Analyse des Programmes et Sémantique (7)

Prof[r] Jacques Malenfant

(cours de J.Malenfant modifié et enseigné en 2013 par Basile Starynkévitch)

janvier-avril 2013

MI030 - APS

© Jacques Malenfant, 2010-2013

♠ avec modifications mineures par Basile Starynkevitch ♣

Cours (en M1) du Professeur Jacques Malenfant
http://pagesperso-systeme.lip6.fr/Jacques.Malenfant/ Professeur en informatique au Laboratoire d'Informatique de Paris 6

Enseigné en 2013 par Basile Starynkevitch
http://starynkevitch.net/Basile/
basile@starynkevitch.net & basile.starynkevitch@cea.fr
ingénieur chercheur au CEA, LIST - http://www-list.cea.fr/
travaille sur gcc-melt.org

♠ Nota Bene

Les transparents à fond rose (pages numérotées ♠) et les mots ♠ signalés ainsi ♣ sont de Basile Starynkevitch (dont les opinions n'engagent que lui) ♠

La plupart des transparents sont [recopiés de ceux] de J.Malenfant 2012, que je remercie. Ces transparents sont disponibles sous http://starynkevitch.net/Basile/

ce 7eme cours est le cours 8 de J.Malenfant.

1. ♠ compléments pratiques
   - ♠ la récursion en pratique
   - ♠ les langages à objets
2. Sémantique dénotationnelle de BOPL

Leitmotiv - pensez “implémentation” !

Un programme source = un texte figé

Un processus Linux qui s’exécute = un tas mémoire en mouvement
graphes de référence, circularités, auto-références

Le lien : les points fixes

1. ♠ compléments pratiques
   - ♠ la récursion en pratique
   - ♠ les langages à objets

2. Sémantique dénotationnelle de BOPL

pratique des récursions dans les langages stricts 1

syntaxes par l'exemple

En *C*, *C++*, *Java*, *Pascal*, *Ada* .... on **déclare** à l'avance les fonctions :

```
// déclarations
bool odd(unsigned),even(unsigned);

// définitions
bool odd(unsigned x) {
  if (x == 0) return false;
  else return even(x-1);
}
bool even(unsigned y) {
  if (y == 0) return true;
  else return odd(y-1);
}
```

NB : Bien que ça soit recursif-terminal ("tail recursive") ce n'est généralement pas traduit en une itération, sauf forte optimisation.

pratique des récursions dans les langages stricts - 2

En Ocaml, pas besoin de “déclarer” mais définir avec un *letrec* :

```
letrec odd x = if x = 0 then true else even (x-1)
and even y = if y = 0 then false else odd (y-1)
(*eventuellement:*) in (odd 321, even 17)
```

En Scheme :

```
(letrec (
  ((odd x) (if (eq? x 0) #t (odd (- x 1))))
  ((even y) (if (eq? y 0) #f (even (- y 1))))
  )
 (display (odd 321)) (newline) (display (even 17)) (newline))
```

(Scheme *exige* le traitement itératif de la récursion terminale)

pratique des récursions dans les langages stricts - 3

En MELT ☺ voir **gcc-melt.org**

```
(letrec (
    (even (lambda (x) (if (>ivi x 0) (odd (-ivi x 1)) :true)))
    (odd (lambda (y) (if  (>ivi y 0) (even (-ivi y 1)) ())))
    (tu (tuple even odd))
  ) tu)
```

Mais *MELT* ne connait pas les appels récursifs terminaux, car il est traduit en du *C*.

Les constantes comme `:true` ou `>ivi` ou `1` ne sont pas closes dans les fermetures (mais dans les “valeurs routines” λ_1 ou λ_2, ou directement dans le code *C* généré)
les deux fermetures vont clôturer `odd`, respectivement `even`

implémentation de la récursion

GCC [1] produit avec `gcc -fverbose-asm -O1 -S evenodd.c`

```
.globl even
.type even, @function
even:
.LFB1:
.cfi_startproc    ## call frame info
movl $1, %eax #, D.1768
testl %edi, %edi # y
je .L6 #,
subq $8, %rsp #,
.cfi_def_cfa_offset 16
subl $1, %edi #, D.1769
call odd #
addq $8, %rsp #,
.cfi_def_cfa_offset 8
.L6:
rep; ret
.cfi_endproc
.LFE1:
.size even, .-even
```

```
.globl odd
.type odd, @function
odd:
.LFB0:
.cfi_startproc
movl $0, %eax #, D.1771
testl %edi, %edi # x
je .L12 #,
subq $8, %rsp #,
.cfi_def_cfa_offset 16
subl $1, %edi #, D.1772
call even #
addq $8, %rsp #,
.cfi_def_cfa_offset 8
.L12:
rep; ret
.cfi_endproc
.LFE0:
.size odd, .-odd
```

1. Version 4.8 parue fin mars 2013 sur Linux x86-64

GCC produit avec `gcc -fverbose-asm -O3 -S evenodd.c`

```
.globl odd
.type odd, @function
odd:
.LFB0:
.cfi_startproc
testl %edi, %edi # x
jne .L4 #,
jmp .L5 #
#pour alignement du code:
.p2align 4,,10
.p2align 3
.L10:
subl $2, %edi #, x
je .L5 #,
.L4:
cmpl $1, %edi #, x
.p2align 4,,5
jne .L10 #,
movl $1, %eax #, D.1781
ret
.p2align 4,,10
.p2align 3
.L5:
xorl %eax, %eax # D.1781
ret
.cfi_endproc
.LFE0:
.size odd, .-odd
.p2align 4,,15
.globl even
.type even, @function
even:
.LFB1:
.cfi_startproc
testl %edi, %edi # y
movl $1, %eax #, D.1792
jne .L21 #,
rep; ret
.p2align 4,,10
.p2align 3
.L21:
subl $1, %edi #, x
je .L17 #,
cmpl $1, %edi #, x
.p2align 4,,5
jne .L14 #,
.p2align 4,,5
jmp .L23 #
.p2align 4,,10
.p2align 3
.L15:
cmpl $1, %edi #, x
je .L19 #,
.L14:
subl $2, %edi #, x
.p2align 4,,5
jne .L15 #,
.L17:
xorl %eax, %eax # D.1792
.p2align 4,,3
ret
.p2align 4,,10
.p2align 3
.L19:
movl $1, %eax #, D.1792
ret
.L23:
.p2align 4,,6
rep; ret
.cfi_endproc
.LFE1:
.size even, .-even
```

Notez la récursion terminale (non exigée par le standard), grâce à l'optimisation poussée !

