The `eval` symbol for axiomatising variadic functions

Lars Hellström

Division of Applied Mathematics, The School of Education, Culture and Communication, Mälardalen University, Box 883, 721 23 Västerås, Sweden; `lars.hellstrom@residenset.net`

Abstract

This paper describes (and constitutes the source for!) the proposed `list4` OpenMath content dictionary. The main feature in this content dictionary is the `eval` symbol, which treats a list of values as the list of children of an application element. This may, among other things, be employed to state properties of variadic functions.

1 Background and motivation

OpenMath is a formal language for (primarily) mathematics. It is not a coherent theory of mathematics, but the standard makes room for and even encourages expressing small fragments of theory in the form of *mathematical properties* of symbols in content dictionaries. The main purpose of these is to nail down exactly what concept a symbol denotes, and they can take the form of a direct definition of the symbol, but mathematical properties may also clarify a concept in more indirect ways, e.g. by stating that a particular operation is commutative.

As a language of formalised mathematical logic, OpenMath is somewhat unusual in allowing application symbols to be variadic—a flexibility that is most commonly used to generalise binary associative operations into general n-ary operations, but it is by no means useful only for that. By contrast, the formal language used in e.g. [2] rather considers the arity to be a built-in property of each function or predicate symbol, and acknowlegdes no particular link between unary function symbol one (f_1^1) and binary function symbol one (f_1^2). Variadic application symbols frequently permit a more natural encoding of formulae than a translation to fixed arity symbols would allow, but they raise the problem of how to define this symbol for all the infinitely many forms that an application of it may take.

In theoretical mathematical logic, this problem is a minor one since a theory is allowed to have infinitely many axioms; usually they would be generated from a finite set of axiom schemata, but there is no requirement that this be the case. Practical reasoning about a concrete theory does on the other hand require that the axioms of the theory are in some sense known, so one still ends up having to present them in some finite fashion. This is where OpenMath can run into trouble, as the Formal Mathematical Properties have no schematic ability whatsoever. (Having first-class functions lets you do some things as single axioms that more theoretical texts would probably employ schemata for, but first-class functions cannot achieve much in basic syntactical matters.) The *OpenMath sequences* proposal [1] can be seen as adding some schematic abilities of immediate relevance to variadic application, but although it may look like a simple meta-level mechanism for generating arbitrarily long argument sequences, it is in fact an extension of the base language. This is troublesome both in that it can have unforseen logical consequences (as the proposal acknowledges [1, Sec. 6]) and in that it renders the base OpenMath language far less simple than is currently the case.

This content dictionary implements an alternative approach for stating formal properties of variadic functions, namely to introduce one new application symbol with the power to treat a

dynamically generated *list* of objects as they would have been treated by an application (XML encoding OMA element) if these objects had been its children. This does not eliminate the use of axiom schemata, but it reduces the need for them to a few places that deal with fundamental functions, as opposed to something that would be needed whenever a custom variadic function was defined. This should increase the chances of formal properties being machine-useable, since it is reasonable for a system to make special cases for a few fundamental symbols (if that is what it needs to make practical use of the claims they make), but less reasonable for it to recognise a wide range of idioms.

The name, as well as the detail semantics, of the new evaluate-list symbol (eval in the list4 content dictionary) is taken from the dynamic programming language Tcl. A discussion of the relevant commands and their use there can be found in [3, p. 130 ff.].

To conserve space on the page, most OMOBJs below are typeset in a semi-formulaic style. @ denotes application (corresponds to an OMA element in the XML encoding) and **bind** denotes bindings (OMBINDs). Italic identifiers are variables, whereas symbols are written as cd.name in the upright text shape.

2 The list4 content dictionary

(This section also constitutes the source from which the .ocd file is generated!)
Description: This CD provides symbols that make it possible to state general FMPs about variadic application symbols, by treating a list of things as the list of children of an OMA element.
CDBase: http://www.openmath.org/cd
Version 0, revision 2.
Standard OpenMath licence terms apply.

2.1 islist predicates

When using an eval symbol for stating formal properties about particular symbols, one needs to quantify variables over sets of lists with the right kind of elements. For this, it is convenient to have a few helper predicates.

