Normalization by Evaluation 入门

本文是对 *Normalization by Evaluation* (NBE) 的入门介绍：

- 本文会介绍 NBE 的各种不同变种
- 会用实现和 formalization 两种风格呈现定义
- 但是不会涉及正确性证明相关的内容

具体的内容列表如下：

- 1：对 normalization 和 NBE 本身的介绍
- 2：以无类型的、使用闭包的 NBE 算法为例，比较详细地解释一个 NBE 算法
- 3：非常简略地介绍使用 HOAS 的 NBE 算法及其数学背景
- 4：介绍 NBE 算法中处理不重名新变量生成的，更严谨的方法
- 5：引入类型，约定后续内容使用的类型系统
- 6：对“支持 η-equivalence 的 normalization”这一问题进行说明
- 7：一个带类型的、支持 η 的 NBE 算法
- 8：另一个带类型的、支持 η 的 NBE 算法
- 9：介绍 NBE 算法的一种共通结构：Tait's Yoga
- 10：从 Tait's Yoga 中导出第三个带类型的、支持 η 的 NBE 算法
- 11：将带类型的 NBE 算法从简单类型系统拓展到 dependent type
- 12：一些拓展阅读材料

本文中代码的部分将使用 OCaml 语法。本文会以最朴素的 λ calculus 为主要例子。接下来的内容会用到如下两个定义：

formalization	OCaml 代码

$\texttt{Type} \ni A, B ::= c \mid A \to B$	`type typ =`
	`  | Base of base_typ`
	`  | Fun  of typ * typ`

$\texttt{Term} \ni t, u ::= x \mid \lambda x.t \mid t\ u$	`type term =`
	`  | Var of string`
	`  | Lam of string * term`
	`  | App of term * term`

1 什么是 NBE

大部分时候，我们只关心一门语言在 empty context，也就是没有上下文时的语义。因为对于编程语言来说，如果运行时有未定义、没有值的自由变量，运行就会出错。但是，有时，我们也需要在有自由变量的情况下对表达式进行求值、化简。这时，当我们看见一个函数 $\lambda x.t$ 时，也需要对 λ 下的 t 进行化简。这种允许自由变量的、会穿越 λ 的求值/化简，就是 *normalization*。Normalization 比较常见的应用有：

- Normalization 可以用来做 partial evaluation，对一个表达式进行优化（事实上，做 partial evaluation 的人就重新发明过 NBE，并将其称为 *Type Directed Partial Evaluation* (TDPE)）
- 支持 dependent type 的语言的类型检查往往需要 normalization。由于类型可以依赖于值，类型中存在化简的可能。例如，我们希望 `Vec 2` 和 `Vec (1 + 1)` 是同一个类型。这时，为了检查两个类型是否相等，就需要先 normalize 它们

为了实现 normalization，最暴力的方法是 *Capture Avoiding Substitution*。也就是不断地寻找形如 $(\lambda x.t)u$ 的 redex，并把它替换为 $t[x \mapsto u]$，直到得到的表达式不能再化简为止。然而，这种做法非常慢，我们希望有一种更高效的办法。此外，反复根据一些规则 rewrite 在理论上处理起来也不是很方便。

在没有自由变量时候，我们不需要使用 substitution、反复执行单步的 reduction，就能写出高效的解释器：

```
type value =
    | VLam of env * string * term

and env = (string * value) list

let rec eval_closed env term =
    match term with
    | Var var        -> List.assoc var env
```

```
    | Lam(var, body) -> VLam(env, var, body)
    | App(f, a)      -> apply_value (eval_closed env f)
                                    (eval_closed env a)

and apply_value vf va =
    match vf with
    | VLam(env, var, body) ->
        eval_closed ((var, va) :: env) body
```

那么，我们能否用同样的思路来写出一个高效的 normalization 算法呢？沿着这种思路达到的，就是 *Normalization by Evaluation* (NBE)。在 NBE 中：

- 我们有一个表示值的类型 `value`。**但 `value` 要能处理自由变量的情况**
- 我们有一个求值函数 `eval : env -> term -> value`。同样地，`eval` 要能处理自由变量
- 我们还需要一个**读回函数**，它能把一个 `value` 转换回一个表达式 `term`，而且转换得到的 `term` 一定是一个 normal form（不能继续化简）

接下来我将介绍 NBE 的各种变种。

2 无类型 + closure

让我们先从最简单的例子开始：不考虑类型，和上面的 `eval_closed` 一样使用 closure 来表示函数值。

2.1 如何定义 value

我们面临的第一个问题是：如何设计带自由变量的 `value`？在没有自由变量的情况下，λ calculus 中的值只有一种：形如 $\lambda x.t$ 的函数。然而，如果有自由变量，那么自由变量们，例如 x、y，同样不能继续化简。所以，它们也应当被看作值。除此之外，如果我们给自由变量应用一些参数，例如 $x\ (\lambda y.t)$，那么得到的表达式同样无法化简、应当被看作值。

