Extension de Langage par les Valeurs Virtuelles

Gary Benattar, Yoann Couillec

3 novembre 2011

Résumé

Cette article est un résumé de l'article *Virtual Values for Language Extension*[1] écrit par Thomas H. Austin, Tim Disney et Cormac Flanagan. Il introduit la possibilité d'étendre les fonctionnalités d'un langage par l'utilisation des valeurs virtuelles.

1 Introduction

Une valeur virtuelle est basé sur le concept de proxy. D'une manière général un proxy est une entité qui se situe entre deux autres qui souhaitent communiquer. Dans un langage de programmation, un proxy est une entité qui encapsule une autre. Ainsi, le proxy a la possibilité de filtrer les demandes effectuées vers l'entité qu'il encapsule. Le travail de Thomas H. Austin fait suite à celui de T. Van Cutsem [2][4], qui présente une implémentation des proxys pour Javascript. Brendan Eich[6], le créateur de *SpiderMonkey*, le moteur d'exécution de Javascript de référence, a aussi créé *Narcissus*, un moteur d'exécution Javascript refléxif implémentant les proxys.

2 Valeur virtuelle

L'objectif des valeurs virtuelles est de virtualiser l'interface entre le code et les données. Lorsqu'une opération est appelée sur une valeur virtuelle, une trappe est appelée. Les opérations classiques entre codes et données sont : application de fonction, accès, affectation, opération unaire, binaire etc. Dans un langage donné le comportement de ces opérations est défini par la sémantique du langage et est fixé. Une valeur virtuelle définie une collection de trappes défini par l'utilisateur qui décrit comment les opérations doivent se comporter sur la valeur virtuelle. Cela implique que le langage doit comporter des règles pour invoquer la trappe appropriée. Il sera ainsi possible de décrire des extensions de langages.

2.1 Trappe

Une valeur virtuelle est un proxy qui comporte une collection de trappes :

– *call*, la valeur est considéré comme une fonction.
– *getr* et setr, elle est considérée comme un enregistrement.
– *geti* et seti, elle est considérée comme un index d'enregistrement.
– *unary*, appelée lorsqu'un opérateur unaire est appelé sur la valeur virtuelle.
– *left*, appelée lorsque la valeur virtuelle se situe à gauche de l'appel à un opérateur binaire.
– *right*, appelée lorsque la valeur virtuelle se situe à droite de l'appel à un opérateur binaire.
– *test*, appelé lorsque la valeur virtuelle est utilisée comme une condition.

Le Listing 1 décrit un exemple de valeur virtuelle, le proxy identité, ainsi que les appels implicites à ses différentes trappes :

```
var handler =  {
  value:x,
  call:function(y){ return x(y);},
  getr:function(n){ return x[n];},
  setr:function(n,y){ x[n] = y; },
  geti:function(r){ return r[x];},
  seti:function(r,y){r[x] = y;},
  unary:function(o){return unaryOps[o](x);},
  left:function(o,r){return binaryOps[o](x,r);},
  right:function(o,l){return binaryOps[o](l,x);},
  test:function(){return x;}
}

var proxy = Proxy.create(handler);
var value, args, index, record;
proxy(args); // handler.call(args);
proxy[index]; // handler.getr(index);
proxy[index] = value; // handler.setr(index,value);
record[proxy]; // handler.geti(record);
record[proxy] = value; // h.seti(record,value);
-proxy; // handler.unary('-');
proxy + value // handler.left('+',value);
value + proxy // handler.right('+',value);
value = (proxy) ? true:false; // p.test();
```

Listing 1 – Exemple de proxy en Javascript

2.2 Secret

Une valeur virtuelle, contient un ensemble de trappes et un secret. Le secret est une valeur partagées par les valeurs virtuelles issues d'une même classe de valeurs virtuelles. Ce qui permet d'introduire un mécanisme de vérification de type.

