Compilation

Implémentation des objets

SI4 — 2018-2019

Erick Gallesio

Introduction

Pour l'implémentation de la version complète de Toy, il faut:

- trouver une représentation de chacun des concepts objet en C.
- probablement implémenter une partie des fonctionnalités au travers de structures/fonctions du support d'exécution (runtime)

En particulier: il faut implémenter

- les attributs des objets (partie données)
- les méthodes des objets (partie code)
- le mot clé **this** (et **super**)
- le polymorphisme
- la liaison dynamique
- les méthodes prédéfinies (`printobj`, `typename`)
- la notion de méta-classe
- l'héritage
- …

Problèmes de production de code (1/3)

Les structures de C ne permettent pas le polymorphisme:

- Pour une classe `A`

```
class A {...};                    ⇒   typedef struct A *A;
                                      struct A { ... };
```

- Pour `B` qui hérite de `A` :

```
class B extends A  {...};         ⇒   typedef struct B *B;
                                      struct B { ... };
```

- Le polymorphisme proprement dit:

```
A a;                              ⇒   A a;
B b;                                  B b;
a = b;                                a = (B) b;
```

Problèmes de production de code (2/3)

C ne propose pas de **méthodes** ni de **surcharge**

```
class A {
    int x;
    void f(int i) { x = i; }
}
class B extends A {
    int x;
    void f(int i) { super.f(1); x = 2 *i; }
}
```

$\Longrightarrow$

```
typedef struct A *A;          struct A { int A_x; };

void _A_f(A this, int i) {              // Implémentation du f de A
  this->A_x = i;
}

typedef struct B *B;        struct B { ...; int B_x; };

void _B_f(B this, int i) {              // Implémentation du f de B
  _A_f((A) this, 1);                    // super.f(1)
  this->B_x = 2 * i;                    // x = 2 * i
}
```

Problèmes de production de code (3/3)

C ne propose pas de **liaison dynamique**:

```
A a; B b;                    A a; B b;
                    ⟹
a.f(1);                      _A_f(a, 1);
b.f(1);                      _B_f(b, 1);
if (ok) a = b;               if (ok) a = (B) b;
a.f(2)                       // Qui appeler ici: _A_f ou _B_f ?????
```

- En général, on doit traduire l'appel `a.f(1)` sans savoir qu'il existe la classe B, donc la production de code suivante ne peut pas être:

```
si type de a == A       alors A_F
sinon si type de a == B alors B_F
sinon si ...
```

- De plus, on veut produire du code **statique** (*i.e.* sans support du *runtime*)
- ⟹ l'objet repéré par `a` **doit connaître son type** pour savoir quelle méthode appeler.

On doit donc être capable d'avoir des objets qui représentent des classes (objets de type **métaclasse**).

Notion de métaclasse (1/3)

Une implémentation possible pour la **liaison dynamique**:

- utiliser la forme des objets que l'on a vu pour représenter les objets de l'arbre syntaxique.
- implémenter les méthodes de l'objet comme des pointeurs sur fonctions.

On aurait quelque chose comme:

```
typedef struct A *A;
struct A {
    int A_x;
    void (*A_f)((A this, int i); // initialisé à _A_f sur new
};

void _A_f(A this, int i) {
    this->A_x = i;
}
```

Peu efficace si on a:

- beaucoup de méthodes virtuelles
- beaucoup d'objets

Idée: Avoir un pointeur dans chaque objet qui pointe vers un descripteur de la classe.

Notion de métaclasse (2/3)

Définition:

> A chaque objet d'une classe donnée est associé un objet permettant de décrire cette classe (cet objet est une instance de la classe `metaclass`).

Selon le langage, cette description est plus ou moins riche, mais elle contient toujours les adresses des méthodes de la classe.