MELT produit [plus de 1300 lignes] en simplifiant - sans le code des λ qui est ailleurs

```
/// allocation d'une zone pour les 3 valeurs
 struct meltletrec_1_st {
   struct MELT_CLOSURE_STRUCT (1) rclo_0__EVEN;
   struct MELT_CLOSURE_STRUCT (1) rclo_1__ODD;
   struct MELT_MULTIPLE_STRUCT (2) rtup_2__TU;
 } *meltletrec_1_ptr = NULL;
 meltletrec_1_ptr = (struct meltletrec_1_st *)
  meltgc_allocate
    (sizeof (struct meltletrec_1_st), 0);

  /*iniclos rclo_0__EVEN */
/*_.EVEN__V15*/ meltfptr[12] =
  (melt_ptr_t) & meltletrec_1_ptr->rclo_0__EVEN;
meltletrec_1_ptr->rclo_0__EVEN.discr =
  MELT_PREDEF (DISCR_CLOSURE);
meltletrec_1_ptr->rclo_0__EVEN.nbval = 1;
meltletrec_1_ptr->rclo_0__EVEN.rout =
  (meltroutine_ptr_t) (meltfptr[6]);   λ1

  /*iniclos rclo_1__ODD */
/*_.ODD__V16*/ meltfptr[10] =
  (melt_ptr_t) & meltletrec_1_ptr->rclo_1__ODD;
meltletrec_1_ptr->rclo_1__ODD.discr =
  MELT_PREDEF (DISCR_CLOSURE);
meltletrec_1_ptr->rclo_1__ODD.nbval = 1;
meltletrec_1_ptr->rclo_1__ODD.rout =
  (meltroutine_ptr_t) (meltfptr[9]);   λ2

   /*inituple rtup_2__TU */
/*_.TU__V17*/ meltfptr[16] =
```

```
  (melt_ptr_t) & meltletrec_1_ptr->rtup_2__TU;
 meltletrec_1_ptr->rtup_2__TU.discr =
MELT_PREDEF (DISCR_MULTIPLE);
 meltletrec_1_ptr->rtup_2__TU.nbval = 2;

  /*putclosurout#1 */
((meltclosure_ptr_t)meltfptr[12])->rout =
  (meltroutine_ptr_t) (meltfptr[6]);
((meltclosure_ptr_t)meltfptr[12])->tabval[0] =
  (melt_ptr_t) (meltfptr[10]);
meltgc_touch (meltfptr[12]);

  /*putclosurout#2 */
((meltclosure_ptr_t)meltfptr[10])->rout =
  (meltroutine_ptr_t) (meltfptr[9]);
 ((meltclosure_ptr_t) meltfptr[10])->tabval[0] =
  (melt_ptr_t) (meltfptr[12]);
 meltgc_touch (meltfptr[10]);

 /*putupl#1 */
((meltmultiple_ptr_t) (meltfptr[16]))->tabval[0] =
  (melt_ptr_t) (meltfptr[12]);

 /*putupl#2 */
((meltmultiple_ptr_t) (meltfptr[16]))->tabval[1] =
  (melt_ptr_t) (meltfptr[10]);
meltgc_touch ( /*_.TU__V17*/ meltfptr[16]);

 /*_.LETREC___V14*/ meltfptr[11] =
  /*_.TU__V17*/ meltfptr[16];
```

1. ♠ compléments pratiques
   - ♠ la récursion en pratique
   - ♠ les langages à objets
2. Sémantique dénotationnelle de BOPL

Envoi de message = invocation de méthode

L'envoi d'un message (ou l'invocation d'une méthode) a un comportement qui dépend du receveur du messsage (ou sujet de la méthode)

- pour dessiner une figure géométrique (rectangle, cercle), la procédure de dessin dépend de la figure considérée

On peut vouloir que le comportement dépende de *plusieurs* receveurs :

- le dessin d'une figure dépend de la figure, mais aussi du support (on ne dessine pas pareil un triangle à l'écran et sur fichier PDF ou en SVG)
- le produit de deux facteurs dépend de chaque facteur :
  - produit d'un entier par un flottant
  - produit d'un flottant par un vecteur
  - produit d'un flottant et d'un polynôme
  - produit scalaire de deux vecteurs
  - produit de deux nombres complexes

  et le type du résultat aussi !

$\Rightarrow$ *bi-méthodes, multi-méthodes* [difficile à implémenter dans le cas général]

Comportements partagés

- plusieurs valeurs “semblables” (ou entités, ou objets) partagent un même comportement :
  c’est la même routine qui va dessiner un carré A et un autre carré B
- des valeurs distinctes peuvent partager le même comportement (pour une certaine méthode) : un carré se dessine comme un rectangle, car un carré “est” un rectangle
- le partage peut être “partiel” : dessiner un carré noir vs un carré coloré

Le receveur doit être un “argument”[2] de la méthode

2. généralement implicite, et noté voire passé spécialement

Langages à classe vs langages à prototype

- langages à classe et héritage : Java, C++, Common Lisp, Smalltalk, Ruby, Python ...
  `http://en.wikipedia.org/wiki/Object-oriented_programming`
  - chaque objet connait sa classe (qui définit le comportement de l'objet), a une identité ("adresse") figée et un état (mutable)
  - les classes sont organisées hierarchiquement, avec héritage simple ou multiple
  - on peut -ou non- avoir une classe racine (`Object` en Java [3])
  - le descripteur de classe est connu de chaque objet et connait les méthodes ; certaines sont héritées de la [des] super-classe[s]
  - une classe peut être instanciée en un nouvel objet ; la classe peut ou non être un objet
  - test `instanceof`
- langages à prototype et délégation : JavaScript, Self, Io...
  `http://en.wikipedia.org/wiki/Prototype-based_programming`
  - chaque objet a son dictionnaire de méthodes et un prototype
  - l'envoi d'une méthode peut déléguer au prototype, etc.
  - création d'objet par clonage d'un prototype