3 Typage

3.1 Primitives

Un langage implémentant les valeurs virtuelles doit fournir trois primitives :

- *Constructeur* d'un proxy, prend en argument un secret et la valeur encapsulée. Le secret est une valeur qui permet d'identifier de manière unique un type de valeur virtuelle.
- *Prédicat*, permettant de savoir si une valeur est un proxy ou non.
- *Dé-constructeur* (Listing 2), prend en argument un secret et une valeur, renvoi la valeur encapsulée si le secret est valide sinon renvoi la valeur entrée en argument. Cette primitive permet de faire une vérification dynamique de type de valeur virtuelle.

```
var unProxy = function(secret,proxy){
  if (isProxy(proxy) && secret == proxy.secret){
    return proxy.handler;
  }
  return false;
}
```

Listing 2 – Dé-constructeur

3.2 Erreur de typage

3.2.1 Valeur virtuelle et valeur de base

```
var c1 = makeComplex(1,2);
var c2 = 3;
c1 + c2; // Erreur de typage
```

Listing 3 – Évaluation paresseuse

$c1 + c2$ implique l'appel de la trappe *left* des nombres complexes. Cette trappe applique *unProxy* sur l'argument $c2$ qui applique *isProxy* or $c2$ n'est pas un proxy de type nombre complexe donc l'appel $c1 + c2$ lève une exception de typage.

3.2.2 Interaction entre valeurs virtuelles

```
var c = makeComplex(1,2);
var u = makeUnit('cm'); // Proxy d'unite de mesure
c + u; // Erreur de typage
```

Listing 4 – Évaluation paresseuse

isProxy renvoi *true* car u est bien un proxy. Mais l'appel de *unProxy* renvoi $false$ car le secret de u est différent de celui de c car il sont issus de proxys

différents.

4 Exemples d'extension de langage

4.1 Identité

Les trappes de la valeur virtuelle identité (Listing 1) définissent le comportement usuel des opérateurs du langage. Par exemple la trappe *call* est défini comme suit : *call* est une fonction prenant un argument et appliquant la valeur virtuelle, alors considérée comme une fonction, sur cet argument. En Javascript cette trappe s'écrit ainsi : *call : function(y) return x(y);* , x étant la valeur virtuelle encapsulée. Les trappes *getr*, *setr*, *geti* et *seti* définissent le comportement normal associé aux enregistrements. les trappes *unary*, *left* et *right* doivent tester l'opérateur qui est appliqué sur la valeur virtuelle afin de réagir correctement. la trappe *test* renvoi simplement la valeur virtuelle.

4.2 Évaluation paresseuse

Le proxy d'évaluation paresseuse prend en valeur une fonction f. Celle-ci est évaluée lorsque qu'un des trappes est évaluées. La fonction interne au proxy z permet d'implémenter ce mécanisme (Listing 5). Lors du premier appel à une trappe, f est évaluée, et lors des appels ultérieurs, la valeur de cette évaluation est renvoyée. Les trappes sont définies de la même manière que dans le proxy identité, à ceci près que la valeur encapsulé est la fonction z. Nous en proposons une implémentation en Javascript (Annexe A.3).

```
var z = function(){
  var r = f();
  z = function() { return r; }
  return r;
}

var handler = {
  getr: function(n) { return z()[n]; },
  ...
}
```

Listing 5 – Évaluation paresseuse

4.3 Contrats

Un contrat plat est la soummission d'une valeur à un prédicat. Si la valeur vérifie le prédicat alors le contrat est rempli. Le contrat d'une fonction est un domaine, c'est à dire l'ensemble des valeurs autorisées en entrée de la fonction, et un intervalle, c'est à dire l'ensemble des valeurs autorisées en sortie de la fonction. Lors d'un appel de fonction sur le contrat, trappe *call*, le domaine et l'intervalle sont vérifiés. Par exemple :

```
var handler =  {
  value:f,
  domain:function(in){ assert(/* true ou false */); return in;},
```

```
  range:function(out){ assert(/* true ou false */); return out;},
  call:function(y){ return range(f(domain(y)));},
  ...
}
var proxy = Proxy.create(handler);
proxy(args);
// appel handler.call(args),
// qui appel range(f(domain(args)))
```

Listing 6 – Contrats

De la même manière, il existe des contrats sur les enregistrements. L'index de l'enregistrement devant appartenir au domaine et la valeur associée à un index quelconque devant appartenir à l'intervalle. On peut aussi souhaiter être plus précis et définir un contrat spécifique par index. Nous proposons une implémentation des contrats en Javascript (Annexe A.4) et en Python (Annexe B.2).