Le compilateur *Toy* représente les métaclasses par la structure suivante:

```
// --- Structure used for toy metaclasses
typedef struct _toy_metaclass {
  char *classname;                      // name of the class
  struct _toy_metaclass *extends_meta;  // meta of inherited class
  void* methods [];                     // array of methods addr
} _toy_metaclass;
```

De plus, le compilateur définit (durant son **bootstrap**) une métaclasse pour la classe *Object*:

```
extern
_toy_metaclass _toy_meta_Object; // The metaclass of Object
```

Notion de métaclasse (3/3)

Par conséquent, pour la méthode `A::f`

```
class A {
    int x;
    void f(int i) { x = i; }
}
```

$\Longrightarrow$ on produit ce code

```
// Code de la Méthode f
void _A_f(A this, int i) {
    this->A_x = i;
}

// Métaclasse de A
_toy_metaclass _toy_meta_A = {
     .classname    = "A",
     .extends_meta = &_toy_meta_Object,       // ⇒ Classe A hérite de Object
     .methods      = {
        _Object_printobj,                     // Méthode prédéfinie
        _Object_typename,                     // Méthode prédéfinie
        _A_f                                  // Méthode A::f
     }
};
```

Implémentation d'un objet (1/4)

Soit la classe:

```
class A {
    int x;
    void f(int i) { x = i; }
};
```

Le type permettant de représenter les objets de la classe est défini comme:

```
typedef struct A *A; // class A implementation typedef
struct A {
  _toy_metaclass *_instance_of;
  int A_x;
};
```

On note que:

- seuls les attributs sont dans la structure `A`
- les méthodes de la classe seront implémentées par des fonctions C
- les pointeurs sur ces fonctions seront dans la métaclasse de `A`
- Tous les objets de la classe `A` se partagent le pointeur sur sa métaclasse.

Implémentation d'un objet (2/4)

La création d'un nouvel objet (par `new` en *Toy*) provoque l'exécution du **constructeur** de la classe.

```
int main () {
   A a = new A;
   ...
}
```

$\Longrightarrow$

```
int main(void) {
  A a = _init_A(_toy_allocate_object(sizeof(struct A)));
  ...
}
```

- `_toy_allocate_object` n'est qu'un *wrapper* autour de la fonction *malloc*.
- Le constructeur de la classe s'appelle `_init_A`
- Il prend en paramètre la zone mémoire qui vient d'être allouée.

Implémentation d'un objet (3/4)

Le **constructeur** produit par le compilateur pour la classe `A` :

```
1: // Class constructor for A
2: A _init_A(A this) {
3:    _init_Object((Object) this);
4:    this->_instance_of = &_toy_meta_A;
5:
6:    this->A_x = 0;
7:    return this;
8: }
```

On voit que:

- **ligne 3**: on appelle le constructeur de la classe parente pour initialiser **this** (ici la classe *Object*)
- **ligne 4** initialisation de la métaclasse de **this** à l'objet décrivant la classe `A`
- **ligne 6**: Initialisation des champs de l'objet à leur valeur par défaut (ici un seul champ: **x**).

Implémentation d'un objet (4/4)

Héritage: Voyons ce que cela donne:

```
class B extends A {
    int x, y;
    void f(int i) { x = y = 2 *i; }
}
```

$\Longrightarrow$

```
B _init_B(B this) {        // B default class constructor impl.
  _init_A((A) this);
  this->_instance_of = &_toy_meta_B;

  this->B_x = 0;
  this->B_y = 0;
  return this;
}

void _B_f(B this, int i) {   // Method B::f implementation
  this->B_x = this->B_y = 2 * i;
}
```

Implémentation des méthodes (1/2)

Les méthodes:

- sont implémentées par des fonctions en C
- la liste de paramètres de la fonction C
  - *a toujours un premier paramètre*
    - de nom **this**
    - du type de la classe où la méthode est définie
  - *les autres paramètres (et le type de retour) sont identiques à la méthode qu'elle implémente.*

```
class A { void f(int i) { print(i); } }
```

$\Longrightarrow$

```
void _A_f(A this, int i) {
  {
    printf("%d", i);
    fflush(stdout);
  }
}
```

