Paradigmes et Interprétation

Sucre syntaxique

Julien Provillard
julien.provillard@univ-cotedazur.fr

Université Nice Sophia Antipolis

Codage

- Il est possible de définir de nouvelles fonctionnalité dans un langage à partir d'autres déjà présente.
- En d'autres termes, on introduit une expression dans le langage qui va se traduire en une composition des autres expressions disponibles.
- La nouvelle expression n'apparaît donc pas dans la représentation interne.

Codage : Exemples

- Nous avons vu deux moyens d'introduire une liaison,
- La liaison locale :

```
{let {[x 1]}
   {+ x 2}}
```

- L'application de fonction :

```
{let {[f {lambda {x} {+ x 2}}]}
   {f 1}}
```

- Ces deux expressions sont équivalentes.
- Simplifions la deuxième.

Codage : Exemples

- Nous avons vu deux moyens d'introduire une liaison,
- La liaison locale :

```
{let {[x 1]}
  {+ x 2}}
```

- L'application de fonction :

```
{{lambda {x} {+ x 2}} 1}
```

- Ces deux expressions sont équivalentes.
- Le corps de la liaison locale et de la fonction jouent le même rôle. On peut les abstraire.

Codage : Exemples

- Nous avons vu deux moyens d'introduire une liaison,
- La liaison locale :

```
{let {[x 1]}
  body}
```

- L'application de fonction :

```
{{lambda {x} body} 1}
```

- Ces deux expressions sont équivalentes.
- De même, le membre droit de la liaison locale et l'argument de la fonction jouent des rôles similaires. On peut à nouveau les abstraire.

Université Nice Sophia Antipolis

Codage : Exemples

- Nous avons vu deux moyens d'introduire une liaison,
- La liaison locale :

```
{let {[x rhs]}
  body}
```

- L'application de fonction :

```
{{lambda {x} body} rhs}
```

- Ces deux expressions sont équivalentes.

Université Nice Sophia Antipolis

Codage : Exemples

- Nous venons de montrer que

```
{let {[name rhs]} body}
```

Est, en toute généralité, équivalent à

```
{{lambda {name} body} rhs}
```

- On peut dès lors retirer la variante `letE` du type `Exp` si l'analyse syntaxique s'occupe de cette traduction.

```
(test (parse `{let {[x 1]} {+ x 2}})
      (appE (lamE 'x (plusE (idE 'x) (numE 2)))
            (numE 1)))
```

Université Nice Sophia Antipolis

Codage : Exemples

- Cette opération s’appelle retirer le **sucre syntaxique**.
- On dit aussi que la liaison locale est **implémentée par sucre syntaxique** (sur les fonctions).

- On pourrait faire de même, par exemple, pour l’opposé.

```
(test (parse `{neg e}) ; <=> {* -1 e}
      (multE (numE -1) (idE 'e)))
```

- Mais dans ce cas, on aurait juste pu définir une nouvelle fonction dans une **bibliothèque** spécialisée.

```
{let {[neg {lambda {n} {* -1 n}}]}
  ... }
```

Université Nice Sophia Antipolis

Codage : Exemples

- Le sucre syntaxique et le développement de bibliothèques sont deux manières de coder de nouvelles fonctionnalités sans modifier le langage interne.
- Le sucre syntaxique permet de coder tout ce qu’on trouverait dans une bibliothèque.
- En agissant directement sur la manière dont est analysé le code, il permet de faire plus que d’ajouter des définitions.
- Sa compréhension peut par contre devenir complexe.

Codage : Exemples

- On a vu que les boîtes pouvait être codée par des variables mutables au sein d'une bibliothèque.

```
(define (crate val)
  (pair (lambda () val)
        (lambda (new-val) (set! val new-val))))

(define (uncrate c)
  ((fst c)))

(define (set-crate! c val)
  ((snd c) val))
```

