ReMoDeL

Functional Programming

Model Specification

Version: 0.5

Date: 02 November 2011

Anthony J H Simons
Department of Computer Science
University of Sheffield

1 Introduction

This document describes the specification for ReMoDeL FUN, a dialect of XML [1] used to encode the Functional Programming Model. This is a foundational model, used as a core subset by other ReMoDeL XML dialects.

1.1 Model Scope

The Functional Programming Model is intended to support a language of pure computation, consisting of types, values, variables, functions and expressions. It should be possible to translate from ReMoDeL FUN into Functional languages, such as Haskell, ML, Miranda, Clean or Hope; and also into the Functional subset of other languages, such as Scheme, Common Lisp and C. However, it is not our intention that FUN should support all the diverse and powerful constructions present in each of these languages individually. Instead, FUN should be a very simple model.

One of the motivations for developing ReMoDeL FUN as a core model was to avoid re-inventing similar constructions when devising the expression-handling subsets of each new ReMoDeL XML dialect. The intention is to reuse, extend and adapt the constructions from ReMoDeL FUN in other ReMoDeL XML dialects. This also promotes a desirable kind of homogeneity across the different models.

1.2 Model Semantics

Since the Functional Programming Model is intended also to form the expression handling subset of other imperative languages, choices were made to harmonise the model semantics with those of imperative languages. The two major issues were the evaluation semantics, and the calling semantics.

Functional languages may support eager evaluation (call by value), in which all sub-expressions are simplified before substitution into function arguments, or lazy evaluation (call by need), in which sub-expressions are substituted directly and are only evaluated on demand. The latter is standard in modern functional languages, such as Haskell, Miranda and Clean. However, older functional languages and imperative languages do not support lazy evaluation, except in the *if*-construction (and in the non-strict evaluation of Boolean compounds). For this reason, FUN adopts eager evaluation semantics.

Functional languages may support automatic deconstruction of arguments (pattern matching), or rely on explicit navigation (access functions) to the parts of a structured variable. The former is standard in modern functional languages, but is absent from older languages such as Scheme and Common Lisp; and was never a feature of imperative languages. Again, for maximum applicability, FUN adopts the simpler strategy of using access functions to deconstruct arguments.

Apart from this, FUN is a simple language of expressions. Every expression denotes a value (literally, or by reduction). Every function returns a value, which is the result of simplifying its body expression, after values have been substituted for the function's arguments. There is no need for an explicit *return* operator in FUN (although this may be necessary in imperative languages that extend FUN).

In keeping with Functional programming, computation is not expressed sequentially as a series of steps, but through the implicit ordering defined by the nesting of function applications. This is carried through to the declaration of local variables (constants) and functions within any given scope. Variable names are logically bound in parallel, upon entry to that scope, and cannot rely on each other for any assumed sequential order of initialisation.

1.3 Common Metamodel

The XML elements and attributes defined in this syntax model correspond to concepts, attributes and relationships in the ReMoDeL Metamodel. The deliberate consequence of this is that XML elements may be mapped directly onto metamodel classes. Elements are not defined in isolation, but may be organised in a conceptual hierarchy according to their similarities and differences. This is intended to support parsers that build syntax trees directly from instances of the metamodel classes, as well as parsers that use conventional XML trees.

A model will be constructed from the terminal elements in the conceptual hierarchy. A consequence of this is that certain XML element names will be reserved to denote abstract concepts in the metamodel, which are never actually present in any instance of the model. These abstract elements are nonetheless defined as part of the model, since they may correspond to strongly typed nodes in syntax trees derived directly from the metamodel.

The intention is for all ReMoDeL XML dialects to be mapped onto a common metamodel. The terminal elements used across different languages, though they may be different, will nonetheless share certain similarities, expressed through their relationships with common abstract elements.

1.4 XML Conventions

ReMoDeL FUN adopts all the W3C conventions for XML [1]. Identifier symbols must observe the rules of Unicode identifiers, attribute values must be enclosed in quotes and special characters must be escaped as entity references. Apart from this, ReMoDeL imposes a certain "house style" on identifiers:

- The names of all XML elements are presented in *CapitalCase*, similar to type names in the Java programming language.

- The names of all XML attributes are presented in *camelCase*, similar to variable names in the Java programming language.

The use of hyphens and underscores as part of identifier names goes against the house style and is strongly discouraged. The use of digits as part of the body of an identifier is legal, but also generally discouraged, unless the application clearly demands this.

2 Types

The kinds of *Types* used in ReMoDeL FUN subdivide into the *Basic* types, the *Symbolic* types, the *Record* types, the *Structure* types and the *Generic* parameters. These all descend from the metaclass *Type*, as illustrated in the fragment of the ReMoDeL Metamodel shown in figure 1.

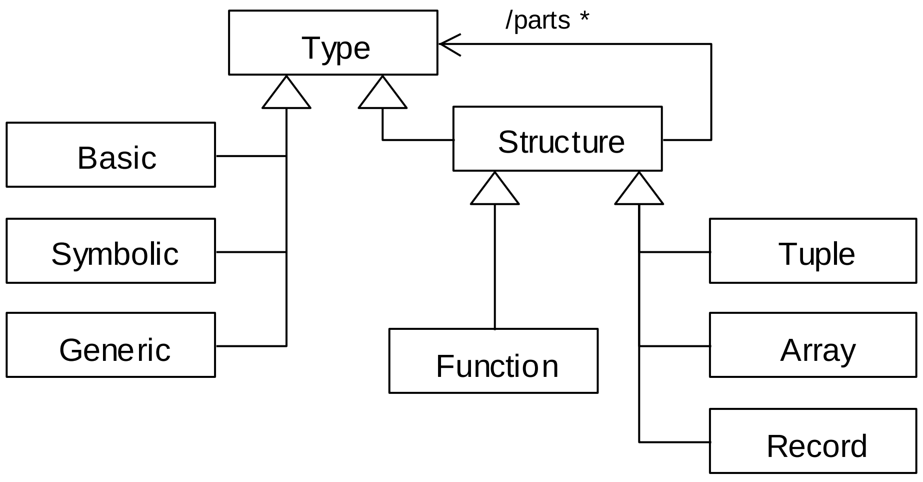

Figure 1: Types in the ReMoDeL Metamodel

All types are defined as instances of one of these metaclasses, which also specify the type's implementation. For example, an *Integer* type is defined as an instance of the *Basic* metaclass; the symbolic *Boolean* type is defined as an instance of the *Symbolic* metaclass; a *Person* record type is defined as an instance of the *Record* metaclass; and the *String* type is defined as an instance of the *Array* metaclass. Instances of the *Generic* metaclass are type parameters, placeholders for types. *Function* defines an executable function, which has an associated type.