4.4 Nombres complexes

Les valeurs virtuelles permettent d'introduire les nombres complexes. Les opérations binaire et unaire auront ainsi un comportement spécifique et qui est celui voulu par les mathématiques.

```
var complexUnaryOps = {
  "-": function(r,i){ return makeComplex(-r, -i); }
}

handler = {
  real: r,
  img: i,
  unary: function(o){return complexUnaryOps[o](r,i);},
  ...
}

var complex1 = Proxy.create(handler);
-complex1;
// appel de handler.unary('-')
```

Listing 7 – Nombres complexes

Nous en proposons une implémentation en Javascript (Annexe A.5) et en Ruby (Annexe C.1).

5 Implémentation

Afin d'implémenter les valeurs virtuelles, il faut que la sémantique du langage utilisé comporte le principe de proxy avec l'appel de trappe. Ce qui implique, que le langage doit être typé dynamiquement. Pourtant nous remarquons, un lien entre trappe et surcharge d'opérateur. Nous avons testé puis comparé l'implémentation des valeurs virtuelles dans différents langages.

5.1 Javascript

La version actuelle de Javascript (1.8.2) n'intègre pas les proxys. Toutefois, la prochaine version de Javascript (1.8.5) intégrera une bibliothèque de proxys. Ainsi, Thomas H. Austin propose quelques exemples [7] d'implémentation fonctionnant avec le moteur d'exécution Narcissus. Il a également écrit un plug-in [8] Firefox intégrant Narcissus. Nous proposons une implémentation (Annexe A) des valeurs virtuelles pour la version actuelle de Javascript, en explicitant l'appel aux trappes. Nous avons implémenté trois extensions de langages, l'évaluation paresseuse (Annexe A.3), les contrats (Annexe A.4) et les nombres complexes (Annexe A.5).

5.2 Python

Python est un langage interprété, objet, fonctionnel. Il dispose de la surcharge d'opérateur. On peut implémenter les valeurs virtuelles soient par surcharge ou en se fiant à la définition formelle. Nous avons opté pour la deuxième solution (Annexe B), qui ressemble beaucoup à l'implémentation Javascript. La différence réside dans l'utilisation de l'héritage, par exemple *FunctionC* (Annexe B.2) hérite de *Proxy*. Quelques difficultés techniques apparaissent, comme le besoin de réflection afin de connaître le type d'une instance, il n'y a pas d' *instanceof* en Python. Nous avons essentiellement implémenter les contrats pour les fonctions afin de pouvroi comparer avec l'implémentation en Javascript, les contrats pour les enregistrements suivants le même schéma.

5.3 Ruby

Ruby est un langage interprété, multi-paradigme et orienté objet. Il comprend entre autre la surcharge et la définition d'opérateur. Comme nous avons pu le voir dans les exemples d'implantations des valeurs virtuelles, il arrive, lorsque les trappes *call*, *geti*, *seti* ne sont pas nécessaires que la programmation orienté objet et la surcharge des opérateurs remplace aisément l'utilisation de proxy.

En effet *Ruby* permet la surcharge des opérateurs :

- unaires (*unary*) : *-@, +@*
- binaires (*left* et *right*) : *+(object), ==(object), -(object)*
- assesseurs et modificateurs (*getr* et *setr*) : *[](y), []=(y, value)*

Nous proposons donc une implémentation des nombres complexes en *Ruby* (Annexe C.1) en ayant redéfini les opérateurs nécessaires :

```
c =  Complex.new(1,2)
c2 = Complex.new(3,4)
c3 = Complex.new(3,4)
// inversion d'un complexe (surcharge de l'operateur unaire '-')
puts (-c)
// ajout de deux complexes (surcharge de l'operateur binaire '+')
puts c + c2
puts (c == c2) // false
puts (c2 == c3) // true
puts c[0] = 3 // Exception sur setr
puts c[0] // Exception sur getr
puts 3 + c // Erreur de typage
```

