The Delta Dynamic Object-Oriented Programming Language

ANTHONY SAVIDIS

Institute of Computer Science,

Foundation for Research and Technology – Hellas,

Heraklion, Crete, GR-71110 GREECE

as@ics.forth.gr

The Delta language

A. Savidis, 2005
Abstract

Delta, i.e. Dynamic Embeddable Language for Extending Applications, is an imperative object-oriented scripting language, designed as an instrument for programmable postproduction extension of software systems. Applications frameworks written in native programming languages can export an arbitrary number of library functions to the Delta language. Programs written in Delta are compiled to portable virtual-machine instructions, while applications may encapsulate and call the virtual machine system to load and execute externally attached Delta compiled code units. Such loaded virtual machine instances can be queried on the fly in Delta programs, subsequently employed for cross virtual-machine function calls. The Delta language exhibits dynamic typing not only for typical program variables, but also for object instances, meaning that class types appear always in the form of first-class values called prototypes, enabling instance creation through programmer-defined factory methods. Inheritance is dynamically supported as a runtime function, without any compile-time semantics, while all member function calls are resolved through late binding. Finally, the Delta language provides elements for the imperative programming of polymorphic higher-order functions, such as generic function composers or the map function.

Keywords


The Delta language  
A. Savidis, 2005
1 Introduction

The Delta language, i.e. Dynamic Embeddable Language for Extending Applications, is a scripting language developed to facilitate postproduction extension of application frameworks, ensuring that the application-core source code in the native language can stay completely unaffected. Such an instrument is particularly critical when there is a basic application kernel constituting a resource of mainly invariant added-value functionality, while the rest of the application logic tends to be more fluid and extensible, while, from a software-engineering perspective, is usually less demanding. The design of such scripting languages is usually driven by the needs: (a) to enable a lower-entrance barrier for application extension programmers; (b) to eliminate the runtime propagation of potential software defects in application extensions to the basic application engine; and (c) to allow application extensions or modifications to be easily created, communicated, installed, and deployed.

![Figure 1: The architecture for postproduction application extensions using Delta. Solid arrows indicate calls, while dashed arrows concern loading of code units.](image)

The outline of the architectural infrastructure for postproduction extension of software applications through Delta is provided in Figure 1. The Delta language is compiled to Delta virtual machine code, being platform-neutral target code like the Java byte code,
which can be executed by the Delta virtual machine system. Extensible applications always encompass the implementation of the Delta virtual machine system. The core application functionality is made available to application extension programmers in the form of Delta library functions written in the native application programming language. The entry point for postproduction application extensions is the Delta primary compiled unit, being a precompiled Delta program at a predefined path and with a predefined file name. As it will be explained later, through the programming of the primary unit, an arbitrary number of additional compiled units may be dynamically loaded and run.

<table>
<thead>
<tr>
<th>No</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statically typed variables</td>
<td>Statically typed values</td>
</tr>
<tr>
<td>Type information in function signatures</td>
<td>Optional argument-name lists</td>
</tr>
<tr>
<td>Compile-time function overloading</td>
<td>Dynamic handling of alternative actual argument lists</td>
</tr>
<tr>
<td>Access qualifiers for class members</td>
<td>All members are public</td>
</tr>
<tr>
<td>Virtual qualifier for member functions</td>
<td>All members are resolved with late binding</td>
</tr>
<tr>
<td>Compile-time object classes</td>
<td>Runtime prototype values</td>
</tr>
<tr>
<td>Compile-time inheritance</td>
<td>Inheritance as a runtime function</td>
</tr>
<tr>
<td>Built-in class constructor</td>
<td>Programmer defined factory patterns</td>
</tr>
<tr>
<td>Explicit instance memory management</td>
<td>Automatic with garbage collection</td>
</tr>
<tr>
<td>Built-in base instance access method</td>
<td>Programmer defined access patterns</td>
</tr>
</tbody>
</table>

Figure 2: The typical features regarding statically typed object-oriented programming support (left), with the alternative incarnations in the Delta language (right).

The design of the Delta language has been based on specific technical objectives, while generally aiming to marry deployment flexibility and syntactic convenience with comprehensive programming support. As a language that can be deployed to manipulate, customize and automate the facilities of an existing system, it belongs to the general family of scripting languages. In this domain, it is believed that it can serve either as an advantageous alternative, or as a proposition for some specific enhanced programming features. The key desirable features of the Delta language, with juxtaposition to typical programming features of statically typed object-oriented
programming languages are shown in Figure 2. The decision to eliminate access
qualifiers was taken since Delta was not originally targeted to supporting large-scale
system development, where encapsulation and information hiding do play a critical
software engineering role. Instead, Delta was designed for the programming of
localized dynamically loaded application extensions, putting particular emphasis on
deployment flexibility and functional polymorphism, with syntactic simplicity. As it
will be discussed at the end, the incorporation of runtime access qualifiers is one of
the possible planned extensions.

2 Related work

Previous work is mainly related to languages commonly known as scripting
languages, most of which are currently widely used and deployed. Many of the
common programming features met in the Delta language have been borrowed or
inspired from such languages.

2.1 Lua

The Lua language, see (Ierusalimschy et al., 1996) and (Lua, 2004), is a powerful and
easy to use scripting language supporting dynamic typing, garbage collection and
extensible semantics (for specific operators) through installable methods (called meta-
tables now, known as tagged methods before), while it offers a C Application
Programming Interface (API) for linkage to C / C++ programs. Lua currently offers
some object-oriented programming features, like: (a) enabling functions to be
collected as normal table fields, implying that they can be syntactically deployed as if
they were class member functions; (b) supporting genuine table methods, with a built-
in implicit self argument carrying the host table reference, however, requiring a
differentiated deployment syntax; and (c) suggesting a hand-made approach to instance inheritance relying on programmer-installed fallbacks for table member access. In Lua, the differentiated syntax for calling member functions gives evidence that it is employed to generate methods as normal function calls, internally introducing automatically as a standard argument the host table. Consequently, member functions do not reflect a new type of first-class values, as for instance in C++, but are similar to normal non-member functions, a fact that, as we will show, limits the capabilities to program generic higher-order functions. Additionally, Lua does not reflect inheritance-specific semantics since it relies on hand-crafting of inheritance hierarchies, it does not support dynamic loading of compile code units into virtual machine instances, neither cross virtual machine function calls. However, in the design of the basic features of the Delta language, various elements of Lua, like unnamed functions and associative tables, have been employed.

2.2 ECMA Script

The ECMA Script language (ECMA Script, 2004) was based on several originating technologies, mainly the well known JavaScript (by Netscape Corp.) and JScript (by Microsoft Corp.) de facto industry standards. The latter have been widely employed for client-side programming, i.e. web document implementation, providing built-in access to the hosting document structure, compliant to the Document Object Model (DOM) specification. However, ECMA Script is not tied to web documents, since it was designed to allow embedding in any host environment for client-side computations. ECMA Script is object-based supporting prototype-based inheritance with constructor functions accepting explicit prototype parameters. The inheritance path towards the topmost prototype is called a prototype chain. One of the negative aspects in ECMA Script is arguably the awkward syntax for implicit class object
definitions through constructor functions, not allowing member functions to be encapsulated in one specification unit. Also, although inheritance is performed at runtime, there are no further facilities to dynamically affect inherited elements, as we will describe in Delta, e.g. to add or remove members, or to inherit from other prototypes after construction. In the design of Delta, many of the prototype-based inheritance features of ECMA Script have been absorbed.

2.3 Unreal Script

Developed by Epic Mega Games Inc. (Sweeney, 1988), is a statically typed domain-reflecting imperative object-oriented language with broad industrial success. It constitutes the primary instrument assisting in the development of interactive scenes for the Unreal™ game franchise, while it allows postproduction extension of the relevant game titles enabling users to add new game scenes and arenas, i.e. the so called “mod” game extensions. Unreal Script borrows elements from C++ and Java, while reflecting game-domain elements like lighting, textures, actors, states, timing, etc., into built-in language constructs. In this sense, it followed a different design path in comparison to the rest of general-purpose scripting languages, by turning typical library objects and functions to explicit language elements.

2.4 ActionScript

Produced by Macromedia Corp. (Action Script, 2004), is the language for developing interactive multimedia applications, compiled to the Flash Player multi-platform binary executable format. It is a statically typed object-oriented imperative language, supporting compile-time inheritance, with a rich set of built-in components for managing multimedia and animation, exported by the Flash advanced multimedia engine. The Flash engine actually constitutes the main execution environment for

*The Delta language*  
A. Savidis, 2005
ActionScript programs. The ActionScript language aims to support small-to-medium scale multimedia applications, emphasizing partial type-safety (weak typing) with static-type checking and compile-time inheritance, in comparison to the non type-safe dynamic typing and prototype-based inheritance scripting languages.

