Dynamic Imperative Languages for Runtime Extensible Semantics and Polymorphic Meta-programming

Anthony Savidis

Institute of Computer Science, Foundation for Research and Technology – Hellas,
GR-70013, Heraklion, Crete, Greece
as@ics.forth.gr

Abstract. Dynamically typed languages imply runtime resolution for type matching, setting-up an effectible ground for type-polymorphic functions. In statically typed object-oriented languages, operator overloading signifies the capability to statically extend the language semantics in the target program context. We show how the same can be accomplished dynamically in the Delta dynamic language, through simple member-function naming contracts. Additionally, we provide a software-pattern for dynamically extensible function semantics, something that cannot be accommodated with static function overloading. We demonstrate how meta-programming, i.e. crafting of parametric program capsules solving generic problems known as meta-algorithms or meta-components, become truly polymorphic, i.e. can accept an open set of parameter values, as far as those dynamically bind to eligible elements compliant to the meta-program design contract. In Delta, inheritance is dynamically supported as a runtime function, without any compile-time semantics, while all member function calls are resolved through late binding. We employ those features to show how Delta supports the imperative programming of polymorphic higher-order functions, such as generic function composers or the map function.

1 Introduction

Statically typed compiled languages have been widely deployed for the implementation of typical stand-alone software applications, while interpreted dynamically typed languages became mostly popular as a means to support web application development. Dynamic languages not only enable rapid development, but also facilitate far more flexible and open component reuse and deployment. The lack of static type matching enables truly polymorphic programming templates, relying on late binding and conformance to predefined design contracts, in the deployment program context. A superset of key features met in existing imperative dynamic languages like Python (http://www.python.org/), Lua (Ierusalimschy et al., 1996), and ECMA Script (ECMA, 2004), encompasses: dynamically typed variables, prototype-based runtime classes, functions as first-class values, support for unnamed functions, dynamic handling of actual arguments, and extensible operator semantics.

We extend this set of features, in the context of the Delta language, by introducing: (a) prototypes with member functions being independent callable first-class values, as

N. Guelfi and A. Savidis (Eds.): RISE 2005, LNCS 3943, pp. 113 – 128, 2006.

atomic pairs holding both the function address and the alterable owner instance; (b) dynamic inheritance, having entirely runtime semantics, in comparison to the traditional compile-time inheritance operators; (c) a programming recipe for runtime extensible function semantics; and (d) an enhanced operator overloading technique. We also show how polymorphic programming of software patterns is possible, relying on the dynamic language features. The latter is only partially accommodated in statically typed OO languages, for types conforming to predefined super-types, once the LSP (Liskov, 1988) design contract is not broken. Finally, we demonstrate the way polymorphic higher-order functions, such as function generators, are implementable through functor object instances.

1.1 Link to Rapid Integration of Software Engineering Techniques

The key contribution of dynamic languages in the context of rapid integration of software engineering techniques concerns their genuine capability to accommodate quickly advanced software patterns like: extensible semantics, higher-order functions, coupling-relieved dynamic inheritance, and polymorphic pattern programming. Due to the inherent type-dynamic nature of such languages, the transition from generic design capsules to concrete algorithmic meta-programs is straightforward, as there are no syntactic constructs that introduce unnecessary type-domain restrictions, like typed arguments, typed function signatures, or compile-time base classes.

It is argued that software engineering is by no means independent of the adopted programming language, as languages may severely affect, infect, advance, or define the particular software engineering code of practice. For instance, Eiffel (Meyer, 1997) by semantically and syntactically reflecting an innovative software design recipe requires explicitly programmers to assimilate and apply the Design by Contract method. This way, the language itself provides an effective safety net ensuring developers cannot deviate from the design prescription itself.

Dynamic languages make it implementationally easier, while syntactically more economic, to practice the implementation of generic functions and directly programmable program patterns, thus leading to easily manageable reusable code units. But in the mean time, due to complete lack of compile-time safety, they require algorithmic type-check safeguards, that, when implemented in a less than prefect manner, may unnecessarily compromise both code quality and runtime performance.

2 Prototypes as Instance Factories

In the Delta language, prototypes are runtime class values, from which instances are dynamically produced through *replication*. In this context, following the recipe of existing dynamic languages, object classes never appear within the source code in the form of compile-time manifested types, but only as first-class runtime values called *prototypes*. The characteristics of prototypes in the Delta language are:

- They are associative table objects, having no prototype-specialized compile-time or runtime semantics; prototypes are normal object instances, chosen by programmers to play the role of class-instance generators, thus prototypes are effectively a design pattern combined with a deployment contract;

- There are no reserved constructor functions; construction is implemented through programmer-decided *factory* member functions, primarily relying on instance cloning.

