

用单子表示计算

1 单子的定义

每一个不同的单子都用于描述一类计算：

- 假设 M 是一个单子，
- $f \in M(A)$ 表示 f 是一个计算结果（或返回值）的类型为 A 的计算，
- M 单子应配有 bind 算子表示计算的复合，即 bind: $\forall AB. M(A) \rightarrow (A \rightarrow M(B)) \rightarrow M(B)$ ；
- M 单子应配有 return 算子表示直接返回一个特定值，即 return: $\forall A. A \rightarrow M(A)$ 。

1.1 状态单子

- 用于表示带存储状态的计算
- 定义: $M(A) \triangleq \Sigma \rightarrow \Sigma \times A$;
- 含义: 如果 $f \in M(A)$, $f(s_1) = (s_2, a)$ 表示从 s_1 状态开始执行计算 f , 那么计算结果是 a , 并且存储状态会被修改为 s_2 ;
- 考虑 $\Sigma = \mathbb{Z}$ 的情况
- Read : $M(\mathbb{Z})$ 表示从存储空间中读出一个整数

$$\text{Read}(s) = (s, s)$$

- WriteAndPlus1 : $\mathbb{Z} \rightarrow M(\mathbb{Z})$ 表示向存储空间中写入一个整数并且将这个整数加一后返回

$$\text{WriteAndPlus1}(n)(s) = (n, n + 1)$$

- ReadAndPlus : $\mathbb{Z} \rightarrow M(\mathbb{Z})$ 表示将存储空间中的值加上某个整数后的结果返回

$$\text{ReadAndPlus}(n)(s) = (s, n + s)$$

- return 算子: 如果 $a \in A$, 那么 $\text{return}(a)(s) = (s, a)$;
- bind 算子: 如果 $f \in M(A)$ 并且 $g : A \rightarrow M(B)$, 那么 $\text{bind}(f, g) \in M(B)$, 而且

$$\text{bind}(f, g)(s_1) = (s_3, b)$$

其中 $f(s_1) = (s_2, a)$, $g(a)(s_2) = (s_3, b)$ 。

1.2 集合单子

- 用于表示非确定性的计算
- 定义: $M(A) \triangleq \mathcal{P}(A)$;
- 含义: 如果 $f \in M(A)$, $a \in f$ 表示 f 的一个可能计算结果是 a ;
- $\text{Any} : \mathcal{P}(A) \rightarrow M(A)$ 表示从集合 A 的某个子集中任意选一个元素返回

$$a \in \text{Any}(X) \text{ if } a \in X$$

- return 算子: $\text{return}(a) = \{a\}$;
- bind 算子: 如果 $f \in M(A)$ 并且 $g : A \rightarrow M(B)$, 那么 $\text{bind}(f, g) \in M(B)$, 而且

$$\text{bind}(f, g) = \{b \mid \exists a. a \in f \text{ and } b \in g(a)\}$$

。

1.3 记录时间花费的单子

- 除了计算结果外, 额外记录计算花费的运算次数
- 定义: $M(A) \triangleq \mathbb{N} \times A$;
- 含义: 如果 $f \in M(A)$, $f = (t, a)$ 表示 f 这一计算需要的时间是 t 并集计算结果是 a ;
- return 算子: $\text{return}(a) = (0, a)$;
- bind 算子: 如果 $f \in M(A)$ 并且 $g : A \rightarrow M(B)$, 那么 $\text{bind}(f, g) \in M(B)$, 而且

$$\text{bind}(f, g) = (t_1 + t_2, b)$$

其中 $f = (t_1, a)$, $g(a) = (t_2, b)$ 。

1.4 描述计算异常退出的单子

- 除了成功执行完成计算外, 也可能计算出错异常终止;
- 定义: $M(A) \triangleq A \cup \{\epsilon\}$;
- 含义: 如果 $f \in M(A)$, 那么 $f \in A$ 表示能够成功完成计算, $f = \epsilon$ 表示计算出错异常终止;
- return 算子: $\text{return}(a) = a$;
- bind 算子: 如果 $f \in M(A)$ 并且 $g : A \rightarrow M(B)$, 那么 $\text{bind}(f, g) \in M(B)$
 - 当 $f = a \in A$ 并且 $g(a) = b \in B$ 时, $\text{bind}(f, g) = b$
 - 当 $f = a \in A$ 并且 $g(a) = \epsilon$ 时, $\text{bind}(f, g) = \epsilon$
 - 当 $f = \epsilon$ 时, $\text{bind}(f, g) = \epsilon$

2 在 Coq 中定义单子结构

单子可以这样在 Coq 中定义为一类代数结构。

```
Class Monad (M: Type -> Type): Type := {
  bind: forall {A B: Type}, M A -> (A -> M B) -> M B;
  ret: forall {A: Type}, A -> M A;
}.
```

我们之后最常常用到的将是集合单子 (set monad) 如下定义。

```
Module SetMonad.

Definition M (A: Type): Type := A -> Prop.

Definition bind: forall (A B: Type) (f: M A) (g: A -> M B), M B :=
  fun (A B: Type) (f: M A) (g: A -> M B) =>
    fun b: B => exists a: A, a ∈ f /\ b ∈ g a.

Definition ret: forall (A: Type) (a: A), M A :=
  fun (A: Type) (a: A) => Sets.singleton a.

End SetMonad.
```

```
#[export] Instance set_monad: Monad SetMonad.M := {
  bind := SetMonad.bind;
  ret := SetMonad.ret;
}.
```

