several presentations for such a definition, as Haskell does, and a specific one is preferred, the preferred variant should be indicated through an annotation.

A declaration is used for what in vernacular mathematical language would be expressed, for example, as "Let d be a divisor of n". The *mrestriction* is then the statement $d \mid n$. If omitted, **true** is assumed, i.e., no restriction.

In a characterization, the *definiendum* is defined to be the unique (up to equivalence) value that satisfies the statement. If there is no unique (up to equivalence) solution, this is ill defined. An omitted *mequivalence* (which must, semantically, denote an equivalence relation of a correct type) corresponds to the equality relation.

There is quite some redundancy here. For example, a definition $x = 3$ should be equivalent to a declaration of x together with an assumed axiom $x = 3$. The underlying idea is to cater for tools that can handle direct definitions, but not axiomatic definitions. Also, an axiom is like a proof rule with zero antecedents; some provers can handle user-supplied axioms, but not general user-supplied proof rules.

5 Theorems and proofs

```
data UFproofRule = UFproofRule {prrx :: X, proofRule :: UFterm}
    -- with restrictions as noted below
```

In the style of HOL, proof rules are denoted as abstractions. The first restriction on a *term* used as *proofRule* is that the constructor must be *UFabstraction*. The second restriction is that the body of that abstraction must either be again a *proofRule*, or else a *statement*, and so on. Repeatedly extracting the body must terminate by finding a *statement* at the end of the chain.

```
data UFproof
  = UFproof {prfx :: X, proof :: UFterm}   -- with restrictions as noted below
```

A proof is then the repeated application to suitable parameters of a proof rule, leading to a statement, which is then considered proved. So a *term* used as a proof must be an *application* whose head is either a proof rule, or again an *application* whose head is a proof rule, and so on, until a statement is reached.

Remark. We want to be able to name proof rules and invoke them by name.

```
data UFtheorem = UFtheorem {thmx :: X, thmContext :: Maybe UFcontext,
  dictum :: UFstatement, justification :: UFjustification}
data UFjustification
  = UFinternalProof {jx :: X, iproof :: UFproof}
  | UFexternalProof {jx :: X, xproof :: String}
```

Remark. We also want to be able to name theorems and invoke them by name. Note that a theorem in a sense defines a proof rule.