Université Nice Sophia Antipolis

Codage : Exemples

- Les boîtes peuvent aussi coder les variables mutables mais uniquement par sucre syntaxique.
- Chaque fois qu'une variable est initialisée (dans une liaison locale ou un appel de fonction) sa valeur est placée dans une boîte.
- Chaque fois qu'une variable est utilisée, on réalise un `unbox`.
- L'instruction `set` est remplacée par `set-box!`.

```
{let {[x 1]}
  {let {[f {lambda {y}
             {+ x y}}]}
    {begin
      {set! x 10}
      {f 2}}}}
```

```
{let {[x {box 1}]}
  {let {[f {box {lambda {y}
                  {+ {unbox x} {unbox y}}}}]}
    {begin
      {set-box! x 10}
      {{unbox f} {box 2}}}}}
```

Université Nice Sophia Antipolis

Sucre syntaxique et langages

- Certains langages autorisent nativement les extensions syntaxiques.
- Les macros C en sont un exemple :

```
#define NEG(x) (* -1 (x))
```

Le préprocesseur remplacera **littéralement** tous les appels à la macro `NEG` par sa définition.

- On a également vu la clause `define-syntax-rule` en `plait` :

```
(define-syntax-rule (with [(v-id sto-id) call] body)
  (type-case Result call
    [(v*s v-id sto-id) body]))
```

Université Nice Sophia Antipolis

Codage

- Pourquoi étudier le codage ?
- Pour identifier les structures réellement fondamentales d’un langage.
- Pour simplifier le langage noyau et donc l’interpréteur (ou le compilateur) associé.
- Mais, il faut être conscient que des questions d’efficacité peuvent prendre le pas. Un codage peut fortement dégrader les performances par rapport à une implémentation native.

Fonctions à plusieurs paramètres

- Peut-on simuler les fonctions à plusieurs paramètres par des fonctions à un paramètre ?

```
{let {[f {lambda {x y}
           {+ x y}}]}
  {f 1 2}}
```

➡

```
{let {[f {lambda {x}
           {lambda {y}
             {+ x y}}}]}
  {{f 1} 2}}
```

Université Nice Sophia Antipolis

Fonctions à plusieurs paramètres

- De manière générale, on transforme les codes sources de cette manière.

```
{lambda {par1 par2 ... parn} body}
```

➡

```
{lambda {par1}
  {lambda {par2}
    ...
      {lambda {parn}
        body}}}
```

```
{f arg1 arg2 ... argn}  ➡  {{{f arg1} arg2} ... argn}
```

- Cette transformation s'appelle la **curryfication**.

Université Nice Sophia Antipolis

Branchement conditionnel

- Peut-on faire de même pour l'expression `if` ?

```
{if test
    if-true
    if-false}
```

- Le branchement conditionnel est une forme du langage, ce n'est pas une fonction. Voyez-vous pourquoi ?
- Ces arguments ne sont pas tous évalués, seule l'une des branche l'est !

Université Nice Sophia Antipolis

Branchement conditionnel

- Peut-on faire de même pour l'expression `if` ?

```
{if* test
     if-true
     if-false}
```

- Essayons de le transformer en une fonction `if*`.
- Il faut pouvoir retarder l'évaluation de ses arguments !

Université Nice Sophia Antipolis

Branchement conditionnel

- Peut-on faire de même pour l'expression `if` ?

```
{if* test
     {lambda {d} if-true}
     {lambda {d} if-false}}
```

- Essayons de le transformer en une fonction `if*`.
- Il faut pouvoir retarder l'évaluation de ses arguments !
- Il faut pouvoir sélectionner une branche.

`{lambda {x} {lambda {y} x}}` si `test` est vrai

`{lambda {x} {lambda {y} y}}` si `test` est faux

Université Nice Sophia Antipolis

Branchement conditionnel

- Peut-on faire de même pour l'expression `if` ?

```
{if* test
     {lambda {d} if-true}
     {lambda {d} if-false}}
```

- Essayons de le transformer en une fonction `if*`.
- Il faut pouvoir retarder l'évaluation de ses arguments !
- Il faut pouvoir sélectionner une branche.

```
{lambda {x} {lambda {y} x}} ≡ true
{lambda {x} {lambda {y} y}} ≡ false
```

Branchement conditionnel

- Peut-on faire de même pour l'expression `if` ?

```
{if* {{test
       {lambda {d} if-true}}
      {lambda {d} if-false}}}
```

- Essayons de le transformer en une fonction `if*`.
- Il faut pouvoir retarder l'évaluation de ses arguments !
- Il faut pouvoir sélectionner une branche.

```
{lambda {x} {lambda {y} x}} ≡ true
{lambda {x} {lambda {y} y}} ≡ false
```

- Il reste juste à forcer l'évaluation de la bonne branche.