Symbol islist (application)

This symbol is a predicate stating that something (the only argument) is a list. This is useful for making claims of the form "for every list L, it holds that ..." in FMPs.

Example. The list whose elements are the integers 3, 6, and 5 is a list:

@(list4.islist, @(list1.list, 3, 6, 5))

The next two FMPs state that the empty list is a list, and prepending any object to a list produces a new list.

Formal Mathematical Property 1.

@(list4.islist, list2.nil)

Formal Mathematical Property 2.

bind(quant1.forall; L, x; @(logic1.implies, @(list4.islist, L), @(list4.islist, @(list2.cons, x, L))))

Remember, however, that mathematical lists need not *be* LISP-style linked lists of cons cells; that is merely one way in which they *could* be encoded under a pointer-style memory model. If they are so encoded, then those `nil` and `cons` symbols express primitive operations, but if lists were instead to be encoded as (say) functions defined on a finite ordinal then these operations become less straightforward.

It may also be observed that these two axioms do not rule out nonstandard models of lists (e.g. circular lists or infinite lists), but users who insist on recognising such things as lists shall bear the responsibility for making sense of the rest of the theory under this interpretation.

Symbol `islistwhere` (application)

This symbol is a predicate of two arguments L and P, where P is a predicate of one argument. The claim expressed is first that L is a list, and second that every element of it satisfies P.

Formal Mathematical Property 3.

@(relation1.eq, list4.islistwhere,
bind(fns1.lambda; L, P; @(logic1.and, @(list4.islist, L), **bind**(quant1.forall; k;
@(logic1.implies, @(logic1.and, @(set1.in, k, setname1.Z), @(relation1.le, 1, k),
@(relation1.le, k, @(list3.length, L))), @(P, @(list3.entry, L, k)))))))

2.2 The `eval` symbol itself

Symbol `eval` (application)

This is an application symbol which takes an arbitrary number of lists as arguments. The concatenation of those lists should be nonempty. To eval those lists produces the same result as an application (OMA element, in the XML encoding) having the concatenation of them as its list of children.

Formal Mathematical Property 4 (nullary application)**.**

bind(quant1.forall; f; @(relation1.eq, @(f), @(list4.eval, @(list1.list, f))))

Formal Mathematical Property 5 (unary application)**.**

bind(quant1.forall; f, x; @(relation1.eq, @(f, x), @(list4.eval, @(list1.list, f, x))))

Formal Mathematical Property 6 (binary application)**.**

bind(quant1.forall; f, x, y; @(relation1.eq, @(f, x, y), @(list4.eval, @(list1.list, f, x, y))))

Formal Mathematical Property 7 (ternary application)**.**

bind(quant1.forall; f, x, y, z; @(relation1.eq, @(f, x, y, z), @(list4.eval, @(list1.list, f, x, y, z)))
)

Commented Mathematical Property. In general, if f is any application symbol, and x1, x2, ..., xN is any sequence of arguments, then f(x1,x2,...,xN) is equal to eval(list1.list(f,x1,x2,...,xN)). This infinite sequence of identities cannot be expressed as an FMP, even if every particular element in that sequence can be so expressed, but by accepting it to hold for the eval symbol in particular, it becomes possible to state similar infinite sequences of claims about other symbols as finite FMPs.

Commented Mathematical Property. The second thing one needs to know about eval is that an eval of two lists is the same as an eval of their concatenation.

Formal Mathematical Property 8 (concatenation)**.**

bind(quant1.forall; L, M; @(logic1.implies, @(logic1.and, @(list4.islist, L), @(list4.islist, M)), @(relation1.eq, @(list4.eval, L, M), @(list4.eval, @(list2.append, L, M)))))

Commented Mathematical Property. Finally, to define eval for more than two arguments, we can use nested evaluation with a list-of-lists argument.

