Normalization with Locally Nameless Representation

```
let nf_wrong (t : term) : term =
  let rec aux (t : term) (l : term list) : term =
    match t with
    | Ap (t1 , t2) -> aux t1 (t2 :: l)
    | Lam t' ->
      begin
        match l with
        | [] -> Lam (aux t' [])
        | u :: l' -> aux (subst_bound t' u) l'
      end
    | Var _ ->
      let norm_l = List.map l ~f:(fun t -> aux t []) in
      List.fold norm_l ~init:t ~f:(fun u v -> Ap (u , v))
    in
  aux t []
```

这是一个错误的 normalization 实现. 为什么错误?

考虑 `nf_wrong` 对下面的这个 λ 表达式的处理. 你可以在 utop 里执行一下, 看看得到的结果是否与你认为的正确结果相符. 建议先自己思考一下再看后面的解释.

```
λ y . λ z . (λ a . λ b . a) (z (λ c . c))
```

这个表达式的 locally nameless 表示为

```
λ . λ . (λ . λ . 1) (0 (λ . 0))
```

根据 `nf_wrong` , 这个表达式被处理的过程是

```
[λ . λ . (λ . λ . 1) (0 (λ . 0))]
λ . [λ . (λ . λ . 1) (0 (λ . 0))]
λ . λ . [(λ . λ . 1) (0 (λ . 0))]
λ . λ . [λ . 0 (λ . 0)]
λ . λ . λ . 0 (λ . 0)
```

最后一步把 `λ . λ . 1` , 即 `λ a . λ b . a` 中的 `a` 换成了那个 `0 (λ . 0)` .

但这样出来的结果是不对的, 因为它最后变成了 `λ y . λ z . λ b . b (λ c . c)` , 而本来应该是 `λ y . λ z . λ b . z (λ c . c)` . 原本的 `z` 被错误地弄成 `b` 了.

错误的原因在于, `z` 表示成了 de Bruijn index 0, 然而换进去的时候进了一个 λ-abstraction (`λ b .`), 而 `z` 的 de Bruijn index 没有增加.

怎么办呢? 一个简单的想法就是在计算 `(λ x . a) b = [b/x]a` 的过程中, 每走进一个 lambda 就把 `b` 里面的 indexes 都加一. 但这样是错误的, 因为要加的其实只有 `b` 往外绑定的那些 (例如上面的 `z`). `b` 内部的绑定是跟着动的, 所以不能加. 比如上面的 `λ . λ . [(λ . λ . 1) (0 (λ . 0))]` , 在进行到下一步时, 第一个 0 要变成 1, 因为它绑定了外面的那个 λ. 第二个 0 不能变成 1, 因为它绑定的是内部的, 跟着动的那个 λ.

解决方法就是, 在做 `[b/x]a` 之前我们进行一个处理, 先把 `b` 中往外面绑定的变量的 de Bruijn index 都换成 de bruijn level, 然后正常做替换, 换完之后再把 level 正确地算回 index.

de Bruijn level 是什么? 其实很简单, 原来我们的 index 是说从当前位置往上几个 λ 是我的绑定位置. 现在说的 level 也是一个数, 它指的是从整个表达式作为树的树根往下数几个 λ 是我的绑定位置. 仍然从 0 开始标号.

我们举个例子. 下面用 `I3` 来表示 index 3, `L4` 来表示 level 4.

```
λ y . λ z . (λ a . λ b . a) (z (λ c . c))
= λ . λ . (λ . λ . I1) (I0 (λ . I0))
= λ . λ . (λ . λ . I1) (L1 (λ . I0))
= λ . λ . (λ . λ . L2) (L1 (λ . L2))
```

我们用第二个等号后的表示来做替换, 就是正确的了.

```
λ . λ . [(λ . λ . I1) (L1 (λ . I0))]
⟶ λ . λ . [λ . L1 (λ . I0)]
═ λ y . λ z . λ b . z (λ c . c)
```

为什么正确? 因为该绝对的东西 (往外绑定的变量) 绝对了 (用了 level), 该相对的东西 (内部绑定的变量) 相对了 (用了 index).

`module Good_nf` 就实现了这个正确的 normalization.

```
module Good_nf = struct

  (* 变量多了一种 level 的可能 *)
  type dvar =
    | DIndex of int
    | DLevel of int
    | DFree of string

  (* 为了加入 level, 重新定义 term *)
  type dterm =
    | DVar of dvar
    | DLam of dterm
    | DAp of dterm * dterm

  (* 你可能需要为 dterm 实现的替换 *)
  let dsubst_bound (a : dterm) (b : dterm) : dterm = … in

  (* 你可能需要 term 和 dterm 相互转换的函数 *)
  let rec to_dterm (t : term) : dterm = … in
  let rec from_dterm (t : dterm) : term = … in

  (* 正确的 normalization 函数 *)
  let nf (t : term) : term = …

end
```