DLP : Développement d'un langage de programmation

Master STL, Université Pierre et Marie Curie

Année 2020–2021

TME 8 – Accès réflexif aux champs des objets

Antoine Miné & Christian Queinnec

1 Présentation

Dans le langage ILP4, l'ensemble des champs d'un objet est défini lors de sa création, par la classe choisie. C'est une donnée statique : l'ensemble des champs n'évolue pas lors de l'exécution du programme (seule la valeur des champs évolue dynamiquement). Par ailleurs, l'accès aux champs se fait par des instructions `obj.field` (en lecture) et `obj.field = expr` (en écriture), en passant comme nom de champ, `field`, une chaîne de caractères constante (et non une expression arbitraire). Ceci permet une interprétation et surtout une compilation efficace des objets comme des tableaux de champs, `fields`, de taille fixe. Le compilateur détermine une fois pour toutes, pour chaque nom de champ `field`, un indice `i` fixe et unique dans tout le programme. Un accès au champ est alors traduit comme un accès de tableau : `fields[i]`. Cette implantation des objets se retrouve dans de nombreux langages, comme C++ et Java.

D'autres langages, comme JavaScript ou Python, proposent des systèmes d'objets plus dynamiques :

1. Il est possible d'accéder à un champ dont le nom est calculé à l'exécution, comme le résultat de l'évaluation d'une expression arbitraire. Nous pouvons par exemple écrire :

   ```
   i = obj["to" + "to"];
   obj[f(x)] = 12;
   obj has ("x" + v);
   ```

   ce qui permet, respectivement, de lire, modifier, ou tester l'existence d'un champ d'un objet étant donné le nom du champ.

2. Il est possible d'ajouter des champs dynamiquement. Ainsi :

   ```
   x["a"] = 12;
   y = x["a"];
   ```

   est correct même si `x` ne possède pas de champ `a` lors de sa création (i.e., la classe de `x` ne déclare pas `var a`).

En un sens, ces deux traits reviennent à considérer les objets comme des tables d'associations indexées par des chaînes de caractères arbitraires, et ces tables sont modifiables et extensibles à souhait.

Objectif. L'objectif de ce TME est d'ajouter ces deux traits d'objets dynamiques au langage ILP4, à son interprète et à son compilateur.

Buts :
- comprendre les modèles objets à champs statiques et à champs dynamiques ;
- revoir la procédure d'ajout d'instructions ou de primitives dans le langage ;
- comprendre et modifier la représentation des objets dans l'interprète ;
- comprendre et modifier la représentation des objets dans la bibliothèque d'exécution C et dans la compilation vers C.

Version d'ILP. Ce TME et les TME suivants seront basés sur le langage ILP4. Vous devrez donc commencer par faire un *fork* personnel du projet ILP4 sur le serveur GitLab du cours, puis par importer localement le projet dans Eclipse.

Dans ce TME, nous travaillerons dans le *package* `com.paracamplus.ilp4.ilp4tme8`.

2 Travail à réaliser

2.1 Syntaxe

Nous allons garder intacte la syntaxe existante, `x.field`, qui utilise un point pour accéder aux champs statiques (définis dans la classe), et nous allons ajouter une syntaxe à base de crochets pour accéder aux champs dynamiquement, comme suit :

1. `object[property]`
2. `object[property] = value`
3. `object has property`

où `object`, `property` et `value` sont des expressions arbitraires. Ces instructions vont respectivement lire un champ, modifier un champ et tester l'existence d'un champ. Nous appellerons les nœuds AST correspondants : `ASTreadProperty`, `ASTwriteProperty`, `ASThasProperty`.

Travail à réaliser : étendez la grammaire ANTLR de ILP4, l'AST, et l'analyse syntaxique avec ces trois instructions.

Note : notre syntaxe utilise le terme *property* par analogie avec JavaScript, où ce terme est synonyme avec la notion de champ.