Implémentation des méthodes (2/2)

La liste des méthodes d'une classe est placée dans sa métaclasse:

- est ordonnée. Elle contient:
  - *les méthodes héritées* ***PUIS***
  - *les méthodes de la classe*
- elle permet de **numéroter les méthodes** de la classe
- **l'ordre doit être pérenne** lorsqu'on a de l'héritage (*polymorphisme*)

```
class A {
 void f() { ... }
 void g() { ... }

class B extends A {
  void h() { ... }
  void f() { ... }
}
```

Dans la classe `B`, la liste de méthodes que l'on peut appeler:

0 - `printobj` *(héritée de Objet)*
1 - `typename` *(héritée de Objet)*
2 - `f` *(attention* ***celle de B****, car surcharge)*
3 - `g` *(héritée de A)*
4 - `h` *(celle de B)*

Table des méthodes virtuelles (1/2)

On reprend les définitions précédentes:

```
class A {
 void f() { ... }
 void g() { ... }

class B extends A {
  void f() { ... }
  void h() { ... }
}
```

La métaclasse de B est donc représentée par la structure C suivante:

```
toy_metaclass _toy_meta_B = {
     .classname    = "B",
     .extends_meta = &_toy_meta_A,
     .methods      = {
        _Object_printobj,
        _Object_typename,
        _B_f,
        _A_g,
        _B_h}
};
```

Table des méthodes virtuelles (2/2)

La table des méthodes virtuelles est construite durant la phase d'analyse de la classe.

Principe de l'algorithme:

- Descendre la hiérarchie d'héritage et construire la liste **M** des méthodes définies:
  - *(`Object::printobj`, `Object::typename`, `A::f`, `A::g`, `B::h`, `B::f`)*
- créer une liste vide **L**
- Parcourir la liste **M** de la gauche vers la droite et pour chaque item de la forme α::β
  - *si ∃ un item de la forme χ::β à sa droite dans **M**,*
    - placer χ::β dans **L**
    - sinon placer α::β dans **L**
  - *ne rien faire si α::β est déjà dans **L***

On obtient donc ici:

- (`Object::printobj`, `Object::typename`, `B::f`, `A::g`, `B::h`)

Appel de méthodes (1/2)

Les méthodes:

- sont numérotées
- sont accessibles dans la table des méthodes virtuelles (**methods**) de la classe.

Lors de **l'analyse** d'un appel, vérifier que la méthode est bien accessible depuis la classe où la méthode est définie.

Lors de la production de code d'un appel de la forme `a.f(10)` il faut

- trouver le numéro `n` de la méthode `f` (ici c'est `2`)
- appeler la méthode `n` dans la table *methods* de la métaclasse de `a`
- passer `a` comme premier paramètre de la fonction trouvée (**this**)

On a donc:

```
a.f(10);                              // en Toy

⟹

a->instance_of.methods[2](a, 10);     // en C
```

Appel de méthodes (2/2)

```
class A           { void f(int i) { };    void g() { } }
class B extends A { void f(int i) { };    void h() { } }
...
A a = new A;
B b = new B;

a.f(1);
b.f(1);
if (ok) a = b;
a.f(12);
```

$\Longrightarrow$

```
A a = _init_A(toy_allocate_...)
B b = _init_B(toy_allocate_...)

a->instance_of.methods[2](a, 1);    // Appel de A::f (numéro 2)
b->instance_of.methods[2](b, 1);    // Appel de B::f (numéro 2)
if (ok)
  a = (A)(b);
a->instance_of.methods[2](a, 12);   // Appel méthode numéro 2 accessible depuis
  a
```

Variables temporaires (1/4)

Problème:

En pratique, l'accès à un attribut d'objet n'est pas aussi simple.

En effet, dans le cas général, pour

```
a.x
```

on ne peut pas produire

```
a->A_x;
```

car le préfixe de `x` peut être arbitrairement complexe:

```
x.g(...).x
```

Ce code provoquerait 2 appels à la méthode `g` (qui peut avoir des effets de bords).