Université Nice Sophia Antipolis

Branchement conditionnel

- Peut-on faire de même pour l'expression `if` ?

```
{{{test
    {lambda {d} if-true}}
   {lambda {d} if-false}} 0} ; argument arbitraire
```

- Essayons de le transformer en une fonction `if*`.
- Il faut pouvoir retarder l'évaluation de ses arguments !
- Il faut pouvoir sélectionner une branche.

```
{lambda {x} {lambda {y} x}} ≡ true
{lambda {x} {lambda {y} y}} ≡ false
```

- Il reste juste à forcer l'évaluation de la bonne branche.

Université Nice Sophia Antipolis

Branchement conditionnel

- Pour résumer, on transforme

```
{if test
    if-true
    if-false}
```

en

```
{{{test
   {lambda {d} if-true}}
  {lambda {d} if-false}} 0} ; argument arbitraire
```

en supposant qu'on a un codage particulier pour vrai et faux

```
{lambda {x} {lambda {y} x}} ≡ true
{lambda {x} {lambda {y} y}} ≡ false
```

Université Nice Sophia Antipolis

Paires

- Peut-on coder les paires avec les fonctions ?

```
{pair x y} ⇒ {lambda {sel} {{sel x} y}}

pair  ≡ {lambda {x}
          {lambda {y}
            {lambda {sel} {{sel x} y}}}}
fst   ≡ {lambda {p} {p true}}
snd   ≡ {lambda {p} {p false}}

{fst {{pair 1} 0}}
  ⇒ {{lambda {p} {p true}} {{pair 1} 0}}
  ⇒ {{{pair 1} 0} true}}
  ⇒ {{lambda {sel} {{sel 1} 0}} true}
  ⇒ {{true 1} 0}
  ⇒ {{{lambda {x} {lambda {y} x}} 1} 0}
  ⇒ {{lambda {y} 1} 0}
  ⇒ 1
```

λ-calcul

- Pour nos codages, nous avons utilisés les symboles ainsi que la définition et à l'application de fonctions. Le langage qui se limite à ces trois expressions s'appelle le λ-calcul. Sa grammaire est :

```
<Exp> ::= <Symbol>
        | λ<Symbol>.<Exp>
        | <Exp> <Exp>
        | (<Exp>)
```

```
λx.λy.x ≡ true
λx.λy.y ≡ false
```

Et les nombres ?

Université Nice Sophia Antipolis

Arithmétique de Peano

- L'arithmétique de Peano est une formalisation minimale de l'arithmétique qui contient :
  - une constante `zero`,
  - une fonction unaire `succ`,
  - deux fonctions binaires `+` et `*`.
- Les entiers sont représentés pas l'application itérée de la fonction `succ` sur la constante `zero`.

```
1 ≡ (succ zero)
2 ≡ (succ (succ zero))
3 ≡ (succ (succ (succ zero)))
```
etc…

Université Nice Sophia Antipolis

Représentation des entiers

- On va s’en inspirer pour coder les entiers

```
0 ≡ zero
1 ≡ (succ zero)
2 ≡ (succ (succ zero))
3 ≡ (succ (succ (succ zero)))
```

- Pour les transformer en λ-expressions, il faut introduire les variables `succ` et `zero`.

Université Nice Sophia Antipolis

Représentation des entiers

- On va s'en inspirer pour coder les entiers

```
0 ≡ λsucc.λzero.zero
1 ≡ λsucc.λzero.succ zero
2 ≡ λsucc.λzero.succ (succ zero)
3 ≡ λsucc.λzero.succ (succ (succ zero))
```

- Pour les transformer en λ-expressions, il faut introduire les variables `succ` et `zero`.
- Mais les noms de variables ne sont pas pertinents, on peut les changer.