2.5 Perl

Perl (Practical Extraction and Report Language), (Perl, 2004), is a widely used imperative scripting language, with a particularly compact syntax and built-in operators for flexible string manipulation. It supports dynamic typing and user-defined functions, while depending on the implementation it offers a large repertoire of built-in system functions ranging from operating system, networking and shell commands. Perl became popular as an early most preferred instrument for Common Gateway Programming (CGI), i.e. server-side logic implementation for web applications. In the design of Delta, the string manipulation facilities of Perl have been mapped to an appropriate collection of library functions.

3 Variables and dynamic typing

In Delta, all variable declarations are implicit by use, except of formal arguments that are optionally listed in the context of function definitions. By default, use of a named variable causes instant declaration at the present scope if the name cannot be resolved either to a previously defined variable within enclosing scopes or as a library function. Variables defined at global scope have static linkage. Additionally a variable name can be qualified with the prefixes: local (i.e. resolve only at current scope, with automatic declaration if not found), static (i.e. local and with static linkage), <n> with \( n \in \mathbb{N} \) (i.e. resolution only at the \( n \)-th outer scope, \(<0>\) is a synonym for local),

The Delta language A. Savidis, 2005
and global (i.e. resolve only at global scope). In Figure 3, examples for variable use and scope resolution are provided.

```javascript
x = y = z = w = 10; // Implicit declarations at global scope

function f(y) {
    // Formal argument "y" at local scope
    y = 10; // Formal argument "y" referenced
    local y = 20; // "local y" resolves to formal argument "y"
    <1>y = 30; // Resolves to global "y"
    <0>x = global w; // Same as "local w = <1>w;"
    static x = <1>x; // Local and static "x" assigned the "<1>x"
}
```

Figure 3: Examples of variable declarations “by use” with scope resolution.

### 3.1 Statically typed values with dynamically typed variables

Variables gain initially the type undefined, which is altered dynamically during runtime according to content modification. While variables are dynamically typed, values are always statically typed; no type definitions are supported in the Delta language. In particular, some data types cannot appear even as compile-time values, but may be only internally manifested as returned values of specific library functions (i.e. the types reference and extern of Figure 4). The complete list of all data types is provided in Figure 4, with simple examples of values pertaining to each type.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>There is a single type for numbers, internally represented as a double precision floating number.</td>
<td>10 20.345 987.0x56</td>
</tr>
<tr>
<td>string</td>
<td>Character sequences, no null terminator.</td>
<td>“this” “this&quot;“ and&quot;” this&quot;</td>
</tr>
<tr>
<td>boolean</td>
<td>True / false value.</td>
<td>true false TRUE FALSE</td>
</tr>
<tr>
<td>table</td>
<td>Associative table. Both keys and values can be of any type.</td>
<td>{} [1,2,3,4] [][]</td>
</tr>
<tr>
<td>reference</td>
<td>Reference to a variable. De-reference functions over non-references affect the supplied variable directly. Reference of a reference returns the original reference</td>
<td>p = reference(q); // “q” ref set(p, “this”); // Sets “q” set(z, get(p)); // “z=get(q)” w = reference(p); // “q” ref</td>
</tr>
<tr>
<td>extern</td>
<td>Used for application-specific object types created only through library functions. The application may supply a descriptive type name.</td>
<td>a = list(10, 20, 30, 40);</td>
</tr>
<tr>
<td>normalfunc</td>
<td>Normal non-member functions</td>
<td>function f(x,y){ return x+y; }</td>
</tr>
<tr>
<td>memberfunc</td>
<td>Member functions inside tables. More comprehensive examples will follow later.</td>
<td>t = { ( member(){}) }; Table “t” has a single element at index 0, being a member function.</td>
</tr>
<tr>
<td>libfunc</td>
<td>A host-environment library function. Those can be manipulated as values too.</td>
<td>f = print; // Gets lib func. f(“hello”); // Calls indirectly.</td>
</tr>
</tbody>
</table>
This type is gained when assigned the constant value \textit{nil} (i.e. singleton type).

\begin{center}
\begin{tabular}{ll}
\textbf{nil} & This type is gained when assigned the constant value \textit{nil} (i.e. singleton type). \\
p = \text{nil} & \\
\end{tabular}
\end{center}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{delta_data_types.png}
\caption{The Delta data types with examples of values.}
\end{figure}

3.2 \textbf{Runtime type identification – the ‘typeof’ function}

To enable runtime type identification, the \textit{typeof} library function is provided, accepting a single expression argument, while returning its corresponding type, taken from the left column of Figure 4, in the form of a string value. For example, the call

\begin{verbatim}
typeof(typeof) returns "libfunc", typeof(reference(x)) returns "reference", and
typeof(static y) returns "undefined" (y is declared by use).
\end{verbatim}

3.3 \textbf{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 \texttt{[]} constructs an empty table, while \texttt{[\{"x":0\}]} makes a table with a single element, with value 0, indexed by the string key “x”. In case of string-type indices, apart from the default syntax to reference stored values, i.e. \texttt{table["key"]} (e.g. \(t["x"]\)), the syntax \texttt{table.key} (e.g. \(t.x\)) and \texttt{table."key"} (e.g. \(t."x"\)) may be alternatively employed. Table elements can be removed by setting the corresponding value to \texttt{nil}, implying that \texttt{nil} cannot be stored within a table. Hence, \(t.x = \text{nil};\) causes the entry indexed by key “x” within \(t\) to be directly removed. Finally, the following library functions are provided:

\textit{The Delta language} 

A. Savidis, 2005
tabindices(t₁), returning a new constructed table t₂ where: ∀ pair of index and associated value ( Kı, Vı) ∈ t₁, j ∈ [0, N), N being the total stored values in t₁, the pair (j, Kı) is added in t₂. 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(t₁), returning N being the total stored values in t₁.

tabcopy(t₁), returning a new constructed table t₂ where: ∀ pair of index and associated value ( Kı, Vı) ∈ t₁, j ∈ [0, N), N being the total stored values in t₁, the pair ( Kı, Vı) is added in t₂. That is, an exact copy of t₁ 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 Figure 5, simple examples for table construction and use are provided. Elements listed in a table constructor without an explicit index are given automatically successive integer indices (starting from zero).

```
lib = [{
  "io": [ { "print" : print} ] },
  {"tables" : [ { "indices" : tabindices}, {"length" : tablength} ] }
};

lib.io.print("hello, world"); // Calls "print"
t = lib.tables.indices(lib); // Calls "tabindices";
lib.math = [ { "sqr" : ( function(x){ return x*x; } ) } ];
y = lib.math.sqr(10); // Calls the previous unnamed function
z = y.sqr(); // lib.math is at [0]
t[0].print(z); // lib.io is at [1]
function Point(x,y) { return { { "x" : x}, { "y" : y} }; }
p1 = Point(-34, -78); // A very simple form of object construction
```

Figure 5: Examples regarding typical uses of associative tables.

3.4 Dynamic management of call by reference

In comparison to other scripting languages, Delta provides a method to accommodate call by reference scenarios dynamically, by supporting runtime deducible references.
to program variables. This is allowed by the following library functions (examples of use are provided in Figure 6):

- `reference(x)`, `x` being an expression, with the following runtime behavior:
  - If the expression is not a program L-value, i.e. it is a temporary expression-evaluation variable or a function (normal, member or library), a runtime error is raised and nil value is returned.
  - Else, if the L-value is a variable of reference type, its stored value is returned.
  - Else, a reference value for the L-value variable is returned.

The need for Lvalue argument checking is inherent in the fact that the reference library function is not encapsulated in the language syntax.

- `get(x)`, accepts an expression and has the following behavior:
  - If `x` is not of reference type, the value of `x` is returned.
  - Else, let `y` be the variable referenced by `x`. The value of `y` is returned.

- `set(x,v)`, accepts two expressions and has the following behavior:
  - If `x` is not of reference type, the call is equivalent to `x = v`.
  - Else, let `y` be the variable referenced by `x`. Then the call is equivalent to `y = v`.

```pseudo
function inc(i) {
  set(i, get(i)+1);
}
p = Point(20, 30);
inc(p.x); // No effect: "p.x" is not a reference
inc(reference(p.x)); // The right way to affect "p.x"
inc(reference(reference(p.x))); // Same. Reference on reference is ignored
z = reference(inc); // "z" gets nil, and an error is raised
function move(p, dx, dy) { p.x += dx; p.y += dy; }
move(reference(p)); // Error: table expected, not a reference
move(p); // Ok: table is always passed by reference
```

Figure 6: Examples of dynamic handling for calls by reference.