Associative tables constitute the sole object model in the Delta language, offering indexed member access through late binding; *member* functions are allowed inside associative-table construction expressions, i.e. enumeration of member elements between `[` and `]`. Such table construction expressions are called *prototype definition* expressions. Member functions have the following key properties:

- They are typical table members, associated by default to the constructed table instance; however, the owner instance of a member function can be dynamically altered, while it is not required to be an instance of the original prototype, i.e. the prototype whose definition syntactically encompasses the member function definition;
- New member functions can be also installed to a table instance dynamically, i.e. outside the syntactic context of the respective prototype definition;
- Within member function definitions, the keyword *self* always resolves dynamically to the runtime owner table instance;
- The value of a member function itself is directly callable, internally, being an atomic pair of the owner instance and function address; upon call, members will resolve *self* references to their owner instance value. This way, member function calls do not syntactically require an object instance expression, as it is the case with C++, Java, ECMA Script or Lua.

The dotted syntax, e.g. `p.x` is syntactically equivalent to `p["x"]`, where member `x` binding within `p` always takes place during runtime. This is similar to name-based late binding in Lua methods and ECMA Script member functions. This dynamic form of late-binding can be openly deployed for any object instance, once the object caters to dynamically resolve to the referred named members. However, this behavior is not accomplishable in statically typed languages, as compile-time conformance of the object instance and the referenced member is required to a specific type inheritance hierarchy and function signature, respectively. Moreover, late binding in dynamic languages is straightforward for data members too, something that is not accommodated in statically typed languages. A *member* function in Delta is an unnamed value; it is referenceable through the programmer decided index value. A function definition inside parenthesis, like for instance `(member() {return copy(self);})`, is a function value expression, internally carrying the function address; the same form is applicable to non-member functions as well.

In Fig. 1, one of the possible ways to implement prototypes in Delta is outlined. Following this method, prototypes are stored in static local variables, inside their respective prototype-returning function, e.g. `PointProto()`. The prototype is constructed as a model-instance, offering a set of members chosen by the programmer. It should be noted that none of the following member function names appearing in Fig. 1, like `clone` instance production function, `new` constructor function, and `class` reflecting the prototype name, is enforced by the Delta semantics, but those are freely chosen by the programmer.

```
function PointProto() { // Prototype extraction function
       static proto;
       if (typeof(proto)=="Undefined") // First time called.
              proto = [
                     {"x", "y" : 0 },
                     {"class"  : "Point" },
                     {"clone"  : (member(){return copy(self);})}
                     {"new"    : (member(x,y) { // Constructor.
                                    p = self.clone();
                                    p.x = x, p.y = y;
                                          return p;
                                    } ) }
              ];
       return proto;
}
p1 = PointProto().new(30, 40);   // Via prototype constructor
p2 = p1.clone();                 // By instance replication
fc = p1.clone;                   // Getting p1 "clone" member
p2 = fc();                       // Calls p1 "clone" member
```

Fig. 1. Examples of simple prototype implementation and use

2.1 Details on Object Oriented Extensions to Deployment of Associative Tables

Associative tables (or tables) play a key role in the Delta language: (a) they are the only built-in aggregate type; and (b) they provide the ground for object-oriented programming. Tables are stored in variables by reference, so assignment or parameter passing semantics does not imply any kind of copy, while comparison is also done by reference. Within a table, indexing keys of any type may be used to store associated values of any type. Tables grow dynamically; they can be constructed through a table constructor expression, while individual elements can be easily added or removed. The expression `[]` constructs an empty table, while `[ {"x":0} ]` makes a table with a single element, with value `0`, indexed by the *string* key `"x"`. Table instance elements can be removed by setting the corresponding value to *nil*, implying that *nil* cannot be stored within a table. Hence, `t.x = nil;` causes the entry indexed by key `"x"` within `t` to be directly removed. Finally, the following library functions are provided:

- *tabindices*(t_1), returning a new constructed table t_2 where: $\forall$ pair of index and associated value $(K_j, V_j) \in t_1$, $j \in [0, N)$, N being the total stored values in t_1, the pair (j, K_j) is added in t_2. That is, a table with all keys indexed by ordered consecutive integer values is returned. The way the ordering of keys is chosen is implementation dependent (i.e. undefined).
- *tablength*(t_1), returning N being the total stored values in t_1.
- *tabcopy*(t_1), returning a new constructed table t_2 where: $\forall$ pair of index and associated value $(K_j, V_j) \in t_1$, $j \in [0, N)$, N being the total stored values in t_1, the pair (K_j, V_j) is added in t_2. That is, an exact copy of t_1 is returned. The *tabcopy* function is implemented by using *tabindices* and *tablength* as follows:

```
function tabcopy(t1) {
   for (t2=[], ti=tabindices(t1), n=tablength(ti)-1; n>=0;
--n)
       t2[ti[n]] = t1[ti[n]];
   return t2;
}
```