2.2 Accès réflexif aux champs de classes

Dans cette partie, nous ne traitons que le premier trait : l'accès aux champs de nom calculés, mais pas (encore) l'ajout dynamique de champ :

— `ASThasProperty` retourne un booléen, `true` ou `false`, selon que le champ donné existe dans la classe de l'objet ou non ;

— `ASTreadProperty` retourne la valeur du champ spécifié de l'objet, ou lève une exception si le champ n'existe pas dans la classe de l'objet ;

— `ASTwriteProperty` modifie la valeur du champ spécifié de l'objet et retourne la valeur précédente du champ, ou lève une exception si le champ n'existe pas dans la classe de l'objet.

Dans tous les cas, une exception sera levée si le calcul du nom de champ donne un objet qui n'est pas une chaîne de caractères.

2.2.1 Interprète

Dans l'interprète, les objets sont représentés par des instances de la classe `ILPInstance`. Cette classe possède déjà toutes les méthodes nécessaires pour manipuler un champ en connaissant son nom. L'extension de l'interprète est donc relativement simple.

Travail à réaliser : étendez la classe `Interpreter` pour gérer les nouvelles instructions. Développez des tests, que vous ajouterez à l'intégration continue sur le serveur GitLab du cours.

2.2.2 Compilateur et bibliothèque d'exécution

La lecture des classes `ASTCfieldRead`, `ASTCfieldWrite` et `Compiler` dans le code d'ILP4 montre que l'accès classique à un champ est transformé, dès la compilation, en un accès à un indice constant dans le tableau de champs de l'objet. Ce mécanisme n'est plus possible si le nom du champ n'est connu qu'à l'exécution. Le code C généré pour nos nouvelles instructions devra donc transmettre à la bibliothèque d'exécution ce champ calculé dynamiquement sous forme d'une valeur `ILP_Object` qui devra contenir une chaîne de caractères. Par exemple, `ASThasProperty` pourra se compiler en un appel à la fonction C suivante, que vous devrez ajouter à la bibliothèque d'exécution :

```
ILP_Object* ILP_has_property(ILP_Object* target, ILP_Object* property);
```

où `target` dénote une instance de classe ILP, `property` dénote une chaîne de caractères ILP, et la fonction retourne une valeur booléenne ILP. De même, des fonctions `ILP_read_property` et `ILP_write_property` pourront être ajoutées à la bibliothèque d'exécution.

La lecture du code de la bibliothèque d'exécution, dans `ilp.h`, montre quant-à-elle que, à toute valeur `ILP_Object` est associée une classe `ILP_Class`, et que cette classe contient une liste chaînée de champs `ILP_Field`, incluant leur nom et position dans le tableau de champs de l'objet. Toute l'information nécessaire pour retrouver un champ connaissant son nom est donc bien disponible à l'exécution.

Travail à réaliser : implantez le support pour `ASThasProperty`, `ASTreadProperty` et `ASTwriteProperty` dans le compilateur, en prenant garde à gérer tous les cas qui génèrent une exception. Nous suggérons de commencer par implanter une fonction utilitaire :

```
ILP_Object* ILP_lookup_property(ILP_Object target, char* property);
```

qui, étant donnés un objet `target` et un nom de champ `property`, retourne l'adresse en mémoire où la valeur du champ correspondant est stockée dans l'objet, ou `NULL` si aucun champ de ce nom n'existe. Retourner un pointeur plutôt qu'un indice permettra de facilement réutiliser le code de `ILP_has_property`, `ILP_read_property`, `ILP_write_property` même si la structure de donnée stockant les champs évolue (ce qui sera le cas à la question suivante).

N'oubliez pas d'ajouter des tests pour vérifier que votre implantation est correcte.

2.3 Champs dynamiques

Dans ces questions, nous ajoutons le support pour l'ajout dynamique de champs. Dans la suite, nous appellerons « champ statique » tout champ qui a été déclaré dans la classe de l'objet (ou un parent de cette classe), et « champ dynamique » un champ qui a été ajouté par la suite à cet objet. Nous avons alors la sémantique suivante :