### 3.5 Built-in ‘assert’ clauses and ‘debug’ qualified statements

In widely used scripting languages there is no specific support for code fragments dedicated to bug diagnosis. This implies that the correctness-checking logic supplied
by programmers is semantically and syntactically indistinguishable, in the context of the language, from the rest of the program source code. In Delta, there are two key features to enable better quality defensive programming: (a) statements prefixed by the debug qualifier; and (b) program assertions in the form of assert clauses. Those two features impose the following regulations:

- **During compilation:**
  - If the program is not compiled in debug mode, debug statements and assert clauses are entirely stripped-off from the produced target code;
  - All variables / functions declared / defined within debug statements and assert clauses can be only employed within debug statements and assert clauses;
  - If there is a return statement in the definition of a function, this is not included in a debug statement;

- **During execution:**
  - The expression of an assert clause evaluates to true;
  - If the execution of a program instruction initiated from a debug statement or an assert clause eventually modifies a program variable, then this variable has been declared within a debug statement or an assert clause;
  - If the execution of a debug statement or an assert clause leads to a library function call, then this function does not alter the application state, nor any program variables declared outside debug statements or assert clauses.

```plaintext
function f(x, y, z) {
    assert x*y==z;
    debug {
        set(x, y+z);        // Runtime error: lib function "set" changes "x"
        return x*y+z;      // Compile error: function return in debug stmt
    }
    debug local w = sqr(x+y+z);
    local wr = nil;
    debug wr = reference(w); // Runtime error: "wr" changed in debug stmt
    return w;              // Compile error: "w" used outside debug stmt
}
```

**Figure 7: Examples for debug statements and assert clauses.**

*The Delta language*  
A. Savidis, 2005
The above runtime regulations ensure that the execution of correctness checking logic never affects the program state. This approach facilitates the development of fully orthogonal defensive programming units with encapsulated defect diagnosis, for higher quality source code, genuinely validated by the language execution semantics. In Figure 7, simple examples of deployment of `debug` statements and `assert` clauses are provided.

4 Functions

Programmer defined functions are first-class objects in Delta, appearing as values within program expressions, while constituting normally assignable content for program variables. Issuing function signatures, like the C function prototypes, is not supported, while function definitions are syntactically treated like statements. The advantageous features of Delta regarding functions are mainly related to:

- The capability of implementing modular runtime function overloading, with an argument matching dispatch pattern, relying upon: runtime type identification, late binding of actual arguments (to be explained later) and associative tables.
- The support for polymorphic higher-order functions, through the employment of table member functions and late binding of actual arguments. This is discussed in the next Section, since it requires the presentation of the object-oriented extensions of associative tables, to play the role of object prototypes.

4.1 Function values and unnamed functions

Although functions are syntactically treated like statements, they do not result on the generation of particular executable code within the control flow at the point of definition. Functions appear as values in two syntactic forms: (a) by direct reference

*The Delta language*  
A. Savidis, 2005
to the function name; and (b) by enclosing a complete function definition in parenthesis (), referred as the expression form of function definition. Unnamed functions can be also defined, however, always in the context of a parenthesized expression form. Representative examples are provided in Figure 8.

```javascript
function f(){}   // Named func defined in stmt from
(p = f)();       // Func value taken by name
(p = (function g(){}))(); // Value of named func defined in expr form
g();            // Func value taken by name ("g" is visible)
(function h(){}); // Named func defined in expr form called directly
add = (function(x,y){ return x+y; }); // Value of unnamed func (expr form)
x = add(10,20);
```

**Figure 8: Examples of function definitions and deployment of function values.**

### 4.2 Static and late binding for library functions

Library functions are also treated as first-class objects, meaning their value can be gained and deployed in ways similar to programmer-defined functions. However, in the Delta language, library functions exhibit more flexible deployment since they may be called in two ways:

- As evaluated expressions of `libfunc` type (i.e. native language function address). The values of such expressions originate from the engagement of library function names in program expressions, being statically bind by the compiler to the corresponding library function values.

- As evaluated expressions of `string` type. In this case, the string value is looked-up within the whole collection of library functions. If a library function matches such a name, it is called. Else a runtime error is posted. Alternatively, the `getlibfunc(id)` library function can be used, `id` denoting an expression of `string` type, which returns the corresponding library function value if any, else `nil`. This way, all calls can be made with `libfunc` expressions, while the programmer may supply error checking for the cases where the dynamic binding to a library function value fails.
The previous approach implements a late binding policy for library functions, being particularly useful for mobile code or applications attachments for diverse machines and operating systems, when there is a need to dynamically identify the range of capabilities offered by the host environment in the form of library functions; examples (excerpts) are provided in Figure 9.

```javascript
io = [
  {"printcon" : getlibfunc("printcon") }, // Console output
  {"printgfx" : getlibfunc("printgfx") }, // GUI output (popup)
  {"printau" : getlibfunc("printau") }, // Audio text to speech
  {"print" : getlibfunc("print") }, // Default, always supplied
  {"call" : ( function(f, s) { if (typeof(f)="$\text{libfunc}$") f(s); } ) }
];
assert typeof(io.print)="$\text{libfunc}$"; // We need the default "print"
io.call(io.printau, "Application started"); // Delegate caller
g = getlibfunc("getlibfunc"); // "g" is "getlibfunc" now!
```

**Figure 9:** Example of dynamic library function use; the “call” function plays the role of a safe delegate caller, ignoring calls to unsupported library functions.

### 4.3 Dynamic manipulation of actual arguments

In the Delta language, there are two categories of facilities for runtime manipulation of actual arguments:

- Allow varying actual arguments to functions, while providing the method to extract the content and type of each individual actual argument. Such a facility is offered by most known scripting languages.

- Enable the runtime construction and manipulation of actual argument lists for function calls, indistinguishable to the called functions from typical explicit lists of actual arguments. Hence, although at compile time a particular function call reflects a statically defined number of actual argument expressions $e_1,\ldots,e_n$, during runtime, those expressions may result in a wider list of actual argument values $v_1,\ldots,v_m$, where $m>n$. This reflects a type of **late binding for actual arguments**, not offered by the previously discussed scripting languages. As it will be shown later,
this technique plays a key role in the programming of polymorphic higher order functions.

4.3.1 Runtime argument extraction and type checking

Although functions may provide argument identifier lists for syntactic convenience, i.e. to easier refer to actual arguments, the number of actual arguments need not conform to the number of such optionally listed formal argument names. Additionally, the original list of actual arguments supplied will always remain intact, i.e. no argument pruning (dropping values) or padding (adding default values) will be performed, as it is the case with Lua.

The list of all actual arguments can be gained through the keyword arguments, returning a table with all argument values indexed through consecutive integer values, according to their (left to right) ordering during the call. Hence, the first actual argument value is taken by arguments[0] and the last actual argument value is taken by arguments[tablengt(arguments)-1]. The type of individual arguments can be queried through the typeof function. Naturally, the increased flexibility and ease of use due to dynamic typing is paid by the complete lack of type safety, inherently requiring the explicit type-error checking encapsulated in the algorithmic function logic. For instance, if function \( f(x, y) \) requires two number arguments, its type checking version has to be implemented as:

```plaintext
function f(x, y) {
  assert tablengt(arguments) == 2 and
typeof(x) == "number" and
typeof(y) == "number;
  ...
}
```

*The Delta language*  
A. Savidis, 2005
4.3.2 Late binding of actual arguments

The semantics for late binding of actual arguments presuppose the presence of a runtime mechanism to dynamically push actual arguments on the runtime stack, upon function calls. As it will be shown later, while discussing the implementation of the novel language features, in compiled scripting-language versions like our Delta implementation, this cannot be accommodated unless a special-purpose virtual machine instruction is introduced. The syntax and semantics for late binding of actual arguments are very simple:

- If an actual argument expression $e$ is syntactically written in the form $|e|$, then before the function call, instead of pushing the value of $e$ on the stack:
  - It is asserted that $e$ evaluates to a table $t$;
  - It is asserted that: $\forall (K, V) \in t, K \in [0, \text{tablelength}(t))$. That is, the table contains only values indexed by successive integers, starting from zero. Table length can be zero.
  - $\forall j \in [0, \text{tablelength}(t)), t[j]$ is pushed as an actual argument. That is, all table values are pushed as actual arguments ordered by their integer index.