ReMoDeL assumes standard default specifications for the types: *Boolean*, *Character*, *Integer*, *Natural*, *Decimal* and *Void*, which may be taken for granted if no alternative definition is provided. *Boolean* occupies a byte, *Character* supports UTF-8, *Integer* assumes a signed 32-bit range, *Natural* assumes an unsigned 32-bit range, *Decimal* assumes double-precision floating point and *Void* is the most specific type of the empty value *null*. These types are declared in the *Core* package for the ReMoDeL FUN dialect, which has other basic types, such as *Byte*, *Short* and *Long*, which may be included in models as desired.

2.1 Basic Type

The *Basic* element is used to declare a primitive value-type with a fixed range of values. Basic types are often built-in types of the target language. The syntax supports declaring types with different ranges (and therefore different precision), which must be preserved by the implementation of the target language.

The grammar for the *Basic* element is:

```
<!ELEMENT Basic (Literal, Literal)>
<!ATTLIST Basic name NMTOKEN #REQUIRED>
```

The attributes of the *Basic* element are:

- *name* – the name of the type (required).

The name must be given in *CapitalCase* (see section 1.4).

The children of the *Basic* element are exactly one each of:

- *Literal* – the lower limit of the type (optional);
- *Literal* – the upper limit of the type (optional);

Examples of the *Basic* element include:

```
<Basic name="Character"/>

<Basic name="Integer">
  <Literal value="-2147483648" limit="low" type="Integer"/>
  <Literal value="2147483647" limit="high" type="Integer"/>
</Basic>
```

In the first example, the range of the declared *Character* type is assumed to be the default range supporting UTF-8. The second example explicitly declares the *Integer* type to have a 32-bit range.

2.2 Symbolic Type

The *Symbolic* element is used to declare a symbolic value-type with enumerable values. These are symbolic constants, which are declared by finite enumeration. Symbolic types are often user-defined, but may be built-in types.

The grammar for the *Symbolic* element is:

```
<!ELEMENT Symbolic (Literal+)>
<!ATTLIST Symbolic name NMTOKEN #REQUIRED>
```

The attributes of the *Symbolic* element are:

- *name* – the name of the type (required).

The name must be given in *CapitalCase* (see section 1.4).

The children of the *Symbolic* element are one or more ordered occurrences of:

- *Literal* – a symbolic constant of the type (one-to-many).

Examples of the *Symbolic* element include:

```
<Symbolic name="Boolean">
  <Literal value="false" type="Boolean"/>
  <Literal value="true" type="Boolean"/>
</Symbolic>
```

```
<Symbolic name="Status">
  <Literal value="closed" type="Status"/>
  <Literal value="open" type="Status"/>
  <Literal value="frozen" type="Status"/>
</Symbolic>
```

Model transformation tools must be able to determine whether any given symbolic type is primitive (built-in), or requires special declaration in each target language.

2.3 Generic Type

The *Generic* element is used to declare generic parameters. These are placeholders for actual types. Generic parameters are included as the first children inside any generic type or function definition.

The Generic element has more attribute options in OOP. The grammar for the *Generic* element in FUN is:

```
<!ELEMENT Generic EMPTY>
<!ATTLIST Generic name NMTOKEN #REQUIRED>
<!ATTLIST Generic satisfy NMTOKEN #IMPLIED>
```

The attributes of the *Generic* element are:

- *name* – the name of the parameter (required);
- *satisfy* – an upper bound interface name (optional).

Names must be given in *CapitalCase* (see section 1.4). By default, any type may be substituted later for a generic parameter. If an upper bound interface is supplied, only types that satisfy this interface may be substituted. ReMoDeL FUN assumes that the interfaces *Equal[T]* and *Compare[T]* are predefined.

The *Generic* element has no children.

Examples of the *Generic* element include:

```
<Generic name="Element"/>

<Generic name="Type" satisfy="Compare[Type]"/>
```

The first example is a parameter for the element type of a general list or tree. The second example is a parameter for the element type of a sorted list or tree, in which values of the supplied type must be compared with each other. Only predefined interfaces exist in FUN – interfaces are more generally available in OOP.

2.4 Record Type

The *Record* element is used to declare record types. These are named structures, consisting of named fields, each of a different type. Record types may declare generic type parameters for the types of one or more of their fields. Records have reference semantics, when passed as values.

The grammar for the *Record* element is:

```
<!ELEMENT Record (Generic*, Field+)>
<!ATTLIST Record name NMTOKEN #REQUIRED>
```

The attributes of the *Record* element are:

- *name* – the name of the type (required).

The name must be given in *CapitalCase* (see section 1.4).

The children of the *Record* element are:

- *Generic* – generic type parameter (optional, zero-to-many);
- *Field* – named field member (required, one-to-many).

Examples of the *Record* element include:

```
<Record name="Person">
  <Field name="forename" type="String"/>
  <Field name="surname" type="String"/>
  <Field name="gender" type="Character"/>
  <Field name="age" type="Natural"/>
</Record>

<Record name="List">
  <Generic name="Element"/>
  <Field name="head" type="Element"/>
  <Field name="tail" type="List[Element]"/>
</Record>
```

The *Record* metaclass is used to create structures with named fields. The first example is of a user-defined *Person* record, with *Fields* of the given names and types. The second example is of a generic *List* record type, the building block for linked lists in FUN. The tail of this record is a reference to another structure of the same type. The *null* value marks the end of the list.

2.5 Tuple Type

The *Tuple* element may be used to declare tuple types. These are optionally named structures, consisting of indexed slots, each of a different type. Tuple types are more usually inferred dynamically from the context. Tuple types may declare generic parameters for one or more of their slots. Tuples have reference semantics, when passed as values.

The grammar for the *Tuple* element is:

```
<!ELEMENT Tuple (Generic*, Slot+)>
<!ATTLIST Tuple name NMTOKEN #IMPLIED>
```

The attributes of the *Tuple* element are:

- *name* – the name of the type (optional).

The name must be given in *CapitalCase* (see section 1.4). While *Tuples* can be named, it is more usual to infer a name for a *Tuple* from its type contents.

The children of the *Tuple* element are:

- *Generic* – generic type parameter (optional, zero-to-many);
- *Slot* – indexed structure slot (required, one-to-many).

Examples of the *Tuple* element include:

```
<Tuple>
  <Slot type="String"/>
  <Slot type="Natural"/>
</Tuple>

<Tuple name="Pair">
  <Generic name="First"/>
  <Generic name="Second"/>
  <Slot type="First"/>
  <Slot type="Second"/>
</Tuple>
```

The first example is a simple entry mapping from a *String* to a *Natural* number. The second is the generalisation of this to a generic *Pair* type. The first example is not explicitly named, but its type name is inferred as *Tuple [String, Natural]* from the context. The named *Pair* type has the equivalent inferred name: *Tuple [?X, ?Y]*, where *?X, ?Y* denote missing type values to be supplied later.