Représentation des entiers

- On va s'en inspirer pour coder les entiers

```
0 ≡ λf.λx.x
1 ≡ λf.λx.f x
2 ≡ λf.λx.f (f x)
3 ≡ λf.λx.f (f (f x))
```

L'entier n est donc codé par n fois l'application d'une fonction f à un argument x.

- Pour les transformer en λ-expressions, il faut introduire les variables `succ` et `zero`.
- Mais les noms de variables ne sont pas pertinents, on peut les changer.
- Ce codage s'appelle les **entiers de Church**.

Manipuler les entiers : incrémentation

```
add1 ≡ λn.λf.λx.f (n f x)

add1 0
➡ (λn.λf.λx.f (n f x)) 0
➡ λf.λx.f (0 f x)
➡ λf.λx.f ((λf.λx.x) f x)
➡ λf.λx.f x
➡ 1
```

On a fait une substitution dans le corps d'un lambda qui n'était pas évalué!

Avez-vous remarqué?

β-réduction

- Une **β-réduction** revient à substituer un paramètre par son argument, `(λx.body) arg` devient `body[x ≡ arg]`.
- On peut faire cette opération n'importe où dans une λ-expression, pas forcément en tête.
- Transforme une λ-expression en une autre λ-expression.
- Une λ-expression est sous **forme normale** si on ne peut lui appliquer aucune β-réduction.
- Si on peut réduire une λ-expression à une forme normale par une suite de β-réductions, cette forme normale est assimilée au résultat de la λ-expression.

Université Nice Sophia Antipolis

Forme normale

- Une λ-expression est dite **normalisable** si elle admet une forme normale par une suite de β-réductions.
- Une λ-expression est dite **fortement normalisable** s’il n’existe pas de chaîne infinie de β-réductions à partir d’elle.
- Une λ-expression est **faiblement normalisable** si elle est normalisable mais pas fortement normalisable.
- **Théorème de confluence :** Si une λ-expression `t` peut se réduire en `u` ou en `v`, alors il existe une λ-expression `w` telle que `u` et `v` se réduisent en `w`.
- Le théorème assure l'unicité de la forme normale quand elle existe.

Université Nice Sophia Antipolis

Exemple

- `(λx.λy.y y) λz.z` admet une unique β-réduction en `λy.y y` qui est une forme normale. L'expression initiale est donc normalisable et même fortement normalisable.

- `(λx.x x) (λx.x x)` se réduit uniquement en `(λx.x x) (λx.x x)`. L'expression n'est donc pas normalisable.

- `(λx.y) ((λx.x x) (λx.x x))` peut se réduire en `y` ou en elle-même. L'expression est normalisable mais pas fortement normalisable.

Manipuler les entiers : addition

- On a défini l'incrémentation `add1 ≡ λn.λf.λx.f (n f x)`.
- On cherche maintenant à définir l'addition de deux nombres.

```
add2 ≡ λn.add1 (add1 n)
add3 ≡ λn.add1 (add1 (add1 n))
```

$$\texttt{add} \equiv \lambda n.\lambda m.\texttt{add1}\ \underbrace{(\texttt{add1}\ \ldots\ (\texttt{add1}}_{\texttt{m}\text{ fois}}\ n))$$

- Comment appliquer `m` fois une fonction à un argument ?
- Mais c'est juste la définition de `m` !

Université Nice Sophia Antipolis

Manipuler les entiers : addition

- On a défini l'incrémentation `add1 ≡ λn.λf.λx.f (n f x)`.
- On cherche maintenant à définir l'addition de deux nombres.

```
add2 ≡ λn.add1 (add1 n)
add3 ≡ λn.add1 (add1 (add1 n))
add ≡ λn.λm.m add1 n

add 1 2
➡ 2 add1 1
➡ (λf.λx.f (f x)) add1 1
➡ add1 (add1 1) ➡ 3
```

Université Nice Sophia Antipolis

Manipuler les entiers : multiplication

- On a désormais accès à l'addition, comment faire pour obtenir la multiplication `mult` ?

$$m \times n = \underbrace{n + n + \ldots + n}_{m \text{ fois}} + 0$$

```
mult ≡ λn.λm.m (add n) 0
```