Formal Mathematical Property 9 (nested evaluation base)**.**

bind(quant1.forall; LL; @(logic1.implies, @(list4.islistwhere, LL, list4.islist), @(relation1.eq, @(list4.eval, @(list1.list, list4.eval, list2.nil), LL), @(list4.eval, @(list1.list, list4.eval), LL))))

Formal Mathematical Property 10 (nested evaluation step)**.**

bind(quant1.forall; L, M, NN; @(logic1.implies, @(logic1.and, @(list4.islist, L), @(list4.islist, M), @(list4.islistwhere, NN, list4.islist)), @(relation1.eq, @(list4.eval, @(list1.list, list4.eval, L), @(list2.cons, M, NN)), @(list4.eval, @(list1.list, list4.eval, @(list2.append, L, M)), NN))))

Commented Mathematical Property. It is a theorem following from the above that an eval of three lists is equal to a two-argument eval of the concatenation of the first two lists and the last list.

Formal Mathematical Property 11 (concatenate two of three)**.**

bind(quant1.forall; L, M, N; @(logic1.implies, @(logic1.and, @(list4.islist, L), @(list4.islist, M), @(list4.islist, N)), @(relation1.eq, @(list4.eval, L, M, N), @(list4.eval, @(list2.append, L, M), N))))

Proof. For all lists L, M, and N:

@(list4.eval, L, M, N)
is by FMP 7 equal to
@(list4.eval, @(list1.list, list4.eval, L, M, N))
is by equality of lists equal to
@(list4.eval, @(list2.append, @(list1.list, list4.eval, L), @(list1.list, M, N)))
is by FMP 8 equal to
@(list4.eval, @(list1.list, list4.eval, L), @(list1.list, M, N))
is by equality of lists equal to
@(list4.eval, @(list1.list, list4.eval, L), @(list2.cons, M, @(list1.list, N)))
is by FMP 10 equal to
@(list4.eval, @(list1.list, list4.eval, @(list2.append, L, M)), @(list1.list, N))
is by FMP 8 equal to
@(list4.eval, @(list2.append, @(list1.list, list4.eval, @(list2.append, L, M)), @(list1.list, N)))
is by equality of lists equal to
@(list4.eval, @(list1.list, list4.eval, @(list2.append, L, M), N))
is by FMP 6 equal to
@(list4.eval, @(list2.append, L, M), N)

□

It would be possible to achieve the basic eval functionality also with other syntaxes for the main operation. Two alternatives that may suggest themselves are:

1. Restrict the eval operation to one argument, and instead use a separate list operation to concatenate multiple lists when that becomes necessary.
2. Make the first child—the function being applied—a separate first argument of eval rather than merely the first list element.

However, neither is how it is done in Tcl, and experience from there suggests that either alternative would have disadvantages. FMP 12 below is one example that constructing a function application from several pieces is useful, and even natural, so the argument for a variadic eval is not unlike that for a variadic arith1.plus over a sum-over-list function. The reason not to treat the first child as special is more subtle, and has to do with the stylistic choice between curried and uncurried function application. In the curried style, the function being applied to something always exists as an explicit object in the formula. In an uncurried style, it is however possible that a "prefix" consisting of an application symbol and some initial arguments (in this context often called 'parameters') is morally "the function" even though they may not technically constitute one object in the formula. To set the first child off as special would favour the curried style, whereas treating it equally gives equal support to both styles.

2.3 An n-associativity predicate

Symbol `nassoc` (application)

This symbol is a predicate which takes one operation as argument. It expresses the claim that this operation is n-associative, i.e., that it is variadic, that the binary case of this operation is associative, that applying it to more than two arguments can be restated as repeated application of the binary form, and also that the unary and nullary cases of the operation follow suit.

Formal Mathematical Property 12 (definition)**.**

$$\begin{array}{l}@(\text{logic1.equivalent}, @(\text{list4.nassoc}, f),\\ \quad @(\text{logic1.and}, \mathbf{bind}(\text{quant1.forall}; x; @(\text{relation1.eq}, @(f, x), x)), \mathbf{bind}(\text{quant1.forall}; L, M;\\ \qquad @(\text{relation1.implies}, @(\text{logic1.and}, @(\text{list4.islist}, L), @(\text{list4.islist}, M)),\\ \qquad\quad @(\text{relation1.eq}, @(\text{list4.eval}, @(\text{list1.list}, f), L, M),\\ \qquad\qquad @(f, @(\text{list4.eval}, @(\text{list1.list}, f), L), @(\text{list4.eval}, @(\text{list1.list}, f), M)))))))\end{array}$$