2.6 Array Type

The *Array* element may be used to declare array types. These are optionally named structures, consisting of indexed slots, each of the same type. Array types are more usually inferred dynamically from the context. Array types may declare a generic parameter for all of their slots. Arrays have reference semantics, when passed as values. Matrix types are constructed from nested arrays.

The grammar for the *Array* element is:

```
<!ELEMENT Array (Generic?, (Slot | Array))>
<!ATTLIST Array name NMTOKEN #IMPLIED>
<!ATTLIST Array size CDATA #IMPLIED>
```

The attributes of the *Array* element are:

- *name* – the name of the type (optional);
- *size* – the length of the array (optional).

The *name* must be given in *CapitalCase* (see section 1.4). While *Arrays* can be named, it is more usual to infer a structured name for an *Array* from its dimension and element type. The *size* must be given as a natural number (unsigned).

The children of the *Array* element are:

- *Generic* – generic type parameter (optional);

- *Slot* – a single exemplar slot (optional, alternates with *Array*);
- *Array* – a nested array type (optional, alternates with *Slot*).

Examples of the *Array* element include:

```
<Array name="String">
  <Slot type="Character"/>
</Array>

<Array size="10">
  <Slot type="Integer"/>
</Array>

<Array name="Table" size="10">
  <Generic type="Element"/>
  <Slot type="Element"/>
</Array>

<Array name="Matrix" size="10">
  <Array size="20">
    <Slot type="Decimal"/>
  </Array>
</Array>
```

The first example illustrates how the *String* type is declared in the *Core* library. The length of the *String* is not predefined, but must be inferred from the context of usage. The second example is of an unnamed array type containing ten *Integer* slots. Its type name may be inferred as: *Array [10, Integer]* from the context. The third example is of a generic *Table* containing ten elements of any homogeneous type. The fourth example is of a *Matrix* with dimensions 10 x 20, containing *Decimal* values.

Explicitly named types are equivalent to their corresponding inferred versions. The *String* type is equivalent to: *Array [?X, Character]*, and the *Table* type is equivalent to *Array [10, ?Y]*, where *?X* denotes the missing dimension and *?Y* the missing type, to be supplied later. The equivalent structural type of the *Matrix* is derived in the same way as: *Array [10, Array [20, Decimal]]*. Inferred array types always include the dimension as the first component.

2.7 Function Type

The *Function* element may be used to declare named functions, each with an associated function type signature. The types of functions are usually inferred from the context. As a kind of *Structure*, a *Function* defines named arguments, each of a possibly different type, and maps to a result type. Functions may declare generic parameters for one or more of their argument or result types. A *Function* is also a kind of named *Property*, and a kind of *Expression*. Refer to *Function Declaration* for the full specification of functions (see section 4.4).

Examples of the *Function* element used to specify function signatures include:

```
<Function name="square" type="Integer">
  <Variable name="value" type="Integer"/>
</Function>
```

```
  <Function name="head" type="Element">
    <Generic name="Element"/>
    <Variable name="list" type="List[Element]"/>
  </Tuple>
```

The first example is of a simple function computing the square of an integer. The inferred type of this function is: *Function [Integer, Integer]*. The second example is of a generic function from the *Lists* library package that returns the head element of a generic list. The inferred type of this function is: *Function [List [?X], ?X]*, where *?X* denotes the missing type, to be supplied later. Inferred function types always list the argument types in the same order that the arguments were declared, and include the result type as their last component.

3 Expressions

The kinds of *Expression* used in ReMoDeL FUN subdivide into *Literal*, *Identifier* and *Function* expressions, the evaluating *Operator, Access* and *Apply* expressions and the compound expression *Select*. These all descend from the metaclass *Expression*, as illustrated in the fragment of the ReMoDeL Metamodel shown in figure 2.

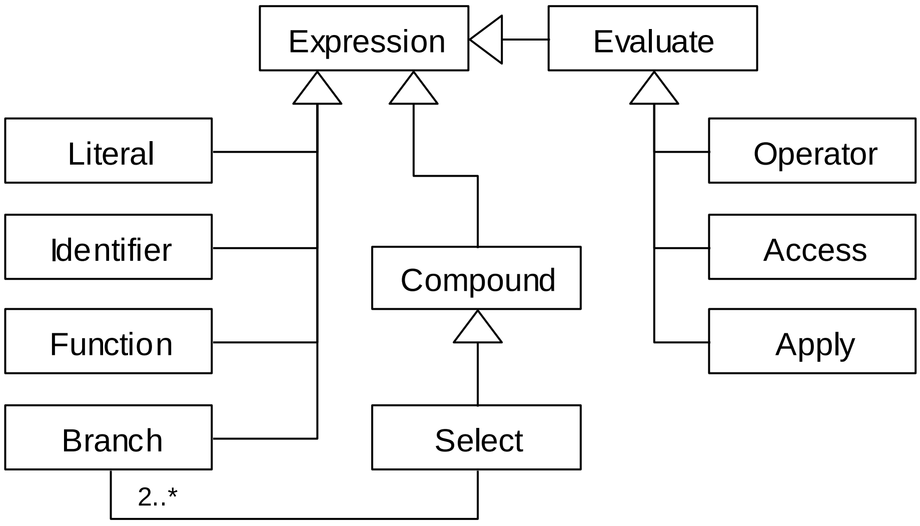

Figure 2: Expressions in the ReMoDeL Metamodel

All expressions are defined as instances of one of these metaclasss. For example, the literal values *3* or *"hello"* are instances of the *Literal* metaclass; the variable name *count* is defined as an instance of the *Identifier* metaclass; a function passed as a value is an instance of the *Function* metaclass; a mathematical operation is an instance of the *Operator* metaclass; a function invocation is an instance of the *Apply* metaclass; and any access into a structured type is defined as an instance of the *Access* metaclass. The *Select* metaclass defines binary and multi-branch selection.

3.1 Literal Expression

The *Literal* element may be used to define a literal expression, or constant. Simple literal values may be declared for any type whose values have a directly printable representation. Literals of structured types may also be declared in FUN by nesting the *Literal* element. Literal elements may state whether they are the low or high limit for their type.

The grammar for the *Literal* element is:

```
<!ELEMENT Literal (Literal | Identifier)*>
<!ATTLIST Literal value CDATA #IMPLIED>
<!ATTLIST Literal limit (low | high) #IMPLIED>
<!ATTLIST Literal type NMTOKEN #REQUIRED>
```

The attributes of the *Literal* element are:

- *value* – the literal value, a constant (optional, alternates with children);
- *limit* – whether this value is the low or high limit (optional);
- *type* – the type of the literal value (required).

Any *value* that has a printed representation is valid, including integral and floating-point numbers, single and multiple character strings or symbols. The *value* is always supplied as text data. The interpretation of the *value* is given by the corresponding *type*, which is the defined name, or inferred name, of any declared ReMoDeL FUN type (see section 2).