但是，并不是对于所有值 v, w，$v\ w$ 都是一个值：如果 v 形如 $\lambda x.t$，那么 $v\ w$ 是可以继续化简的。只有当 v 是一个“不正常”的、被一些自由变量“卡住”的值的时候，$v\ w$ 才是一个值。所以，我们发现，在 NBE 中有两种值。一种是普通的、无自由变量时也存在的值，一种是被自由变量卡住的特殊的值。我们把后者成为中性（neutral）的值。现在，NBE 中的 `value` 可以如此定义：

formalization	OCaml 代码

$\texttt{Value} \ni v, w ::= (\rho, \lambda x.t) \mid n$	`type value =`
	`  | VLam  of env * string * term`
	`  | VNeutral of neutral`

$\texttt{Neutral} \ni m, n ::= x \mid n\ v$	`and neutral =`
	`  | NVar of string`
	`  | NApp of neutral * value`

$\rho \in \texttt{Env} = \texttt{Var} \rightharpoonup \texttt{Value}$	`and env = (string * value) list`

在 formalization 中简单起见，把求值环境定义成一个从变量到值的 partial function。我们用 $\rho, x \mapsto v$ 表示把 ρ 中 x 指向的东西换成 v 得到的新求值环境。

2.2 如何求值

定义出 `value` 之后，就应该考虑如何实现求值函数 `eval : env -> term -> value` 了。当被求值的表达式 t 中有自由变量 x 时，由于我们已经把所有自由变量 x 加入到值的集合当中了，我们可以在求值环境中把 x 映射到自己。所以，我们可以像 `eval_closed` 中那样，假设被求值的表达式中的所有自由变量都在求值环境中有定义。这么一来，求值函数的主体相比 `eval_closed` 就完全不需要更改了：

```
let rec eval env term =
    match term with
    | Var var        -> List.assoc var env
    | Lam(var, body) -> VLam(env, var, body)
    | App(f, a)      -> apply_value (eval env f) (eval env a)
```

由于值的集合发生了变化，我们需要重新定义 `apply_value` 函数。不过，它的定义非常直观：

```
and apply_value vf va =
    match vf with
    | VLam(env, var, body) -> eval ((var, va) :: env) body
    | VNeutral neutral     -> VNeutral(NApp(neutral, va))
```

如果 `vf` 真的是一个函数，那么就正常地进行求值。如果 `vf` 是被自由变量卡住的表达式，那么就把新的参数叠上去。上述两个函数在 formalization 中定义如

下：

$$
\begin{aligned}
[\![x]\!]_\rho &= \rho(x) \\
[\![\lambda x.t]\!]_\rho &= (\rho, \lambda x.t) \\
[\![t\ u]\!]_\rho &= [\![t]\!]_\rho \cdot [\![u]\!]_\rho
\end{aligned}
$$

$$
\begin{aligned}
(\rho, \lambda x.t) \cdot v &= [\![t]\!]_{\rho, x \mapsto v} \\
n \cdot v &= n\ v
\end{aligned}
$$

2.3 如何把值转换回表达式

现在，我们已经得到了一个带自由变量的值的定义，以及一个能够处理自由变量的求值函数。为了得到一个完整的 normalization 算法，我们欠缺的最后一环是：把一个值转换回一个（不能再化简的）表达式。

中性值的转换非常简单：把自由变量转换成它自己，在把它的参数分别转成表达式即可。转换出的表达式会被这个自由变量卡住，因而是不能化简的 normal form。问题在于，闭包 $(\rho, \lambda x.t)$ 应当如何转换呢?

由于我们在进行 normalization，所以不能简单地直接返回 $\lambda x.t$：必须把 t 也 normalize 了，不能让 t 中有可以化简的部分。但是要触发 t 的求值，就必须提供一个参数，而我们不能遍历所有可能的参数！幸运的是，**值的定义中有自由变量**。所以我们可以将一个新的自由变量 y 作为参数提供给 $\lambda x.t$！这么一来，就可以触发对 t 的求值，从而对它进行 normalize。而且，因为我们提供的参数是一个新的自由变量 y，它可以“代表”所有其他参数，我们只需要在外面裹上一层 λy 就能得到 $(\rho, \lambda x.t)$ 对应的表达式：

```
let rec readback value =
    match value with
    | VLam _ ->
        let x = fresh_var () in
        let x_value = VNeutral(NVar x) in
        Lam(x, readback (apply_value value x_value))
    | VNeutral ne ->
        readback_neutral ne

and readback_neutral ne =
    match ne with
    | NVar x     -> Var x
    | NApp(f, a) -> App(readback_neutral f, readback a)
```

$$
\begin{aligned}
\downarrow (\rho, \lambda x.t) &= \lambda y.[\![t]\!]_{\rho, x \mapsto y} \qquad y \text{ fresh} \\
\downarrow x &= x \\
\downarrow (n\ v) &= (\downarrow n)\ (\downarrow v)
\end{aligned}
$$