In prototype definitions through associative tables, there is no support for a built-in destructor function in the Delta language. Although Delta is a language supporting automatic garbage collection, it was decided to separate memory disposal, taking place when tables can no longer be referenced via program variables, from the particular application-specific object destruction or clean-up logic. This decision is backed-up by the following remark:

- Application objects need to be cancelled exactly when the application logic decides that the relevant destruction conditions are met. In such cases, all corresponding cancellation actions, which actually implement the application-specific policy for the internal reflection of the cancellation event, are performed as needed. Once application-specific actions are applied, memory disposal takes place only at the point there is no program variable assigned to particular the subject object instance. Hence, it is clear that memory disposal is semantically thoroughly separated from application-oriented object lifetime control and instance cancellation (i.e. destruction).

It should be noted that the use of *tabcopy* should be avoided when there are member functions in tables, since their internal owner table reference is not changed but is copied as it is. Instead, the *copy*(t_1) should be employed, which in addition to *tabcopy*, performs the following:

- Let t_2 be the returned copy of t_1. Then, $\forall$ member function value $(F,T) \in t_1$, if $T = t_1$, then add (F, t_2) in t_2 else add (F, t_1) in t_2. In other words, member function values of the original table become member function values of the table copy. The functioning of *copy* is not recursive, meaning in case of member instances programmers have to take care for proper instance *copy* as well.

3 Dynamic Inheritance

Inheritance is based on dynamic associations of the form $\alpha \swarrow \beta$ (α derived from β) and $\beta \nearrow \alpha$ (β inherits to α), to reflect that table instance α inherits directly from table instance β. This association defines an inheritance tree, where, if γ is a predecessor of δ, then we define that δ is derived from γ, symbolically $\delta \leftarrow \gamma$, while γ is also said to be a *base* instance for δ. The establishment of an inheritance association $\alpha \swarrow \beta$ is regulated by the precondition:

$$\alpha \neq \beta \wedge \neg \beta \leftarrow \alpha \ \wedge \neg \ (\exists \gamma : \ \gamma \neq \beta \wedge \ \gamma \swarrow \beta)$$

This precondition formalizes the fact that an instance: (a) cannot inherit from its self; (b) cannot inherit from any of its derived instances; and (c) can inherit to at most one instance. In the Delta language, the following basic library functions are provided for dynamic management of inheritance associations among table instances:

- *inherit*(t_α, t_β), which establishes the associations $t_\alpha \swarrow t_\beta$ and $t_\beta \nearrow t_\alpha$
- *uninherit*(t_α, t_β), which cancels the association $t_\alpha \swarrow t_\beta$ and $t_\beta \nearrow t_\alpha$
- *isderived*(t_α, t_β), returning, *true* if $t_\alpha \leftarrow t_\beta$, else *false*

In Delta, inheritance is a runtime function applied on instances, establishing an augmented member-binding context for derived instances. The metaphoric *isa* connotation of base and derived classes are not entirely adopted in Delta, since *inherit*(*x*, *y*) doesn't state that *x isa y*, neither that *x* depends implementationally on *y*; it only defines augmented member binding for both *x* and *y*, i.e. if a member requested for *x* or *y* is not found in *x* (derived), then try to find it in *y* (base).

3.1 Dynamic Virtual Base Classes

In a given inheritance hierarchy *I* with most derived class *C*, a virtual base class *B* is a class required to be inherited only *once* by *C*, irrespective of how many times *B* appears as a base class in *I*. In statically typed OOP languages, compilers "know" the static inheritance hierarchy, so they construct appropriate memory models for derived classes having a single constituent instance per virtual base class. In the context of dynamic inheritance, the same behavior is accomplished with the special form of virtual inheritance programmed as shown in Fig. 2.

```
function virtually_inherit(derived, base) {
       t = allbaseinstances(derived);
       for (n = tablength(t) - 1; n >= 0; --n)
              if (t[n].class == base.class)
                     return;
       inherit(derived, base);
}
```

Fig. 2. Implementation of virtual dynamic inheritance

Its implementation uses the `allbaseinstances(x)` library function, returning a numerically indexed table encompassing references to all base instances of `x`. The function `virtually_inherit` is actually supplied as a library function in Delta for convenience. In the implementation of Fig. 2, we need only seek for a base instance whose class name matches the supplied base instance argument. If such an instance is found, i.e. `derived` already inherits from `base`, inheritance is not reapplied. However, we have also extended the `virtually_inherit` library function to enable dynamically the conditional update of the current virtual base instance with the supplied `base` argument.