3. Sauf pour les tableaux....

vtable = tableau descripteur des méthodes virtuelles

En C++, le premier champ implicite d'un objet pointe vers une table de pointeurs de méthodes virtuelles

```
class A {
  virtual int meth1();
  virtual void meth2(int);
  float meth3(float,int);
  int champ;
}
```

≈

```
struct A {
  struct {
    int (*p_meth1)(struct A*);
    void (*p_meth2)(struct A*, int);
  } *_vtable;
  int champ;
}
```

Et si on a déclaré `A una;` l'appel `una.meth2(7);` est implémenté comme `(una._vtable->p_meth1) (&una, 7);` mais pas pour `una.meth3(3.14,-1)`

Le compilateur génère les vtables, et initialise la vtable de chaque objet à sa construction. La vtable d'une sous-classe `PointColoré` reprend *parfois des* pointeurs de sa superclasse `Point`. L'appel à la superclasse s'explicite `Point::bouger`

Pour optimiser, le compilateur peut dans certains cas dévirtualiser un appel (car l'appel indirect coûte cher en temps CPU)

Langages à classes réifiées

Il est agréable que les classes soient aussi des objets du language (Java, Common Lisp, Smalltalk, Ruby, Python, MELT, ...).
La classe des classes = l'objet `Class` [4] est souvent magiquement instance d'elle-même. $\Rightarrow$ c'est un point fixe.
Voir notamment

- `http://en.wikipedia.org/wiki/Smalltalk`
- `http://en.wikipedia.org/wiki/Metaclass`
- `http://en.wikipedia.org/wiki/Metaobject_protocol`
- `http://en.wikipedia.org/wiki/Eigenclass_model`

La création d'une sous-classe applicative `PointColoré` peut se faire par une primitive ou bien par envoi d'un message à l'objet `Class` ou à sa super-classe `Point`

Le système ne peut pas démarrer à vide : il faut "charger un état mémoire initial" qui contient au moins les objets-classes `Class` et `Object` etc etc etc....

L'implémentation connait incestueusement `Class` et `Object` (difficilement modifiables)

4. créée magiquement, "avant" le démarrage du système !

instanceof avec classes réifiés et héritage simple

Chaque classe connait le vecteur de ses ancêtres ; alors

ω `instanceof` $\kappa \equiv \quad class(\omega) = \kappa \vee class(\omega)$`issubclassof` κ

```
bool
melt_is_subclass_of (meltobject_ptr_t subclass, meltobject_ptr_t superclass) {
/* verifier que subclass et superclass sont des objets
 de la taille d'une classe, puis...*/
  if (superclass_p == (meltobject_ptr_t) MELT_PREDEF (CLASS_ROOT))
    return TRUE;
  subanc =
    (struct meltmultiple_st *) subclass->obj_vartab[MELTFIELD_CLASS_ANCESTORS];
  superanc =
    (struct meltmultiple_st *) superclass->obj_vartab[MELTFIELD_CLASS_ANCESTORS];
  if (melt_magic_discr ((melt_ptr_t) subanc) != MELTOBMAG_MULTIPLE
      || melt_magic_discr ((melt_ptr_t) superanc) != MELTOBMAG_MULTIPLE)
     return FALSE;
  unsigned subdepth = subanc->nbval;
  unsigned superdepth = superanc->nbval;
  if (subdepth <= superdepth)
    return FALSE;
  return (subanc->tabval[superdepth] == superclass); }
```

Sélecteurs de message

Dans certains languages (C++, Java, ...) on connait statiquement toutes les méthodes possibles sur une instance
$\Rightarrow$ le compilateur peut organiser statiquement la vtable (ou la classe)

Dans d'autres langages (Common Lisp, Smalltalk, Ruby, Javascript, ...) on peut ajouter **dynamiquement** et ôter des méthodes.
Par exemple, chaque classe a son dictionnaire de méthodes et connait sa super-classe
L'envoi de message est alors plus complexe ; on connaît le sélecteur de message (= le "nom" de la méthode invoquée) et le receveur.

- on cherche la méthode dans [le dictionnaire de] la classe du receveur, ou sinon....
- on la cherche récursivement dans les super-classes
- on applique la méthode trouvée
- on doit traiter le cas d'un sélecteur inconnu

Techniques JIT ou mémoisation pour accélérer...
(empiriquement il y a seulement quelques classes de receveurs sur un site d'appel précis)

Etat persistant = image mémoire

Les machines Lisp ou Smalltalk des années 1980 conservaient la totalité de leur état sur disque. (c'est très utile pour la circularité d'`Object` et `Class`)

- image système `http://en.wikipedia.org/wiki/System_image`
- persistence orthogonale (et sérialisation) `http://en.wikipedia.org/wiki/Orthogonal_persistence`
- suivre et restorer tout ce qui est accessible depuis la continuation courante `http://en.wikipedia.org/wiki/Continuation`
  donc persister aussi [ou non !] la pile d'appel courante !
- voir aussi le `unexec` d'Emacs ou l' `export` de PolyML ou le `save-lisp-and-die` de SBCL/Common Lisp :

```
% poly
> fun f () = print "Hello World\n";
val f = fn : unit -> unit
> PolyML.export("hello", f);
val it = () : unit
```

```
$ cc -o hello hello.o -lpolymain -lpolyml
$ ./hello
Hello World
```

- aujourdhui : hibernation, checkpoint/restart

Sémantique Dénotationelle de BOPL

(reprise du course de J.Malenfant)

Principes généraux

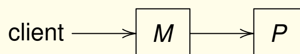
En tant que langage à objets, BOPL introduit deux concepts importants par rapport au mini-langage impératif précédent :

- l'héritage, par la clause `extends` et l'utilisation de `super`, et
- l'auto-référence des objets, par l'utilisation de `self`.

