Métaprogrammation dans un langage à 2 niveaux

0. Introduction

Le typage des programmes est un outil central pour réduire les erreurs de programmation. Cependant, les systèmes de types conduisent souvent le programmeur à écrire du code trop spécifique, qui doit être écrit pour un type de données particulier quand une solution générale aurait été adaptée.

Par exemple, en C, il faut dupliquer les fonctions lorsque les types de données diffèrent :

```c
struct IntList {
    int x;
    struct IntList* next;
};

IntList* int_cons(int h, IntList* t);
```

```c
struct StrList {
    char* x;
    struct StrList* next;
};

StrList* str_cons(char* h, StrList* t);
```

Une solution possible est d'utiliser le système de macros :

```c
#define List(name, T)       \
    struct name {           \
        T x;                \
        struct name* next;  \
    };

#define cons(h, t) ...
```

Mais cette approche est peu lisible, peu pratique à utiliser et difficilement maintenable.

1. Présentation générale du langage

On se propose de définir et d'implémenter un langage de programmation stratifié en 2 niveaux. Le niveau 0 est le niveau d'exécution et le niveau 1 est le niveau de typage, évalué à la compilation. Ce design est inspiré du `comptime` de Zig [1] et de certains éléments de la compilation par étapes de András Kovács [2].

Le langage admet les constructions primitives usuelles retrouvées dans un langage fonctionnel pur : variables immutables, fonctions capturant la portée lexicale dans lesquelles elles sont définies et pouvant être récursives, filtrage par motif sur les valeurs d'un type somme. Le langage se rapproche d'OCaml dans sa sémantique générale mais s'inspire de Rust pour la syntaxe. Il est également fortement typé.

Le langage n'a que des variables immutables par simplicité, notamment car la mutation pose des problèmes de choix de sémantique sur passage par valeur ou par référence. Des problèmes de variance dus à la présence de sous-typage émergent également.

On présente les éléments principaux de la syntaxe :

```
let x = 41 // Les nouvelles lignes sont significatives.
let y = "abcd"

// Il est possible de créer des constructeurs de label optionnel
//   et avec éventuellement des données.
// Contrairement à OCaml, les constructeurs ne sont pas
//   rattachés à un type nominal.
let constructeur1 = .Foo
let constructeur2 = .Abc(43, "bonjour")
let constructeur3 = .("hello", 12)
let constructeur4 = . // Homologue de ()
```

```
let z = { // Un bloc-expression à la Rust.
    let a = 4
    let b = 5
    a + b // == 9
}

let zero = 0
let un = 1

// On définit une fonction récursive (rec) qui capture sa portée parente (zero, un).
let rec fib = fun(n: Int) -> Int {
  match n {
      0 -> zero,
      1 -> un,
      n -> fib(n + -1) + fib(n + -2)
  }
}
```

2. Niveaux

Le langage est stratifié en deux niveaux.

Le niveau 0 représente le *runtime*, c'est-à-dire l'exécution conventionelle du programme. Ce niveau est fortement et statiquement typé.

Le niveau 1 est exécuté à la compilation lors du typage du niveau 0. Les valeurs au niveau 1 permettent de typer le niveau 0 : un type est une valeur conventionnelle manipulable au niveau 1.

On crée par exemple une variable au niveau 1 avec le mot clé `const` au lieu de `let`. Elle sera seulement accessible au niveau 1, et pas au niveau 0.

```
const s = "une string"
```

Le niveau 1 peut manipuler les types comme des valeurs. En particulier, on peut assigner des types aux variables au niveau 1 :

```
const T = Int // Le type des entiers.
```

Les variables du niveau 1 ne sont pas héritées au niveau 0 car elles peuvent contenir des types, qui ne sont pas manipulables au niveau 0.