3.2 Member Resolution in Dynamic Inheritance Chains

Inheritance associations define an augmented way for late binding of instance members, reflecting the fundamental priority of member versions in *derived* instances over the member versions of *base* instances, within inheritance hierarchies. Additionally, programmers may qualify member bindings as *bounded*, when there is a need to employ the original member versions of base instances, as opposed to the refined ones. The member-binding algorithm is a tree search algorithm, as shown in Fig. 3.

```
bind (t, x, bounded) {
        if (bounded = true and x ∈ t ) then
                return t.x
        V = {}              /* V holds visited base instances */
        r = resolve({ t.root }, x)   /*'root' denotes the most derived instance */
        if ( r ≠ nil ) then
                return r.x
        else
                return nil
}
resolve (S, x) {
        L = {}    /* Set of all base instances for the instances of S*/
        for ( each t ∈ S where t ∉ V ) do {
                if ( x ∈ t ) then
                        return t
                V = V ∪ { t }
                L = L ∪ t.base    /* 't.base' is a set of 't' base instances */
        }
        if ( L ≠ ∅ )  then
                return resolve(L, x)
        else
                return nil
}
```

Fig. 3. Member binding logic within instance inheritance chains; notice that *root* denotes the most derived instance in an instance inheritance hierarchy

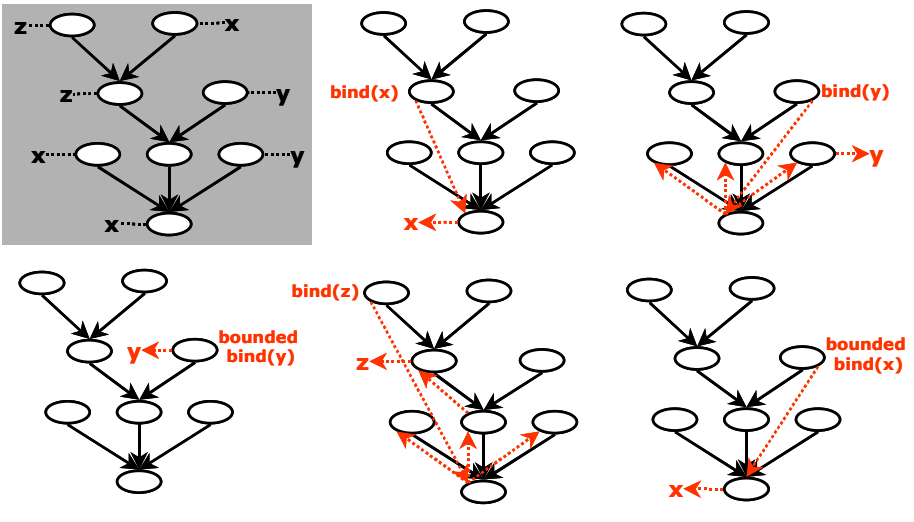

Fig. 4. Examples of member search paths (dotted arrows) for late binding of different members in an instance inheritance hierarchy; the shaded tree indicates the real member storage, while the most derived instance is actually the *root* instance

Following Fig. 3 (left), in case the *bounded* flag (i.e. bounded use) of the *bind* function is *true*, resolution of x directly in instance t is performed. Otherwise, i.e. $\neg$ *bounded* $\vee$ $x \notin t$, the resolution function *resolve* is called, which performs a breadth-first search starting from the root, i.e. the most derived instance in the runtime inheritance tree. This search always returns the first member resolution closest to the inheritance root (most derived instance). In Fig. 4, a few examples are provided regarding the alternative search paths, to resolve particular members within an instance inheritance hierarchy. The distinction of table instances into either object prototypes or object instances is a semantic separation in the context of the program design, not reflecting any particular built-in language semantics for associative tables. Similarly, the semantics of the inheritance-association management functions concern table instances in general, without any operational differentiation for either object prototypes or object instances. This feature allows:

- *Dynamically installable/removable inheritance*, facilitated by connecting/disconnecting a complete instance inheritance sub-hierarchy to/from the target instance, through a call to *inherit*/*uninherit* library function.