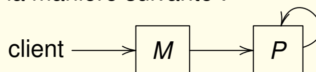
L'auto-référence est une forme de récurrence qui va nécessiter l'utilisation d'un point fixe dans la définition de sa dénotation.

Vers l'auto-référence

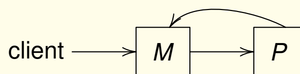
- Considérons une méthode *M* qui doit redéfinir une méthode *P* héritée.
- Par exemple, *M* fait un prétraitement, appelle *P* et enfin fait des post-traitements après *P* avant de retourner à son appelant.
- Du point de vue du client, la situation est la suivante :



- Si la méthode *P* est récursive, elle possède une référence à elle-même, immédiatement résolue de la manière suivante :

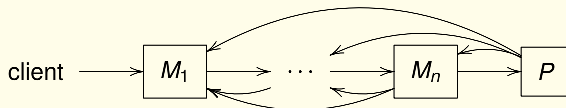


- Mais ce n'est pas le comportement attendu dans les langages à objets, puisque l'auto-référence de *P* devrait en fait voir la redéfinition opérée par *M*. Donc, le schéma souhaité serait plutôt :



Vers l'auto-référence 2

- En généralisant, on voudrait obtenir, après une succession de redéfinitions M_1, M_2, ..., M_n, le schéma suivant :



où le schéma pour la redéfinition M_i ne conserve que les auto-références à M_i pour tous les éléments $M_{i+1,\ldots,n}$ et P.

Comment appliquer ce patron aux objets ?

- Pour les objets, supposons qu'on adopte une représentation sous la forme d'une fonction de l'ensemble des identifiants vers des λ-termes notée :

$$\{i_1 \mapsto \Lambda_1, \ldots, i_1 \mapsto \Lambda_n\}$$

- Soit la classe `Point` :

```
class Point(a, b)
   method x = a
   method y = b
   method distFromOrig() = sqrt(self.x^2 + self.y^2)
   method closerToOrig(p) =
                 self.distFromOrig() < p.distFromOrig()
```

Comment appliquer ce patron aux objets ? 2

- Elle ♠ la classe `Point` ♣ sera représentée par une fonction engendrant ce que Cook et Palsberg ont appelé un *générateur* de points :

$$
\begin{aligned}
MakeGenPoint(a, b) = \lambda self.\ & \{x \mapsto a, \\
& \ y \mapsto b, \\
& \ distFromOrig \mapsto \sqrt{self.x^2 + self.y^2} \\
& \ closerToOrig \mapsto \\
& \qquad \lambda p.(self.distFromOrig < p.distFromOrig)\}
\end{aligned}
$$

Comment appliquer ce patron aux objets ? 3

- Le générateur est une fonction prenant *self* en paramètre et retournant l'instance de la classe `Point` selon la représentation choisie.
- *self* représentant la référence à l'objet lui-même, de la même manière qu'une fonction récursive fait référence à elle-même, on peut « résoudre » cette auto-référence en prenant le point fixe de ce générateur. Le point p, de coordonnées 3 et 4, sera alors représenté par :

$$
\begin{aligned}
p &= \mathbf{fix}(\mathit{MakeGenPoint}(3,4)) \\
&= \mathbf{fix}(\lambda \mathit{self}.\{x \mapsto 3, \\
&\qquad\qquad y \mapsto 4, \\
&\qquad\qquad \mathit{distFromOrig} \mapsto \sqrt{\mathit{self}.x^2 + \mathit{self}.y^2} \\
&\qquad\qquad \mathit{closerToOrig} \mapsto \\
&\qquad\qquad\quad \lambda p.(\mathit{self}.\mathit{distFromOrig} < p.\mathit{distFromOrig})\})
\end{aligned}
$$

Comment tenir compte de l'héritage ?

- Que se passe-t'il lorsqu'on veut étendre la classe `Point` ?
- Considérons l'extension suivante dans le pseudo-code précédent :

```
class Circle(a, b, r) inherits Point(a, b)
   method radius = r
   method distFromOrig() =
              max(super.distFromOrig - self.radius, 0)
```

- En plus des références à *self*, il faut être en mesure de résoudre les références à *super*.

  ♠ informatiquement **self** est un pointeur (le receveur `this` de C++ ou Java) mais **super** est une *notation* qui "pointe" la méthode *Point::* `distFromOrig` effectivement appellée ! ♣

Comment tenir compte de l'héritage ? 2

♠ Pour résoudre le **`super`**.... ♣

- À cette fin, cette extension va être représentée par ce que Cook et Palsberg ont appelé une *enveloppe* (ou « *wrapper* ») :

$$
\begin{aligned}
CircleWrapper(a, b, r) = \lambda self.\lambda super \\
\{radius \mapsto r, \\
distFromOrig \mapsto \\
max(super.distFromOrig - self.radius, 0)\}
\end{aligned}
$$

- L'idée est de « passer en paramètre » la « partie » de l'objet cercle qui est héritée de la classe `Point` de manière à y référer par *super*.
- La construction du générateur pour la classe `Circle` va être un peu plus ardue que dans le cas de `Point`. Cela demande la composition entre l'enveloppe de `Circle` et le générateur de `Point` avant de prendre le point fixe.