Listing 8 – Nombre complexe en Ruby

5.4 Conclusion

On constate que les valeurs virtuelles ne sont pas implémentées dans les langages de programmation mais que l'on peut les implémenter en partie quelque soit le langage. La surcharge d'opérateur ressemble à ce mécanisme, on peut l'utiliser en Python, Ruby mais aussi C# et C++. En Javascript, nous avons souhaité être fidèle à la définition formelle des valeurs virtuelles, en acceptant la contrainte suivante : appeler explicitement les trappes. Cela nous a permis de constater la puissance de ce concept et également sa grande facilité d'utilisation. En effet, en Javascript les valeurs virtuelles sont des objets instanciés, classiquement par *new*, les mécanismes sous-jacents étant masqués. Le concept de valeur virtuelle n'est pas totalement nouveau mais permet de formaliser des concepts déjà existants dans les langages de programmation. Nous avons vu que leur implémentation n'est possible qu'en partie dans la plupart des langages. Pour implémenter totalement les valeurs virtuelles il faut que la sémantique du langage considérée intègre l'appel des trappes appropriées. A notre connaissance, la surcharge n'est pas possible pour les trappes *call* et *test* n'est pas permis dans les langages que nous avons testés. Á l'inverse la trappe *binary* est redéfinnissable dans presque tous les langages. Il faut donc, qu'un langage prévoit ce mécanisme nativement. L'intégration des proxys dans la prochaine version de Javascript[3] ouvrira certainement la porte à d'innombrables extensions.

Références

[1] T.H. Austin, T. Disney, C. Flanagan, Virtual Values for Language Extension OOPSLA (2011)

[2] T. Van Cutsem, M. S. Miller, Proxys : Design Principles for Robust Object-oriented Intercession APIs (2010)

[3] `https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Proxy`

[4] `http://wiki.ecmascript.org/doku.php?id=harmony:proxies`

[5] `https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Proxy`

[6] `http://jsconf.eu/2010/speaker/be_proxy_objects.html`

[7] `http://slang.soe.ucsc.edu/proxy-values/`

[8] `http://mozillalabs.com/zaphod/`

A Javascript

A.1 Assert

```
function assert (v){
  if ( ! v){
    throw "Assert false";
  }
}
```

A.2 Proxy

```
function Proxy (s, h){ // Constructeur
  var secret = s;
  var handler = h;
  this.secret = secret;
  this.handler = handler;
}

function unProxy (s, p){ // Deconstructeur
  if (isProxy(p) && s == p.secret){
    return p.handler;
  }
  else{
    return false;
  }
}

function isProxy (p){ // Predicat
  return typeof p.secret != "undefined"
  && typeof p.handler != "undefined"
  && typeof p.handler.call != "undefined"
  && typeof p.handler.getr != "undefined"
  && typeof p.handler.setr != "undefined"
  && typeof p.handler.geti != "undefined"
  && typeof p.handler.seti != "undefined"
  && typeof p.handler.unary != "undefined"
  && typeof p.handler.left != "undefined"
  && typeof p.handler.right != "undefined"
  && typeof p.handler.test != "undefined" ;
}
```

A.3 Lazy

```
Lazy.secret = {};
function Lazy(f){
  var z = function() {
    var r = f(); // 1er appel
    z = function() { return r; } // appels suivants
    return r;
  };
  return new Proxy(Lazy.secret,{
    call:function(y) { return z()(y); },
    getr:function(n) { return (z()[n]); },
    setr:function(n,y) { z()[n] = y; },
    geti:function(r) { return r[z()]; },
    seti:function(r,y) { r[z()] = y; },
    unary:function(o) { return unaryOps[o](z()); },
    left:function(o,r) { return binaryOps[o](z(),r); },
    right:function(o,l) { return binaryOps[o](l,z()); },
    test:function() { return x; }
  })
}

// EXEMPLES
var f = function(){
  return {"a":3};
}
var l = new Lazy(f);
console.log(unProxy(Lazy.secret, l).getr("a")); // l["a"]
console.log(unProxy(Lazy.secret, l).getr("a"));
```