这里关于如何生成不重名的新变量（`fresh_var`）的细节被省去了。后面将会介绍比起生成全局新变量更好的处理方式。实现了读回函数之后，把它和求值函数组合起来，就能得到一个 normalization 算法了：

```
let rec normalize free_variables term =
    let env = List.map (fun x -> x, VNeutral(NVar x)) free_variables in
    readback (eval env term)
```

$$\uparrow \vec{x} = y \mapsto \begin{cases} y & y \in \vec{x} \\ \bot & \text{otherwise} \end{cases}$$

$$\texttt{norm}^{\vec{x}} t = \downarrow\ (\llbracket t \rrbracket_{\uparrow \vec{x}})$$

下面通过一个例子：$\lambda x.(\lambda y.\lambda x.y)\ x$ 来演示我们得到的 normalization 算法：

$$\begin{aligned}
& \underline{\texttt{norm}^{\varnothing}(\lambda x.(\lambda y.\lambda x.y)\ x)} \\
=\ & \downarrow\ \underline{\llbracket \lambda x.(\lambda y.\lambda x.y)\ x \rrbracket_{\varnothing}} \\
=\ & \underline{\downarrow (\varnothing, \lambda x.(\lambda y.\lambda x.y)\ x)} \\
=\ & \lambda z_1.\downarrow\ \underline{\llbracket (\lambda y.\lambda x.y)\ x \rrbracket_{x \mapsto z_1}} \\
=\ & \lambda z_1.\downarrow \left(\underline{\llbracket \lambda y.\lambda x.y \rrbracket_{x \mapsto z_1}} \cdot \underline{\llbracket x \rrbracket_{x \mapsto z_1}} \right) \\
=\ & \lambda z_1.\downarrow \left(\underline{(x \mapsto z_1, \lambda y.\lambda x.y) \cdot z_1} \right) \\
=\ & \lambda z_1.\underline{\downarrow \llbracket \lambda x.y \rrbracket_{x \mapsto z_1, y \mapsto z_1}} \\
=\ & \lambda z_1.\lambda z_2.\downarrow\ \underline{\llbracket y \rrbracket_{y \mapsto z_1, x \mapsto z_1}} \\
=\ & \lambda z_1.\lambda z_2.\underline{\downarrow z_1} \\
=\ & \lambda z_1.\lambda z_2.z_1
\end{aligned}$$

3 HOAS 与 domain

上面的 NBE 实现中，代表函数的值是用 `env * string * term` 的闭包的形式表达的。在实现解释器时，另一种常见的选择是，直接用一个元语言中 `value -> value` 的函数来表达函数，这种做法叫 Higher Order Abstract Syntax（HOAS）：

```
type value =
    | VLam     of (value -> value)
    | VNeutral of neutral

...

let apply_value vf va =
    match vf with
```

```
    | VLam f     -> f va
    | VNeutral n -> VNeutral(NApp(n, va))

...
```

在代码实现中，使用 HOAS 的版本和之前使用 closure 的版本几乎没有区别。但是，在 formalization 上就不一样了。由于这里的 `value` 的定义涉及到了 `value -> value` 的函数，所以我们不能简单地把 `value` 用 AST 的形式定义了。假设使用 HOAS 的值是 V，那么 V 必须满足：

$$V \simeq [V \to V] + N$$

其中 N 是中性值的集合，定义为：

$$N \simeq \texttt{Var} + N \times V$$

（这里 V 和 $[V \to V]$ 应当解释成 *domain* 和 domain 间的 *continuous function*。解释成集合的话，这个式子会因为集合大小问题无解。不过，这不是本文的重点，所以下面不对这个问题和 domain theory 相关的内容进行讨论）。对于任意的 domain V，只要它满足上面的等式，那么我们就可以定义出三个函数：

$$\begin{aligned} \uparrow &: N \to V \\ [\![\_]\!]\_ &: \texttt{Term} \times (\texttt{Var} \rightharpoonup V) \to V \\ \downarrow &: V \to \texttt{Term} \end{aligned}$$