Il faudrait donc produire:

```
{
  A tmp = x.g(...);
  tmp->A_x;
}
```

Variables temporaires (2/4)

Mais on peut être amené à produire plusieurs variables temporaires.

Soit

```
x.g(...).x = z.h(...).y;
```

$\Longrightarrow$

```
{
  A tmp = x.g(...);
  tmp-> A_x;
} = {
      B tmp = z.h(...);
      tmp->B_y;
    }
```

ne marche pas en C car les blocs ne renvoient pas de valeur.

On doit donc produire quelque chose comme:

```
{
   A tmp1 = x.g(...);
   B tmp2 = z.h(...);
   tmp1->A_x = tmp2->y;
}
```

Variables temporaires (3/4)

Si on produit des temporaires, il faut en minimiser le nombre:

- On peut déclarer toutes les temporaires de type `Object`
- Au moment de l'affectation on fait un cast en `Object`
- Il faut faire un passage sur l'arbre pour déterminer le nombre de temporaires nécessaires.

Cela complique notablement la production de code.

GCC propose une extension qui permet de transformet des blocs en expressions (CLANG dispose aussi de cette extension).

Exemple d'utilisation: la macro suivante:

```
#define max(a,b) ((a) > (b) ? (a) : (b))
```

évalue 2 fois `a` ou `b` (ce qui peut être génant si il y a des effets de bord).

Elle peut être réécrite en :

```
#define maxint(a,b) \
  ({int _a = (a), _b = (b); _a > _b ? _a : _b; })
```

Variables temporaires (4/4)

Soit

```
x.g(...).x = z.h(...).y;
```

$\Longrightarrow$

```
({ A tmp = x.g(...); tmp-> A_x;}) =
            ({ B tmp = z.h(...); tmp->B_y;} )
```

Pour simplifier le code produit; on passe par des macros (définies dans `toy_runtime.h`).

On dispose principalement de 2 macros:

- `_TOY_ACCESS(_obj, _klass)` est utilisée pour accéder à un attribut d'une instance `_obj` de la classe `_klass`
- `_TOY_INVOKE(_type, _obj, _idx, ...)` permet l'appel de la méthode de numéro `_idx` de l'objet `_obj`

De plus, ces macros vérifient que l'objet `_obj` n'est pas NULL et déclenchent une erreur sinon.

Exemples de code produit (1/3)

Soit

```
a.x = b.y;
```

$\Longrightarrow$

```
_TOY_ACCESS(a,A)->A_x = _TOY_ACCESS(b,B)->B_y;
```

$\Longrightarrow$

```
({ A _tmp_ = a;
   if (! _tmp_) _toy_nullaccess("foo.c", 102);
   _tmp_;})->A_x =
({ B _tmp_ = b;
   if (! _tmp_) _toy_nullaccess("foo.c", 102);
   _tmp_;})->B_y;
```

Exemples de code produit (2/3)

Soit

```
a.f(1234);      // f est une méthode de type void
```

$\Longrightarrow$

```
_TOY_INVOKE(void, a, 2, _this, 1234);  // Indice fonction = 2
```

$\Longrightarrow$

```
({ Object _this = (Object) a;
   if (! _this) _toy_nullaccess("foo.c", 102);
   ((void (*)()) (_this->_instance_of->methods[2]))(_this, 1234);})
```

Exemples de code produit (3/3)

```
a.g().x = b.h().y;     // g renvoit un A et h un B
```

$\Longrightarrow$

```
_TOY_ACCESS(_TOY_INVOKE(A, a, 3, _this),A)->A_x =
_TOY_ACCESS(_TOY_INVOKE(B, b, 4, _this),B)->B_y;
```

$\Longrightarrow$

```
({ A _tmp_ = ({ Object _this = (Object) a;
      if (! _this) _toy_nullaccess("foo.c", 102);
      ((A (*)())(_this->_instance_of->methods[3]))(_this);});
  if (! _tmp_) _toy_nullaccess("foo.c", 102); _tmp_; })->A_x =
({ B _tmp_ = ({ Object _this = (Object) b;
      if (! _this) _toy_nullaccess("foo.c", 102);
      ((B (*)())(_this->_instance_of->methods[4]))(_this);});
  if (! _tmp_) _toy_nullaccess("foo.c", 102); _tmp_; })->B_y;
```

Intérêt des macros:

- code plus facile à produire;
- (accessoirement) code produit plus lisible;
- possibilité de désactiver les contôles lorsque le programme est testé.