Following those definitions, the number of actual arguments pushed on the stack due to a function call, if dynamically resolved arguments of this form are engaged, is only calculated at the time the call is actually performed. Examples are shown in Figure 10.

```java
function f(x, y){ return x+y; }
f(10, |[20]|); // 20 pushed dynamically
f([30, 40]); // 30 ad 40 pushed dynamically
f([50], [60]); // 50 and 60 pushed dynamically
f(10, 10, |[]|); // |[]| pushes no argument
t = [1, 2];
f(|t|); // 1 and 2 pushed dynamically
```

Figure 10: Examples where the late binding of actual arguments is employed.
4.3.3 Dynamic function overloading

The deployment of unnamed functions, dynamic manipulation of actual arguments, runtime type identification, and associative storage, allows the implementation of a 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 Figure 11, 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 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. In the Delta language, this type of signature-independent automatic propagation of actual arguments to proper delegate functions is commonly referred as transit actual arguments, playing a critical role in supporting polymorphic higher-order functions.

```plaintext
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") // First time called ?
        dispatcher = [
```

The Delta language
A. Savidis, 2005
function added (x, s) {...} 
overloaded().install("NumberString", added);

Figure 11: The dynamic function-overloading software-pattern recipe in the Delta language for modular extension of function semantics.

4.3.4 Dynamic operator overloading

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

- \( \text{eval} (t_1 \ op \ t_2) \). 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.

- \( \text{eval}(op \ \text{ unary } t_1) \). 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.

This method applies for all operators in Delta, such as arithmetic, boolean, assignment and associative operators, as well as the function call \( () \) and the table member access \( [ ] \) and \( . \) operators. For prefix and postfix unary operators \(--\) and \(++\), the \( + \) and \( - \) binary operators need to be only overloaded. 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 Error!

Reference source not found.

```
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;
}
```

Figure 12: Dynamic operator overloading for extensible language semantics.

4.3.5 Multiple returned values and multiple assignments

Functions may return multiple values, as in Lua, syntactically supplied as a list of expressions separated with commas. A function call expression returning multiple values is subject to the following runtime semantics:

- When supplied as an actual argument, i.e. pushed on the runtime stack, each such returned value is effectively pushed as a separate actual argument;
- When engaged as an un-indexed element in table construction, it is treated as a list of separately supplied values;
- In any other case, it is simply used as a unary value, resolving to the first element of the list, while also removing it; this implies that subsequent uses of a multi-value list effectively access and remove each successive values in a well defined order;

```
function f(x) { return x/3, y/3, z/3; }
function g(x, y, z) { return x+y+z; }
w = g(f(15)); // i.e. g(15/3, 15/3, 15/3)
t = [ f(w ) ]; // i.e. [ 15/3, 15/3, 15/3 ]
```

The Delta language A. Savidis, 2005
\[ i = f(20); \quad // \text{i.e. } i = 20/3, \text{ rest values dropped} \]
\[ x, y, z = f(x, y, z); \quad // \text{i.e. } x=x/3, \ y=y/3, \ z=z/3; \]

Figure 13: Examples showing how multiple returned values are treated.

The implementation of the method to accommodate the expansion of multiple values in table construction to successive numerically indexed values is straightforward though the introduction of a special operand type being “auto table indexing”. This reflects an internal integer counter for associative tables within the virtual machine, which is increased with each element insertion with index of the above new type. Effectively, when the inserted content is of multi-value type, multiple insertions will take place, increasing this table counter according to the list size.

Multiple assignments are more of a “syntactic sugar”, not requiring amendments or special purpose instructions at the virtual machine side. A multiple assignment is of the generic syntactic form \(l\text{-value}_1, \ldots, l\text{-value}_n = r\text{-value}_1, \ldots, r\text{-value}_k\). This statement is effectively treated as two parts: (a) the first part is the construction of a multi-value list \(L\) consisting of all expression values \(r\text{-value}_1, \ldots, r\text{-value}_k\); (b) the second part is a generated list of \(n\) assignments of the form \(l\text{-value}_i = L\).

5 Prototypes

In the Delta language, in comparison to compile-time object classes, object prototypes are runtime class values, from which instances are dynamically produced through replication. In this context, object classes never appear within the source code in the form of compile-time manifested types, but only as first-class runtime values. In some languages, prototypes do not map to first-class values during runtime, neither to statically typed classes at compile-time. In such cases, the most common technique is the delivery of a constructor function having additional semantics in comparison to

The Delta language

A. Savidis, 2005
normal functions, where member entries are introduced through special-purpose statements. For instance, in ECMA Script, the assignment on `this.x`, for a legal ECMA Script identifier `x`, leads to the introduction of a member referenced by name `x`. Additionally, in all ECMA Script constructors, there is a reserved prototype property, which can be set during constructor function definition as a standard invariant argument value. The characteristics of prototypes in the Delta language are:

- They are first-class values, implemented using associative tables. They are actually table instances, with no prototype-specialized compile-time or runtime semantics.
- There are no specialized constructor functions for object instances or prototypes. Construction can be implemented through normal programmer-defined functions.

In the Delta language primary emphasis is put in delivering an appropriate software pattern to effectively implement prototypes, while minimizing the special-purpose syntactic or semantic extensions for object-oriented programming. This is in accordance to the previous suggested method to address dynamic function overloading, without introducing particular language extensions.

### 5.1 Object oriented extensions to tables

To allow associative tables play effectively the role of both prototypes and instances, *member* functions with a parenthesized expression form can be defined inside table construction expressions. Member functions have the following key properties:

- They are first class values, instantiated upon table construction, being automatically and invariantly associated to the constructed table instance. In other words, member function values are instantiated with every host table construction, as pairs of the member function address and the constructed table instance.
They comply with the same semantics as all types of values stored in an associative table, e.g. can be extracted, assigned to variables, or removed from the owner table. In all such cases, the associated table instance cannot be altered.

Within member function definitions, the keyword `self` always resolves to the runtime associated table instance.

When the function value of a function call expression evaluates to a member function, it is called by resolving all `self` references to the associated table instance. This implies that member functions do not syntactically require an object instance to be called, like for instance, in C++, Java, ECMA Script or Lua.

There is no built-in destructor function. Although Delta is a language with automatic garbage collection (discussion follows in the implementation details at the end), 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 logic. This is based on the following argument:

Application objects need to be cancelled exactly when the application logic decides that the relevant conditions are met. Then, all corresponding cancellation actions actually implement the policy for the internal reflection of the cancellation event. Once application-specific actions are applied, memory disposal takes place only at the point there is no program variable assigned to the subject table. Hence, it is argued that memory disposal is semantically separated from application-specific object cancellation (i.e. destruction).

Examples of prototypes and instance construction are shown in Figure 14. As it is shown, instance construction is realized by programmer-defined construction functions. 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 `objcryp(t1)` should be employed, which in addition to `tabcopy`, performs the following:

- Let t2 be the returned copy of t1. Then, ∀ member function value \((F, T) \in t_1\), if
  
  \[ T = t_1, \]
  
  then add \((F, t_2)\) in t2 else add \((F, t_1)\) in t2. In other words, member function values of the original table become member function values of the table copy.

```javascript
function PointProto() { // Prototype extraction function
  static proto;
  if (proto=="undefined")
    proto = [
      { "x", "y" : 0 },
      { "class" : "Point" },
      { "clone" : ( member() { return objcryp(self); } ) }
    ];
    return proto;
  }

  p1 = PointProto().new(30, 40); // Instance construction via prototype
  p2 = PointProto().new(60, 70); // Same as before
  p3 = p1.clone(); // Instance construction by replication
  fc = p1.clone; // Getting "clone" member of "p1"
  p3 = fc(); // Calls "clone" in the context of "p1"
}

// Can make a dedicated constructor function
function Point(x,y) { return PointProto.new(x,y); }

// Can collect prototypes or constructors together in a table
prototypes = [ { "Point" : PointProto() }, other prototypes go here ];
constructors = [ { "Point" : Point }, other constructors go here ];

p6 = prototypes.Point.new(p2.x, p2.y);
p7 = constructors.Point(p3.x, p3.y);
```

**Figure 14:** Example of simple prototype implementation in the Delta language, illustrating possible alternative deployment styles. Runtime type checking and constructor overloading have been removed for clarity.