然后通过它们的复合得到一个 normalization 函数。相比 closure 版本，这种描述 NBE 的方式可以利用 domain theory 中现有的结论，但另一方面也必须拥抱 domain theory 本身的复杂性。

4 如何生成新鲜的变量

当我们将一个 closure 转回表达式时，需要通过生成一个新的、和任何现存变量都不重名的变量，作为生成的 λ 表达式的参数，以避免名字的冲突。实现上，只需要一个全局计数器就能做到这一点。但是，从 formalization 的角度来说，全局状态难以处理。所以，我们需要一种更简单的生成新鲜变量的方式。

4.1 记录已经用过的变量

第一种方式是，在读回函数中，额外提供一个参数。这个参数是目前已经使用过的变量的列表。当我们需要生成新的变量时，找一个不在这个列表中的变量即可：

```
let fresh_var (free_vars : string list) : string = ...

let rec readback free_vars value =
    match value with
    | VLam _ ->
```

```
        let x = fresh_var free_vars in
        let x_value = VNeutral(NVar x) in
        Lam(x, readback (apply_value value x_value))
    | VNeutral ne ->
        readback_neutral free_vars ne

and readback_neutral free_vars ne =
    ...

let normalize free_vars term =
    let env = List.map (fun x -> x, VNeutral(NVar x)) free_vars in
    readback free_vars (eval env term)
```

$$\downarrow^{\vec{x}} (\rho, \lambda y.t) = \lambda z.\downarrow^{\vec{x},z} [\![t]\!]_{\rho,y\mapsto z} \qquad z \notin \vec{x}$$
$$\downarrow^{\vec{x}} y = y$$
$$\downarrow^{\vec{x}} (n\ v) = (\downarrow^{\vec{x}} n)\ (\downarrow^{\vec{x}} v)$$

4.2 使用 de Bruijn index/level

另一种解决变量名冲突问题的方式，是直接抛弃“名字”本身，转而使用 de Bruijn index/level。也就是说，我们通过“这是第几个变量”的序号来表示变量本身。在 de Bruijn index 中，这个序号是从新的变量往旧的变量数的。在 de Bruijn level 中，这个序号是从旧的变量往新的变量数的。

那么，我们应该使用 de Bruijn index 还是 level 来实现 NBE 呢？最流行的办法是将两者进行组合：

- 在 `term` 中使用 de Bruijn index。因为在 de Bruijn index 中，closed term 可以不做任何修改嵌入到任意 context 中，这相对来说更方便
- 在 `value` 中使用 de Bruijn level。如此一来，求值中向求值环境 `env` 中添加新的变量时，就不需要修改现有的值了

当然，混用两者意味着需要在两者之间进行转换。幸运的是，这一转换很简单。假设一共有 n 个变量，某个变量的 de Bruijn index 是 i，de Bruijn level 是 l，那么 $i+l+1=n$。由此，只要知道 n，就可以在 index 和 level 之间互相转换。使用 de Bruijn index/level 的 NBE 实现如下：

```
type term =
    | Idx of int
    | Lam of term (* 不需要指定参数名字：由当前变量总数自动决定 *)
    | App of term * term

type value =
    | VLam     of env * term
    | VNeutral of neutral
```

```
and neutral =
    | NLvl of int
    | NApp of neutral * value

and env = value list

let rec eval env term =
    match term with
    | Idx idx   -> List.nth env idx
    | Lam body  -> VLam(env, body)
    | App(f, a) -> apply_value (eval env f) (eval env a)

and apply_value vf va =
    match vf with
    | VLam(env, body)  -> eval (va :: env) body
    | VNeutral neutral -> VNeutral(NApp(neutral, va))


let rec readback n_free_vars value =
    match value with
    | VLam _ ->
        let var = VNeutral(NLvl n_free_vars) in
        Lam(readback (n_free_vars + 1) (apply_value value var))
    | VNeutral ne ->
        readback_neutral n_free_vars ne

and readback_neutral n_free_vars ne =
    match ne with
    | NLvl lvl   -> Idx(n_free_vars - lvl - 1)
    | NApp(f, a) ->
        App(readback_neutral n_free_vars f, readback n_free_vars a)


let normalize n_free_vars term =
    let env = List.init n_free_vars
        (fun idx -> VNeutral(NLvl(n_free_vars - idx - 1)))
    in
    readback n_free_vars (eval env term)
```

