Open, extensible composition models

For the workshop on Free Composition at ECOOP 2011. The proceedings were published by the ACM digital library.

Ian Piumarta

VPRI Technical Report TR-2011-002

This material is based upon work supported in part by the National Science Foundation under Grant No. 0639876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Viewpoints Research Institute, 1209 Grand Central Avenue, Glendale, CA 91201 t: (818) 332-3001 f: (818) 244-9761

Open, extensible composition models

(extended abstract)*

Ian Piumarta

Academic Center for Computing and Media Studies, Kyoto University, Japan
Viewpoints Research Institute, Glendale, CA, USA

ian@vpri.org

ABSTRACT

Simple functional languages like LISP are useful for exploring novel semantics and composition mechanisms. That usefulness can be limited by the assumptions built into the evaluator about the structure of data and the meaning of expressions. These assumptions create difficulties when a program introduces a composition mechanism that differs substantially from the built-in mechanism of function application. We explore how an evaluator can be constructed to eliminate most built-in assumptions about meaning, and show how new composition mechanisms can be introduced easily and seamlessly into the language it evaluates.

1. INTRODUCTION

Adding a new composition mechanism to a programming language can entail defining a corresponding data type T, creating and maintaining values of that type, adding a syntactic operator to combine those values with one or more other values, and providing an algorithm that determines the compositional meaning of those combinations. Values of T retain persistent information needed by the composition. These values can also act as syntactic operators, if the intrinsic evaluation mechanism associates their presence in a combination with behaviour specific to T. The algorithm provides semantics for the composition, expressed as further compositions or as primitive operations of the language, according to the values being combined.

The above can be achieved within the basic abstractions of some programming languages, although unnecessary complexity and obfuscation arise whenever those abstractions limit direct access to the data and algorithms implementing the composition. If supporting mechanisms can be introduced at the meta level of the host language then these limitations do not arise, the interface presented to the programmer can deal directly and efficiently with relevant information, and the new composition appears as a natural extension of the language—qualitatively indistinguishable from an intrinsic mechanism.

*The full paper [4] is available online.

FREECO'11, July 26, 2011, Lancaster, UK

The introduction of supporting mechanisms at the meta level will be illustrated using a functional language derived from McCarthy's LISP [1, 2]. Section 2 introduces this language and its evaluator. Section 3 describes modifications in its meta level to accommodate user-defined compositions. Section 4 presents several examples of composition mechanisms added to the language. Section 5 discusses the work and places it in context. Section 6 offers concluding remarks.

1.1 Typographic conventions

URLs and code are set `monospaced`. In code the name of a `<type>` identifier is enclosed by angle brackets and access to the fields of its values is designated `<type>-field`. (`<`, `>`, `%` and `-` are letters having no special significance.) Primitive behaviour is written as *`{pseudo code}`*.

2. A MINIMAL FUNCTIONAL LANGUAGE

Figure 1 defines an evaluator for a functional language of symbolic expressions represented as lists in polish notation. The evaluator corrects several semantic inadequacies of LISP (described by Stoyan [5]) and is metacircular (written in the language it evaluates). The language provides:

- lists and atomic values, including symbols
- primitive functions called `<subr>`s
- symbolic functions (closures) called `<expr>`s
- a `<fixed>` object that encapsulates another applicable value and prevents argument evaluation
- predicates to discriminate between the above types
- built-in `<subr>`s to access the contents of these values
- a way to *call* the primitive behaviour of a `<subr>`
- a quotation mechanism to prevent evaluation of literals

The interpretation of structures is defined by the usual pair of functions `eval` and `apply`. `eval` takes an `exp`ression (simple or complex) and yields its value in the context of an `env`ironment of bound names. `apply` takes a complex expression, split into a `fun`ction part and its `arg`ument`s`, and yields the result of applying the former to the latter in the context of an `env`ironment of bound names. (As in LISP [2]: `evlis` evaluates each element in a list and returns a list of the results, `pairlis` extends an environment by binding a list of names to a list of values, and `assoc` finds a previously-bound name in an environment.)