- `ASThasProperty` retourne le booléen `true` si un champ statique ou un champ dynamique existe avec le nom spécifié pour l'objet spécifié, `false` sinon ;
- `ASTreadProperty` retourne la valeur du champ statique ou dynamique spécifié de l'objet, ou lève une exception si aucun champ statique ni dynamique n'existe pour cet objet ;
- `ASTwriteProperty` modifie la valeur du champ spécifié de l'objet spécifié, et retourne la valeur précédente du champ. Si un champ statique existe, c'est celui-là qui est modifié. Sinon, si un champ dynamique existe, il est modifié. Enfin, si aucun champ statique ni dynamique avec ce nom n'existe, un nouveau champ dynamique est créé [1] ; la valeur précédente du champ (retournée par l'instruction `ASTwriteProperty`) sera par défaut le booléeen `false` pour un nouveau champ. Notez qu'il n'est pas possible d'avoir un champ statique et un champ dynamique de même nom, ou deux champs dynamiques de même nom. Si un champ statique existe, aucun champ dynamique de même nom ne sera créé.

Par ailleurs les accès classiques aux champs par les instructions `ASTfieldRead` et `ASTfieldWrite` sont inchangés : ils ne concerneront que les champs statiques et ignoreront les champs dynamiques, afin d'assurer la compatibilité.

2.3.1 Interprète

Le modèle d'exécution suggère l'implantation suivante : les objets sont enrichis par une table d'associations, associant aux champs dynamiques leur valeur. Ils gardent néanmoins le tableau de champs, qui servira toujours à stocker les champs statiques. Les opérations `ASThasProperty`, `ASTreadProperty`, `ASTwriteProperty` doivent maintenant parcourir d'abord le tableau de champs statiques puis, en cas d'échec, parcourir la table de champs dynamiques. L'avantage de cette approche est que le code existant (`ASTfieldRead`, etc.), qui se base sur l'existence d'un tableau de champs, n'a pas à être modifié.

Travail à réaliser : enrichissez la classe `ILPInstance` et l'interprète en y ajoutant la gestion des champs dynamiques. Développez des tests, que vous ajouterez à l'intégration continue sur le serveur GitLab du cours.

2.3.2 Compilateur et bibliothèque d'exécution C

Pour le compilateur, il est nécessaire d'enrichir le type des valeurs `ILP_Object` déclaré dans `ilp.h` avec une table d'associations dynamique (pour simplifier l'implantation, on pourra utiliser une simple liste chaînée). A priori, deux choix semblent possibles :

1. ajouter la table uniquement dans les instances de classe, `ILP_Object->_content.asInstance`, ce qui est cohérent avec l'interpréteur ; ou
2. ajouter la table globalement à toutes les valeurs `ILP_Object`, ce qui est cohérent avec la vision « tout objet » de la bibliothèque d'exécution, où une classe et des méthodes sont associées à toutes les valeurs, même elles ne sont pas des instances de classes ILP (comme par exemple les entiers ou les chaînes de caractères).

Un avantage de la deuxième solution est la possibilité d'ajouter un champ également à une valeur entière ou chaîne de caractères. Nous vous laissons libre dans le choix entre ces deux implantations.

À priori, peu de modifications sur la partie Java du compilateur sont nécessaires, puisque le compilateur délègue à la bibliothèque d'exécution tout le travail. Attention cependant : une modification du type `ILP_Object` peut nécessiter

1. Le champ n'est créé que pour cet objet, pas pour toute la classe. Les autres instances de cette classe, créées antérieurement ou postérieurement à l'ajout de champ, ne seront pas affectées.

une modification dans le compilateur de certaines parties qui semblent à première vue indépendantes de la gestion de nos nouvelles instructions. En particulier, le compilateur émet dans le code C généré des définitions statiques d'objets (en particulier pour représenter les classes, les champs, les méthodes), et cette génération est très sensible au nombre et à l'ordre des champs dans la structure C `struct ILP_Object`.

Travail à réaliser : ajoutez la gestion des champs dynamiques au compilateur et à la bibliothèque d'exécution. Testez.

2.4 Rendu

Vous effectuerez un rendu en vous assurant que tout le code développé a été envoyé sur le serveur GitLab (*push*) dans votre *fork* d'ILP4. Vous vous assurerez que les tests d'intégration continue développés ont bien été configurés et fonctionnent sur le serveur. Vous ajouterez un tag « rendu-initial-tme8 » en fin de séance, puis un tag « rendu-final-tme8 » quand le TME est finalisé.