A.4 Contrat

```
// PREDICATES
function isFunction(f) {
  return (typeof f == "function");
}
function isRecord(x){
  return typeof x == "object";
}
function isBool(val){
  return typeof val == "boolean";
}
function isNum(val){
  return typeof val == "number";
}
function isTrue(val){
  return true;
}
function isPositive(x){
  return x >= 0;
}

// FLAT CONTRACT
function flatc (pred, x){
  assert(pred(x));
  return x;
}

// FUNCTION CONTRACT
FunctionC.secret = {};
function FunctionC(domain,range,x){
  assert(isFunction(x));
  return new Proxy(FunctionC.secret, {
    call:function(y) { return range(x(domain(y))); }, ///
    getr:function(n) { return (x[n]); },
    setr:function(n,y) { x[n] = y; },
    geti:function(r) { return r[x]; },
    seti:function(r,y) { r[x] = y; },
    unary:function(o) { return unaryOps[o](x); },
    left:function(o,r) { return binaryOps[o](x,r); },
    right:function(o,l) { return binaryOps[o](l,x); },
    test:function() { return x; }
  });
}

// MAP CONTRACT
MapC.secret = {}
function MapC(domain,range,x){
  assert(isRecord(x));
  return new Proxy(MapC.secret, {
    call:function(y) { return x(y); },
    getr:function(n) { return range(x[domain(n)]); }, ///
    setr:function(n,y) { x[domain(n)] = range(y); }, ///
    geti:function(r) { return r[x]; },
    seti:function(r,y) { r[x] = y; },
    unary:function(o) { return unaryOps[o](x); },
    left:function(o,r) { return binaryOps[o](x,r); },
```

```
    right:function(o,l) { return binaryOps[o](l,x); },
    test:function() { return x; }
  });
}

// RECORD CONTRACT
RecordC.secret = {}
function RecordC(contracts, x){
  assert(isRecord(x));
  return new Proxy(RecordC.secret, {
    call:function(y) { return x(y); },
    getr:function(n) { return contracts[n](x[n]); }, ///
    setr:function(n,y) { x[n] = contracts[n](y); }, ///
    geti:function(r) { return r[x]; },
    seti:function(r,y) { r[x] = y; },
    unary:function(o) { return unaryOps[o](x); },
    left:function(o,r) { return binaryOps[o](x,r); },
    right:function(o,l) { return binaryOps[o](l,x); },
    test:function() { return x; }
  });
}

// Useful contracts
Bool = function(x){return flatc(isBool,x)};
Num = function(x){return flatc(isNum,x)};
Any = function(x){return flatc(isTrue,x)};
var natc = function(x){
  return flatc(isPositive, x);
}
NatFun = function(f){ return new FunctionC(natc,natc,f); }

// EXEMPLE 1
var domain = function(v){
  assert(v > 0);
  return v;
}
var range = function(v){
  assert(v > 10);
  return v;
}
var f = function(x){
  return x + 10;
}
var fc = new FunctionC(domain,range,f);
try{
  unProxy(FunctionC.secret,fc).call(1);
}catch(e){
  console.log(e + ": FunctionC");
}

// EXEMPLE 2
var domain = function(i){
  assert(i == "a" || i == "b");
  return i;
}
var range = function(res){
  assert(res > 0);
```

```
  return res;
}
var rec = {"a":5, "b":-6, "c":8}
var mc = new MapC(domain, range, rec);
try{
  unProxy(MapC.secret, mc).getr("a");
  //unProxy(MapC.secret, mc).getr("aa");
  //unProxy(MapC.secret, mc).getr("b");
}catch(e){
  console.log(e + ": MapC");
}

// EXEMPLE 3
var contracts = {"a" : function(x){assert(x>0); return x;},
  "b" : function(x){assert(x<0); return x;},
  "c" : function(x){assert(x==0); return x}
};
var rec = {"a":1, "b":2, "c":0}
var rc = new RecordC(contracts,rec)
try{
  unProxy(RecordC.secret, rc).getr("a");
  //unProxy(RecordC.secret, rc).getr("b");
}catch(e){
  console.log(e + ": RecordC");
}

Bool(true);
//Bool(5); // error
Num(8);
//Num("ee"); // error
Any(5);
Any(true);
Any("ss");
Any({});
var f = function(x){
  console.log(x);
  return x - 1;
}
var nf = NatFun(f);
try{
  unProxy(FunctionC.secret, nf).call(1);
  //unProxy(FunctionC.secret, nf).call(0);
}catch(e){
  console.log(e + ": NatFun");
}
```