A global initial environment contains bindings for primitives (`<subr>` values) and control structures (`<subr>`s or `<expr>`s wrapped in a `<fixed>`). Assignment and mutable state are supported via primitives (as they were in LISP [2, pp. 70*ff*]).

```
(define eval (lambda (exp env)
  (cond
    ((symbol? exp) (cdr (assoc exp env)))
    ((atom?   exp) exp)
    ('t            (let ((fn (eval (car exp) env)))
                     (if (fixed? fn)
                         (apply (<fixed>-function fn) (cdr exp) env)
                         (apply fn (evlis (cdr exp) env) env)))))))

(define apply (lambda (fun args env)
  (cond
    ((subr? fun) {call (<subr>-implementation fun) args env})
    ((expr? fun) (eval (<expr>-body fun) (pairlis (<expr>-formals fun) args (<expr>-environment fun)))))))
```

Figure 1: Evaluation of symbolic expressions in a minimal functional language

```
(define %type-names  (tuple))
(define %type-sizes  (tuple))
(define %type-fields (tuple))

(define %allocate-type
  (let ((last-type number-of-builtin-types))
    (lambda (name fields)
      (let ((type (set last-type (+ 1 last-type))))
        (set-tuple-at %type-names  type name)
        (set-tuple-at %type-sizes  type (list-length fields))
        (set-tuple-at %type-fields type fields)
        type))))

(define <point> (%allocate-type '<point> '(x y)))

(define <point>-x (lambda (value)
  (and (= <point> (type-of value))
       (tuple-at value 0))))
```

Figure 2: Adding an aggregate type to the language

`<expr>`s are closures carrying the environment in which they were defined. Variable lookup is lexically scoped. (A trivial change to the last line in Figure 1 gives LISP's dynamic scoping.) Additional atomic types such as numbers, and primitives that act on them, will be present in a practical language of this kind. They are omitted here for brevity.

3. SUPPORTING OPEN COMPOSITION

The language just described supports one combining form (the list). When a list is evaluated it causes a composition in which each of the element(s) being combined is recursively evaluated yielding one or more values, the first of which is then applied (as a function) to the rest (the arguments). "Open composition" means the ability to add new compositions (or replace existing ones) corresponding to the evaluation of new (or existing) combining forms.

Modifications to the language will be made to support:

- defining new types, to represent the syntactic operators, state and semantics of composition mechanisms, and
- associating meaning with combinations involving these new types.

3.1 Extensible aggregate types

Predicates in `eval` and `apply` use some property of a value to discriminate between types. The simplest generalisation is to identify each type with a unique integer. Incrementing a counter suffices to allocate a new type. A primitive function `type-of` yields the type identifier for a given value.

Three types (`<expr>`, `<fixed>` and `<pair>`s for constructing lists) appear in the evaluator that are aggregates of values. Aggregation can be generalised to a single mechanism: infinitely-sized, indexable N-tuples containing *undefined* at all uninitialised indices.[1]

[1]Assignment at an uninitialised index extends the tuple as necessary.

With these mechanisms addition of a new aggregate type can be effected in user code, as shown in Figure 2.[2] No restrictions have been placed on the structure or semantics of objects. Intrinsic types (those used by the evaluator) are built from the same parts: there is no disparity between built-in and user-defined types and values.

3.2 Extensible composition rules

Two kinds of composition occur in the evaluator. Simple atomic expressions (`symbol`s in particular) are composed with the environment by `eval` to yield a value. Combinations of one or more values in complex expressions are composed by `apply` by application of a primitive or closure value to the remaining values. Both of kinds of composition are made extensible by applying an appropriate combination mechanism to every expression according to its type. Two tuples, *evaluators* and *applicators*, are indexed by type in `eval` and `apply`, respectively.

$$\mathrm{eval}(x, e) = \mathrm{apply}(\boldsymbol{evaluators}[\mathrm{type}(x)], \mathrm{cons}(x, nil), e)$$
$$\mathrm{apply}(f, a, e) = \mathrm{apply}(\boldsymbol{applicators}[\mathrm{type}(f)], \mathrm{cons}(f, a), e)$$

Four consequences of this decomposition are:

- any applicable value can supply the composition semantics for any expression, simple or complex,
- any value can be made applicable, with semantics determined by its type and its value,
- the meaning of a complex expression (explicit combination of values, e.g., as a list) is not fixed, or even supplied, by the evaluation mechanism, and