4 Function Overloading Pattern

The deployment of unnamed functions, dynamic manipulation of actual arguments, runtime type identification, and associative storage, allows the implementation of a

```
function sig(t) {
   for (s = "", n = tablength(t), i = 0; i < n; ++i)
       s += typeof(t[i]);
   if (s == "")   return "void";
   else           return s;
}

function overloaded() {
   static dispatcher;
   if (typeof(dispatcher) == "Undefined")
       dispatcher = [
           {"NumberNumber" : ( function(x,y) {...}) },
           {"StringString" : ( function(a,b) {...}) },
           {"Table"        : ( function(t)   {...}) },
           {"void"         : ( member() { return self; })},
           {"install"      : ( member(sig, f) {
                                 self[sig] = f; })}
       ];
       return dispatcher[sig(arguments)] (|arguments|);
}

function added (x, s) {...}
overloaded().install("NumberString", added);
```

Fig. 5. Dynamic function overloading for extensible function semantics

dynamic function-overloading pattern, relying on runtime management of alternative function versions through string-based signatures. It is a software pattern in the sense that it is not a built-in language mechanism, but an accompanying language-specific programming recipe for dynamically extensible function semantics. The programming pattern for function overloading is illustrated in Fig. 5, with an example function supporting three alternative signatures. As it is shown, overloaded functions encapsulate a static local dispatch table, named `dispatcher`, storing the alternative

```
function sig(t) {
       for (s = "", n = tablength(t), local i = 0; i < n; ++i)
              if (typeof(t[i])=="table" and t[i].class != nil)
                     s += t[i].class;
              else
                     s += typeof(t[i]);
       return s=="" ? "void" : s;
}


function metaconstructor() {
       return [
              {"construct" : (member() {
                     f = self.constructors[sig(arguments)];
                     return f(|arguments|);
              })}
       ];
}
proto = [
   {  "constructors" : [
        { "void" : (function(){ return copy(proto); }) },
        { "numbernumber" :  // Parameterized constructor
          (function(x,y){
              p = copy(proto); p.x = x; p.y = y; return p; }) },
        { "Point" :         // Copy constructor
         (function(pt){ return proto.construct(pt.x, pt.y); })}
       ]
   }
];
inherit(proto, metaconstructor());

function midpointconstructor(p1,p2) {
       p = copy(PointProto());      // Instantiate from prototype
       p.x = (p1.x+p2.x)/2;         // Initialize members
       p.y = (p1.y+p2.y)/2;
       return p;                    // Return the new instance
}

// Installing the constructor at the prototype
PointProto().constructors.PointPoint = midpointconstructor;
```

Fig. 6. The meta-constructor pattern for dynamic constructor overloading. Notice that to allow dynamically installed constructors, those are turned to non-member functions.

implemented versions as embedded unnamed functions. The actual argument expression |*table*| unrolls all elements of *table*, as if those where supplied by distinct actual expressions (i.e. "pushed" on the arguments' stack); this is similar to the * operator for sequences in Python. Also, `arguments` is a reserved local variable (of table type) carrying all actual arguments of the current call (numerically indexed). Thus, `|arguments|` propagates the actual arguments of the present call to an encapsulated delegate function invocation.

If `overloaded` is called without arguments, it returns a handle to the internal dispatcher table, offering the `install` member function to dynamically add / remove / update a function version for arguments signature `sig` (in Delta, removal of a table element is equivalent to setting `nil` as the element value).

4.1 Dynamic Constructor Overloading

Overloaded constructors basically follow the dynamic function-overloading pattern previously discussed (see Fig. 5). Additionally, such alternative constructors can be dynamically extensible, meaning argument conversion and instance initialization functions can be installed on the fly as needed, either at an instance or prototype level.

We will slightly modify the `sig` function which extracts the type-signature of the actual argument list to cater for object-instance arguments in the following manner: if an argument is of type `"table"`, then if it has a `"class"` member, its value is asserted to be of `"string"` type and its content is returned as the type value; else, the `"table"` type is returned. As it is shown in Fig. 6, all constructors are dynamically collected in one member-table of the object prototype named `"constructors"`, while the dynamic installation of a particular constructor requires the provision of a unique signature and a corresponding constructor function. Also, the meta-construction functionality is named `"construct"`, internally dispatching to the appropriate signature-specific constructor, is implemented as an inherited member function, meaning it can be directly re-used. One important modification of this overloaded constructor in comparison to the function-overloading pattern is that the overloaded constructor functions are now non-member functions. The latter is necessary once we decide to allow dynamically installable constructors, effectively requiring that such constructor functions can be defined externally to table constructor definitions, i.e. being non-member functions.