Le bloc ci-dessous est évalué au niveau 1 car c'est le côté droit d'un assignement `const`, c'est-à-dire au niveau 1 :

```
const Result = {
    let A = .Ok(Int)
    let B = .Err(String)

    A | B   // Type somme: .Ok(Int) | .Err(String)
}
```

Les expressions au niveau 1 peuvent servir à typer les expressions au niveau 0.

```
let error: Result = .Err("418 IM_A_TEAPOT")
```

Une version générique de l'identité :

```
const id = fun(T: Type) {
    // La fonction ci-dessous est annotée par des expressions évaluées au niveau 1.
    fun (x: T) -> T {
        x
    }
}
```

Les blocks `const { ... }` exécutent l'expression au niveau 1 et placent le résultat dans l'arbre de syntaxe typé du niveau 0 :

```
let id_int = const { id(Int) }
```

On peut maintenant utiliser la fonction instantiée `id_int` au niveau 0 :

```
let x = id_int(17)
```

Le niveau peut être inféré dans le cas d'un appel d'une fonction au niveau 1.

```
let y = id(String)("(A = B) ≃ (A ≃ B)")
```

`id` est uniquement définie au niveau 1, on insère donc un bloc const implicitement. La ligne ci-dessus est transformée en :

```
let y = const { id(String) }("(A = B) ≃ (A ≃ B)")
```

Ce qui permet d'instantier `id` pour le type `String` et de réduire le résultat au niveau 0.

Il n'est pas possible de réduire un type au niveau 0. Les lignes suivantes ne compilent pas :

```
// S'évaluerait en Int, qui ne peut pas s'échapper du niveau 1.
let illegal1 = const { Int }
// S'évaluerait en String, qui ne peut pas s'échapper non plus.
let illegal2 = const { id(Type)(String) }
```

Car le niveau 0 ne peut pas manipuler les types.

On définit une fonction au niveau 1. Elle peut manipuler les types comme des valeurs.

```
const foo = fun(s: String) -> Type {
    Int
}
```

Les annotations de types sont évaluées au niveau 1. Ici `12` est évalué au niveau 0 (côté droit d'un `let` qui crée une variable au niveau 0), alors que `foo("unused")` est évalué au niveau 1 et son résultat sert au typage de la ligne.

```
let i: foo("unused") = 12
```

3. Typage

Seul le niveau 0 est typé statiquement. Les types des expressions de niveau 0 sont des expressions évaluées à la compilation, i.e. au niveau 1. On note `a: A` la relation de typage.

Le type des entiers est `Int`. Ex : `43: Int`.
Le type des chaînes de caractères est `String`. Ex : `"crab": String`.
Le type des types est `Type`.

- `String: Type`
- `.Foo(Int, String): Type`
- `Type: Type`

Les constructeurs sont typés par des constructeurs ayant comme données les types des données des valeurs (voir l'exemple ci-dessous). Ex : `.Foo(41, "Curry-Howard"): .Foo(Int, String)`.

Il est possible de combiner plusieurs types constructeur à labels disjoints en un type somme de la forme `.A | .B(String) | .C(Int)`. Les termes d'un type somme sont sous-types de ce type ; par exemple, `.B(String)` est un sous-type de `.A | .B(String) | .D(Int)`.

- `.B("foo"): .B(String)`.
- `.B("foo"): .A | .C(Int) | .B(String)`.
- `.B("foo"): .A | .B(String) | .D(Int)`.

Le type des fonctions s'écrit `Fun(A1, ..., An) -> B`. On utilise un F majuscule pour distinguer les fonctions des types de fonction car les deux sont des expressions au niveau 1.

Par exemple, si on définit :

```
let f = fun (x: Int) -> String {
    "0 = 1"
}
```

Alors `f: Fun(Int) -> String`.

Le typage est structurel, assigner un type à une variable au niveau 1 n'a pour effet que de créer un raccourci pour l'expression du type. Dans l'exemple ci-dessous, `A` et `B` sont les mêmes types.

```
const A = .Point(Int, Int)
const B = .Point(Int, Int)

let x: A = .Point(14, 70)
let y: B = x
```

L'opérateur de sommation des types évalue paresseusement ses cas afin de permettre les types récursifs :

```
const rec List = fun(T: Type) {
    .Nil | .Cons(T, List(T))
}

let xs: List(Int) = .Cons(1, .Cons(2, .Cons(3, .Nil)))
```

4. Implémentation

Le langage est implémenté en ≈3200 lignes de Rust par un interpréteur. Le niveau 0 et le niveau 1 partagent le même interpréteur pour des raisons de simplicité, avec la différence que le niveau 0 ne peut pas manipuler les types. Le niveau 0 a cependant été conçu dans l'idée d'avoir un langage implémentable par un compilateur. L'analyse syntaxique produit un arbre de syntaxe représentant le programme.

