Object Structure

Lambert Meertens

Department of Algorithmics and Architecture, CWI, Amsterdam, and

Department of Computing Science, Utrecht University, The Netherlands

lambert@cwi.nl

Version of June 23, 1998

What this is about

The following is a sketch of a formalism for object structure, eventually to be cast in API form, that should enable different structure-handling components to cooperate. It should be fairly easy to give a concrete XML representation for the structures sketched abstractly here.

Issues not addressed here are object creation and object types.

1 Terms

Terms may need a *context* to interpret them. If we abstract from the interpretation we get "raw" terms.

We assume that a universe of *constructors* is given. Constructors are like tags: they are inherently meaningless. Some constructors, however, are special: the formalism assigns a meaning to them. They are represented here by tags prefixed with "`!`".

Using Gofer notation, terms are defined as an adt:

$$\textbf{data}\ \ Term\ =\ \mathsf{Struct}\ Constructor\ [Attribute]\ [Term]$$

If we know we are dealing with a term, the initial Gofer constructor "Struct" carries no information and so is redundant. We shall from now on omit it. Some possible terms are:

$$\texttt{BIBLIOGRAPHY}\ [\texttt{STYLE=alpha}]\ [Bibitem \cdots\ ]$$

and

$$\texttt{PRIMVAL}\ [\texttt{TYPE=Float},\ \texttt{VALUE=3.14}]$$

So terms are trees of constructors-cum-attributes.

Basic procedures — but the details are negotiable — are:

$\mathsf{getTermConstructor}\,(term)$

$\mathsf{getTermAttrNames}\,(term)$

$\mathsf{getTermValOfAttr}\,(term,\ attrname)$

$\mathsf{getTermNmbOfChildren}\,(term)$

$\mathsf{getTermChild}\,(term,\ i)$

2 Term grammars

The language of *all* terms is produced by the grammar

$$Term\ ::=\ Constructor\ [Attribute \cdots\ ]\ [Term \cdots\ ]$$

in which "$[X \cdots\ ]$" stands for a list of zero or more productions of X. This language takes no account of the context, and includes presumably meaningless terms like

`BILBIOGRAPHY` [`TYPE=Float`]

— which, however, may be quite meaningful in a sufficiently weird context. Depending on the context in which terms are interpreted, it may be possible to give a grammar for the meaningful terms, something like

Picture	::=	`TEXTPIC`	[ ]	[*Text*]
	|	*Enclosure*		
	|	*ComposedPicture*		
Enclosure	::=	`BOX`	[*Linestyle*]	[*Picture*]
	|	`OVOID`	[*Linestyle*]	[*Picture*]
ComposedPicture	::=	`ROW`	[*Direction*]	[*Picture* ···]
CompanyLogo	::=	`OVOID`	[`STYLE=Flashy`]	
				[`TEXTPIC` [ ] [*CompanyName*]]
CompanyName	::=	`PRIMVAL`	[`TYPE=String`, `VALUE="Coca"`]	

A nonterminal of a tree grammar can be viewed as a *sub-type* of the type of all terms. *CompanyLogo* above is a sub-type of *Picture*.

In a traditional parse tree, the node labels are nonterminals, while in an abstract syntax tree they are more like constructors. We combine both in the parse tree of a well-formed term. For example, here is the parse tree (which happens to be linear) for the single production of *CompanyLogo*:

$$\begin{array}{r} \mathit{CompanyLogo} \to \texttt{OVOID}\ [\texttt{STYLE=Flashy}] \\ | \qquad\qquad \\ \texttt{TEXTPIC}\ [\,] \qquad\qquad \\ | \qquad\qquad \\ \mathit{CompanyName} \to \texttt{PRIMVAL}\ [\texttt{TYPE=String}, \texttt{VALUE="Coca"}] \end{array}$$

And here is the parse tree for the same term as produced by *Picture*:

$$\begin{array}{r} \mathit{Picture} \to \mathit{Enclosure} \to \texttt{OVOID}\ [\texttt{STYLE=Flashy}] \\ | \qquad\qquad \\ \mathit{Picture} \to \texttt{TEXTPIC}\ [\,] \qquad\qquad \\ | \qquad\qquad \\ \mathit{Text} \to \texttt{PRIMVAL}\ [\texttt{TYPE=String}, \texttt{VALUE="Coca"}] \end{array}$$

The idea, however, is that in a given context the meaning of a well-formed term only depends on the term *per se* and not on its parse tree. The function of the grammars, if any, is primarily to describe restrictions that ensure meaningfulness.