### 5.2 Only late binding for members

The indexing of members through string values serves a two-fold role: (a) it allows members to be used with the readable dotted syntax, e.g. \( p.x \) in place of \( p["x"] \); (b) it implies that the binding of the member-referring expression to the target member object is always taking place at runtime. This fundamentally late binding policy...
allows the easy construction of polymorphic functions, since there is no required compile-time type-conformance regarding the list of parameters for polymorphic functions. Instead, there is only a design contract reflected in the support of the expected member naming conventions together with their associated semantics; this is similar to name-based binding in Lua methods and ECMA Script member functions. Following the connotation of late binding for statically typed object-oriented languages: if virtual function $f$ of base class $B$ is called through a $B$ reference variable, internally denoting an object of a derived class $D$ that refines $f$, then the refined $f$ is called. Without this specialized form of late binding, statically typed object-oriented languages simply fail to support polymorphic functions. However, dynamic typing is more powerful since it can be deployed for all object instances as far as they can dynamically offer the required members, where statically typed languages require compile-time conformance to type inheritance hierarchies. Additionally, late binding is allowed also for data members, something that is not accommodated in statically typed languages.

```plaintext
function refreshall(displays) {
    // Polymorphic function on display objects
    for (local n = tablelength(output), local i = 0; i < n; ++i)
        dispays[i].refresh();
}

if (displayvm = vmload("/ext/display.dbc")) // Dynamic VM loading
displayprotos = vmcall(displayvm, "displayprotos"); // Cross VM call
if (displayprotos.text) // Do we have a display prototype for text output?
textdisplay = displayprotos.text.new(); // Instantiate a text display
```

Figure 15: A simple example of polymorphic algorithms applicable to dynamically loaded prototypes.

Examples for polymorphic functions via late binding are provided in Figure 15. In this example taken from the 2WEAR Project (see acknowledgments), migratory clients written in the Delta language, upon start-up load any resident host-environment utility virtual machines, including the one offering implemented display object prototypes (i.e. `displayprotos` in Figure 15). The `vmload` and `vmcall` library functions for loading

*The Delta language*  
A. Savidis, 2005
virtual machines and performing cross machine calls are explained later. Once display object instances are made, like `textdisplay`, the `refreshall` function of Figure 15 is regularly called to refresh updated display contents. This shows that such polymorphic functions are made dynamically reusable for external dynamically loaded precompiled code units, as far as the latter supply a `refresh` function.

### 5.3 Dynamic constructor overloading

Overloaded constructors basically follow the dynamic function-overloading pattern previously discussed (see Figure 11). Additionally, such alternative constructors can be dynamically extensible, meaning argument conversion and instance initialization functions can be installed on the fly as needed.

```javascript
function sig(t) {
  assert typeof(t) == "table";
  for (local s = "", local n = tablelength(t), local i = 0; i < n; ++i)
    if (typeof(t[i]) == "table" and t[i].class != nil) {
      assert typeof(t[i].class) == "string";
      s += t[i].class;
    }
  else
    s += typeof(t[i]);
  return s == "" ? "void" : s; // Empty argument list is "void"
}
// This shows how the proto of Figure 14 is extended to encompass // dynamically installable constructor functions.
proto = [
  { "constructors" : [
    { "void" : (function(){ return objcopy(proto); }) },
    { "numbernumber" : // Parameterized constructor
      (function(x,y){ p=objcopy(proto); p.x=x; p.y=y; return p; }) },
    { "Point" : // Copy constructor
      (function(pt){ p=objcopy(proto); p.x=pt.x; p.y=pt.y; return p; }) }
  ]},
  { "new" : ( member() {
    f = self.constructors[sig(arguments)];
    assert f;
    return f(|arguments|); // Transit actual arguments
  }) }
],
...

// Implementing a new constructor
function midpointconstructor(p1,p2) {
  p = objcopy(PointProto()); // Instantiate from the 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
```

*The Delta language*  

---

A. Savidis, 2005
PointProto().constructors.PointPoint = midpointconstructor;

Figure 16: The meta-constructor pattern for dynamic constructor overloading. Notice that to allow dynamically installed constructors, those are turned to non-member functions.

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 Figure 16, 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. As it is shown, the resulting constructor function named “new” is primarily a meta-constructor, internally dispatching to the appropriate signature-specific constructor. Moreover, as it will be explained later, this meta-constructor can be inherited and 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.

5.4 Polymorphic higher-order functions

Higher-order functions are well known as functions taking functions as arguments or delivering functions as results. Functional programming languages like Haskell (Peyton Jones, 2003) or Scheme (Abelson et al., 1998) genuinely support the definition of generic (polymorphic) higher-order functions. Although many statically
or dynamically typed imperative programming languages do support the programming of higher-order functions, there are particular limitations:

- Existing statically typed languages cannot support purely polymorphic higher-order functions, i.e. directly applicable to all function types, even when parametric polymorphism is applied (like C++ templates, C++ ISO Standard 1998), since they always have to rely on statically matched function signatures.

- Existing dynamically typed scripting languages overpass the signature-checking barrier, however, they particular fail to support higher-order functions, when, depending on the arguments of call, they return functions with differentiated behavior. The main reason is that existing scripting languages merely return the same function address value for the same named function. Instead, an object-oriented model is needed, treating function values as object instances with their own local copy of data values, syntactically accessible within the function definition. This model allows distinct function values, when relating to the same function definition, to have differentiated semantic behaviors. In statically typed object-oriented languages, such function objects are known as *functors* (in C++, a functor is a class that overloads the function call operator \( () \)).

Next we present the imperative implementation of four key polymorphic higher-order functions in the Delta language (see Figure 17): (a) the *mapping* function, applying an argument function to all entries of a table – label 1; (b) the *const* function, transforming a value parameter to a mathematical constant function (i.e. always returning this value when called) – label 2; (c) the *composition* function, returning the functional composition of two function arguments – label 3; and (d) the *delayed call* function, which accepts a function and its actual arguments, returning a function which is equivalent to this call– label 4. The programming of such higher-order functions
functions in the Delta language is enabled by the deployment of two key features: (a) member functions as distinct first-order values internally carrying both the member function address and the associated table instance; and (b) 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 Figure 17, label 2) 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. Following the same technique, the `compose` and `call` higher-order functions also return such member function values for internally constructed tables.

```javascript
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 compose(f, g) {
    t = [ { "f" : f }, { "g" : g },
          { "comp" : (member(){ return self.f(self.g(|arguments|)); })} ];
    return t.comp;
}

mul = (function(x,y){ return x*y; });
sqrpair = (function(x,y){ return sqr(x), sqr(y); });
mulsquares = compose(mul, sqrpair);
print(mulsquares(3, 2)); // Prints "36"

function call(f) {
    for (args = [], n = tablength(arguments) - 1; n > 0; --n)
        args[n-1] = arguments[n]; // Ignore f and shift indices left
    t = [ { "args" : args }, { "f" : f },
          { "call" : (member(){ return self.f (|self.args|); })} ];
}
```

The Delta language A. Savidis, 2005
Figure 17: Examples of key polymorphic higher-order functions, with an imperative implementation in the Delta language.

6 Dynamic inheritance

In statically typed object-oriented languages, inheritance is a compile-time operation where the source and target domains are always compile-time class types. In ECMA Script, single dynamic inheritance is facilitated upon instance construction, through the supplement of a base prototype argument in constructor functions. In the Delta language, inheritance is pushed one step further by being offered as a runtime function engaging table instances, the latter playing either the role of distinct object instances or class prototypes, as freely decided by the programmer. Following this functional form of inheritance, single or multiple inheritance hierarchies of table instances can be dynamically constructed, by only augmenting the basic semantics of the Delta language for the runtime resolution of table members.

6.1 Runtime instance inheritance trees

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

$$\alpha \neq \beta \land \neg \beta \triangleleft \alpha \land \neg (\exists \gamma : \gamma \neq \beta \land \gamma \triangleleft \beta)$$
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) cannot inherit to more than one instance. In the Delta language, the following library functions are provided for dynamic management of inheritance associations among table instances:

- `inherit(t_α, t_β)`, which establishes the associations $t_α \leftarrow t_β$ and $t_β \rightarrow t_α$
- `uninherit(t_α, t_β)`, which cancels the association $t_α \leftarrow t_β$ and $t_β \rightarrow t_α$
- `isderived(t_α, t_β)`, returning, true if $t_α \leftrightarrow t_β$, else false.