Le typage s'effectue par un parcours de l'arbre de syntaxe. Les annotations de type, les blocks `const` et les variables au niveau 1 `const v = ...` sont évalués avant de vérifier les types. Utiliser un système de métaprogrammation pour émuler des types et fonctions génériques a pour avantage de rendre le système de types monomorphe, *i.e.* le niveau 0 est typé uniquement par des types concrets. Cela permet d'inférer le type des expressions de manière très satisfaisante sans unification. L'arbre de syntaxe obtenu à la fin du typage est complètement typé par des types monomorphes, et aucune construction propre au niveau 1 n'y apparaît. L'arbre de syntaxe typé peut être enfin évalué au niveau 0.

Par exemple,

```
// Identité générique
const id = fun(T: Type) {
    fun (x: T) -> T {
        x
    }
}

// Note: on a défini cette fonction au niveau 1.
const rec fib = fun(n: Int) -> Int {
    match n {
        0 -> zero,
        1 -> un,
        n -> fib(n + -1) + fib(n + -2)
    }
}

const an_int = fun() -> Type {
    Int
}

let fib5 = const { fib(9) }
let thing: an_int() = id(Int)(45)
```

Devient, après typage et évaluation du niveau 1 :

```
let fib5 = 34
let thing: Int = (fun (x: Int) -> Int { x })(45)
```

5. Conclusion

Il est donc possible d'implémenter le polymorphisme par un système de métaprogrammation basé sur le langage des objets, sans passer par un langage distinct pour les types. Ce système permet de réaliser des fonctions polymorphiques et des types génériques assez avancés. On peut par exemple implémenter un analogue de `List.map` en OCaml :

```
const rec List = fun(T: Type) {
    .Nil | .Cons(T, List(T))
}

const map = fun(A: Type, B: Type) {
    let rec map_specialise = fun (xs: List(A), f: Fun(A) -> B) -> List(B) {
        match xs {
            .Nil -> .Nil,
            .Cons(h, t) -> .Cons(f(h), map_specialise(t, f))
        }
    }
    map_specialise
}

let add_one = fun (x: Int) -> Int {
    x + 1
}
                // [1; 2; 3]
let xs: List(Int) = .Cons(1, .Cons(2, .Cons(3, .Nil)))

map(Int, Int)(xs, add_one) // Le programme s'évalue en [2; 3; 4].
```

J'aurai aimé avoir le temps d'étendre ce langage en ajoutant des modules comme des valeurs à la compilation, permettant d'obtenir une forme de polymorphisme *ad hoc* et d'émuler les foncteurs d'OCaml.

```
// Note : ces idées n'ont pas été implémentées.
const StringModule = module {
    const t = String
    let eq = ...
}

// Un équivalent d'un foncteur
const SetModule = fun (Elt: Module) -> Module {
    module {
        const t = some_set_type(Elt.t)
        let insert = fun (set: t, element: Elt.t) -> t { ... }
    }
}
```

On pourrait également permettre au niveau 1 de manipuler des fragments d'arbre de syntaxe pour obtenir un système de macros (qui a été partiellement implémenté par des quasicitations).

De manière plus générale, l'avantage d'avoir accès au langage complet au niveau 1 rend les techniques de métaprogrammation très variées et expressives en ajoutant peu de constructions manipulables comme des valeurs par le niveau 1.

Références

- **[1]** Zig, Zig Language Reference
  - https://ziglang.org/documentation/0.13.0/#comptime
- **[2]** András Kovács, Proceedings of the ACM on Programming Languages (2022), article n°110, p.540-569, Staged Compilation with Two-Level Type Theory, DOI: 10.1145/3547641
  - https://dl.acm.org/doi/10.1145/3547641