5 Operator Overloading Contract

In Delta the semantics of all binary operators are dynamically extensible for table object instances through the following implementation technique:

- *eval*(t_1 *op* ^{binary} t_2). If there is a t_1 member named *op* being actually a function *f*, the result of evaluation is $f(t_1, t_2)$. Otherwise, the original semantics for t_1 *op* t_2 are applied.
- *eval*(*op* ^{unary} t_1). If there is a t_1 member named *op* being actually a function *f*, the result of evaluation is $f(t_1)$. Otherwise, the original semantics for t_1 *op* t_2 are applied.

In the current implementation, this method applies to most binary operators in Delta, like arithmetic and associative operators, as well as the function call `()` and the table

member access operators. For prefix and postfix unary operators **--** and **++**, the **+** and **-** binary operators need to be only overloaded. Boolean operators are excluded as short-circuit boolean evaluation diminishes boolean operators from the target code. However, different Delta implementations may override short-circuit code and introduce boolean instructions in the virtual machine, meaning overloading can be also supported in this case. Regarding table member access, we distinguish among read / write access through "**[]**" (read access) and "**[]=**" (write access), also covering the use of "**.**" supplied in place of "**[]**" for syntactic convenience. Once table member access is overloaded, the native operator is hidden unless the member is temporarily removed and reinstalled again; this is possible with the explicit non-overloaded member access functions `tabget` and `tabset`. Finally, for unary operators: **not** requires overloading of **!=**, and unary minus requires overloading of multiplication operator with numbers (`-x` it is calculated as `x*-1`).

To make binary operators more efficient we distinguish the position of the primary table argument with a dot, so there are two member versions; e.g. `".+"` and `"+."`. The operator overloading approach in Delta is very simple, yet very powerful. *Overloaded operators constitute normal members distinguishable uniquely through a naming contract being part of the language semantics*. This makes operators directly derivable through dynamic inheritance, since, as normal object members, they are also subject to late binding; finally, operator functions as first-class values are dynamically extractable, removable or substitutable. An example showing operator overloading is provided in Fig. 7.

```
function Polygon() {
    static proto = [
       {"area"       : (member(){...})},
       {".<="        : (function(p1,p2) {
                              return p1.area() <= p2.area(); })},
       {".+"         : (function(p,x)   {
                              p[".+dispatch"][sig(x)](p,x); })},
       {".+dispatch": [
              {"Point"  : (function(a,b){…})},
              {"Number" : (function(a,b){…})},
              ]},
    ];
}

p1 = Polygon().new();
p2 = Polygon().new();
if (p1 <= p2)
       p1 = p2 + 10;
```

Fig. 7. Dynamic operator overloading for dynamically extensible language-operator semantics

6 Polymorphic Pattern Programming

Software patterns are defined as recurring solutions to common design problems mostly provided as recipes having a standardised documentation, rather than as

directly reusable code. Since software patterns constitute meta-solutions, the capability to turn their documentation to an equally generic programmed artefact is really a matter of appropriate abstraction choices in the context of pattern implementation, and effective support for polymorphism in the context of pattern deployment. Theoretically, patterns are meta-programs, where *meta* accounts to type abstraction and polymorphism for constituent content or logic elements. Arguably, once the necessary type-abstraction and type-polymorphism support is provided, polymorphic pattern programming is directly accomplishable. It is clear that to enable generic polymorphism, the compile-time matching barrier needs to be effectively bypassed.

```
// Returns an instance of the 'State' pattern.

function StatePattern() {
    return [
       { "setstate" : (member(newState) {
          uninherit(self, self.State);
          inst = prototypes[self.class].States[newState].new();
          inherit(self, self.State = inst);
       })}
    ];
];

inherit(a, StatePattern());
a.setstate("foo");
```

Fig. 8. The reusable polymorphic *State* pattern implementation

We demonstrate the capability for polymorphic pattern programming for the *State* pattern (Gamma *et al*., 1995), concerning classes supporting runtime updateable behaviors, the latter implemented as distinct classes. It is interesting to note that the *State* pattern implicitly exposes the need to support dynamic inheritance, since the *State* pattern was born as a design recipe to craft classes conditionally reflecting, during runtime different behavioral pictures. The implementation of a directly deployable polymorphic *State* pattern is shown in Fig. 8. Following Fig. 8, we choose to store at runtime any state-related prototype named *S*, for class-specific prototype named *A*, within `prototypes[A].States[S]`. The State pattern logic is actually consolidated in a single function performing the following actions:

- Cancels the inheritance association with the current base *State* instance `self.State`;
- Makes a new instance corresponding to the prototype of the new state, that is `prototypes[self.class].States[newState]`;
- Establishes an inheritance association with the new base *State* instance, while setting the current *State* name, i.e. `inherit(self, self.State = inst)`;

The runtime associations for the *State* pattern are shown in Fig. 9. The *"owns"* label indicates the instance in which members are actually stored, *"binds"* denotes members resolved via late-binding to a base / derived instance, while *"refers"* signifies members being instance references.

