Mescal Architectural Sketches

Lambert Meertens*
Department of Algorithmics and Architecture, CWI, Amsterdam, and
Department of Computing Science, Utrecht University, The Netherlands
www.cwi.nl/~lambert

Version of June 22, 1998

0 What this document is about

Mescal is a (not yet existing) system for mechanical support in constructing and exploring formal theories, with an emphasis to calculational theories for software development. The envisioned functionality of *Mescal* is described in a companion paper. This document is an attempt to sketch a possible architecture for *Mescal*.

The envisioned architecture for *Mescal* is that of a *kernel system* "empowered" by libraries.

Shood go sumwear:

- Forms are generated by certain "formation rules" of a (multi-sorted) algebra, giving rise to tree-like structures. A form belongs to a formalism that is richer than just the syntactic aspects. In general, forms appear in some *context*. The context may impose certain requirements on the forms that may appear there, but the meaning of a form is *independent of the context in which it appears*. Forms may carry, in addition, "certificates" issued by some formalism.
- "Text" is just another formalism not having certified forms.

*Work performed while visiting Kestrel Institute, Palo Alto.

- For a Haskell program, one possible kind of certificate is IsTypeCorrect, another kind being the parametrized certficate ImplementsSpecification(S), where S is a form in some specification formalism. In either case, the certification would presumably depend on assumptions concerning for example definitions from a Haskell prelude applied in the program.

- While Mescal is *about* formalisms, to describe the Mescal system we also need formalisms. To keep the two apart, we use the prefix "M-" for the latter. It can be read as meaning "*meta-*"; in any case, it is used for things that pertain to the Mescal *architecture*, as opposed to the kind of things (text, formulas, proofs, etc.) the user manipulates through Mescal.

- The Mescal architecture, then, comprises (modules for handling) an M-formalism for describing formalisms. A formalism would at least comprise:
  - a formalism "identifier";
  - the declaration of possible sub-formalism relationships;
  - multi-sorted formation rules (possibly mixing in forms from other formalisms!);
  - presentation rules;
  - certification rules, to be distinguished in basic rules (e.g., definitions) and derived rules (e.g., theorems).

- An M-database for storing the formalisms persistently.

- Pro Memoria: M-certified derived rules (cf. Automath).

- Examine the merits of "bootstrapping" using the M-formalism (partially) described in itself, versus making the PaT-reflexion found a general and canonical feature.

1 Forms

As a first approach to making forms more concrete, we give a Haskell data definition, extended with a data type {_} for sets, analogous to [_] for lists.

$$\begin{array}{lll}
\textbf{data}\ \mathit{Form} & = & \mathit{Fnode}\ \ \mathit{Head}\ \ [\mathit{Form}] \\
\textbf{data}\ \mathit{Head} & = & \mathit{Hnode}\ \ \mathit{Symbol}\ \ \{\mathit{Certificate}\}\ \ \{\mathit{Attribute}\} \\
\textbf{data}\ \mathit{Symbol} & = & \mathit{Snode}\ \ \mathit{FormalismId}\ \ \mathit{SortId}\ \ \mathit{SymbolId}
\end{array}$$

$$\begin{array}{lll}
\textbf{type}\ \mathit{Certificate} & = & \mathit{Form} \\
\textbf{type}\ \mathit{Attribute} & = & (\mathit{AttrId},\ \mathit{AttrVal}) \\
\textbf{type}\ \mathit{AttrVal} & = & \mathit{Form}
\end{array}$$

(There is no deep reason why *Head* is presented as a separate datatype. The definition might be unfolded inside that for *Form*, but this separation is felt to serve clarity. The same holds for *Symbol*. The positioning of the attributes with the head is also somewhat arbitrary. The attributes of a form are basically a mapping of a finite set of attribute names to attribute values, We refrain from giving a more formal description. In a first approximation attributes may be ignored, since they do not influence the meaning of forms. They may, however, influence their presentation.)

A form has one *head* and zero or more *children*, which are forms again. The order of the children is significant.

Some possible symbols (values of type *SymbolId*) are subsection, figure, footnote, binarySum, dummyVariable, forAll, calculation, where the names are meant to suggest a role. However, they have no intrinsic meaning; any semantics exists by virtue of their sort, which identifies a formalism.

Sorts are like syntactic categories of a formalism. Some possible sort identifiers are Text, Formula, Proposition, Equation, HaskellType, JavaClassDefinition.