Comment tenir compte de l'héritage ? 3

- Le point fixe devra permettre de résoudre correctement les références à *self* de `Point`, c'est-à-dire faire en sorte que ces références regardent d'abord les définitions de `Circle` avant celles de `Point`.
- Cook et Palsberg ont défini un opérateur de composition $\triangleright$ prenant l'enveloppe de la sous-classe et le générateur de la superclasse pour retourner le générateur de la sous-classe :

$$W \triangleright G = \lambda self.(W(self)(G(self)) \boxplus G(self))$$

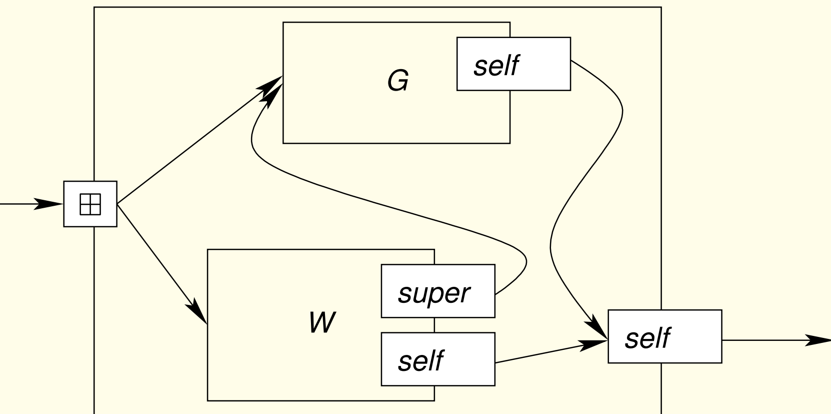
où $\boxplus$ est l'opérateur de combinaison des fonctions représentant les objets qui, conceptuellement, remplace toutes les définitions communes de son opérande de droite par celles de son opérande de gauche.

Comment tenir compte de l'héritage ? 4

Exemple :

$$
\begin{array}{rcl}
P & = & (MakeGenPoint(3,4)\ self) \\
 & = & \{x \mapsto 3,\ y \mapsto 4, \\
 & & \ distFromOrig \mapsto \sqrt{self.x^2 + self.y^2} \\
 & & \ closerToOrig \mapsto \lambda p.(self.distFromOrig < p.distFromOrig)\}) \\
C & = & ((CircleWrapper(3,4,2)\ self)\ P) \\
 & = & \{radius \mapsto 2, \\
 & & \ distFromOrig \mapsto max(super.distFromOrig - self.radius, 0)\} \\
C \boxplus P & = & \{x \mapsto 3,\ y \mapsto 4, \\
 & & \ closerToOrig \mapsto \lambda p.(self.distFromOrig < p.distFromOrig)\}) \\
 & & \ radius \mapsto 2, \\
 & & \ distFromOrig \mapsto max(super.distFromOrig - self.radius, 0)\}
\end{array}
$$

Visualisation de l'opérateur de combinaison



1 ♠ compléments pratiques

2 Sémantique dénotationnelle de BOPL

- Représentations des objets, classes et méthodes pour BOPL

Principes de base

- Inspirée du modèle des générateurs et enveloppes de Cook et Palsberg.
- Deux entités plutôt qu'une :
  - les classes, et leurs enveloppes pour gérer le *super*,
  - les objets et leurs générateurs pour gérer le *self*.
- Les méthodes : entités qui prennent *super* et *self* en paramètres :
  - le *super* est lié statiquement par la classe, alors que
  - le *self* est lié à la création des instances

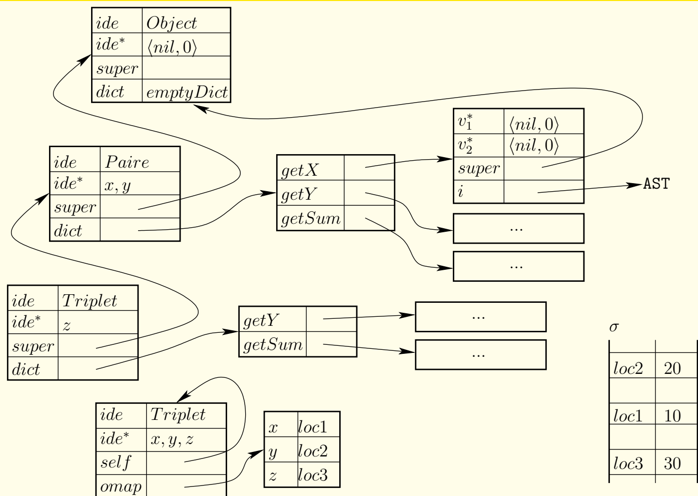
1. ♠ compléments pratiques

2. Sémantique dénotationnelle de BOPL
   - Représentations des objets, classes et méthodes pour BOPL

Un exemple de programme

```
program
  Class Paire is
  vars  Int x, y ;
  methods
    Int getX() begin return self.x end
    Int getY() begin return self.y end
    Int getSum() begin return self.getX() + self.getY() end
  end
  Class Triplet entends Pair is
  vars  Int z ;
  methods
    Int getZ() begin return self.z end
    Int getSum() begin return super.getSum() + self.getZ() end
  end
let
  Paire p ;
  Triplet t ;
in begin
  #...
end
```

Représentation des entités dans la SD



Les catégories syntaxiques et la grammaire abstraite

p	$\in$	*Program*	p	$::=$	`program` $c^* \; v^* \; i$
c	$\in$	*Class*	c	$::=$	`class` $id \; ce \; v^* \; m^*$
ce	$\in$	*CExp*	ce	$::=$	`cexp` id
v	$\in$	*Var*	v	$::=$	`var` $ce \; id$
m	$\in$	*Method*	m	$::=$	`method` $id \; v_1^* \; ce \; v_2^* \; i$
i	$\in$	*Instructions*	i	$::=$	`seq` $i_1 \; i_2$ | `assign` $id \; e$ | `writefield` $e_1 \; id \; e_2$ |
e	$\in$	*Expressions*			`if` $e \; i_1 \; i_2$ | `while` $e \; i$ | `return` e | `writeln` e
id	$\in$	*Identifiers*	e	$::=$	n | `true` | `false` | `not` e | `nil` | `self` | `super` |
n	$\in$	*Numbers*			`new` ce | `instanceof` $e \; ce$ | `methodcall` $e \; id \; e^*$ |
					`readfield` $e \; id$ | `plus` $e_1 \; e_2$ | `minus` $e_1 \; e_2$ |
					`times` $e_1 \; e_2$ | `equal` $e_1 \; e_2$ | `and` $e_1 \; e_2$ | `or` $e_1 \; e_2$ |
					`less` $e_1 \; e_2$