A.5 Nombres complexes

```
Complex.secret = {};
Complex.complexUnaryOps = {
  "-": function(r,i){ return new Complex(-r, -i); }
}
Complex.complexBinOps = {
  "+": function(r1,i1,r2,i2){return new Complex(r1+r2,i1+i2);},
  "==": function(r1,i1,r2,i2){return r1==r2 && i1==i2;}
}
Complex.isComplex = function(x){
  if (unProxy(Complex.secret,x)){
    return true;
  }
  else{
    return false;
  }
}
Complex.Complex = function(x){
  return flatc(Complex.isComplex,x);
}
Complex.i = new Complex(0,1);

Complex.tostring = function(c){
  var h = unProxy(Complex.secret, c);
  return "(r = " + h.real + ", i = " + h.img+")";
}

// CONSTRUCTEUR
function Complex (r,i){
  return new Proxy(Complex.secret, {
    real: r,
    img: i,
    unary: function(o){return Complex.complexUnaryOps[o](r,i);},
    left: function(o,y){
      var h = unProxy(Complex.secret,y);
      if (h){
        return Complex.complexBinOps[o](r,i,h.real,h.img);
      }
      else{
        return Complex.complexBinOps[o](r,i,y,0);
      }
    },
    right: function(o,y){
      var h = unProxy(Complex.secret,y);
      if(h){
        return Complex.complexBinOps[o](h.real,h.img,r,i);
      }
      else{
        return Complex.complexBinOps[o](y,0,r,i);
      }
    },
    test: function(){return true;},
    call: function(){throw "no call for a complex";},
    getr: function(){throw "no getr for a complex";},
    setr: function(){throw "no setr for a complex";},
    geti: function(){throw "no geti for a complex";},
```

```
    seti: function(){throw "no seti for a complex";}
  });
}

// EXEMPLES
var c1 = new Complex(1,2);
console.log("c1" + Complex.tostring(c1));
var c2 = new Complex(3,2);
console.log("c2" + Complex.tostring(c2));
var c3 = unProxy(Complex.secret, c1).unary('-'); // c3 = -c1
console.log("c3" + Complex.tostring(c3));
var c4 = unProxy(Complex.secret, c1).left('+',c2); // c4 = c1 +
    c2
console.log("c4" + Complex.tostring(c4));
var c5 = unProxy(Complex.secret, c1).right('+',c2); // c5 = c2 +
    c1
console.log("c5" + Complex.tostring(c5));
Complex.Complex(c2);
Complex.Complex(Complex.i);
console.log(Complex.tostring(Complex.i));
```

B Python

B.1 Proxy

```
class Proxy:
   def __init__(self,s,h):
      self.secret = s
      self.handler = h

def unProxy (s, p):
   if (isProxy(p) and s == p.secret):
      return p.handler
   else:
      return False

def isProxy (p):
   if (p.__class__.__name__ == "Proxy"):
      return True
   for s in p.__class__.__bases__:
      print s.__name__
      if (s.__name__ == "Proxy"):
         return True
   return False
```

B.2 Contrat

```
def isFunction(f):
    return type(f) == type(isFunction)
def isRecord(r):
    return type(r) == dict
def isNum(v):
    return type(v) == int or type(v) == float

# FLAT CONTRACT
def flatc (pred,x):
    assert(pred(x))
    return x

def Num(v):
    return flatc(isNum, v)
Num(3) # Ok
Num("e") # Error

# FUNCTION CONTRACT
class FunctionC(Proxy):
    secret = "secret"
    def __init__(self,d, r, f):
        assert(isFunction(f))
        Proxy.__init__(self,FunctionC.secret,{"call": lambda y:
            rang(f(domain(y)))})
        self.domain = d
        self.rang = r
        self.x = f

def domain(v):
```

```
    assert(v > 0)
    return v
def rang(v):
    assert(v > 15)
    return v
def f(x):
    return x + 10

fc = FunctionC(domain,rang,f)
unProxy(FunctionC.secret, fc)["call"](6) # Ok
unProxy(FunctionC.secret, fc)["call"](1) # Error
```

C Ruby

C.1 Nombres complexes

```ruby
class Complex
attr_reader :real, :img

def initialize(real, img)
@real, @img = real, img
end

def -@
@real, @img = -real, -img
end

def +(other)
@real, @img = (@real + other.real), (@img + other.img)
end

def ==(other)
(@real == other.real) && (@img == other.img)
end

def []=(y, value)
raise "not setr for complex"
end

def [](y)
raise "not getr for complex"
end

def to_s
"r = #@real i = #@img"
end

end
```