The children of the *Literal* element, which alternate with the *value* attribute, are zero or more ordered occurrences of:

- *Literal* – a nested literal value (optional, zero-to-many);
- *Identifier* – a nested bound identifier (optional, zero-to-many).

Nested *Literal* and *Identifier* elements are only used when constructing literal values of structured types. In this case, the *value* attribute is not used in the enclosing *Literal* element. Nested *Identifier* elements are used when constructing the result of a function from values bound in some of the function's arguments.

Simple examples of the *Literal* element include:

```
<Literal value="false" type="Boolean"/>

<Literal value="e" type="Character"/>

<Literal value="-10" type="Integer"/>

<Literal value="42" type="Natural"/>

<Literal value="3.1415926" type="Decimal"/>

<Literal value="null" type="Void"/>
```

These examples show how to express literals of simple types. The value *null* is the only legal value of the *Void* type. Examples of structured *Literal* elements include:

```
<Literal type="Array[3, Integer]">
  <Literal value="56" type="Integer"/>
  <Literal value="-2" type="Integer"/>
  <Literal value="66" type="Integer"/>
</Literal>

<Literal type="Person">
  <Literal value="John" type="String"/>
  <Literal value="Smith" type="String"/>
  <Literal value="M" type="Character"/>
  <Literal value="32" type="Natural"/>
</Literal>
```

These examples illustrate structured literal values of array and record types. The types of the nested literals must correspond to the expected types of the structure's components. Examples of *Literal* elements used as limits include:

```
<Literal value="-2147483648" limit="low" type="Integer"/>

<Literal value="2147483647" limit="high" type="Integer"/>

<Literal value="0" limit="low" type="Natural"/>

<Literal value="4294967295" limit="high" type="Natural"/>
```

The first two examples define the limits of a signed 32-bit Integer type. The second two examples define the limits of an unsigned 32-bit Natural type. Refer to *Basic Types* for more examples (see section 2.1).

3.2 Identifier Expression

The *Identifier* element may be used to define an occurrence of a variable that was previously declared, and which is currently in scope.

The grammar for the *Identifier* element in FUN is:

```
<!ELEMENT Identifier EMPTY)>
<!ATTLIST Identifier name NMTOKEN #REQUIRED>
<!ATTLIST Identifier type NMTOKEN #REQUIRED>
<!ATTLIST Identifier scope (local | global) "local">
```

The attributes of the *Identifier* element are:

- *name* – the name of the identifier (required);
- *type* – the type of the identifier (required);
- *scope* – the scope of the identifier (optional).

The *name* must be supplied in *camelCase*, and the *type* must be a legal ReMoDeL FUN type name (see section 2). The *scope* attribute is used when it is desired to override the default *local* scope of an identifier. In ReMoDeL FUN, the only alternative scope is *global* (other scopes may be defined in different ReMoDeL dialects).

The *Identifier* element has no children.

Examples of the *Identifier* element include:

```
<Identifier name="majority" type="Natural"/>

<Identifier name="version" type="String" scope="global"/>
```

The first example is of an identifier referring by default to a locally scoped variable called *majority* of the type *Natural*. The second example is of an identifier referring to a globally scoped variable called *version* of the type *String*.

3.3 Operator Expression

The *Operator* element may be used to define an operator expression. This is any standard operation using the predefined mathematical, Boolean or comparison operator symbols supported by ReMoDeL FUN. All such symbols have standard ReMoDeL names, which are mapped to the appropriate symbol by model transformation tools.

The grammar for the *Operator* element in FUN is:

```
<!ELEMENT Operator ((Literal | Identifier | Operator |
          Access | Apply | Select),
    (Literal | Identifier | Operator | Access | Apply |
          Select)?)>
<!ATTLIST Operator symbol (not | or | and | implies |
    equals | notEquals | lessThan | moreThan |
    noMoreThan | noLessThan | negate | plus | minus |
    times | divide | modulo) #REQUIRED>
<!ATTLIST Operator type NMTOKEN #REQUIRED>
```

The attributes of the *Operator* element are:

- *symbol* – the standard name of the operator symbol (required);
- *type* – the result type of the operator expression (required).

The *type* must be any legal ReMoDeL FUN type name (see section 2). The predefined legal *symbol* names to be used include the following (where *noMoreThan* denotes less than or equal, and *noLessThan* denotes greater than or equal):

- logical operators: *not*, *or*, *and* and *implies*;
- comparison operators: *equals*, *notEquals*, *lessThan*, *moreThan*, *noMoreThan* and *noLessThan*;
- arithmetic operators: *negate*, *plus*, *minus*, *times*, *divide* and *modulo*.

The logical and comparison operators yield a *Boolean* valued result. The arithmetic operators are polymorphic, returning some *Number* type consistent with the argument types, such as *Integer*, *Natural* or *Decimal*.

The children of the *Operator* element include one or more occurrences of:

- any kind of expression apart from a function (required).

The majority of the predefined operators are binary and expect two operands. The operators *not* and *negate* are unary operators and only expect a single operand.

Examples of the *Operator* element include:

```
<Operator symbol="negate" type="Integer">
  <Literal value="42" type="Integer"/>
</Operator>
```

```
<Operator symbol="plus" type="Integer">
  <Literal value="7" type="Integer"/>
  <Literal value="35" type="Integer"/>
</Operator>

<Operator symbol="lessThan" type="Boolean">
  <Literal value="7" type="Integer"/>
  <Operator symbol="minus" type="Integer">
    <Literal value="12" type="Integer"/>
    <Literal value="6" type="Integer"/>
  </Operator>
</Operator>
```

The first example is of a unary operation, negating the value of the operand 42. The second example is of a simple binary operation, adding two integer operands. The third example is of a boolean comparison of two values, where the second value is a nested operation.

3.4 Access Expression

The *Access* element may be used to navigate to a sub-part of a structured value. It generalises projections from tuples, the indexing of single- and multi-dimensional arrays, and access paths to the named fields of records. *Access* is a basic operation that bypasses any visibility restrictions (see 4.2).

The grammar for the *Access* element is:

```
<!ELEMENT Access ((Literal | Identifier | Access |
          Apply | Select),
    (Literal | Identifier | Operator | Access |
          Apply | Select)+)>
<!ATTLIST Access type NMTOKEN #REQUIRED>
```

The attributes of the *Access* element are:

- *type* – the result type of the access expression (required).

The *type* may be any legal ReMoDeL FUN type name (see section 2). The result type of the access expression must correspond to the type of the value selected by the indexing-expression(s). Likewise, the type of the indexing-expression must be suitable for the kind of structure-expression being navigated.

The children of the *Access* element are two or more expressions of suitable types:

- the structure-expression (required);
- further indexing-expressions (required).