Les domaines sémantiques

$$
\begin{array}{rcll}
\mathbf{T} & = & \{\mathit{true}, \mathit{false}, \bot_{\mathbf{T}}\} & \\
\mathbf{Z} & = & \{\ldots, -2, -1, 0, 1, 2, \ldots\} \cup \{\bot_{\mathbf{Z}}\} & \\
\mathbf{V} & = & \mathbf{T} \oplus \mathbf{Z} & \\
\mathbf{Ide} & = & \textit{non-spécifié} & \\
\mathbf{Address} & = & \textit{non-spécifié} & \\
\mathbf{Loc} & = & \mathbf{Address} & \\
\mathbf{Oid} & = & \mathbf{Address} \oplus \{\mathit{nil}, \bot_{\mathbf{Nil}}\} & \\
\mathbf{LV} & = & \mathbf{Loc} & \text{// Left Values} \\
\mathbf{RV} & = & \mathbf{V} \oplus \mathbf{Oid} & \text{// Right Values}
\end{array}
$$

$$
\begin{array}{rcll}
\mathbf{U} & = & \{\mathit{unbound}, \bot_{\mathbf{U}}\} & \\
\mathbf{DV} & = & \mathbf{LV} \oplus \mathbf{Class} \oplus \mathbf{Object} \oplus \mathbf{U} & \text{// Denotable Values} \\
\mathbf{EV} & = & \mathbf{RV} & \text{// Expression Values} \\
\mathbf{PV} & = & \mathbf{RV} & \text{// Parameter Values} \\
\mathbf{UU} & = & \{\mathit{undefined}, \mathit{unused}, \bot_{UU}\} & \\
\mathbf{SV} & = & \mathbf{RV} \oplus \mathbf{Object} \oplus \mathbf{UU} & \text{// Storable Values}
\end{array}
$$

Les fonctions sémantiques : environnement

$$
\begin{array}{rcl}
\mathbf{Env} & = & \mathbf{Ide} \to \mathbf{DV} \\
\mathit{emptyEnv} & : & \mathbf{Env} \\
 & = & \lambda \mathit{ide}.\mathit{in}\mathbf{DV}_4(\mathit{unbound}) \\
\mathit{extendEnv} & : & \mathbf{Env} \to \mathbf{Ide} \to \mathbf{DV} \to \mathbf{Env} \\
 & = & \lambda\rho.\lambda \mathit{ide}.\lambda \mathit{dv}.\lambda \mathit{ide}_1.\mathbf{if}\ \mathit{ide}_1 = \mathit{ide}\ \mathbf{then}\ \mathit{dv}\ \mathbf{else}\ (\rho\ \mathit{ide}_1) \\
\mathit{extendEnv}* & : & \mathbf{Env} \to \mathbf{Ide}^* \to \mathbf{DV}^* \to \mathbf{Env} \\
 & = & \lambda\rho.\lambda \mathit{ide}^*.\lambda \mathit{dv}^*. \\
 & & \quad ((((\mathbf{fix}\ \lambda f.\lambda \mathit{ide}^*.\lambda \mathit{dv}^*.\lambda\rho. \\
 & & \quad\quad \mathbf{if}\ \neg \mathit{null}(\mathit{ide}^*) \\
 & & \quad\quad \mathbf{then}\ (((f\ (\mathit{tail}\ \mathit{ide}^*))\ (\mathit{tail}\ \mathit{dv}^*)) \\
 & & \quad\quad (((\mathit{extendEnv}\ \rho)\ (\mathit{head}\ \mathit{ide}^*))\ (\mathit{head}\ \mathit{dv}^*))) \\
 & & \quad\quad \mathbf{else}\ \rho) \\
 & & \quad \mathit{ide}^*)\ \mathit{dv}^*)\ \rho)
\end{array}
$$

Les fonctions sémantiques : la mémoire 1

$$
\begin{array}{rcl}
\Sigma & = & \mathbf{Loc} \rightarrow \mathbf{SV} \\
\mathit{emptyStore} & : & \Sigma \\
 & = & \lambda l.in\mathbf{SV}_3(\mathit{unused}) \\
\mathit{updateStore} & : & \Sigma \rightarrow \mathbf{Loc} \rightarrow \mathbf{SV} \rightarrow \Sigma \\
 & = & \lambda\sigma.\lambda l.\lambda sv.\lambda l_1.\mathbf{if}\ l = l_1\ \mathbf{then}\ sv\ \mathbf{else}\ (\sigma\ l_1) \\
\mathit{updateStore}* & : & \Sigma \rightarrow \mathbf{Loc}^* \rightarrow \mathbf{SV}^* \rightarrow \Sigma \\
 & = & \lambda\sigma.\lambda l^*.\lambda sv^*. \\
 & & \quad ((((\mathbf{fix}\ \lambda f.\lambda l^*.\lambda sv^*.\lambda\sigma. \\
 & & \qquad \mathbf{if}\ \neg null(l^*) \\
 & & \qquad \mathbf{then}\ (((f\ (tail\ l^*))\ (tail\ sv^*)) \\
 & & \qquad\qquad (((\mathit{updateStore}\ \sigma)\ (head\ l^*))\ (head\ sv^*))) \\
 & & \qquad \mathbf{else}\ \sigma)\ \ l^*)\ sv^*)\ \sigma)
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{allocate} & : & \Sigma \rightarrow \Sigma \otimes \mathbf{Loc} \\
 & = & \lambda\sigma.\langle(((\mathit{updateStore}\ \sigma)\ l)\ in\mathbf{SV}_3(\mathit{undefined})), l\rangle \\
 & & \quad \mathit{where}\ l \in \mathbf{Loc} \mid is\mathbf{UU}(\sigma\ l) \wedge out\mathbf{UU}(\sigma\ l) = \mathit{unused}
\end{array}
$$