Formalization 风格的描述如下：

$$
\begin{aligned}
[\![i]\!]_{\vec{v}} &= v_i \\
[\![\lambda.t]\!]_{\vec{v}} &= (\vec{v}, t) \\
[\![t\ u]\!]_{\vec{v}} &= [\![t]\!]_{\vec{v}} \cdot [\![u]\!]_{\vec{v}}
\end{aligned}
$$

$$
\begin{aligned}
(\vec{v}, t) \cdot v &= [\![t]\!]_{\vec{v},v} \\
n \cdot v &= n\ v
\end{aligned}
$$

$$
\begin{aligned}
\downarrow^N (\vec{v}, t) &= \lambda.\downarrow^{N+1} [\![t]\!]_{\vec{v},N} \\
\downarrow^N l &= n - l - 1 \\
\downarrow^N (n\ v) &= (\downarrow^N n)\ (\downarrow^N\ v)
\end{aligned}
$$

5 加入类型

接下来，我们讨论加入了类型信息的 NBE。使用的类型系统是 Simply Typed λ Calculus (STLC)，其类型规则如下：

$$\texttt{Context} \ni \Gamma, \Delta ::= \cdot \mid \Gamma, x : A$$

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B}$$

6 η-equivalence

上面给出的 NBE 算法都是无类型的。即使输入的表达式其实是有类型的，也会被当作无类型来处理。使得 NBE 算法的描述变得简单，但却无法处理一些依赖类型的等式，例如 η-equivalence：

$$\lambda x.t\ x = t \qquad (x \notin FV(t))$$

其中 $FV(t)$ 是 t 中的自由变量的集合。无类型的 NBE 之所以无法处理 η，是因为 η 的成立是由类型条件的。如果语言中除了函数还有其他类型，例如整数，那么 $\lambda x.1\ x$ 和 1 显然不应该相等：前者是类型错误的。

所以，为了处理 η-equivalence，就需要向 NBE 中引入类型。首先，在“什么才是 normal form”这个问题上，我们就需要借助类型：

$\Gamma \vdash t \Leftarrow A$ $\quad$ t 是 Γ 下类型为 A 的 **normal form**

$\Gamma \vdash t \Rightarrow A$ $\quad$ t 是 Γ 下类型为 A 的**中性 normal form**

$$\frac{\Gamma, x : A \vdash t \Leftarrow B}{\Gamma \vdash \lambda x.t \Leftarrow A \to B} \qquad \frac{\Gamma \vdash t \Rightarrow c}{\Gamma \vdash t \Leftarrow c}$$

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A} \qquad \frac{\Gamma \vdash t \Rightarrow A \to B \qquad \Gamma \vdash u \Leftarrow A}{\Gamma \vdash t\ u \Rightarrow B}$$

注意到，只有对 base type c，中性 normal form 才是 normal form。对于函数类型，normal form 必然形如 $\lambda x.t$。所以，上述类型规则定义出的 normal form 在加入了 η-equivalence 的意义上是唯一的。这种 normal form 被成为 η-long/η-expanded，因为在 η-equivalence 的两侧中，它选择了较长的（带 λ 的）一侧。反过来，如果选择较短的、不带 λ 的一侧，则称为 η-short/η-contracted。

7 支持 η 的 NBE：从外侧传入类型

接下来，我们尝试改动 NBE 算法，让它能够产生 η-expanded 的 normal form。为此，NBE 过程中需要用到类型信息。第一种做法，也是本节的做法，是从外部传入需要的类型信息。

首先，让我们考虑 η-expansion 应该发生在哪个阶段：求值时，还是 readback 时。如果我们在求值时过早地进行 η-expansion，例如我们对 $f\ x$ 中的变量 f 进行了 η-expansion，使它变成了 $\lambda y.f\ y$，那么就创建了一个新的 redex $(\lambda y.f\ y)\ x$，而且这个 redex 求值后又会变回原来的 $f\ x$。所以，在求值时进行 η-expansion 是过早的，应当在 readback 时进行 η-expansion。

为了在 readback 时进行 η-expansion，我们需要提供被 readback 的值的类型。除此之外，我们还需要记录变量类型的上下文 Γ 以获得变量的类型。所以，可以如此定义 readback：

$$\begin{aligned}
R_{\mathrm{nf}}(\Gamma \vdash A \to B, v) &= \lambda x.R_{\mathrm{nf}}(\Gamma, x : A \vdash B, v \cdot x) && x \notin \Gamma \\
R_{\mathrm{nf}}(\Gamma \vdash c, n) &= t && R_{\mathrm{ne}}(\Gamma, n) = t, A \\
R_{\mathrm{ne}}(\Gamma, x) &= x, A && (x : A) \in \Gamma \\
R_{\mathrm{ne}}(\Gamma, n\ v) &= (t\ R_{\mathrm{nf}}(\Gamma \vdash A, v)), B && R_{\mathrm{ne}}(\Gamma, n) = t, A \to B
\end{aligned}$$