Université Nice Sophia Antipolis

Manipuler les entiers : test de nullité

- On cherche à trouver une λ-expression qui renvoie `true` si son argument est `0`, `false` sinon.

```
iszero ≡ λn.n (λx.false) true
```

- Appliquer 0 fois `λx.false` à `true` produit `true`.
- Appliquer au moins une fois `λx.false` à `true` produit `false`.

```
iszero 0 ⟶ 0 (λx.false) true ⟶ true
iszero 1 ⟶ 1 (λx.false) true ⟶ (λx.false) true ⟶ false
```

Manipuler les entiers : décrémentation

- Peut-on faire la décrémentation sur le modèle de l'incrémentation ?

```
add1 ≡ λn.λf.λx.f (n f x)
sub1 ≡ λn.λf.λx. … (n f x) …
```

- On a appliqué `n` fois `f` à `x`. On voudrait qu'elle ne le soit que `n-1` fois.
- Il n'est pas possible d'annuler l'application d'une fonction !

- Solution : travailler sur des couples et se rappeler le prédécesseur.

```
pair 0 1 ⟶ pair 1 2 ⟶ pair 2 3 ⟶ … ⟶ pair n-1 n
```

Manipuler les entiers : décrémentation

- Comment passer d'un couple au suivant ?

```
shift ≡ λp.pair (snd p) (add1 (snd p))
```

- Comment obtenir la décrémentation ?
- En appliquant n fois le shift à partir du couple (0, 0), on obtient le couple (n-1, n). Il suffit alors de projeter la première composante.

```
sub1 ≡ λn.fst (n shift (pair 0 0))
```

- On en déduit la soustraction.

```
sub ≡ λn.λm.m sub1 n
```

Université Nice Sophia Antipolis

Pour résumer

- Le λ-calcul permet d'avoir une représentation pour :
  - les fonctions,
  - la liaison locale,
  - les booléens et les primitives associées,
  - les entiers et l'arithmétique.

- Que manque-t-il ?

Récursion : exemple de la factorielle

```
(local [(define fac (lambda (n)
                      (if (zero? n)
                          1
                          (* n (fac (- n 1))))))]
  (fac 6))
```

- `local` lie le nom `fac` dans l'environnement mais aussi dans le corps de la fonction.

Université Nice Sophia Antipolis

Récursion : exemple de la factorielle

```
(letrec ([fac (lambda (n)
                (if (zero? n)
                    1
                    (* n (fac (- n 1)))))])
  (fac 6))
```

- `letrec` a une forme plus proche de `let` mais garde les effets de la structure `local` sur l'environnement.

Université Nice Sophia Antipolis

Récursion : exemple de la factorielle

```
(let ([fac (lambda (n)
             (if (zero? n)
                 1
                 (* n (fac (- n 1)))))])
  (fac 6))
```

- Ne fonctionne pas, `fac` est un identificateur libre dans le corps de `fac`.
- Nous avons vu que pour lier un nom dans l'environnement, il suffisait de le passer en argument.

Récursion : exemple de la factorielle

```
(let ([facX (lambda (f n)
               (if (zero? n)
                   1
                   (* n (f f (- n 1)))))])
  (facX facX 6))
```

- Ne fonctionne pas, `fac` est un identificateur libre dans le corps de `fac`.
- Nous avons vu que pour lier un nom dans l'environnement, il suffisait de le passer en argument.

Récursion : exemple de la factorielle

```
(let ([facX (lambda (f n)
              (if (zero? n)
                  1
                  (* n (f f (- n 1)))))])
  (facX facX 6))
```

- Et si l'on souhaite obtenir `fac` de nouveau ?

Récursion : exemple de la factorielle

```
(let ([fac (lambda (n)
             (let ([facX (lambda (f n)
                           (if (zero? n)
                               1
                               (* n (f f (- n 1)))))])
               (facX facX n)))])
  (fac 6))
```

- Et si l'on souhaite obtenir `fac` de nouveau ?
- Notre langage ne contient que des fonctions à un paramètre !