If the structure-expression is a tuple or simple array, the indexing-expression can only yield a natural number index. If the structure-expression is a multi-dimensional array, there may be a sequence of indices, up to the total number of dimensions. If the structure-expression is a simple record, the indexing-expression can only be a label. If the structure is a nested record, there may be a sequence of labels corresponding to an access path into the record structure.

Examples of the *Access* element include:

```
<Access type="String">
  <Identifier name="pair" type="Tuple[Integer, String]"/>
  <Literal value="1" type="Natural"/>
</Access>

<Access type="Integer">
  <Identifier name="vector" type="Array[10, Integer]"/>
  <Literal value="3" type="Natural"/>
</Access>

<Access type="Integer">
  <Identifier name="matrix"
      type="Array[10, Array[20, Integer]]"/>
  <Literal value="3" type="Natural"/>
  <Literal value="2" type="Natural"/>
</Access>

<Access type="Array[20, Integer]">
  <Identifier name="matrix"
      type="Array[10, Array[20, Integer]]"/>
  <Literal value="3" type="Natural"/>
</Access>

<Access type="String">
  <Identifier name="employee" type="Person"/>
  <Literal value="surname" type="Label"/>
</Access>

<Access type="String">
  <Identifier name="company" type="Company"/>
  <Literal value="manager" type="Label"/>
  <Literal value="surname" type="Label"/>
</Access>
```

The first example accesses the *second* projection (at index 1) of a tuple. The first projection would be at index 0. Note how the types must correspond, so it is rare to use anything other than a constant for the index, whereas when using arrays, it is more common to compute the index.

The second example accesses the *fourth* value (at index 3) from an array of integers. The third example shows an access path inside a matrix to return the value of the *fourth* row and *third* column. Note how the indices are applied in the same order that the dimensions are nested. The fourth example shows how fewer indices than the maximum may be applied, returning the *fourth* row of the matrix.

The fifth example accesses the *surname* field of a record. The sixth example applies a sequence of labels, navigating the access path: *company.manager.surname* to yield the name, a *String*. Shorter paths would return a record type. Note that record field names are considered literal values of the predefined *Symbolic* type *Label*.

3.5 Function Expression

The *Function* element may be used to declare an unnamed function-expression, to be passed as a value (or returned as a result). This is the anonymous lambda term, used in higher-order functional programming. As a kind of *Expression*, a *Function* has reference semantics when it is passed by value. Refer to *Function Declaration* for the full specification of functions (see section 4.4).

Examples of the *Function* element used to define anonymous lambda terms include:

```
<Function type="Boolean">
  <Variable name="number" type="Integer"/>
  <Operator symbol="lessThan" type="Boolean">
    <Identifier name="number" type="Integer"/>
    <Literal value="0" type="Integer"/>
  </Operator>
</Function>

<Function type="Integer">
  <Variable name="number" type="Integer"/>
  <Operator symbol="times" type="Integer">
    <Identifier name="number" type="Integer"/>
    <Literal value="2" type="Integer"/>
  </Operator>
</Function>
```

The first anonymous function is a predicate to test whether an integer number is less than zero. This function could be mapped over a list of integers to filter the list. The second anonymous function is a transformer that doubles its integer argument. This function could be mapped over a list to transform the whole list.

3.6 Apply Expression

The *Apply* element may be used to define a function application expression. *Apply* denotes a function- or procedure-call in languages with functions and procedures. It is an evaluating expression that applies a function to suitable argument expressions to return a result. The *Apply* element may refer to a declared function by name, or may include the function-expression as its first child.

The grammar for the *Apply* element in FUN is:

```
<!ELEMENT Apply ((Function | Identifier)?,
    (Literal | Identifier | Function | Operator |
          Access | Apply | Select)+)>
<!ATTLIST Apply function NMTOKEN #IMPLIED>
<!ATTLIST Apply type NMTOKEN #REQUIRED>
```

The attributes of the *Apply* element are:

- *function* – the name of the function (optional, alternates with first child);
- *type* – the result type of the application (required).

The *function* name, if supplied, must be given in *camelCase*, and should refer to a previously defined function that is currently in scope. The *type* may be any legal ReMoDeL FUN type name (see section 2), but must also be identical to the expected result type of the function being applied.

The children of the *Apply* element are one or more expressions of suitable types:

- the function-expression (optional, alternates with *function*);
- further argument-expressions (required).

The argument-expressions must evaluate to suitable values, whose types correspond to the types expected by the function's arguments.

Examples of the *Apply* element include:

```
<Apply function="isEven" type="Boolean">
  <Literal value="7" type="Natural"/>
</Apply>

<Apply function="maximum" type="Natural">
  <Literal value="7" type="Natural"/>
  <Literal value="13" type="Natural"/>
</Apply>

<Apply type="Boolean">
  <Identifier name="test" type="Function[Natural, Boolean]"
  <Literal value="7" type="Natural"/>
</Apply>

<Apply type="Boolean">
  <Function type="Boolean">
    <Variable name="number" type="Natural"/>
    <Operator symbol="lessThan" type="Boolean">
      <Identifier name="number" type="Natural"/>
      <Literal value="0" type="Natural"/>
    </Operator>
  </Function>
  <Literal value="7" type="Natural"/>
</Apply>
```

The first example applies the named function *isEven* (which must be defined and in scope) to the single argument-expression. The second example applies the named function *maximum* to a pair of argument-expressions (the function must accept a pair of arguments of these types).

The third and fourth examples have no named function, but apply the first expression (the function-expression) to the remaining argument-expressions. The third example applies the function stored in the identifier *test* to its argument; and the fourth example applies a locally scoped anonymous function-expression to its argument (see also section 3.5).

3.7 Branch Expression

The *Branch* element may be used to define a lazy-evaluating expression. The *Branch* element is a kind of *Expression*, representing a guard or trigger protecting the enclosed expression, whose evaluation is delayed. In FUN, *Branch* is used in conjunction with *Select*, a generalised conditional branching construction. When a selection choice-expression is evaluated, and the result matches the trigger for a particular branch, this identifies which branch of the program to follow.

The grammar for the *Branch* element in FUN is:

```
<!Element Branch (Literal | Identifier | Function |
    Operator | Access | Apply | Select)>
<!ATTLIST Branch when CDATA #REQUIRED>
<!ATTLIST Branch type NMTOKEN #REQUIRED>
```

The attributes of the *Branch* element are:

- *when* – the trigger value for the branch (required);
- *type* – the type of the delayed expression (required).

The trigger *when* is a value determining when the branch should be executed. This can be a *Boolean* value (*false*, *true*), a scalar value (a symbolic or integral value) or the result of any other choice-expression tested by *Select* expressions (in FUN; other ReMoDeL dialects may also test choice-expressions in *Iterate* expressions). The *type* may be any legal ReMoDeL type name (see section 2), but must correspond to the type of the wrapped expression, whose evaluation is delayed.