这里定义了两个 readback 函数：

$$\begin{aligned}
R_{\mathrm{nf}} &: \texttt{Context} \times \texttt{Type} \times \texttt{Value} \to \texttt{Term} \\
R_{\mathrm{ne}} &: \texttt{Context} \times \texttt{Neutral} \to \texttt{Term} \times \texttt{Type}
\end{aligned}$$

可以看到，转换一个**值**需要提供一个类型，而转换一个**中性值**可以产生一个类型。这和上面 normal form 的类型规则中箭头的方向（$\Leftarrow$ 和 $\Rightarrow$）是一致的。注意到，在 R_{nf} 的定义中，如果提供的类型是 $A \to B$，那么**无论被转换的值 v 是什么**，R_{nf} 都会产生一个 λ。所以，$R_{\mathrm{nf}}(f : c_1 \to c_2 \vdash c_1 \to c_2, f)$ 会产生 $\lambda x.f\ x$：

这正是我们想要的 η expansion 的效果。代码实现如下（只有 `readback` 的代码需要改动）：

```
type context = (string * typ) list

let fresh (ctx : context) : string = ...

let rec readback (ctx : context) typ value =
    match typ, value with
    | Fun(param_typ, ret_typ), _ ->
        let var = fresh ctx in
        let value' = apply_value value (VNeutral(NVar var)) in
        Lam(var, readback ((var, param_typ) :: ctx) ret_typ value')
    | Base _, VNeutral ne ->
        fst (readback_neutral ctx ne)
    | _ ->
        failwith "type error"

and readback_neutral ctx ne =
    match ne with
    | NVar var ->
        Var var, List.assoc var ctx
    | NApp(f, a) ->
        match readback_neutral ctx f with
        | f', Fun(param_typ, ret_typ) ->
            App(f', readback ctx param_typ a), ret_typ
        | _ ->
            failwith "type error"
```

8 在值中直接写入类型

上一节中,我们通过在 readback 时从外侧传入类型的方式,实现了带 η-expansion 的 NBE。另外一种实现方式，则是把类型直接写入值中，并使得求值时的各种操作都尊重类型。

那么，应该在值和中性值的什么位置插入类型呢？我们希望，插入类型之后，在 readback 时就不需要额外传入类型信息了。此外，插入的类型信息在求值过程中要能正常维护。下面是一种满足上述需求的非常优雅的定义方式：

$$
\begin{aligned}
\texttt{Normal} \ni \downarrow^A v & \\
\texttt{Value} \ni v, w \quad & ::= \lambda x.t \mid \uparrow^A n \\
\texttt{Neutral} \ni n \quad & ::= x \mid n \downarrow^A v
\end{aligned}
$$

这个定义的意义是什么呢？如果把这个定义和上一节的 readback 函数比对，就会发现，所有出现 $\downarrow^A v$ 的地方都是 readback 需要类型的地方。所有 $\uparrow^A n$ 都是

readback 产生类型的地方。我们直接在这些位置加上类型，就不需要在 readback 时额外维护类型了：

$$
\begin{aligned}
R_{\mathrm{nf}} \downarrow^{A \to B} v &= \lambda x. R_{\mathrm{nf}}(v \cdot x) && x \text{ fresh} \\
R_{\mathrm{nf}} \downarrow^{c} \uparrow^{A} n &= R_{\mathrm{ne}} n \\
R_{\mathrm{ne}}\ x &= x \\
R_{\mathrm{ne}}(n\ \downarrow^{A} v) &= (R_{\mathrm{ne}} n)\ (R_{\mathrm{nf}} \downarrow^{A} v)
\end{aligned}
$$

此外，还需要在求值时维护类型信息。为此，我们需要修改求值时的 application 的定义：

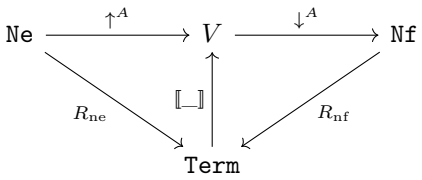
$$
\begin{aligned}
(\rho, \lambda x.t) \cdot v &= [\![t]\!]_{\rho, x \mapsto v} \\
\uparrow^{A \to B} n \cdot v &= \uparrow^{B} (n\ \downarrow^{A} v)
\end{aligned}
$$

代码实现见 A。

9 Tait's Yoga

上面的两种方式虽然细节不同，但整体构造和类型信息的流向都非常相似。所以，NBE 算法的背后有着某种共通的结构。这种结构在有些地方被称作 *Tait's Yoga*。Tait's Yoga 中：