Gestion des désignations (1/3)

Syntaxe des désignation de Toy:

```
expr:           ...
                designator
                ...

var:            identifier
        |       designator '.' identifier
        ;

call:           var '(' eparam_list ')'
        ;

designator:     var
        |       call
        |       KNEW identifier
        |       KTHIS
        |       KSUPER
        ;
```

Avec l'introduction des objets dans Toy, on peut avoir:

- `a.b.p(i,c.k).d.q().x`
- `this.a`
- `super.b`
- `new A.init(1, 2, 3)`

Gestion des désignations (2/3)

Au niveau syntaxique, pour implémenter les désignations, on ne fait qu'ajouter le préfixe à l'identificateur.

Ainsi, on a:

```
var:       identifier
                    { $$ = make_identifier($1); }
         |   designator '.' identifier
                    { $$ = add_prefix_to_identifier($3, $1);}
         ;
```

Lorsqu'un identificateur est préfixé, le préfixe sera donc

- accessible depuis l'identificateur le plus à droite (de proche en proche)
- typé lors de l'analyse sémantique (avec vérification que l'identificateur situé à droite est bien un membre du préfixe).

Gestion des désignations (3/3)

Suite des règles sur les désignations:

```
call:           var '(' eparam_list ')'
                                { $$ = make_call($1, $3); }
        ;

designator:     var             { $$ = $1; }
        |       call            { $$ = $1; }
        |       KNEW identifier { $$ = make_instance_new(
                                          make_type($2, "NULL"));}
        |       KTHIS           { $$ = make_this(); }
        |       KSUPER          { $$ = make_super(); } // ABSENT
        ;
```

Ces règles permettent donc bien la construction de l'AST pour

- `a.b.p(i,c.k).d.q().x`
- `this.a`
- `super.b`
- `new A.init(1, 2, 3)`

Désignations et arbre abstrait (1/3)

Pour représenter un identificateur dans l'arbre:

En *Toy base*:

```
struct s_identifier {
  ast_node header;  ///< AST header
  char *value;      ///< value of the identifier (a string)
};

ast_node *make_identifier(char *id);
```

En *Toy*:

```
struct s_identifier {
  ast_node header;      ///< AST header
  char *value;          ///< value of the identifier (a string)
  ast_node *in_class;   ///< Embedding class for attributes or NULL
  ast_node *prefix;     ///< prefix of the identifier (or NULL)
  ast_node *qualified;  ///<  'prefix.ident' object after analysis.
};

ast_node *make_identifier(char *id);
ast_node *add_prefix_to_identifier(ast_node *ident,
                                   ast_node *prefix);
```

Désignations et arbre abstrait (2/3)

Trois nouveaux champs:

- **prefix**: contient l'arbre syntaxique.
  Si on a `x.y.z`
  - *z est présent dans l'arbre*
  - *le préfixe de z est l'identificateur y (qui a comme préfixe x)*

  Si on a `this.x`
  - *x est présent dans l'arbre*
  - *son préfixe est un nœud de type `this`.*
- **in_class**: contient un pointeur sur la classe de définition d'un champ Ce champ sert principalement à la production de code:
  Si on a: `c.x` on produit `_TOY_ACCESS(c,C)->_A_x)`
  - *si `c` est de type C*
  - *si `x` est déclaré dans la classe `A` héritée par `C`*
- **qualified**: pourrait être omis.
  Utilisé pour éviter plusieurs recherches dans la table des symboles.