### 6.2 Member resolution in inheritance chains

Inheritance associations define an augmented way for dynamic binding of table members, reflecting the fundamental priority of member versions in derived instances over the member versions of base instances, within inheritance hierarchies. The latter is also reflected by the semantics of the late binding mechanism within statically typed object-oriented languages like C++ or Java. 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 latter is also supported in statically typed languages through static binding, by providing specialized base-member resolution syntax (e.g. in C++, the scope resolution operator for the base class instance is applicable), but cannot be supported by the hand-crafted inheritance in Lua. The member-binding algorithm is provided in Figure 18.
Figure 18: Member binding logic within instance inheritance chains; notice that root denotes the most derived instance in an instance inheritance hierarchy.

Following Figure 18, in case the $b$ flag (i.e. bounded use) of the $bind$ function is $true$, the resolution of the member $x$ in table $t$ is performed by giving higher priority to the binding of $x$ directly within table $t$. Otherwise, i.e. $¬ bounded ∨ x ∉ t$, 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 Figure 19, a few examples are provided regarding the alternative search paths, to resolve particular members within an instance inheritance hierarchy.

Figure 19: 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.
6.3 Prototype-based inheritance pattern

The employment of the dynamic inheritance-association management functions to cater for prototype-based inheritance is based on a specific proposed software pattern, with the following key characteristics:

- Replication of the base prototypes, through the "new" member function (see also Figure 14) to produce the inherited base instances, inside the constructor of the derived-prototype;
- Establishment of inheritance associations between the newly created base instances and the newly created derived instance;
- Reservation for extra standard members within the derived instance, providing syntactic access to base instances.

In statically typed object-oriented languages, the memory model (i.e. structure and size) for derived class instances is assembled during compilation, while upon runtime instantiation of a derived class, the instantiation of the various constituent base instances is automatically performed, following a well defined execution order (i.e. base instances are constructed first). Additionally, such memory models typically reflect the packaging of instances of an inheritance hierarchy into a single block of memory. In contrast, in the Delta language, inheritance is viewed as a dynamic functional property, semantically separated from instance memory structures, without exposing such memory organization semantics for constituent instances (the latter are to be decided by programmers). The proposed pattern emphasizes the presence of instance inheritance hierarchies as first-class values.

```
function MetaConstructor() {
    function sig() { As it is implemented in figure 16 }
    return [ Only the "new" member of figure 16 ]; // Constructed by every call
}

function CicleProto() {
    static proto;
```

*The Delta language*  A. Savidis, 2005
if (undefined(proto)) { // Same as: typeof(proto)=="undefined"
    proto = [
        { "radius" : 0 },
        { "class" : "Circle" },
        { "clone" : (member(){ return proto.new(self); })},
        { "area" : (member(){ return pi()*sqr(self.radius); })}
    ];
    (a) inherit(proto, MetaConstructor());

    function constructor() { // A base proto delegate constructor
        (b) circle = objcopy(proto);
        (c) circle.Point = PointProto().new({arguments});
        (d) inherit(circle, circle.Point);
        return circle;
    }

    (e) proto.constructors = [
        (e)   { "void", "Point", "numbernumber": constructor },
        (e)   { "Circle": (function(c){ return proto.new(c.Point); })}
    ];
    (e) return proto;
}

c1 = CircleProto().new(); // "void" empty constructor
c2 = CircleProto().new(c1); // "Circle" copy constructor
p1 = c1.Point; // Getting base instance
c3 = CircleProto().new(p1); // "Point" constructor
c4 = p1.clone(); // Calls derived "clone" for c1 "Circle"

Figure 20: A derived class prototype, to construct “Circle” objects deriving from “Point”: (a) base prototype inheriting the meta-constructor; (b) derived prototype replication; (c) base prototype replication and base instance reserved member; (d) inheritance association; and (e) dynamic constructor installation.

In Figure 20, the implementation of the Circle prototype inheriting from Point is provided. Apart from the prototype table instance, normally introducing all the desirable Circle members, the rest of the code reflects: base and derived prototype replication, dynamic constructor installation and the establishment of the inheritance association. The constructor function implements a derived-class constructor, internally making a delegate call to the base prototype constructor; this function is installed within the constructors sub-table of the Circle prototype, following the dynamic constructor-overloading pattern. The latter is also known as the virtual constructor pattern (Coplien, 1992).
6.4 Instance targeted inheritance and polymorphic pattern programming

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. Consequently, dynamic inheritance associations are applicable over instances as they are for prototypes. This features allows the following programming patterns to be easily accomplished, hardly accommodated in languages without dynamic inheritance:

- **Dynamically removable inheritance.** This is facilitated by disconnecting a complete instance inheritance sub-hierarchy from the target instance, with a call to the `uninherit` library function.

- **Dynamically installable inheritance.** Similarly, this is made possible by connecting an instance inheritance sub-hierarchy to the target instance, with a call to the `inherit` library function.

Although the implementation of such patterns in Delta is straightforward, the practical need for such “nomadic” instance inheritance sub-hierarchies is not directly obvious. If object instances need to dynamically substitute runtime behaviors implemented by particular inherited constituent instances, the instance-targeted manipulation of inheritance chains offers an appropriate solution. From the implementation point of view, this functionality can be also accommodated with specific programming patterns like the State pattern (Gamma et al., 1995). However, apart from the evident need to explicitly implement the pattern itself, it has to be: (a) remade every time the
same need emerges for different classes; and (b) extended when new classes for the updateable behaviors are implemented.

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.

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 Figure 21. Following Figure 21, 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 patterns logic is actually consolidated in a single function performing the following actions:

*The Delta language*  
A. Savidis, 2005
- Cancels the inheritance association with the current base State instance
  ```javascript
  self.State;
  ```

- Makes a new instance corresponding to the prototype of the new state, i.e.
  ```javascript
  prototypes[self.class].States[newState];
  ```

- Establishes an inheritance association with the new base State instance, while setting the current State name, i.e.
  ```javascript
  inherit(self, self.State = inst);
  ```

The runtime associations for the State pattern are shown in **Error! Reference source not found**. 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.

```javascript
function StatePattern() {
  // Returns an instance of the 'State' pattern.
  return {
    "setstate": (member(newState) {
      // The instance has a single member.
      uninherit(self, self.State);
      inst = prototypes[self.class].States[newState].new();
      inherit(self, self.State = inst);
    })
  }
}
```

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

Figure 21: The polymorphic reusable pattern for dynamically updateable base instances, following the semantics of the State pattern (i.e. a state reflects a distinct behavior).
**Figure 22:** Dynamic inheritance of the State pattern, to support varying inherited behavior instances. Labels indicate the distinction between member ownership, use, binding and reference to other instances.

### 6.5 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 Figure 23.

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

**Figure 23:** 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 Figure 23, 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. 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.
7 Dynamic deployment of precompiled units

Precompiled code written in the Delta language can be loaded and executed during runtime using a special-purpose set of library functions, within the same thread of execution as the particular calling code unit. Each loaded compiled unit is allocated a new virtual machine instance, with its own separate runtime stack. More specifically, the following library functions are provided:

- **vmload(s, f)**, which loads the compiled code from file path f, instantiates a virtual machine associated with the unique optionally supplied name s (if s is not supplied, f is used as a name). The newly created virtual machine instance is returned as a value of `extern type with "virtualmachine"` as the descriptive type name. Finally, before returning, the `vmload` function executes all globally defined statements of the loaded code unit;

- **vmthis()**, returning the virtual machine instance in the context of which it is called during runtime;

- **vmget(s)**, returning the virtual machine instance having name equal to s, if one exists, else *nil*;

- **vmgetfunc(v, f)**, returning the address of the global programmer-defined function with name f, in the context of the loaded code for either the virtual machine instance v (i.e. `typeof(v)=="virtualmachine"`) or the virtual machine name v (i.e. `typeof(v)=="string"`), else returns *nil*.

- **vmcall(v, f, ...)**, calling global function named f, for either the virtual machine instance v or virtual machine name v, with actual arguments the optionally supplied arguments denoted with dots.

Values for instances of loaded virtual machines are always copied by reference, as it is also the case with tables and values of `extern type`, while virtual machine instances

*The Delta language*  
A. Savidis, 2005
are also subject to garbage collection. In Figure 24 examples of use for the previously
described library functions are provided.

vmload("math", "math.dbc"); // "dbc" stands for Delta Binary Code
n = vmcall("math", "sum", 1,2,3,4,5,6);
sum = vmgetfunc("math", "sum"); // Instead of calling we get the function
n = sum(1,2,3,4,5,6); // Cross VM call, same as previous
sum = vmgetfunc(mathvm = vmget("math"), "sum");
n = vmcall(mathvm, "mul", n, n, n); // Returns n*n*n

function f(){
    vmcall(vmthis(), "f"); // Calls "f" with no arguments
callglobal("f"); // Same as "vmcall(vmthis(), "f")"

Figure 24: Examples for dynamic loading of compiled code units and application of
cross virtual machine calls.