Different formalism may have sorts that happen to have the same sort identifier. For example, Matrix might be a sort of the Text formalism, as well as a sort of a MatrixAlgebra formalism with an entirely different meaning. Sorts may be subsorts of other sorts (including sorts of other formalisms!). For example, several sorts of a Picture formalism may be subsorts of the sort picture of a Text formalism.

Possible formalism identifiers are Text, PropCalc, SpecWare, Java.

2 Certificates

In general, when a form carries a certificate, that certificate will have the meaning of a (certified) statement that the form is "healthy" in some respect. Some possible forms of healthiness are

- well-formedness according to a collection of syntactic formation rules;
- typability according to some typing discipline;
- correctness with respect to a given specification;
- having a proof according to some collection of proof rules.

The certificate identifies the issuing formalism, the "assumptions" relative to which the healthiness is asserted, and usually (but not necessarily) a "witness" of the healthiness (for example the proof whose existence was asserted, or at least sufficient information to reconstruct it).

3 Contexts

A context can be viewed as a collection of certified forms that are "in force" in a certain scope, usually a form with its descendant forms. Scopes may, however, have "holes" (excluding some subforms). Contexts play a role in the certification process.

4 Certification rules

A certification rule is rather like an inference rule in logic. An inference rule can be presented as a schema consisting of a number of "antecedent" terms and a "consequent" term containing "placeholders". If there is a consistent matching of previously derived terms to the antecedent terms of the schema, meaning a substitution for the placeholders that turns the antecedents into terms already proved, then performing the same substitution on the consequent term gives a newly derived term.

A major difference, though, is that Mescal certification rules can perform rather arbitrary computations to derive certified forms. As envisaged, the certification rules are embodied in *edit steps*. An edit step (think of it for the time being as an edit *command*) takes zero or more forms as parameters, in general with different roles, and then performs some computation resulting in the creation of a new form, which may for instance be inserted in or pasted over an existing form. If the parameter forms are certified, an edit step may use the certificates, as well as the forms themselves, in determining one or more certificates for the new form. In addition, the forms from the context may be used for that purpose. Some edit steps will never look at or issue certificates; that all depends on hoe the step has been defined.

Not only does a certifying edit step create a new form, it also creates a small network linking the forms used for the certification attempt – it need not succeed – to the new form. Normally that network would not be visibly presented on the screen, but exist behind the scene. On any change that might bear on the certification process, the process is repeated. The form may thereby gain certificates it did not have, while losing other certificates.

Comparison with *Propositions as Types* The question could be asked why we use the certificate approach, instead of the propositions-as-types paradigm. From a logical point of view the two are rather similar (if we are prepared to model consulting an external engine as the consultation of an oracle, or equivalently as the discovery of yet another axiom in the proof system), but pragmatically they are quite different, In the propositions-as-types paradigm we find the following analogy:

$$\begin{array}{ccc} \text{proof} & : & \text{proposition} \\ & = & \\ \text{inhabitant} & : & \text{type} \end{array}$$

Extending it to certificates and forms, we have:

$$\begin{array}{ccc} \text{proof} & : & \text{proposition} \\ & = & \\ \text{inhabitant} & : & \text{type} \\ & = & \\ \text{certificate} & : & \text{form} \end{array}$$

Since forms can have several certificates, maintaining this analogy means forms can have several propositional readings, depending on the viewpoint chosen. Of course, by moving some things around we can make the propositional reading unique. For example,

$$\textbf{has-type}\ \tau\ \textbf{since}\ J\ :\ F$$

(in which J is the justification of the typing judgment) can be changed into

$$J \ : \ F \ \mathbf{has\text{-}type} \ \tau$$

However, this judgment now pertains to a *different* form, F **has-type** τ. Although such forms *may* exist, it seems pointless to *require* their existence while prying apart the certificates from the forms to which they pertain.

Using *Propositions as Types*, the situation becomes even more precarious when proofs are the very forms a user is editing. Under the certificate approach, certificates and forms live in the same "universe", and the coexistence of

$$\begin{array}{l} \mathbf{is\text{-}a\text{-}correct\text{-}proof} \ : \ P \\ \mathbf{is\text{-}a\text{-}valid\text{-}statement\text{-}since} \ P \ : \ S \end{array}$$

is not problematic; on the contrary, the first certification would be *required* for the second to succeed.

5 Presentation

6 Related work

- Foobaz

7 Where should this go?

- Foobaz

References

- Wile's and Gibbons' work on tree traversal
- Cordy & Carmichael: TXL

TODO

`[[TODO]]`