The children of the *Branch* element are:

- *Expression* – the expression whose evaluation is delayed.

Boolean-triggered examples of the *Branch* element include:

```
<Branch when="true" type="String">
  <Literal value="True branch selected" type="String"/>
</Branch>

<Branch when="false" type="String">
  <Literal value="False branch selected" type="String"/>
</Branch>
```

Scalar-triggered examples of the *Branch* element include:

```
<Branch when="closed" type="String">
  <Literal value="Account is closed" type="String"/>
</Branch>

<Branch when="open" type="String">
  <Literal value="Account is open" type="String"/>
</Branch>

<Branch when="frozen" type="String">
  <Literal value="Account is frozen" type="String">
</Branch>
```

These examples are the cases for a multi-branch selection on some expression of the type *Status*, each wrapping an expression, here a *String* value to be returned. Cases of other scalar types are possible, such as *Integer*, *Natural* or *Character*, so long as all cases are mutually exclusive and exhaustive. There is no default case.

3.8 Select Expression

The *Select* element may be used to define a conditional branching expression. *Select* is the generalisation of the binary branching if-statement and the multi-branching switch-statement (or case-statement) in programming languages. *Select* is the only kind of *Compound* expression used in ReMoDeL FUN, which does not have *Sequence*, *Parallel* or *Iterate* expressions. (There are no sequences, since these only exist in languages that evaluate expressions for their side effects; and all repetition is accomplished through recursive definitions). *Select* is used in conjunction with lazy-evaluating *Branch* expressions in FUN.

The grammar for the *Select* element in FUN is:

```
<!ELEMENT Select ((Literal | Identifier | Operator |
          Access | Apply | Select),
    Branch, Branch+)>
<!ATTLIST Select choice (simple | multiple) #REQUIRED>
<!ATTLIST Select type NMTOKEN #REQUIRED>
```

The attributes of the *Select* element are:

- *choice* – whether the choice is *simple* or *multiple* (required);
- *type* – the result type of the selection expression (required).

The *choice* indicates whether a simple or multiple-choice selection is intended. The *type* may be any legal ReMoDeL FUN type name (see section 2), but must also be identical to the expected result type of each branch of the *Select* expression.

The *Select* element has one *Expression* child and two or more *Branch* children:

- the choice-expression, having a *Boolean* type for a simple choice and a scalar type for a multiple-choice (required);
- the branch expressions, of which one will be taken (required, two-to-many).

The first child is always the choice-expression. This has a *Boolean* value in binary selections (*if-then-else* statement), or any scalar value in a multi-branching selection (*switch, case* statement).

Examples of the *Select* element include the following simple choice and multiple-choice:

```
<Select choice="simple" type="String">
  <Operator symbol="moreThan" type="Boolean">
    <Identifier name="count" type="Natural"/>
    <Literal value="0" type="Natural"/>
  </Operator>
  <Branch when="true" type="String">
    <Literal value="More than zero" type="String">
  </Branch>
  <Branch when="false" type="String">
    <Literal value="Equal to zero" type="String">
  </Branch>
</Select>
```

This first example has a *Boolean*-valued choice expression, so is a binary branching selection. In FUN, binary branching expressions must have two branches and single-branch *if*-statements are not permitted. Both branches must return the same String type, which is also the type of the Select-expression.

```
<Select choice="multiple" type="String">
  <Identifier name="status" type="Status"/>
  <Branch when="closed" type="String">
    <Literal value="Account is closed" type="String"/>
  </Branch>
  <Branch when="open" type="String">
    <Literal value="Account is open" type="String"/>
  </Branch>
  <Branch when="frozen" type="String">
    <Literal value="Account is frozen" type="String"/>
  </Branch>
</Select>
```

This second example has a scalar-valued choice expression, of the enumerated type *Status*, so is a multi-branching selection. In FUN, multi-branching expressions must exhaustively cover all possible branches (missing branches are not permitted). So, it is useful to be able to use finite *Symbolic* types as the values of guards.

4 Declarations

The kinds of *Declaration* used in ReMoDeL FUN include all *Type*, *Variable* and *Function* declarations. Significant kinds of *Declaration* include *Classifier*, which defines all things with a namespace; and *Property*, which defines named properties with an associated *Type*. Descendants of *Property* include *Slot*, an indexed property, and *Member*, an owned property. Functions and variables are free properties that are not owned by other types (although they are organised in packages). Figure 2 illustrates the fragment of the ReMoDeL Metamodel dealing with declarations.

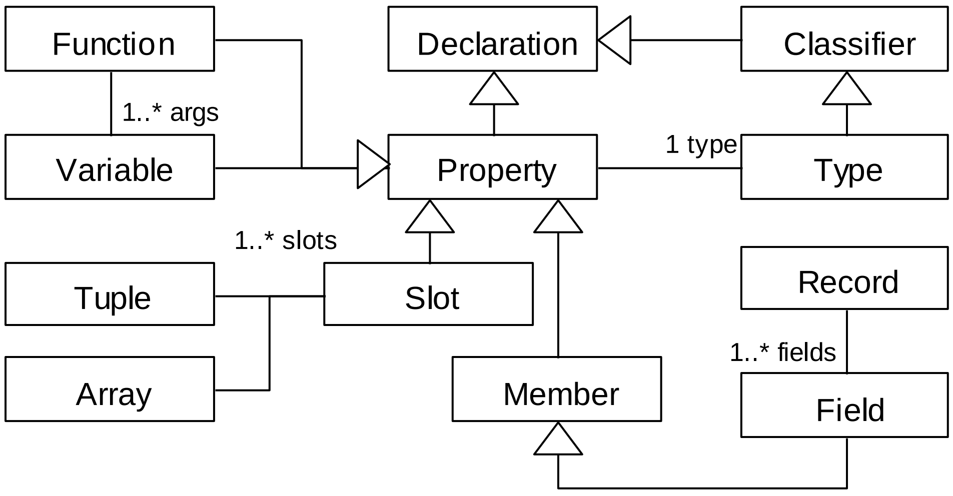

Figure 3: Declarations in the ReMoDeL Metamodel

Program declarations are instances of these metaclasses. For example, a function definition is an instance of the *Function* metaclass; and a global or local variable definition is an instance of the *Variable* metaclass. The types of the language, which were considered above (see section 2) are also kinds of *Declaration*.

4.1 Slot Declaration

The *Slot* element may be used to declare storage for program values. *Slot* is a basic concept in the ReMoDeL Metamodel, used to declare storage for values in *Array* and *Tuple* types. A slot is a degenerate kind of unnamed *Property*, which may instead be indexed by its owning type.

The grammar for the *Slot* element is:

```
<!ELEMENT Slot EMPTY>
<!ATTLIST Slot type NMTOKEN #REQUIRED>
```

The attributes of the *Slot* element are:

- *type* – the type of the slot (required).