8 Implementation

In the design of the Delta language there are various elements that are not met in
existing scripting languages like: debug and assert clauses, late binding of actual
arguments, member functions with associated table instances, dynamic inheritance
management, and dynamic loading of compiled units. For such a compiled language
running on top of a platform-independent virtual machine, it is considered critical to
explicitly supply the appropriate implementation patterns, so as to effectively realize
all such new features, without however being specific to the particular language of
implementation. The development of the most common features like dynamic typing,
unnamed functions, associative tables, and linkage of library functions have been
already reported in the context of existing scripting language implementations like the
freely distributed implementation of Lua (see http://www.lua.org/download.html).

8.1 Virtual machine instruction set

The virtual machine instructions have three operands: the first two are usually
employed as instruction source values while the last one typically playing the role of a
result operand. In Figure 25, the simple sub-set of instructions for control flow and
arithmetic operations is provided.

The Delta language A. Savidis, 2005
In Figure 26 the rest of the instructions for table construction, table member access, and function call management are provided. Apart from the instruction set, which is very compact and simple, the following key implementation issues remain to be addressed, so as actually accommodate all the demanding runtime semantics of the Delta language: (a) the type of information encapsulated within different categories of instruction operands (e.g. expressions, tables, arguments, etc.) and the way those are resolved during runtime; (b) the virtual machine architecture; and (c) the detailed execution semantics of each instruction.
Figure 26: Instructions to handle table construction, table member indexing, and function calls. The operand categories are shown in bold typeface. The multi value flag is true when the function returns multiple values (i.e. Expr is a linear table with all values).

<table>
<thead>
<tr>
<th>Operand flags (1 byte)</th>
<th>Operand interpretation (4 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Target instruction address (jump label)</td>
</tr>
<tr>
<td>01</td>
<td>Global variable offset</td>
</tr>
<tr>
<td>02</td>
<td>Formal argument offset</td>
</tr>
<tr>
<td>03</td>
<td>Local function variable offset</td>
</tr>
<tr>
<td>04</td>
<td>self</td>
</tr>
<tr>
<td>05</td>
<td>arguments</td>
</tr>
<tr>
<td>06</td>
<td>self&lt;\textit{j}&gt;, value is \textit{j}</td>
</tr>
<tr>
<td>07</td>
<td>Program function, address</td>
</tr>
<tr>
<td>08</td>
<td>Library function, index</td>
</tr>
<tr>
<td>09</td>
<td>Return value register</td>
</tr>
<tr>
<td>10</td>
<td>Constant string, index</td>
</tr>
<tr>
<td>11</td>
<td>Constant number, index</td>
</tr>
<tr>
<td>12</td>
<td>Constant nil</td>
</tr>
<tr>
<td>13</td>
<td>Constant boolean</td>
</tr>
</tbody>
</table>

Figure 27: The data operand bit flags and their interpretation for the various operand types of virtual machine instructions.

Following Figure 27, \textit{variable} instruction operands denote typical program variables, whose stack address is computed at runtime based on statically associated address offsets, engaging a compile-time resolved stack offset. Apart from program variables, the rest of fall in any of the following categories:

- Const literal value, i.e. boolean, number, string or \textit{nil};
- \textit{arguments}, i.e. the table encompassing all actual call arguments;
- Function value, i.e. library function (a native function address for the host environment language), or a programmer defined normal / member function (a virtual machine code address);
- `self`, used in member functions as the owner table instance, and `<j>self`, used in nested table construction expressions to refer to the appropriate outer constructed table instance (for `j=0` it is the currently constructed table);
- the return value register, used to put or get the result upon function calls.

Bonded member access and dynamic binding of actual arguments are handled through special purpose instructions. Also, the construction of multi-value lists is handled with special virtual machine instructions as well.

### 8.2 Virtual machine architecture

The architecture of the virtual machine is illustrated in Figure 28. All globally defined variables and function activation records are stored in the stack - STACK (i.e. no global data segment), while the loaded compiled code is stored in a reserved code segment - CODE. The virtual machine encompasses typical registers such as: the program counter (`pc`), the stack registers (`top` and `topsp`), the function return value register (`retvalue`), and a counter of the runtime active debug statements (`debugnest`). Additionally, there are two special-purpose stacks: (a) the ACTS stack, to temporarily collect all actual arguments of a function call, before those are finally pushed on the STACK to effectively perform the call; and (b) the TABS stack, to push newly constructed table instances, during the evaluation of their respective table construction expression. The detailed structure of activation records is provided in Figure 29, depicting also the way the `self` and `arguments` elements can be easily resolved. Following this architectural overview, the way instruction operands are resolved to runtime values is discussed.
8.3 Runtime operand resolution

The way the less trivial instruction operands are bind during execution to appropriate runtime values is shown in Figure 30. As it is shown:

- Library functions are simple values carrying the native function address, as it is internally represented in the native application-programming language;
- Normal non-member programmer-defined function values are pairs of the code address and the respective virtual machine instance in which the operand is resolved;
- Member functions are triplets of function code address, virtual machine instance and associated table instance. Since member functions appear as function values...
with the parenthesized form only inside table constructors, the table owner instance is resolved at runtime as the top constructed table from the TABS stack;

- *self* and *arguments* resolution is mostly trivial, i.e. artificially pushed arguments before the programmer supplied arguments.

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Resolution method</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>self</em></td>
<td>STACK[topsp-M] of type is <em>table</em>, see also Figure 29.</td>
</tr>
<tr>
<td>&lt;j&gt;*self</td>
<td>TABS[TABS_top-j] of type is <em>table</em>, getting an outer table constructed instance in nested table construction expressions.</td>
</tr>
<tr>
<td><em>arguments</em></td>
<td>STACK[topsp-K] of type is <em>table</em>, see also Figure 29.</td>
</tr>
<tr>
<td><em>Library function</em></td>
<td>S = name of the library function in the operand Return getlibfunc(S) of type is <em>libfunc</em></td>
</tr>
<tr>
<td><em>Normal function</em></td>
<td>VM = the virtual machine instance executing the instruction ADDR = the code address of the function in the operand Return the pair (VM, ADDR) of type is <em>normalfunc</em></td>
</tr>
<tr>
<td><em>Member function</em></td>
<td>VM = the virtual machine instance executing the instruction ADDR = the code address of the function in the operand TAB = TABS[TABS_top], i.e. top constructed table instance Return the triplet (VM, ADDR, TAB) of type is <em>memberfunc</em></td>
</tr>
</tbody>
</table>

Figure 30: Resolution methods for the less trivial operand categories.

Following the discussion of the method to resolve the most important operand categories to appropriate runtime values, the implementation algorithms for the execution of most critical instructions, as listed in Figure 26, is provided.

### 8.4 Instruction execution algorithms

The implementation of the most important instructions is provided in Figure 31. Table construction instructions practically maintain the TABS stack. Debug control instructions handle the debugnest register; if debugnest > 0, then the current instruction was reached from an active debug qualified statement. When an actual argument is collected, a latebind flag with true value or a multi-value expression cause an unrolling of all table-argument elements on the ACTS stack.

```plaintext
TABLEMAKESTART (L) {
    L = new TABLE
    push(TABS, L)
}
pushactuals (VM, t) {
    \forall j \in [0, tablelength(t)) push(VM.STACK, t[j])
}
producearguments () {
```
Figure 31: Implementation of the key instructions for debugging support, late binding of actual arguments and function calls.