[2]The steps shown allocate the type, record information for printing and instantiating, and define an accessor for an instance field. In practice these steps are generated automatically from a single `define-type` expression, given a type name and a list of field names, along with the implied set of `<type>-field` accessors.

```
(define eval (lambda (exp env)
  (apply (tuple-at evaluators (type-of exp)) (list exp env) env)))

(define apply (lambda (fn args env)
  (if (subr? fn)
    {call (<subr>-implementation fn) args env}
    (apply (tuple-at applicators (type-of fn)) (list fn args env) env))))
```

Figure 3: Generalised assignment of meaning to expressions

```
(set-tuple-at evaluators <symbol> (lambda (exp env) (cdr (assoc exp env))))

(set-tuple-at evaluators <number> (lambda (exp env) exp))

(set-tuple-at evaluators <pair> (lambda (exp env)
  (let ((fn (eval (car exp) env)))
    (if (eq (type-of fn) <fixed>)
      (apply (<fixed>-function fn) (cdr exp) env)
      (apply fn (evlis (cdr exp) env) env)))))

(set-tuple-at applicators <expr> (lambda (fn args env)
  (eval (<expr>-body fn) (pairlis (<expr>-formals fn) args (<expr>-environment fn)))))
```

Figure 4: The original language semantics expressed as composition rules

```
(define-type <form> (function))

(define form (lambda (function)
  (let ((self (new <form>)))
    (set (<form>-function self) function)
    self))

(set-tuple-at *applicators* <form> (lambda (fn args env)
  (eval (apply (<form>-function fn) args env) env)))
```

Figure 5: Form type for defining macros

- *apply* is an infinitely-recursive function.

Infinite recursion is avoided by short-circuiting the semantics of applying a primitive, as shown in Figure 3. Four entries in the *evaluators* and *applicators* tuples, as shown in Figure 4, restore the original semantics to the language.

Associating *evaluators* and *applicators* with `env` (effectively binding them in the environment) permits great flexibility in assigning meaning to program constructs, including incremental extension or modification of existing composition mechanisms, and multiple context-sensitive semantics for any given type. (The latter is pivotal when modelling language implementation as a series of partial evaluations, which is the motivation for an "open composition model".)

4. DEFINING NEW COMPOSITIONS

Three new compositions will be added to the language: a `<form>` value (for defining macros), object-oriented message passing and generic functions. (We assume a quasiquotation mechanism, which is straightforward given `<form>`.)

4.1 Forms and macros

Figure 5 introduces an applicable `<form>` type, encapsulating some other applicable value. The encapsulated value is applied to the arguments and then the result is re-evaluated. Placing a `<form>` inside a `<fixed>` creates a macro.

4.2 Message passing

Message passing sends a message with zero or more arguments to an object that executes a corresponding method. The method is typically chosen by combining the type of the object with the name (*selector*) of the message:

$$type \times selector \rightarrow method$$

Objects scan store (usually in their type) a table of methods indexed by selector, or selectors can store a table of methods indexed by type.[3] The implementation in Figure 6 chooses the latter.[4]

4.3 Generic functions

A generic function contains several function implementations. When applied, the generic function uses some property of each of its arguments to choose which implementation will be executed. The type of each argument is often used.[5]

Figure 7 shows a simple model of generic functions.[6] Tuples are organised as a sparse multi-dimensional array. Successive dimensions correspond to successive argument positions. For each dimension the tuple maps a type id to the tuple for the next argument position, until the final tuple which maps type to the implementation function.

5. DISCUSSION

The preceding compositions can be modelled in the language of Section 2 by a closure implementing dispatch through a closed-over list-based structure containing the required map. Accessing (for inspection or manipulation) the structure is one source of the "complexity and obfuscation" mentioned in the introduction, demanding knowledge of the internal structure and implementation of closures and environments.

[3]Each has advantages compared with the other. Both would be present in a comprehensive model of message passing.

[4]This breaks encapsulation, a precept of object-orientation. Order can be restored by storing an association list with each type to map selector names to methods. Then `<selector>` looks up the method and memoises it, in the structure shown here, before invoking it. Lookup need not be efficient because of the memoisation. Chaining `<selector>`s together permits enumeration to flush memoised results whenever method dictionaries are manipulated. Such an implementation of `<selector>` is one line longer than that shown.

[5]The simplest property to use is equality with a constant but the resulting behaviour is of limited use.