下面是另一个例子状态单子的定义：

```
Module StateMonad.

Definition M (Σ A: Type): Type := Σ -> Σ * A.

Definition bind (Σ: Type):
  forall (A B: Type) (f: M Σ A) (g: A -> M Σ B), M Σ B :=
    fun A B f g s1 =>
      match f s1 with
      | (s2, a) => g a s2
      end.

Definition ret (Σ: Type):
  forall (A: Type) (a: A), M Σ A :=
    fun A a s => (s, a).

End StateMonad.
```

```
#[export] Instance state_monad (Σ: Type): Monad (StateMonad.M Σ) := {
  bind := StateMonad.bind Σ;
  ret := StateMonad.ret Σ;
}.
```

3 bind 算子的记号

以下是一些集合单子的例子

- 任取一个整数:

```
Definition any_Z: SetMonad.M Z := Sets.full.
```

- 整数乘二:

```
Definition multi_two: Z -> SetMonad.M Z :=
  fun x => ret (x * 2).
```

- 整数加一:

```
Definition plus_one: Z -> SetMonad.M Z :=
  fun x => ret (x + 1).
```

- 任取整数再乘二:

```
Definition bind_ex0: SetMonad.M Z :=
  bind any_Z multi_two.
```

- 任取整数乘二再加一:

```
Definition bind_ex1: SetMonad.M Z :=
  bind (bind any_Z multi_two) plus_one.
```

```
Definition bind_ex2: SetMonad.M Z :=
  bind any_Z (fun x => bind (multi_two x) plus_one).
```

下面是用单子描述计算时常用的记号:

```
Notation "x <- c1 ;; c2" := (bind c1 (fun x => c2))
  (at level 61, c1 at next level, right associativity) : monad_scope.
```

用这些 Notation 可以重写前面的一些例子。

- 任取整数再乘二:

```
Definition bind_ex0': SetMonad.M Z :=
  x <- any_Z;; ret (x * 2).
```

- 任取整数乘二再加一:

```
Definition bind_ex1': SetMonad.M Z :=
  x <- any_Z;; y <- multi_two x;; ret (y + 1).
```

注意，在上述定义中，

```
x <- any_Z;; y <- multi_two x;; ret (y + 1)
```

等价于说

```
bind any_Z
  (fun x => bind (multi_two x) (fun y => ret (y + 1)))
```

习题 1. 请写出下面标记背后表达的单子定义。

```
Definition bind_ex2': SetMonad.M Z :=
x <- any_Z;;
y <- any_Z;;
z1 <- multi_two y;;
z2 <- plus_one y;;
ret (x + y + z1 + z2).
```

4 集合单子

4.1 集合单子的简单例子

```
Definition choice {A: Type} (f g: SetMonad.M A):
SetMonad.M A :=
f ∪ g.
```

```
Definition assume (P: Prop): SetMonad.M unit :=
fun _ => P.
```

```
Definition compute_abs (z: Z): SetMonad.M Z :=
choice
(assume (z >= 0);; ret z)
(assume (z <= 0);; ret (-z)).
```

4.2 用集合单子定义表达式指称语义

在整数类型表达式中添加 ANY :

- ANY + 1 表示任取一个整数再加一；
- ANY * 2 表示任取一个整数再乘二。

```
Inductive expr_int : Type :=
| EConst (n: Z): expr_int
| EVar (x: var_name): expr_int
| EAdd (e1 e2: expr_int): expr_int
| ESub (e1 e2: expr_int): expr_int
| EMul (e1 e2: expr_int): expr_int
| EAny: expr_int.
```

下面定义语义算子：

```
Definition any_sem: state -> SetMonad.M Z :=
  fun s => any_Z.
```

```
Definition const_sem (n: Z): state -> SetMonad.M Z :=
  fun s => ret n.
```

```
Definition var_sem (X: var_name): state -> SetMonad.M Z :=
  fun s => ret (s X).
```

```
Definition add_sem (D1 D2: state -> SetMonad.M Z):
  state -> SetMonad.M Z :=
  fun s =>
    x <- D1 s;; y <- D2 s;; ret (x + y).
```

```
Definition sub_sem (D1 D2: state -> SetMonad.M Z):
  state -> SetMonad.M Z :=
  fun s =>
    x <- D1 s;; y <- D2 s;; ret (x - y).
```

```
Definition mul_sem (D1 D2: state -> SetMonad.M Z):
  state -> SetMonad.M Z :=
  fun s =>
    x <- D1 s;; y <- D2 s;; ret (x * y).
```

```
Fixpoint eval_expr_int (e: expr_int): state -> SetMonad.M Z :=
  match e with
  | EConst n =>
    const_sem n
  | EVar X =>
    var_sem X
  | EAdd e1 e2 =>
    add_sem (eval_expr_int e1) (eval_expr_int e2)
  | ESub e1 e2 =>
    sub_sem (eval_expr_int e1) (eval_expr_int e2)
  | EMul e1 e2 =>
    mul_sem (eval_expr_int e1) (eval_expr_int e2)
  | EAny =>
    any_sem
  end.
```