While committing function calls, all actual arguments collected into the ACTS stack are finally pushed on the program runtime stack (i.e., STACK). Cross virtual machine calls are easily handled by: (a) pushing actual arguments on the target virtual machine stack; (b) adding to the `debugnest` register of the target machine the value of the `debugnest` register from the source machine, and restoring to the original value upon function return; and (c) by assigning, after the call, the value of the `retval` register of the target machine to the `retval` register of the source machine. The details regarding setting and restoring runtime stack pointers before and after function calls have been omitted for clarity, since they follow the common implementation practice. The latter functionality is embedded in the implementation of the `calllibrary` and `callfunction` virtual machine functions.
### 8.5 Code generation patterns

In Figure 32, a few representative examples for code generation are shown, for specifically selected code fragments deploying some of the distinctive features of the Delta language, like member functions, debugging statement, assert clauses and late binding of actual arguments.

```
a = [ 10, [<1>self[0], <0>self[0]] ];
b = [ (member(){ return self.x; }) ];
t = [ "hello", ",world" ];
print(|t|);
```

| TABLEMAKESTART - - _t1 | TABLEMEMBERSET _t1 0 10 | TABLEMEMBERSET _t1 0 "hello" |
| TABLEMEMBERGET self[1] 0 _t3 | TABLEMEMBERSET _t1 1 ",world" |
| TABLEMEMBERSET _t2 0 _t3 | TABLEMAKEEND - - |
| TABLEMEMBERSET _t2 1 _t3 | ASSIGN _t1 = t |
| TABLEMAKEEND - - |
| ASSIGN _t1 = a |
| JUMP - - [+5] |
| FUNCENTER |
| TABLEMEMBERGET self "x" _t1 |
| FUNCRET _t1 [+1] |
| FUNCEnd |
| TABLEMAKESTART - - _t1 |
| TABLEMEMBERSET _t1 0 member[address] |
| TABLEMAKEEND - - |
| ASSIGN _t1 = b |
| TABLEMEMBERSET _t1 0 "hello" |
| TABLEMEMBERSET _t1 1 ",world" |
| TABLEMAKEEND - - |
| ASSIGN _t1 = t |
| ACTUALARG latebind[t] - |
| CALLFUNC libfunc[print] - |
| assert (x+y)==z; |
| debug [ local i=x*y*z; ] |

Figure 32: Sample code fragments and the resulting code generation patterns; in this example, jump statements are shown with relative instruction indexing for clarity.

### 8.6 Garbage collection

The garbage collection method employed exploits the specific runtime virtual machine memory model and the runtime information regarding the storage of programmer-defined variables, deciding to collect table instances when those cannot be referred directly or indirectly through program variables. This approach is a variant of an original implementation in the context of defensive smart pointers – see implementation details in (Savidis, 2004) - combining reference counting with referral lists, to perform detection of cyclic references with an $O(N)$ algorithm for fast tracing of external references (i.e. references to programmer-defined variables). The selection

*The Delta language*  
A. Savidis, 2005
of the appropriate garbage collection policy is a concern of the particular implementation, for which numerous alternative implementation recipes are currently available: see (McCarthy, 1960), (Dijkstra et al., 1976), (Pirinen, 1998), and a survey in (Jones and Lins, 1996).

8.7 Runtime inheritance

The implementation of the runtime inheritance mechanism is particularly compact, effectively being realized through:

- The augmentation of the basic associative-table functionality to cater for inheritance-based member binding and dynamic inheritance associations. Within the basic associative table data structure, the following additional members are introduced:
  - \textit{derived} of type \textit{Table} reference, denoting the directly derived table instance;
  - \textit{root} of type \textit{Table} reference, denoting the bottom most-derived table instance;
  - \textit{base} of type \textit{list} of \textit{Table} references, denoting the direct base table instances.

  Additionally, the inheritance-aware member binding function of Figure 18 substitutes the traditional direct associative look-up logic for table members.

- The delivery of the inheritance management library functions. Those functions basically install and un-install dynamically inheritance associations, affecting the \textit{derived}, \textit{root} and \textit{base} members of the engaged table-instances.

9 Conclusions and future work

Scripting languages constitute a domain of programming languages loosely defined to encompass languages designed to support the manipulation, customization, and automation of the facilities of an existing system. In this context, the design objectives

\textit{The Delta language} \\
A. Savidis, 2005
of the Delta language, aiming to allow the post-production extension of legacy applications, seem to better match the profile of scripting languages, rather than general-purpose programming languages. Although such a non-theoretical definition of scripting languages implicitly declares a secondary role in software development, scripting languages are currently widely used, while gaining a largely increasing acceptance in the context of key sectors of the software industry like: entertainment (e.g. Cg, Unreal Script, Quake Script, etc.), interactive multimedia (Lingo, Action Script) and web services (Java Script, Visual Basic Script, PERL, Python, PHP, etc.).

The fact that scripting languages tend to be so popular is mainly due to the marriage of: (a) deployment simplicity, usually because of dynamic typing and garbage collection; and (b) the capability to exploit the added-value functionality of an existing popular system.

In this context, the overall objective for the design of the Delta language has been the delivery of programming elements for larger-scale development, in comparison to existing scripting languages, aiming to support: (i) dynamically-typed object-oriented programming; (b) embedded bug defense; (c) imperative programming of polymorphic higher-order functions; and (d) dynamic loading of compiled code units. Additionally, particular emphasis has been put to only provide the absolutely necessary elements to accommodate the above features while addressing specific mechanisms through pattern-reflecting programming recipes, as opposed to introducing customized language constructs. The way those programming facilities are offered in the Delta language is summarized below:

- **Dynamically typed object-oriented programming**
  - Prototype values as dynamic instances of associative tables;
Late binding for all members;
Member function values associated at runtime with table-instances;
Dynamic inheritance links for table instances via inheritance management functions;
Inheritance-aware late member binding for table instances;
Table-instance targeted inheritance.

**Embedded bug defense**
- Assert clauses;
- Debug qualified statements;
- Making definitions within the above defensive units syntactically unreachable to normal program code;
- Forbidding program state updates from within defensive code.

**Polymorphic higher-order function**
- Member function values automatically called in the context of their runtime associated table-instance;
- Multiple returned values by functions;
- Dynamic actual argument manipulation and transit actual arguments.

**Dynamic loading of compiled code**
- Virtual machine instance information within function values;
- Library functions to:
  - Load compiled code into new virtual machine instances;
  - Query virtual machine instances by names;
  - Query global functions in virtual machine instances by names.
In the Delta language the programming flexibility offered by dynamic typing has to be 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 need to be introduced to support typical member access qualifiers such as \texttt{private} or \texttt{const}, to guard pattern conformance during execution.

The new elements of the Delta language can be selectively embedded in other scripting languages, without actually requiring the adoption of the overall language as a monolith design. The need for designing and implementing a complete scripting language has reflected the development challenges in the specific demanding contexts where the Delta language had to be actually deployed. Overall, the Delta language exhibits many features commonly met in dynamically typed scripting languages, while it introduces some novel special-purpose constructs designed to better promote polymorphic reusable components and dynamic software deployment in post-production application extensions.
Appendix – Delta language grammar

code ::= { def }
def ::= [ 'debug' ] ( [ stmt ] ';' | namedfunc )
namedfunc ::= 'function' id funcbody
unnamedfunc ::= 'function' funcbody
funcbody ::= '(' [ id { ',', id } ] ')' block
block ::= '{' code '}'
stmt ::= expr | whilest | forst | ifst | 'break' | 'continue' |
       'return' [ expr ] | block | assertion | 'const' id '=' expr
whilest ::= while '{' expr '}' stmt | 'do' stmt 'while' '{' expr '}'
forst ::= 'for' '{' exprlist ';' expr ';' exprlist '}' stmt
ifst ::= 'if' '{' expr '}' stmt [ 'else' stmt ]
exprlist ::= [ expr { ',', expr } ]
expr ::= assign | primary | boolean | arith
assign ::= lval '=' expr | lval '+' expr | lval '-' expr |
       lval '*' expr | lval '/' expr
lval ::= [ 'local' | 'static' | 'global' | '<' number '>'] id | tableitem
tableitem ::= ['bound'] ( expr '.' id | expr '[' expr ']' | expr '.' string)
primary ::= lval | lval '+' | '+' lval | lval '-' | '-' lval | const
       funccall | '(' (namedfunc | unnamedfunc ) ')' | table | 'self'
       | '<' number '>' | 'self' | 'arguments' | '=' expr | 'not' expr
funccall ::= expr '(' [ actual { ',', actual } ] ')
actual ::= expr | '|' expr '
const ::= 'nil' | 'true' | 'false' | number | string
boolean ::= expr boolop expr
arith ::= expr arithop expr
arithop ::= '+' | '-' | '*' | '/' | '%'
boolop ::= 'or' | 'and' | '<' | '>' | '<=' | '>= ' | '=' | '!='
table ::= '[' [ field { ',', field } ] ']
field ::= '(' expr { '|' expr } ':' fieldval ')' | fieldval
fieldval ::= expr | '(' 'member' funccall ')
assertion ::= 'assert' expr | 'assert' '{' expr '}'

The Delta language

A. Savidis, 2005
References