Example. The claim that the arith1#plus operation is n-associative:

```
<OMOBJ>
  <OMA> <OMS cd="list4" name="nassoc"/>
    <OMS cd="arith1" name="plus"/>
  </OMA>
</OMOBJ>
```

To elaborate, here are the steps of a proof that $f(x, y, z) = f\big(f(x, y), z\big)$ if nassoc(f):

$@(f, x, y, z)$
is by FMP 7 equal to
$@(\text{list4.eval}, @(\text{list1.list}, f, x, y, z))$

is by equality of lists equal to
  $@(\text{list4.eval}, @(\text{list2.append}, @(\text{list1.list}, f, x, y), @(\text{list1.list}, z)))$
is by FMP 8 equal to
  $@(\text{list4.eval}, @(\text{list1.list}, f, x, y), @(\text{list1.list}, z))$
is by equality of lists equal to
  $@(\text{list4.eval}, @(\text{list2.append}, @(\text{list1.list}, f), @(\text{list1.list}, x, y)), @(\text{list1.list}, z))$
is by FMP 11 equal to
  $@(\text{list4.eval}, @(\text{list1.list}, f), @(\text{list1.list}, x, y), @(\text{list1.list}, z))$
is by (the second part of) FMP 12 and nassoc(f) equal to
  $@(f, @(\text{list4.eval}, @(\text{list1.list}, f), @(\text{list1.list}, x, y)), @(\text{list4.eval}, @(\text{list1.list}, f), @(\text{list1.list}, z))$
)
is by FMP 8 equal to
  $@(f, @(\text{list4.eval}, @(\text{list2.append}, @(\text{list1.list}, f), @(\text{list1.list}, x, y))),$
    $@(\text{list4.eval}, @(\text{list2.append}, @(\text{list1.list}, f), @(\text{list1.list}, z))))$
is by equality of lists equal to
  $@(f, @(\text{list4.eval}, @(\text{list1.list}, f, x, y)), @(\text{list4.eval}, @(\text{list1.list}, f, z)))$
is by FMPs 6 and 5 equal to
  $@(f, @(f, x, y), @(f, z))$
is by (the first part of) FMP 12 and nassoc(f) equal to
  $@(f, @(f, x, y), z)$

It should be observed that this definition of nassoc(f) will make $f()$ the identity element for the binary f operation, i.e., f is the operation of a monoid. If one merely wants the semigroup kind of n-associativity, then one should exclude empty lists in the second part of the definition.

3 Concluding remarks

The mechanism introduced here does not operate by giving functions access to their lists of syntactic arguments. Rather, it works by relating various explicitly constructed lists to the result of an application with those lists as arguments. Programming-wise, there is probably not much difference between these, but the latter feels more mathematical.

It could be argued that the use of the variadic list1.list symbol when stating FMPs above is suboptimal, as using list2.cons and list2.nil throughout instead would reduce the number of schemata needed in a fully formalised axiomatisation. This is true, but I chose to use the variadic list1.list to emphasise the fact that the concept of list is not necessarily implemented in the LISP fashion as a linked list of cons cells. list1.list is an every bit as foundational symbol as list4.eval, so one should not be afraid to equip it with a scheme or two of axiomatic properties, if that is what defining it takes.

References

[1] Fulya Horozal, Michael Kohlhase, and Florian Rabe. Extending OpenMath with Sequences, pp. 58–72 in: *Intelligent Computer Mathematics, Work-in-Progress Proceedings*, Technical Reports of University of Bologna UBLCS-2011-04, 2011. `http://kwarc.info/frabe/Research/HKR_sequences_11.pdf`

[2] Elliot Mendelson. *Introduction to Mathematical Logic*. Wadsworth & Brooks, 1987 (3rd ed.).

[3] Brent B. Welch, Jeffrey Hobbs, and Ken Jones. *Practical programming in Tcl/Tk* (4th ed.). Prentice Hall PTR, 2003. ISBN 0-13-038560-3.