[6]This model uses equality to compare the list of actual argument types and the type signature associated with each implementation. If an ordering relation can be defined for types then more interesting comparisons can be used, to find the 'closest admissible' implementation when actual argument types do not precisely match an implementation function signature. As with selectors, the comparison need not be efficient because the result can be memoised in the `<generic>`'s array.

```
(define-type <selector> (name methods))

(define make-selector (lambda (name)
  (let ((self (new <selector>)))
    (set (<selector>-name    self) name)
    (set (<selector>-methods self) (tuple))
    self)))

(define define-selector (fixed (form (lambda (name)
  `(define ,name (make-selector ',name))))))

(define %add-method (lambda (self type method)
  (set-tuple-at (<selector>-methods self) type method)))

(define define-method (fixed (form (lambda (selector type args . body)
  `(%add-method ,selector ,type (lambda (self ,@args) ,@body))))))

(set-tuple-at *applicators* <selector> (lambda (self . arguments)
  (apply (or (tuple-at (<selector>-methods self) (type-of (car arguments)))
             (error "no method in "(<selector>-name self)" for "(type-of (car arguments))))
         arguments)))
```

Figure 6: Selector type and its applicative meaning function

```
(define-type <generic> (name methods))

(define generic (lambda (name)
  (let ((self (new <generic>)))
    (set (<generic>-name    self) name)
    (set (<generic>-methods self) (tuple))
    self)))

(define define-generic (fixed (form (lambda name) `(define ,name (generic ',name)))))

(define %add-multimethod (lambda (mm types method)
  (if types
      (let ((methods (or (<generic>-methods mm) (set (<generic>-methods mm) (tuple)))))
        (while (cdr types)
          (let ((type (eval (car types))))
            (set methods (or (tuple-at methods type) (set-tuple-at methods type (tuple)))))
          (set types (cdr types)))
        (set-tuple-at methods (eval (car types)) method))
      (set (<generic>-methods mm) method))))

(define define-multimethod (fixed (form (lambda (method typed-args . body)
  (let ((args  (map cadr typed-args))
        (types (map car  typed-args)))
    `(%add-multimethod ,method (list ,@types) (lambda ,args ,@body)))))))

(set-tuple-at *applicators* <generic> (lambda (self . arguments)
  (let ((method (<generic>-methods self))
        (arg     arguments))
    (while arg
      (set method (tuple-at method (type-of (car arg))))
      (set arg (cdr arg)))
    (apply method arguments))))
```

Figure 7: Generic function type and its applicative meaning function

5.1 Metacircularity and extensibility

A metacircular evaluator is self-extensible through direct manipulation of its own implementation. Eventually it must be grounded in an executable representation, terminating the recursion in its implementation. Typically this means translating all or part of the evaluator into another language. The translated parts of the evaluator become inaccessible to direct manipulation.[7] Minimising the number of, and semantic assumptions made by, these translated parts is therefore desirable.

Grounding the original evaluator of Figure 1 fixes:

- the types permissible in simple expressions (implicit combination of atom and environment),
- the meaning of simple expressions,
- the types in which complex expressions (explicit combinations of values) can be represented,
- the composition rule associated with complex expressions of each permissible type,
- the types that can appear as operators in complex expressions, and
- the semantics associated with an operator in a complex expression.

Grounding the generalised evaluator of Figure 3 fixes only the mechanism associating meaning with the type of an expression and the semantics of applying a primitive `<subr>`.

Additional grounded mechanisms (Figure 4) are needed to supply meaning for:

- atomic values (identifiers and literals),
- a composition rule for combination of explicit combination of values via a list of `<pair>`s, and
- the semantics of applying a closure `<expr>`.

Modifications to these last three mechanisms can be made by programs, with care. With almost no restrictions: new types can be given meaning as simple expressions, new types

[7] In the absence of a dynamic translator from expressions to the executable representation.

defined to combine values into complex expressions, specific meanings assigned to those combinations, new applicable types defined to serve as composition operators, and new semantics associated with those compositions.

5.2 Compositionality

In compositional languages, the meaning of a complex expression is determined from the meanings of its lexical components and the syntactic operator used to combine them [6]. In computer languages, which are usually compositional, the lexical components are expressions and the syntactic operator is the punctuation (or reserved word) that combines several expressions into a complex expression.[8] The meaning of an expression is its value or effect. The syntactic operator determines a rule of *composition* for the lexical components.