- 我们会有一个中性 **normal form** 的集合 `Ne`，并且中性 normal form 可以通过一个函数 R_{ne} 转换回 `Term`
- 我们会有一个 **normal form** 的集合 `Nf` 并且 normal form 可以通过一个函数 R_{nf} 转换回 `Term`
- 在 `Ne` 和 `Nf` 之间，有一个值的集合 V，满足：



其中 $\uparrow^A$ 被称为 *reflection*，$\downarrow^A$ 被称为 *reification*。上面的两个算法都可以看成 Tait's Yoga 的一个实例：

- 从外部传入类型的算法（7）中：
  - `Ne` 是算法中的 `Neutral`
  - `Nf` 就是 `Term` 中的 normal form 这一子集

  - Tait's Yoga 图中的 R_{ne} 对应算法中的 R_{ne}（忽略它额外返回的类型）
  - Tait's Yoga 图中的 R_{nf} 对应从子集到父集的嵌入
  - Tait's Yoga 图中的 $\uparrow^A$，对应从 `Neutral` 到 `Value` 的嵌入
  - Tait's Yoga 图中的 $\downarrow^A$，对应算法中的 R_{nf}
- 在值内部标记类型的算法（8）中：
  - Tait's Yoga 中的 `Ne` 和 `Nf` 就是算法中的 `Neutral` 和 `Normal`
  - Tait's Yoga 中的 R_{nf} 和 R_{ne} 就是算法中的 R_{nf} 和 R_{ne}
  - Tait's Yoga 中的 $\uparrow^A$ 和 $\downarrow^A$ 就是算法中的构造器

Tait's Yoga 不仅能容纳之前看到的两个算法，通过将之前的算法放入这个框架中，我们还可以找到新的实现方式的灵感！可以看到，对于 R_{ne} (R_{nf}) 和 $\uparrow^A$ ($\downarrow^A$)，有两种设计选择：

- (A). 把 R_{ne} (R_{nf}) 做成从子集到父集的嵌入，在 $\uparrow^A$ ($\downarrow^A$) 中进行计算
- (B). 把 $\uparrow^A$ ($\downarrow^A$) 中做成 constructor，在 R_{ne} (R_{nf}) 进行计算

通过排列组合我们可以得到四个算法：

Ne \ Nf	A	B
A	?	?
B	7	8

可以看到，这里出现了两个空白！而且，有一种可能性我们之前完全没有考虑到：在 $\uparrow^A$ 中进行计算的可能性！

10 复用现有的解释器

通过 Tait's Yoga，我们意识到，也许从中性值到值的过程也可以发生一些计算。事实上，最早的 NBE 算法正是这么做的。而且，它们有一个非常现实的目的：复用一个现成的、不能进行 noramlization 的解释器，来进行 normalization！

这是怎么做到的呢？首先，不放考虑为什么普通解释器**不能**被拿来做 normalization。在 NBE 中，类型为 $A \to B$ 的值**不一定真的是一个函数**。它还有可能是一个中性值。但是，我们能不能把它变成一个函数呢？假如我们有 $f : A \to B$，那么只需要 η 展开 f，就能得到 $\lambda x.f\ x$：一个函数！

所以，如果我们在求值**之前**，先根据类型进行这样的 η 展开，就能把中性值从函数中彻底踢出去（当然，这些中性值不会消失，但它们只会在 base type 上出现）。这步求值前的 η 展开，正是我们从 Tait's Yoga 中看到的、带计算的

reflection ($\uparrow^A$):

$$\begin{aligned}
\uparrow^{A\to B} n &= v \mapsto \uparrow^B (n\ \downarrow^A v) \\
\uparrow^c n &= n
\end{aligned}$$

$$\begin{aligned}
\downarrow^{A\to B} v &= \lambda x.\downarrow^B (v \cdot \uparrow^A x) && x \text{ fresh} \\
\downarrow^c n &= n
\end{aligned}$$

这里，由于我们在复用 evaluator，所以我们不再使用 closure 自己实现一切，而是使用了形如 $v \mapsto \ldots$ 的、宿主语言的函数。这种做法的更多内容可以参考 3。这里的中性值 $n \in \texttt{Ne}$ 全部都是 `Term`，所以在 $\downarrow^c n = n$ 中无需对 n 再做更多处理。

11 Dependent type

（不考虑正确性证明的话）将上面的带类型的 NBE 算法拓展到 dependent type 并不困难。假设普通函数类型被替换为了 dependent 的函数类型 $\Pi_{x:A}B$。假设 $f : \Pi_{x:A}B$，那么，$f\ a$ 的类型应当是 $B[x \mapsto a]$。然而，我们不想要直接计算 substitution，因为这样需要考虑变量重名问题，而且非常低效。我们希望把类型上的计算也用 NBE 来实现。