Désignations et arbre abstrait (2/3)

Les préfixes sont construit lors de la phase d'analyse syntaxique.

Le compilateur peut aussi introduire des préfixes lors de l'analyse.

```
class B {
    int x;
    A a;

    void foo() {
      print(x);       // équivalent this.x
      print(a.z);     // équivalent this.a.z
    }
    ...
}
```

Ici le compilateur

- doit introduire **this** dans le préfixe de `x` et de `a`.
- produira du code pour accéder à
  - *this->_B_x;*
  - *this->_B_a->_A_z*

Analyse d'un identificateur préfixé

```
static void prefix_analysis(ast_node *node, char *id){
  analysis(IDENT_PREFIX(node));

  ast_node *prefix_type = AST_TYPE(IDENT_PREFIX(node));

  AST_TYPE(node) = NULL;        // will be changed if no error
  if (!valid_type(prefix_type)) return;

  if (TYPE_IS_STANDARD(prefix_type)) {
    error_msg(node, "prefix of '%s' is not a class instance", id);
    return;
  }

  // Verify that the class has the given identifier as field
  ast_node *member = symbol_table_search_member(id, TYPE_NAME(prefix_type));
  if (!member) {
    error_msg(node, "member does not exist in class ");
    return;
  }

  // Copy the class definition of member in the identifier
  IDENT_IN_CLASS(node) = IDENT_IN_CLASS(member);
  // Fill the member slot for further analyzes and type the node
  IDENT_QUALIFIED(node) = member;
  AST_TYPE(node)        = AST_TYPE(member);
}
```

Dénotation: production de code

Tout est dans la production de code d'un identificateur.

```
void produce_code_identifier(ast_node *node){
  ast_node* klass= IDENT_IN_CLASS(node);

  // Produce prefix if there is one
  if (IDENT_PREFIX(node)) {
    ast_node* prefix = IDENT_PREFIX(node);
    char* type       = TYPE_NAME(AST_TYPE(prefix));
    emit("_TOY_ACCESS(");
    code(prefix);
    emit(",%s)->", type);
  }

  // Produce code of the identifier
  if (klass) // Add prefix if it's a field
     emit("_%s_", IDENT_VAL(CLASS_NAME(klass)));
  emit("%s", IDENT_VAL(node));
}
```

Si on a: `c.x` on produit `_TOY_ACCESS(c,C)->_A_x)` si

- `c` est de type `C` et
- `x` est déclaré dans la classe `A` héritée par `C`

Appel: production de code (1/2)

La même fonction gère les appels de fonctions et de méthodes:

```
void produce_code_call(ast_node *node) {
  struct s_call *n = (struct s_call *) node;
  int comma = 0;

  if (IDENT_PREFIX(n->callee) == NULL) {    // - Function call
    code(n->callee); emit("(");
  }
  else {                                    // - Method call
    // Prochain slide ....
  }

  // produce the parameters
  FORLIST(p, n->parameters) {
    if (comma++) emit(", ");
    code_expr_cast(list_item_data(p));
  }
  emit(")");
}
```

Appel: production de code (2/2)

Le code pour les méthode est:

```
{
  ast_node *method = IDENT_QUALIFIED(n->callee);

  emit("_TOY_INVOKE(");
  code(AST_TYPE(method)); emit(", ");   // Result type
  code(IDENT_PREFIX(n->callee));        // this
  emit(", %d", FUNCTION_INDEX(method)); // method index in vtable
  emit(", _this");                      // First method parameter
  comma++;                              // if method has params
}
```

Ainsi, `int x = a.foo(10, 20)`

```
⟹    int x = _TOY_INVOKE(int, a, 2, _this, 10, 20);
```

```
⟹    int x = ({ Object _this = (Object) a;
       if (! _this) _toy_nullaccess("zoo.c", 40);
       ((int (*)()) (_this->_instance_of->methods[2]))
                                    (_this, 10, 20);});
```