MÉTAPROGRAMMATION DANS UN LANGAGE À 2 NIVEAUX

_

Plan

1. **Motivation**
2. **Solutions possibles**
3. **Un langage de programmation**
4. **Réalisation de l'interpréteur**
5. **Résultats**

Motivation

```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);
```

→ **Duplication de code**

Solutions possibles

- Macros : C

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

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

  - Peu lisible ni ergonomique

- Polymorphisme intégré au système de types : OCaml

```
type 'a list = Nil | Cons of 'a * 'a list
```

  - Langages des types et des objets distincts

Un langage de programmation

- Purement fonctionnel
  - Clôtures
  - Filtrage par motif
  - Variables immutables
- Structuré en 2 niveaux : niveau 0 et niveau 1

Un langage de programmation

Syntaxe

```
let x = 41
let y = "abcd"

let constructeur1 = .Foo
let constructeur2 = .Abc(43, "bonjour")
let constructeur3 = .("hello", 12)
let constructeur4 = . // Homologue de ()
```

Un langage de programmation

Syntaxe

```
let z = {
    let a = 4
    let b = 5
    a + b
}

let zero = 0
let un = 1

let rec fib = fun(n: Int) -> Int {
    match n {
        0 -> zero,
        1 -> un,
        n -> fib(n + -1) + fib(n + -2)
    }
}

let u = fib(7)
```

Un langage de programmation

Typage

- Types des littéraux :
  - `42` a pour type `Int`
  - `"crocodile"` a pour type `String`
- Types de constructeur :
  - `.Abc(42, "bonjour")` a pour type `.Abc(Int, String)`
- Types somme :
  - `.Foo(61)` a pour type `.Foo(Int) | .Bar(String)`
- Types de fonction :
  - ```
    let double = fun(n: Int) -> Int {
        n + n
    }
    ```

    `double` a pour type `Fun(Int) -> Int`
    ```

Un langage de programmation

Métaprogrammation

- Structuré en 2 niveaux : niveau 0 et niveau 1
- On introduit le type des types : `Type`.

```
let x = 2+2                 // niveau 0
const Entier: Type = Int    // niveau 1

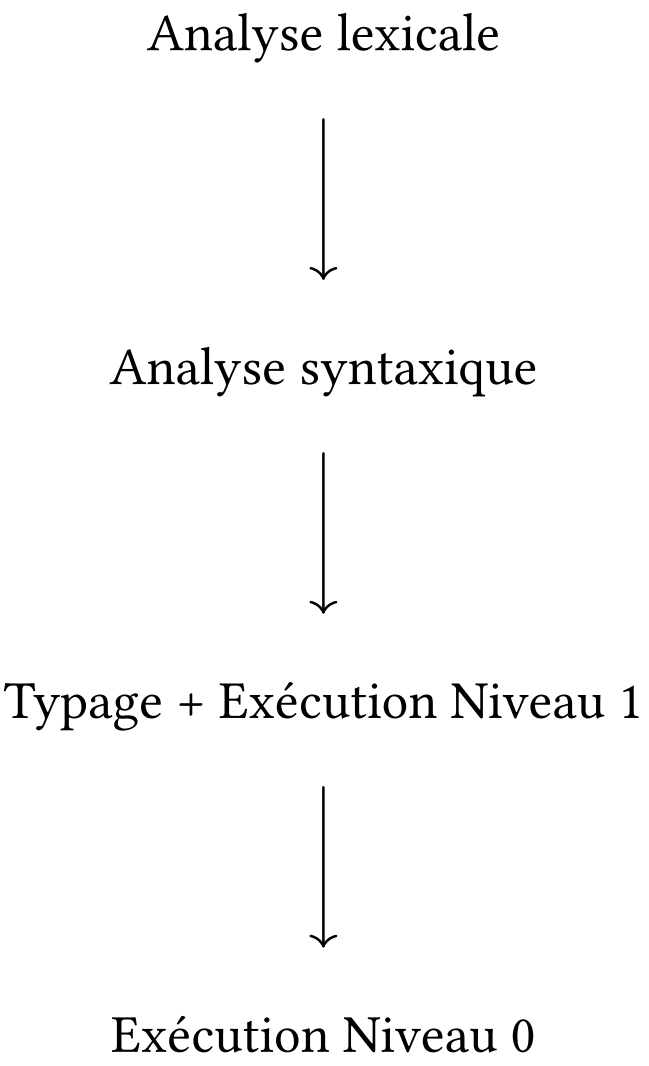
const UnType: Type = {      // niveau 1
    let T = .A(Int)
    let U = .B(String)

    T | U // .A(Int) | .B(String)
}
       // 1      // 0
let y: UnType = .A(16)