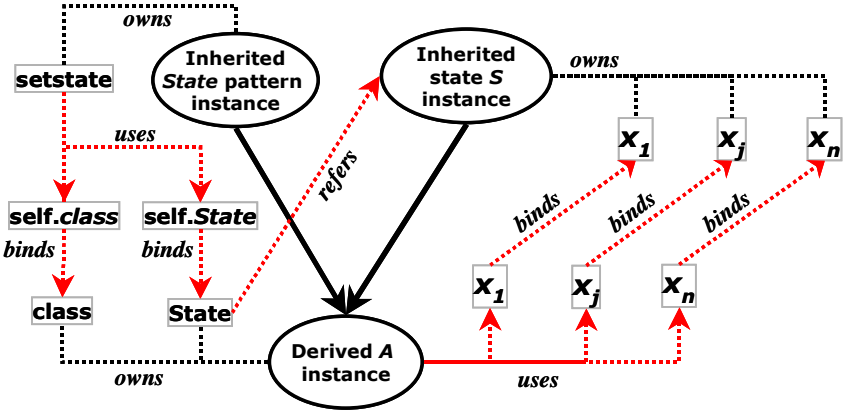

Fig. 9. The runtime associations for the *State* pattern and the way the various members are dynamically resolved; notice that the state-specific instance constitutes a dynamic base instance, rather than a delegate local instance as it is in the original implementation recipe (Gamma *et al*., 1995)

7 Polymorphic Higher-Order Functions

Higher-order functions are functions taking functions as arguments and / or delivering functions as results; the most challenging case concerns function generators. Functional programming languages like Haskell (Peyton Jones, 2003) or Scheme (Abelson *et al*., 1998) genuinely support the definition of generic (polymorphic) higher-order functions through the λ-lambda operator. Although dynamically typed languages easily overpass the signature-checking barrier, they only manage to allow polymorphic function generators once functions are treated as object instances by the underlying implementation, with their own call-persistent data members. In Python this is possible with an interpreted implementation, however compiled dynamically typed languages like Lua (Ierusalimschy et al., 1996) fail in this respect because they do not provide a truly object-oriented model for functions. In Delta, the implementation of

```
function compose(f, g) {
   return [
       { "f"  : f },
       { "g"  : g },
       { "()" : (member(){
              return self.f(|self.g(|arguments|)|); })}
   ];
}
mul          = function(x,y){ return x*y; });
sqrpair      = function(x,y){ return [ sqr(x), sqr(y) ]; });
mulsquares   = compose(mul, sqrpair);
x            = mulsquares(3, 7));
```

Fig. 10. A polymorphic function composer

function generators is straightforward: *they are functions returning table instances overloading the function call* `()` *operator*; *the latter are commonly called functors.* In Fig. 10, the implementation of a generic function composer is shown.

The `compose` function returns a table object instance which stores in local members the two functions (those need not be "normal" functions, but can be functors as well), while also overloading the call operator `()`, binding to a an appropriate member function. This member function performs firstly a call to *g*, propagating to it the actual arguments of the composed function via the `|arguments|` expression. Then, the return values of this call, collected in a table, are supplied as actual arguments to the *f* call, as `|self.g(…)|`. The result of *f* invocation is by definition the correct result of the composition.

Next we present the imperative implementation of three additional key polymorphic higher-order functions in the Delta language (see Fig. 11): (a) the *mapping* function, applying an argument function to all entries of a table; (b) the *const* function, transforming a value parameter to a mathematical constant function (i.e. always returning this value when called); and (c) the *delayed call* function, which accepts a function and its actual arguments, returning a function which is equivalent to this call.

```
function map(f, t) {
   for (ti = tabindices(t), n = tablength(ti)-1; n >= 0; --n)
        t[ti[n]] = f(t[ti[n]]);
}
function sqr(x) {
      if (typeof(x)=="number") return x*x; else return x;
}
t = [0, 1, 2, 3, 4, {"x", "y" : 4}, {"name" : "t"} ];
map(sqr, t); // Affects only numeric values, for all indices
function const(c) {
      t = [ {"c" : c}, {"f" : (member(){ return self.c; })} ];
      return t.f;
}
c_10 = const(10);
print(c_10());              // Prints "10"
c_hello = const("hello");
c_hello();                  // Prints "hello"

function call(f) {
      for (args=[], n=tablength(arguments) - 1; n > 0; --n)
            args[n-1] = arguments[n]; // Shift indices left
      t = [  { "args" : args }, { "f" : f },
            { "call" : (member(){
                  return self.f (|self.args|); })}
          ];
      return t.call;
}
c = call(compose, mul, sqrpair);
print(c()(3, 2));   // Prints "36"
```