Les fonctions sémantiques : la mémoire 2

$$
\begin{array}{rcl}
\textit{allocate}* & : & \Sigma \rightarrow \mathbb{N} \rightarrow \Sigma \otimes \mathbf{Loc}^* \\
& = & \lambda\sigma.\lambda n. \\
& & \quad (((\textit{fix}\lambda f.\lambda\sigma.\lambda n. \\
& & \qquad \textbf{if } n = 0 \textbf{ then } \langle \sigma, \langle \textit{nil}, 0 \rangle \rangle \\
& & \qquad \textbf{else} \\
& & \qquad\quad \textbf{let } p_1 = ((f\ \sigma)\ (-\ n\ 1)) \\
& & \qquad \textbf{and } p_2 = (\textit{allocate}\ (\textit{first}\ p_1)) \\
& & \qquad \textbf{in } \langle (\textit{first}\ p_2), (\textit{prefix}\ (\textit{second}\ p_2)\ (\textit{second}\ p_1)) \rangle) \\
& & \quad \sigma)\ n)
\end{array}
$$

$$
\begin{array}{rcl}
\textit{deallocate} & : & \Sigma \rightarrow \mathbf{Loc} \rightarrow \Sigma \\
& = & \lambda\sigma.\lambda l.\lambda l_1.\textbf{if } l = l_1 \textbf{ then } \textit{in}\mathbf{SV}_3(\textit{unused}) \textbf{ else } (\sigma\ l_1) \\
\textit{deallocate}* & : & \Sigma \rightarrow \mathbf{Loc}^* \rightarrow \Sigma \\
& = & \lambda\sigma.\lambda l^*.(((\textbf{fix}\ \lambda f.\lambda\sigma.\lambda l^*. \\
& & \qquad \textbf{if } \textit{null}(l^*) \textbf{ then } \sigma \\
& & \qquad \textbf{else } ((f\ ((\textit{deallocate}\ \sigma)\ (\textit{head}\ l^*)))\ (\textit{tail}\ l^*))) \\
& & \qquad \sigma)\ l^*)
\end{array}
$$

Les fonctions sémantiques : dictionnaire de méthodes

$$
\begin{array}{rcl}
\mathbf{Method} & = & \mathbf{Object} \rightarrow \mathbf{Env} \rightarrow \Sigma \rightarrow \mathbf{Out} \rightarrow \mathbf{PV}^* \rightarrow (\mathbf{EV} \otimes \Sigma \otimes \mathbf{Out}) \\
\mathbf{Dict} & : & \mathbf{Ide} \rightarrow (\mathbf{Method} \oplus \mathbf{U}) \\
\mathit{emptyDict} & : & \mathbf{Dict} \\
 & = & \lambda ide.in(\mathbf{Method} \oplus \mathbf{U})_2(\mathit{unbound}) \\
\mathit{extendDict} & : & \mathbf{Dict} \rightarrow \mathbf{Ide} \rightarrow (\mathbf{Method} \oplus \mathbf{U}) \rightarrow \mathbf{Dict} \\
 & = & \lambda d.\lambda ide.\lambda m. \\
 & & \quad \lambda ide_1.\mathbf{if}\ ide = ide_1\ \mathbf{then}\ in(\mathbf{Method} \oplus \mathbf{U})_1(m)\ \mathbf{else}\ (d\ ide_1)
\end{array}
$$

Les fonctions sémantiques : dictionnaire de méthodes 2

$$
\begin{array}{rcl}
\mathit{make\text{-}method} & = & \lambda ide_1^*.\lambda ide_2^*.\lambda i.\lambda super. \\
& & \quad \lambda self.\lambda\rho.\lambda\sigma.\lambda o.\lambda pv^*. \\
& & \qquad \textbf{let}* \ \langle \sigma_1, loc^* \rangle = (allocate* \ \sigma \ \sharp pv^*) \\
& & \qquad \textbf{and} \ \rho_1 = (((extendEnv* \ \rho) \ ide_1^*) \ in\mathbf{DV}_1^*(in\mathbf{LV}_1^*(loc^*))) \\
& & \qquad \textbf{and} \ \sigma_2 = (((updateStore* \ \sigma_1) \ loc^*) \ \mathbf{PV}^* to\mathbf{SV}^*(pv^*)) \\
& & \qquad \textbf{and} \ \langle \sigma_3, loc_3^* \rangle = (allocate* \ \sigma_2 \ \sharp v_2^*) \\
& & \qquad \textbf{and} \ \rho_3 = (((extendEnv* \ \rho_1) \ ide_2^*) \ in\mathbf{DV}_1^*(in\mathbf{LV}_1^*(loc_3^*))) \\
& & \qquad \textbf{and} \ \langle \sigma_4, loc_4 \rangle = (allocate \ \sigma_3) \\
& & \qquad \textbf{and} \ \rho_4 = (((extendEnv* \ \rho_3) \ \mathscr{I}[\![\mathtt{return}]\!]) \ in\mathbf{DV}_1(in\mathbf{LV}_1(loc_4))) \\
& & \qquad \textbf{and} \ \rho_5 = (((extendEnv \ (((extendEnv \ \rho_4) \ \mathscr{I}[\![\mathtt{self}]\!]) \ in\mathbf{DV}_3(self))) \\
& & \qquad\qquad\qquad\qquad \mathscr{I}[\![\mathtt{super}]\!]) \ in\mathbf{DV}_3(super)) \\
& & \qquad \textbf{and} \ \langle \sigma_5, o_1 \rangle = \mathscr{C}[\![i]\!]\rho_5\sigma_4 o \\
& & \qquad \textbf{in} \ \langle \mathbf{SV} to\mathbf{EV}(\sigma_5 \ out\mathbf{Loc}(out\mathbf{LV}(\rho_5 \ \mathscr{I}[\![\mathtt{return}]\!]))), \\
& & \qquad\quad (deallocate* \ (deallocate* \ (deallocate \ \sigma_5 \ loc_4) \ loc_3^*) \ loc^*), \\
& & \qquad\quad o_1 \rangle
\end{array}
$$