3 Attributes

Attributes are basically (*Name*, *Value*) pairs. Names are again like tags. As for constructors, special attribute names are represented by tags prefixed with "`!`".

We can't say too much about the universe of attribute values, except that (0) for the purpose of interchange some lexical format for basic values must be agreed upon, and (1) terms should be a subset of that universe.

Which attributes are meaningful — and perhaps mandatory — for a given constructor typically depends on the context. However, here is what I think is an important rule:

> *In addition to attributes that are meaningful for some application in some context, terms may carry* any *further attributes. They are simply disregarded if they cannot be interpreted.*

Perhaps some other application with some other context can make good use of these attributes. Or perhaps they are just passed on in some attribution schema to nodes for which they are meaningful.

4 Locators

An object has an "identity" and possibly a value. The identity is immutable, whilst the value may change dynamically. Conceptually, we may think of this as follows: there is *one* universal (world-wide) object "table" which is indexed by object IDs. That table is the only mutable entity. Its physical embodiment may be distributed.

There is some procedure which, given an object ID, tries to return its value. The procedure may fail because the object has no value, or is hidden (secret), or because of communication problems.

However, we never see object IDs directly. They are not accessible values. Instead, we have *locators*. A locator is a value that purports to determine an object ID. There is some procedure which, given a locator, tries to return the value of the corresponding object. There is no general guarantee, even in the case of success, that applying the procedure twice will give the same object twice.

Terms are suitable for modelling locators:

$$\mathit{Locator} \quad ::= \quad \texttt{!LOCATOR}\,[\mathit{Protocol} = \mathit{ProtoLoc} \cdots\ ]$$

There should be at least one attribute, but alternate ways to locate the object may be additionally supplied.

Basic procedures are:

$$\mathsf{getObjVal}\,(\mathit{loc})$$

$$\mathsf{setObjVal}\,(\mathit{loc},\ \mathit{val})$$

To be elaborated: locator + selection path is also locator.

5 Sub-objects

We can use terms for the object values, but we need something extra to model the possibility of *sub-objects*.

When object $\mathcal{S}$ is a sub-object of object $\mathcal{O}$ any change to the value of $\mathcal{S}$ entails an immediate corresponding change to the value of object $\mathcal{O}$. The reverse is not necessarily true: an assigment to $\mathcal{O}$ can cause $\mathcal{S}$ to become "orphaned".

The extra now is that terms have an optional attribute

$$\texttt{!OBJLOC} = \mathit{Locator}$$

If present, this signifies that this term is (or, rather, at some time was) the value of an object (then) locatable through the locator given. If sub-terms of the value of an object have this attribute set, they correspond to sub-objects.

The advantage of this "trick" is that it coalesces terms and objects in a rather simple way.

Typically, in a call getObjVal (loc), the term returned will have the `!OBJLOC` attribute set. The locator that is the value of that attribute may be completely different from loc. It may be a more efficient locator for the same object. Another possibility are "stationary locators": each access creates a new object initialised to some default value.

In a call setObjVal (loc, val), any `!OBJLOC` attribute of the term val should be discarded.

6 Lazy terms

A lazy term is a term temporarily represented by a locator. Possible syntax is:

$$LazyTerm \quad ::= \quad \texttt{!DEFERRED}\,[\texttt{!REFERTO} = Locator]$$

In shipping a term, the sender need not transmit the whole term necessarily at once, but may represent sub-terms lazily. The idea is that we normally never see lazy terms, but that they are transparently handled by the getTerm$X$ calls, with some local caching so that when we refer a second time to the locator the response is fast.

However, one layer below the API they may be visible as genuine terms with constructor `!DEFERRED`.