Fig. 11. Examples of additional polymorphic higher-order functions, with an imperative implementation in the Delta language

The programming of such higher-order functions in the Delta language is enabled by the deployment of two key features: (i member functions as distinct first-order values internally carrying both the member function address and the associated table instance; and (ii) late binding of actual arguments, supporting "transit" actual argument passing in a functional programming style. For instance, every call to the `const` higher-order function (see Fig. 11) constructs a table encompassing both the supplied value indexed by `"c"`, and a member function indexed by `"f"`, the latter returning the `"c"` member of its associated runtime table; effectively, the `const` function returns the member function value of the newly constructed table, i.e. a pair of the function address and constructed table instance.

8 Discussion and Conclusions

The benefits of introducing enhanced dynamic-language features, towards directly deployable polymorphic program capsules, can be argued and demonstrated, however, they cannot be largely predicted and projected. Since we lack theoretical frameworks to assess the computational necessity of constructs like polymorphic higher-order functions, dynamically extensible function semantics, or dynamic inheritance hierarchies, in an imperative programming context, it is hard to formally prove that their introduction always leads to an enhanced code of programming practice. Intuitively, truly dynamic languages enable a more natural and convenient mapping of abstract designs to source code units, while effectively enabling the accommodation of computable design decisions injected in the runtime logic, as static invariant associations and dependencies are diminished. This remark implies that there is a very strong impact of truly dynamic imperative languages on the software engineering of meta-programs and polymorphic code capsules. Practically, the main implications lay on the fact that meta-elements and parametric polymorphism become directly implementable, turning design patterns and software recipes to concrete program units. However, increased flexibility is usually paid by decreased safety. This is also true for the Delta language, as the programming flexibility offered by dynamic typing has to be eventually paid by the manual embedding of runtime type checking logic. This implies that all potential type conflicts are only detectable during runtime, meaning that the test units have to be designed in a way ensuring the exhaustive execution of all type safety guards. In this context, the dynamic function-overloading pattern provides a standard entry point to attack type conflicts, as well as potential functional extensions, either during development (manually encapsulating functions) or during runtime (signature-based installation of overloaded functions). While at present the object-oriented support offered by the Delta language is primarily focused on re-usability and polymorphism, the language misses the ingredients to facilitate encapsulation and information hiding. Although programmers currently follow the software pattern of accessing member variables only through member functions, language extensions may need to be introduced to support typical member access qualifiers such as `private` or `const`, to guard pattern conformance during execution. However, considering the semantics of Delta tables, such guards can be only accommodated in the form of runtime member-access check-points, meaning that more testing code is needed, to ensure that "information hiding" related qualifiers are always respected.

References

H. Abelson, R.K. Dybvig, C.T. Haynes, G.J. Rozas, N.I. Adams IV, D.P. Friedman, E. Kohlbecker, G.L. Steele Jr., D.H. Bartley, R. Halstead, D. Oxley, G.J. Sussman, G. Brooks, C. Hanson, K.M. Pitman, M. Wand (1998). *Revised Report on the Algorithmic Language Scheme*. Journal of Higher-order and Symbolic Computation, 11(1), pp. 7–105.

ECMA (2004). ECMA Script Language Specification. Available electronically from: http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf.

Gamma, E., Helm, R., Johnson,R., Vlissides, J. (1995). Design Patterns: Elements of Re-Usable Object-Oriented Software. Addison-Wesley.

Ierusalimschy, R., Henrique de Figueiredo, L., Celes Filho, W. (1996). Lua – an extensible extension language. Journal of Software Practice & Experience 26(6), pp. 635–652.

Liskov, B. (1988). Data Abstraction and Hierarchy, *SIGPLAN Notices*, 23,5 (May, 1988).

Meyer, B. (1997). *Object-Oriented Software Construction - Second Edition*. Prentice Hall, Santa Barbara, CA.

Peyton Jones, S. (2003). *Haskell 98 Language and Libraries*, Cambridge University Press.