The *type* may be any legal ReMoDeL FUN type name (see section 2).

The *Slot* element has no children.

Examples of the Slot element include:

```
<Slot type="Integer"/>

<Slot type="Person"/>

<Slot type="Array[12, Integer]"/>
```

The first example defines a slot for a basic *Integer* type, which will store its contents by value (have value-semantics). The second example defines a slot for a *Person* record type, which will store its contents by reference (have reference semantics). All values of *Structure* types are stored by reference.

The third example is rare, since *Array* types of more than one dimension are usually defined by nesting the *Array* element, rather than by typing each *Slot* individually. This form is useful when declaring a *Tuple* with *Slots* of different lengths, to create a ragged array. Refer to *Tuple Types* (see section 2.5) and *Array Types* (see section 2.6) for more details.

4.2 Field Declaration

The *Field* element may be used to define the named fields of records. A *Field* is a kind of *Member* (which is more significant in ReMoDeL OOP, where this metaclass bestows a *visibility* on all members) and transitively a kind of *Property* (from which it obtains a name and typed storage).

The grammar for the *Field* element in FUN is:

```
<!ELEMENT Field ((Literal | Identifier | Function |
    Operator | Access | Apply | Select)?)>
<!ATTLIST Field name NMTOKEN #REQUIRED>
<!ATTLIST Field type NMTOKEN #REQUIRED>
```

The attributes of the *Field* element are:

- *name* – the name of the field (required);
- *type* – the type of the field (required).

The *name* must be supplied in *camelCase*, the expected style for all property names. By default, the *visible* value is always *private* (visibility comes more into play in other ReMoDeL dialects). However, FUN also supplies the *Access* expression, which ignores all visibility restrictions. The *type* may be any legal ReMoDeL FUN type name (see section 2).

The *Field* element optionally has the child:

- the initial value expression to bind to the field (optional).

Initial value expressions may be used to declare and initialise a field at the point of introduction. It is more usual to initialise a whole record variable to a structured literal value; but the above syntax may be used to initialise constant record types.

4.3 Variable Declaration

The *Variable* element may be used to define named storage for program values. *Variable* is used to declare both global and local variables, and also formal arguments to functions. A *Variable* is a kind of *Property* (from which it obtains a property name and associated typed storage). In other ReMoDeL dialects supporting iteration, a Variable may have further attributes and child elements.

The grammar for the *Variable* element in FUN is:

```
<!ELEMENT Variable (Literal | Identifier | Function |
          Operator | Access | Apply | Select)?>
<!ATTLIST Variable name NMTOKEN #REQUIRED>
<!ATTLIST Variable type NMTOKEN #REQUIRED>
```

The attributes of the *Variable* element are:

- *name* – the name of the variable (required);
- *type* – the type of the variable (required).

The *name* must be supplied in *camelCase*, the expected style for all property names. FUN only uses the *scope* values *local* and *global*. The *step* relates to how a variable may be reset automatically. This feature is not used in FUN, since variables cannot be reassigned. The *type* may be any legal ReMoDeL FUN type name (see section 2).

In FUN, the *Variable* element optionally has the child:

- the initial value expression to bind to the variable (optional).

Initial value expressions may be used to declare and initialise a global or local variable at the point of introduction. The semantics of binding is identical to the binding of values to formal arguments upon entry to a function. Each variable introduced in the same scope is bound independently to its initial value.

Initial value expressions may also be used to express default values for formal arguments, to be used in certain circumstances when no supplied value is available. This is used with *folding* or *reducing* operations in higher-order functional programming.

Simple examples of the *Variable* element used as a parameter include:

```
<Variable name="amount" type="Integer"/>

<Variable name="primes" type="Array[5, Natural]"/>

<Variable name="function" type="Function[Natural, Boolean]"/>
```

These examples show how it is possible to declare variables of both simple and structured types. The last example shows how it is possible to type a variable so that it may receive a function expression as its value. Further examples of the *Variable* element include:

```
<Variable name="rate" type="Decimal">
  <Literal value="3.25" type="Decimal"/>
</Variable>

<Variable name="manager" scope="global" type="Person">
  <Literal type="Person">
    <Literal value="John" type="String"/>
    <Literal value="Smith" type="String"/>
    <Literal value="M" type="Character"/>
    <Literal value="32" type="Natural"/>
  </Literal>
</Variable>

<Variable name="primes" type="Array[5, Natural]">
  <Literal type="Array[5, Natural]">
    <Literal value="2" type="Natural"/>
    <Literal value="3" type="Natural"/>
    <Literal value="5" type="Natural"/>
    <Literal value="7" type="Natural"/>
    <Literal value="11" type="Natural"/>
  </Literal>
</Variable>
```

These examples illustrate how variables of both simple and structured types may be initialised, respectively to simple or structured literal values. These are implicitly global variables, if declared in package scope.

4.4 Function Declaration

The *Function* element may be used to define named functions for execution in a program. A *Function* declares one or more formal parameters and defines a body expression to compute its result, which is the value returned. A function may also declare generic type parameters, if it is a generic function. Defining a function binds the function's name to the function's definition (which is treated exactly like binding a variable to a value). A *Function* is a kind of *Property* (from which it obtains a property name and an associated type).

The grammar for the *Function* element is:

```
<!ELEMENT Function (Generic*, Variable+,
    (Literal | Identifier | Function |
          Operator | Access | Apply | Select))>
<!ATTLIST Function name NMTOKEN #IMPLIED>
<!ATTLIST Function type NMTOKEN #REQUIRED>
```

The attributes of the *Function* element are:

- *name* – the name of the function (optional, if anonymous);
- *type* – the result type of the function (required).

The *name*, if supplied, must be in *camelCase*, the expected style for all property names. Functions may be anonymous (see section 3.5). The *type* records the result type of the function and may be any legal ReMoDeL FUN type name (see section 2). This is different from the function's complete type signature (see section 2.7).

The *Function* element has the children:

- *Generic* – generic type parameter (optional, zero-to-many);
- *Variable* – the formal parameters to the function (required, one-to-many);
- the function's single body expression (required).

A function will have generic parameters, if it manipulates a type, which also contains generic parameters. A function must have one to many formal parameters. Some of these may be given default values (allowing the function to be called on fewer arguments). A function always has exactly one body expression, whose type is the same as the function's result type. No special syntax is required to indicate the result of the *Function*, since this is always clear in FUN.