This can be written as a homomorphism between syntax and semantics [3]:

$$m(F(e_1, \ldots, e_k)) = G(m(e_1), \ldots, m(e_k))$$

Our minimal language has no syntax, so the *type* of the first expression in a combination acts as the syntactic operator F.[9] The semantic function G associated with F, and the values (meanings) of the sub-expressions, $m(e_i)$, determine the value (meaning) of the overall composition. So:

- lists (of `<pair>`s) create a combination of sub-expressions (they are not syntax),
- compositional syntax is associated with the type of the value of the first sub-expression in a combination,
- each compositional syntax has exactly one compositional semantics associated with it, and
- the compositional semantics is parameterised by the combined sub-expressions, including the value that determined the compositional syntax.

The morphology of the above rule is imposed by the function stored at *evaluators*`[<pair>]` (Figure 4) which defines the meaning of complex expressions *combined as a list of* `<pair>`*s*. Rules leading to forms of composition very different to that above can be expressed easily as new aggregate types (to combine sub-expressions) with associated meaning functions (imposing compositional forms) in *evaluators*.

5.3 Implementation

The language used in the examples, implementing the extensible types and composition rules described here, can be downloaded from: `http://piumarta.com/software/maru`

The language hosts a small compiler that translates a metacircular definition of its evaluator from S-expressions to IA32 machine code. Several composition mechanisms are defined and used in the compiler for brevity, clarity and simplicity in its implementation.

[8]Punctuation and reserved words may also be associated with declarations. These are not syntactic operators, but may nonetheless create or modify the environment in which compositions are subsequently performed. Some languages may also have a meta language, with its own set of syntactic operators, that determine the meanings of meta expressions involving declarations or types. Such meta languages are usually compositional and therefore subject to the same considerations presented here for the evaluation of "normal" expressions.

[9]So `set` and `+` are different syntactic operators, but `+` and `-` are the same.

5.4 Performance

The translation of S-expressions to IA32 machine code in the above metacircular implementation provides a convenient benchmark for measuring the cost of extensibility. The compiler was run twice, once using the original evaluator of Section 2 and again using the extensible evaluator described in Section 3. Translation took 30% longer when the compiler was run using the extensible evaluator.

6. CONCLUSIONS

A simple, metacircular, symbolic, functional language was restructured to remove assumptions about types and composition mechanisms. The original behaviour was restored by indirectly associating evaluation rules with three types and applicable behaviour with a fourth type. New composition rules can be defined in the resulting language, with no privileged status accorded to the built-in types. Additional indirections in the evaluator caused a 30% loss in performance. (Techniques beyond the scope of this paper, involving staged evaluation of expressions and their associated compositions, can more than recover this loss. Supporting such techniques flexibly was the reason for developing and refining the open composition mechanism described here.)

The restructuring follows general principles that could be adapted for any small language in which code can be manipulated under program control. The language presented here is very simple, approaching the simplest in which the restructuring for extensible composition is possible, but provides a compelling demonstration of the expressive power gained. Its metacircular evaluator, runtime library, and compiler generating IA32 machine code are expressed in less than 1800 distinct lines of code.

7. REFERENCES

[1] J. McCarthy (1960) *Recursive Functions of Symbolic Expressions and Their Computation by Machine*, CACM, Vol. 3, No. 3, pp. 184–195

[2] J. McCarthy *et al* (1961) *LISP 1.5 Programmer's Manual*, MIT AI Project, Cambridge, MA

[3] R. Montague (1970) *Universal grammar*, Theoria, Vol. 36, Issue 3, pp. 373–398

[4] I. Piumarta (2011) *Open, extensible composition models*, `http://piumarta.com/papers/freeco11`

[5] H. Stoyan (1991) *The Influence of the Designer on the Design—J. McCarthy and Lisp*, in V. Lifschitz (Ed.), *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, Academic Press Professional, Inc.

[6] Z. G. Szabó (2008) *Compositionality*, in E. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy*

Acknowledgements

The author is greatly indebted to Kita Laboratory, Kyoto University, for supporting this work. Mark Rafter and Yoshiki Ohshima provided invaluable comments on an early draft of this paper. The anonymous reviewers made many excellent suggestions of which, unfortunately, only half could be given proper consideration in the space available for this version of the paper.