为此，我们需要给类型也设计一套值。这里，我们采用语法上分开类型和表达式的做法，类型对值的依赖通过 base type $c(\vec{t})$ 来表达，并使用 8 中的实现风格：

$$\begin{aligned}
\texttt{Type} \ni A, B &::= c(\vec{t}) \mid \Pi_{x:A}B \\
\texttt{Term} \ni t, u, A, B &::= x \mid \lambda x.t \mid t\ u \\
\texttt{TValue} \ni V, W &::= c\left(\overrightarrow{\downarrow^V v}\right) \mid \texttt{Pi}(\rho, V, x, B) \\
\texttt{Nf} \ni \downarrow^V v & \\
\texttt{Value} \ni v, w &::= \uparrow^V n \mid \texttt{Lam}(\rho, x, t) \\
\texttt{Ne} \ni n &::= x \mid n\ \downarrow^V v
\end{aligned}$$

这里的 $\overrightarrow{\downarrow^V v}$ 是一个 `Nf` 的列表。对应的 readback 的定义如下（R_{nf} 的记号被同时用于转换 `TValue` 和 `Nf`）：

$$
\begin{aligned}
R_{\text{nf}}\ c\left(\overrightarrow{\downarrow^V v}\right) &= c(\overrightarrow{R_{\text{nf}} \downarrow^V v}) \\
R_{\text{nf}}\ \texttt{Pi}(\rho, V, x, B) &= \prod_{y:R_{\text{nf}}V} R_{\text{nf}}\ [\![B]\!]_{\rho, x\mapsto\uparrow^V y} && y \text{ fresh} \\
R_{\text{nf}}\ \downarrow^{\texttt{Pi}(\rho,V,x,B)}\ v &= \lambda y. R_{\text{nf}} \downarrow^{[\![B]\!]_{\rho,x\mapsto y}} (v \cdot \uparrow^V y) && y \text{ fresh} \\
R_{\text{nf}}\ \downarrow^{c\left(\overrightarrow{\downarrow^V v}\right)}\uparrow^W n &= R_{\text{ne}}\ n
\end{aligned}
$$

$$
\begin{aligned}
R_{\text{ne}}\ x &= x \\
R_{\text{ne}}(n\ \downarrow^V v) &= (R_{\text{ne}}\ n)\ (R_{\text{nf}} \downarrow^V v)
\end{aligned}
$$

代码实现留作练习。

12 一些拓展阅读材料

- https://www.cse.chalmers.se/~abela/habil.pdf：一份关于 NBE（包括 dependent type/system F 版本）的很详尽的 thesis。对正确性证明的方法也有很完整的介绍
- https://github.com/AndrasKovacs/elaboration-zoo:一系列使用 NBE 的 dependent type 类型检查器的实现
- https://okmij.org/ftp/tagless-final/NBE.html：一份使用了非 λ calculus 的简单 rewrite rule 为例子的很特别的 NBE 教程。和实现编程语言的 NBE 关系不大，但对于理解 NBE 中“evaluation”的含义很有帮助
- http://hjemmesider.diku.dk/~andrzej/papers/NaPE.pdf：一份关于 NBE 在 partial evaluation 中的应用的综述

A 值中标记类型的 NBE 实现

（简单起见，新变量的生成使用了全局的方式。正文中提到的另外两种方式同样可以使用）

```
type nf = typ * value

and value =
    | VLam     of env * string * term
    | VNeutral of typ * neutral

and neutral =
```

```
    | NVar of string
    | NApp of neutral * nf

and env = (string * value) list


let rec eval env term =
    match term with
    | Var var        -> List.assoc var env
    | Lam(var, body) -> VLam(env, var, body)
    | App(f, a)      -> apply_value (eval env f) (eval env a)

and apply_value vf va =
    match vf with
    | VLam(env, var, body)   -> eval ((var, va) :: env) body
    | VNeutral(Fun(a, b), n) -> VNeutral(b, NApp(n, (a, va)))
    | _                      -> failwith "type error"


let rec readback (typ, value) =
    match typ, value with
    | Fun(a, b), _ ->
        let x = fresh_var () in
        let x_value = VNeutral(a, NVar x) in
        Lam(x, readback (b, apply_value value x_value))
    | Base _, VNeutral(_, ne) ->
        readback_neutral ne
    | _ ->
        failwith "type error"

and readback_neutral ne =
    match ne with
    | NVar x     -> x
    | NApp(f, a) -> App(readback_neutral f, readback a)
```