Simple examples of the *Function* element include:

```
<Function name="sumTwoValues" type="Integer">
  <Variable name="first" type="Integer"/>
  <Variable name="second" type="Integer"/>
  <Operator symbol="plus" type="Integer">
    <Identifier name="first" type="Integer"/>
    <Identifier name="second" type="Integer"/>
  </Operator>
</Function>

<Function name="isEven" type="Boolean">
  <Variable name="number" type="Natural"/>
  <Operator symbol="equals" type="Boolean">
    <Operator symbol="modulo" type="Natural">
      <Identifier name="number" type="Natural"/>
      <Literal value="2" type="Natural"/>
    </Operator>
    <Literal value="0" type="Natural"/>
  </Operator>
</Function>
```

These examples illustrate how the body can be any expression. The second example has a nested body expression, where the inner operator expression returns its value to the outer operator expression, which returns as the result of the function.

Some more complex examples are given below. The first example defines the higher-order *map* function, which maps a function over the elements of a list, returning a list of the transformed results. It illustrates the use of generic parameters, to generalise the types of the input and output lists, relating these to the transformer function's signature. It also demonstrates how to pass a function as an argument to another function, and how to call it internally. Finally, it illustrates the use of recursion when traversing a list.

```
<Function name=map type="List[Result]">
  <Generic name="Argument"/>
  <Generic name="Result"/>
  <Variable name="func" type="Function[Argument, Result]"/>
  <Variable name="list" type="List[Argument]"/>
  <Select choice="simple" type="List[Result]">
    <Operator symbol="equals" type="Boolean">
      <Identifier name="list" type="List[Argument]"/>
      <Literal value="null" type="List[Argument]"/>
    </Operator>
    <Branch when="true" type="List[Result]">
      <Literal value="null" type="List[Result]"/>
    </Branch>
    <Branch when="false" type="List[Result]">
      <Apply function="add" type="List[Result]">
        <Apply function="func" type="Result">
          <Apply function="head" type="Argument">
            <Identifier name="list" type="List[Argument]"/>
          </Apply>
        </Apply>
        <Apply function="map" type="List[Result]">
          <Apply function="tail" type="List[Argument]">
          <Identifier name="func"
              type="Function[Argument, Result]"/>
          <Apply function="tail" type="List[Argument]">
            <Identifier name="list" type="List[Argument]"/>
          </Apply>
        </Apply>
      </Apply>
    </Branch>
  </Select>
</Function>
```

The next example is of a function with default values for its arguments. This function *sum* is normally called on two arguments, which it adds together. However, if it is called on fewer arguments, it uses the default values in place of any missing actual values. The reason for this flexibility is to support a different kind of higher-order functional programming, when reducing a list.

```
<Function name="sum" type="Integer">
  <Variable name="next" type="Integer">
    <Literal value="0" type="Integer"/>
  </Variable>
  <Variable name="total" type="Integer">
    <Literal value="0" type="Integer"/>
  </Variable>
  <Operator symbol="plus" type="Integer">
    <Identifier name="next" type="Integer"/>
    <Identifier name="total" type="Integer"/>
  </Operator>
</Function>
```

The final example is of the *reduce* function, a higher-order function that expects to perform some aggregating operation over a whole list. If it is called with *sum* as its first argument, this will sum the values of the list, by adding the head of the input list to the result of summing the recursive call. Eventually, the input list will be empty, so *sum* is defined to return zero, when invoked on no arguments.

```
<Function name=reduce type="Result">
  <Generic name="Argument"/>
  <Generic name="Result"/>
  <Variable name="func"
      type="Function[Argument, Result, Result]"/>
  <Variable name="list" type="List[Argument]"/>
  <Select choice="simple" type="Result">
    <Operator symbol="equals" type="Boolean">
      <Identifier name="list" type="List[Argument]"/>
      <Literal value="null" type="List[Argument]"/>
    </Operator>
    <Branch when="true" type="Result">
      <Apply function="func" type="Result"/>
    </Branch>
    <Branch when="false" type="Result">
      <Apply function="func" type="Result">
        <Apply function="head" type="Argument">
          <Identifier name="list" type="List[Argument]"/>
        </Apply>
        <Apply function="reduce" type="Result">
          <Identifier name="func"
              type="Function[Argument, Result, Result]"/>
          <Apply function="tail" type="List[Argument]">
            <Identifier name="list" type="List[Argument]"/>
          </Apply>
        </Apply>
      </Apply>
    </Branch>
  </Select>
</Function>
```

4.5 Package Declaration

ReMoDeL FUN uses the ReMoDeL Package and Namespace Model for organising function and variable definitions into separate packages. See the separate model specification [2] for full details.

A ReMoDeL FUN package is declared using the *Package* element to wrap the contents of the package. A skeleton example is the following:

```
<Package model="FUN" name="Func" location="lib.func">

  <!-- other definitions inserted here -->

</Package>
```

This declares that the definitions contained within belong to the package whose namespace is *Func*, whose contents are to be stored at the logical location *lib.func* and that the package model type is *FUN*. This information may be interpreted differently by different translators: they may use the namespace identifier to qualify all definitions, or use the logical location to store all compiled definitions at a given physical location under a root directory for the target language.

The entire contents of a package may be imported into the current namespace using the global import instruction:

```
<Import model="FUN" package="Func" location="lib.func"/>
<Import model="FUN" package="Lists" location="lib.lists"/>
```

Alternatively, a package may express its dependency on other packages and then certain elements may be imported selectively:

```
<Consult model="FUN" package="Func" location="lib.func"/>
<Consult model="FUN" package="Lists" location="lib.lists"/>

<Employ refer="map" kind="Function" from="Func"/>
<Employ refer="reduce" kind="Function" from="Func"/>
<Employ refer="List" kind="Record" from="Lists"/>
<Employ refer="head" kind="Function" from="Lists"/>
<Employ refer="tail" kind="Function" from="Lists"/>
```

Model-checking tools should be able to identify suitable matching elements in the consulted package.

4.6 Program Declaration

In ReMoDeL FUN, the notion of a program is deliberately simple. A program consists of a set of function and variable declarations, possibly imported from various foreign packages. The main program is simply a distinguished variable, called *result*, of some appropriate type (here *ResultType* is just a template), whose initialisation forces the evaluation of the whole program:

```
<Variable name="result" type="ResultType">

  <!-- initialisation expression inserted here -->

</Variable>
```

How the result of the computation is communicated to the end-user is a matter for different translations. The FUN model says nothing about input or output, since the language is one of pure computation. Input and output are imperative operations, and different functional languages have different ways of finessing I/O. Some expect the result to be stored in a variable with a predictable name. Others use more arcane functional machinery, such as monads.

A translation of FUN into a Functional language may have to identify the program entry point in some way to the translator. Alternatively, the translator may use *result* directly in the target language, or identify the initialisation expression for *result* as the main function to call in the target language.

5 References

[1] World Wide Web Consortium (W3C), *Extensible Markup Language (XML) 1.0, 5th edition*, eds. T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, 26 November, 2008. http://www.w3.org/TR/2008/REC-xml-20081126/

[2] A. J. H. Simons, *ReMoDeL Package and Namespace Model Specification*, Technical Report, Department of Computer Science, University of Sheffield, 2011.