const Point = fun(T: Type) -> Type { // niveau 1
    .UnPoint(T, T)
}
                // 1              // 0
let origin:    Point(Int)    = .UnPoint(0, 0)
let position:  Point(String) = .UnPoint("48.0448998N", "1.7460588W")
```

Réalisation de l'interpréteur

- Langage interprété
- $\approx$ 3200 lignes de Rust

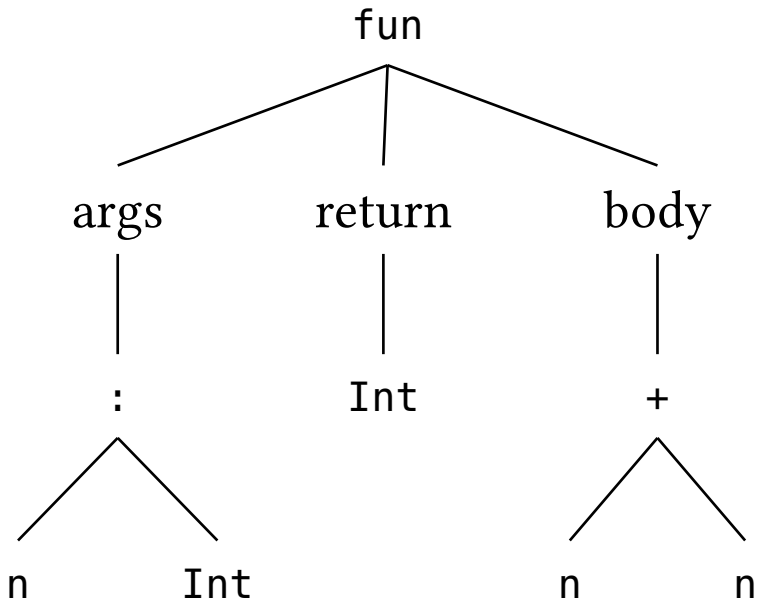


Réalisation de l'interpréteur

Analyse lexicale/syntaxique

```
fun(n: Int) -> Int {
    n + n
}
```

Code



Arbre de syntaxe

Réalisation de l'interpréteur

Typage + Niveau 1

- Niveau 1
  - Exécutée lors du typage du programme.
  - Pure, aucun effet de bord.
  - Peut manipuler des types comme des valeurs.
  - Pas de typage statique.

```
const s = "une string"

const T = Int // Le type des entiers.

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

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

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

Réalisation de l'interpréteur

Typage + Niveau 1

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

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

let x = id_int(17)
```

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

Réalisation de l'interpréteur

Exécution - Niveau 0

- Niveau 0
  - Constitue le programme final.
  - Autorise les effets de bords, mais ne peut pas manipuler les types.
  - Interprété mais serait réalisable comme un langage compilé.

```
let rec countdown = fun (n: Int) {
    match n {
        0 -> . ,
        n -> {
            print(Int)(n)
            countdown(n + -1)
        }
    }
}

countdown(10)
```

Réalisation de l'interpréteur

Évaluation du niveau 1

- Parcours de l'arbre de syntaxe
  - Évaluation des blocs `const { ... }`
  - Typage des expressions résultantes

Avant évaluation du niveau 1

```
const rec fib = fun(n: Int) -> Int {
    match n {
        0 -> 0,
        1 -> 1,
        n ->
          fib(n + -1) + fib(n + -2)
    }
}

let fib5 = const { fib(9) }
```

Après évaluation du niveau 1

```
let fib5 = 34
```

Réalisation de l'interpréteur

Évaluation du niveau 1

Avant

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

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

let id_int = const { id(Int) }

let thing: an_int() = id_int(45)
```

Après

```
let id_int = fun (x: Int) -> Int {
    x
}

let thing: Int = id_int(45)
```

Réalisation de l'interpréteur

Interaction entre les niveaux

Ne compile pas :

```
const n = 17

let m = n              // Variable inconnue: n
let m = const { n }    // Ok

const MonType = .Abc(Int, String)

let impossible1 = MonType
let impossible2 = const { MonType }

const x = 85
let   x = "une chaîne"
```

Résultats

```
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].
```

Résultats

Extensions

- Système de macros
  - Blocs `quote { ... }`
  - Partiellement implémenté
- Modules
  - Blocs `module { ... }`
- Vérifier les types du niveau 1
  - Inférence d'arguments
  - Types dépendants

Annexe

Système de macros :

```
const {
    let msg = "Hello, World!"

    quote {
        print(String)($msg)
    }
}


print(String)("Hello, World!")
```

Annexe

Système de macros

```
const assert_is = fun (T: Type) {
    fun (x: T) {}
}

const create_variable = fun(name: String, value: .) {
    let name = name

    quote {
        let $name = $value

        assert_is(Int)($value)
    }
}

create_variable("abc", 42)

assert_is(Int)(abc)
assert_is(String)(just_created) // Pas d'hygiène

.Result(abc, just_created)
```

Annexe

Modules

- Note : ce code ne compile pas.

```
const StringModule = module {
    const t = String
    let eq = ...
}

// Foncteur
const SetModule = fun (Elt: Module) -> Module {
    module {
        const t = some_set_type(Elt.t)
        let insert = fun (set: t, element: Elt.t) -> t { ... }
    }
}
```

Annexe

Inférence d'arguments et types dépendants

```
const id = fun(T: Type) -> (Fun(T) -> T)  {
    fun(x: T) -> T {
        x
    }
}


let y = id(56)
// ↓
let y = id(Int)(56)
```

Annexe

Mutabilité

- Note : ce code ne compile pas.

```
const T = .A(Int) | .B(String)

let f = fun(x: T) {

    x = .B("??")
}

let y: .A(Int) = .A(16)

f(y)



let g = fun(x: .A(Int)) { ... }

let z: T = .B("abc")

g(z)
```

Annexe

Portée des niveaux

```
const assert_is = fun (T: Type) {
    fun (x: T) {}
}

const x = "aaa"
let x = 42

assert_is(Int)(x)
const { assert_is(String)(x) }

let x = const { x }
assert_is(String)(x)
```

Annexe

Code

Index :

- `strati/src/main.rs`
- `strati/src/ast.rs`
- `strati/src/interpreter.rs`
- `strati/src/stage1.rs`
- `strati/src/parser.rs`
- `strati/src/lexer.rs`
- `strati/src/tests.rs`
- `strati/syntax.ebnf`