Les fonctions sémantiques : classes et objets

$$
\begin{array}{rcll}
\textbf{Class} & : & \mathbb{N} \rightarrow (\textbf{Dict} \otimes & \text{// recherche de méthodes} \\
 & & \quad (\Sigma \rightarrow \textbf{Oid} \otimes \Sigma) \otimes & \text{// instantiation} \\
 & & \quad (\cdot \rightarrow \textbf{Ide}^{*})) & \text{// variables d'instance} \\
\textbf{ClassWrapper} & : & \textbf{Class} \rightarrow \textbf{Class} & \\
\textbf{ObjectMap} & : & \textbf{Ide} \rightarrow \textbf{Loc} & \\
\textbf{Object} & : & \mathbb{N} \rightarrow ((\textbf{Ide} \rightarrow \Sigma \rightarrow \textbf{EV}) \otimes & \text{// readfield} \\
 & & \quad (\textbf{Ide} \rightarrow \textbf{SV} \rightarrow \Sigma \rightarrow \Sigma) \otimes & \text{// writefield} \\
 & & \quad (\textbf{Ide} \rightarrow \textbf{Env} \rightarrow \textbf{Method}) \otimes & \text{// recherche de méthodes} \\
 & & \quad (\textbf{Ide} \rightarrow \textbf{T})) & \text{// test instanceof}
\end{array}
$$

Les fonctions sémantiques : classes et objets 2

$$
\begin{aligned}
\textit{make-ClassWrapper} \quad = \quad & \lambda ide.\lambda ide^*.\lambda dict.\lambda super. \\
& \lambda n. \\
& \quad \textbf{if } (=\ n\ 0)\ \textbf{then} \\
& \quad\quad \lambda ide_1.\textbf{let}*\ found = (dict\ ide_1) \\
& \quad\quad\quad\quad \textbf{in if } is\textbf{U}(found)\ \textbf{then } ((super\ 0)\ ide_1) \\
& \quad\quad\quad\quad\quad \textbf{else } out\textbf{Method}(found) \\
& \quad \textbf{else if } (=\ n\ 1)\ \textbf{then} \\
& \quad\quad \lambda.(append\ ide^*\ (super\ 1)) \\
& \quad \textbf{else if } (=\ n\ 2)\ \textbf{then} \\
& \quad\quad \lambda\sigma.(make\text{-}object\ ide\ (append\ ide^*\ (super\ 1))\ \sigma) \\
& \quad \textbf{else } erreur
\end{aligned}
$$

Les fonctions sémantiques : la classe Object et le générateur d'objets

$$
\begin{array}{rcl}
\textit{theObjectClass} & = & \lambda n.\textbf{if}\ (=\ n\ 0)\ \textbf{then}\ \textit{emptyDict} \\
& & \quad\quad \textbf{else if}\ (=\ n\ 1)\ \textbf{then}\ \lambda.\langle \textit{nil}, 0\rangle \\
& & \quad\quad\quad \textbf{else}\ \textit{erreur}
\end{array}
$$

$$
\begin{array}{rcl}
\textit{make-ObjectGen} & = & \lambda \textit{ide}.\lambda \textit{ide}^*.\lambda\sigma. \\
& & \quad \textbf{let}\ (\textit{omap}, \sigma_1) = (\textit{make-ObjectMap}\ \textit{ide}^*\ \sigma) \\
& & \quad \textbf{in}\ \langle \lambda \textit{self}.\lambda n. \\
& & \quad\quad \textbf{if}\ (=\ n\ 0)\ \textbf{then} \\
& & \quad\quad\quad \lambda \textit{ide}.\lambda\sigma.\textbf{let}\ \textit{loc} = (\textit{omap}\ \textit{ide}) \\
& & \quad\quad\quad\quad \textbf{in if}\ \textit{is}\textbf{Address}(\textit{loc})\ \textbf{then SV}\textit{to}\textbf{EV}(\sigma\ \textit{loc}) \\
& & \quad\quad\quad\quad\quad \textbf{else}\ \textit{erreur} \\
& & \quad\quad \textbf{else if}\ (=\ n\ 1)\ \textbf{then} \\
& & \quad\quad\quad \lambda \textit{ide}.\lambda \textit{sv}.\lambda\sigma. \\
& & \quad\quad\quad\quad \textbf{let}\ \textit{loc} = (\textit{omap}\ \textit{ide}) \\
& & \quad\quad\quad\quad \textbf{in if}\ \textit{is}\textbf{Address}(\textit{loc})\ \textbf{then}\ (((\textit{updateStore}\ \sigma)\ \textit{loc})\ \textit{sv}) \\
& & \quad\quad\quad\quad\quad \textbf{else}\ \textit{erreur}
\end{array}
$$

Les fonctions sémantiques : générateurs d'objets II

$$
\begin{array}{l}
\quad \textbf{else if } (= \ n\, 2) \textbf{ then} \\
\qquad \lambda ide_1.\lambda\rho.(((out\textbf{Class}(\rho\ ide)\ 0)\ ide_1)\ self) \\
\quad \textbf{else if } (= \ n\, 3) \textbf{ then} \\
\qquad \lambda ide_1.in\textbf{EV}_1(in\textbf{RV}_1(in\textbf{V}_1(=\ ide\ ide_1))) \\
\quad \textbf{else } erreur, \\
\sigma_1\rangle
\end{array}
$$

Les fonctions sémantiques : création d'objets

$$
\begin{array}{lcl}
\textit{make-Object} & = & \lambda \textit{ide}.\lambda \textit{ide}^*.\lambda\sigma. \\
& & \quad \textbf{let*}\ \langle og, \sigma_1\rangle = (\textit{make-ObjectGen ide ide}^*\ \sigma) \\
& & \quad \textbf{and}\ o = (\textbf{fix}\ og) \\
& & \quad \textbf{and}\ \langle \sigma_2, loc\rangle = (\textit{allocate}\ \sigma_1) \\
& & \quad \textbf{in}\ \langle \textbf{Loc}\textit{to}\textbf{Oid}(loc), (((\textit{updateStore}\ \sigma_1)\ loc)\ \textit{in}\textbf{SV}_2(o))\rangle
\end{array}
$$

à suivre

Equations sémantiques, la prochaine fois....