Récursion : exemple de la factorielle

```
(let ([fac (lambda (n)
             (let ([facX (lambda (f)
                              (lambda (n)
                                (if (zero? n)
                                    1
                                    (* n ((f f) (- n 1))))))])
               ((facX facX) n)))])
  (fac 6))
```

- Et si l’on souhaite obtenir fac de nouveau ?
- Notre langage ne contient que des fonctions à un paramètre !

Récursion : exemple de la factorielle

```
(let ([fac (lambda (n)
             (let ([facX (lambda (f)
                           (lambda (n)
                             (if (zero? n)
                                 1
                                 (* n ((f f) (- n 1))))))])
               ((facX facX) n)))])
  (fac 6))
```

- On peut simplifier `(lambda (n) (let ([f ... ]) ((f f) n)))` par `(let ([f ... ]) (f f))`

Récursion : exemple de la factorielle

```
(let ([fac
       (let ([facX (lambda (f)
                     (lambda (n)
                       (if (zero? n)
                           1
                           (* n ((f f) (- n 1))))))])
         (facX facX))])
  (fac 6))
```

Ressemble beaucoup à la définition de `fac`

- On peut simplifier `(lambda (n) (let ([f ... ]) ((f f) n)))` par `(let ([f ... ]) (f f))`

Récursion : exemple de la factorielle

```
(let ([fac
       (let ([facX (lambda (f)
                     (lambda (n)
                       (if (zero? n)
                           1
                           (* n ((f f) (- n 1))))))])
         (facX facX))])
  (fac 6))
```

Ressemble beaucoup à la définition de `fac`

- On peut introduire une liaison pour `(f f)`.

Récursion : exemple de la factorielle

```
(let ([fac
       (let ([facX (lambda (f)
                     (let ([fac (f f)])
                      (lambda (n)
                        (if (zero? n)
                            1
                            (* n (fac (- n 1)))))))])
         (facX facX))])
  (fac 6))
```

Exactement la définition de fac

- On peut introduire une liaison pour (f f).
- Problème : (f f) est évalué même pendant le cas d’arrêt !
- Il faut retarder l’évaluation de (f f).

Université Nice Sophia Antipolis

Récursion : exemple de la factorielle

```
(let ([fac
       (let ([facX (lambda (f)
                     (let ([fac (lambda (n) ((f f) n))])
                      (lambda (n)
                        (if (zero? n)
                            1
                            (* n (fac (- n 1)))))))])
         (facX facX))])
  (fac 6))
```

- On peut introduire une liaison pour `(f f)`.
- Problème : `(f f)` est évalué même pendant le cas d'arrêt !
- Il faut retarder l'évaluation de `(f f)`.

Université Nice Sophia Antipolis

Récursion : généralisation

```
(define (mk-rec body-proc)
  (let ([fX (lambda (f)
              (let ([name (lambda (x) ((f f) x))])
                (body-proc name)))])
    (fX fX)))

(let ([fac (mk-rec
            (lambda (fac)
              ; Exactement le corps de fac
              (lambda (n)
                (if (zero? n)
                    1
                    (* n (fac (- n 1)))))))])
  (fac 6))
```

Université Nice Sophia Antipolis

Récursion : Fibonnaci

```
(let ([fibo (mk-rec
              (lambda (fibo)
                (lambda (n)
                  (if (or (= n 0) (= n 1))
                      1
                      (+ (fibo (- n 1)) (fibo (- n 2)))))))])
  (fibo 6))
```

Université Nice Sophia Antipolis

Récursion : somme d’une liste

```
(let ([sum (mk-rec
            (lambda (sum)
              (lambda (l)
                (if (empty? l)
                    0
                    (+ (first l) (sum (rest l)))))))])
  (sum (list 1 2 3 4 5)))
```

Université Nice Sophia Antipolis

Implémentation de la récursion

- On vient de voir que

```
{letrec {[fac
          {lambda {n}
            {if {zero? n}
                1
                {* n {fac {- n 1}}}}}]}
  {fac 6}}
```

pouvait être remplacé de manière équivalente par

```
{let {[fac {mk-rec
            {lambda {fac}
              {lambda {n}
                {if {zero? n}
                    1
                    {* n {fac {- n 1}}}}}}}]}
  {fac 6}}
```

Université Nice Sophia Antipolis

Implémentation de la récursion

- De manière générale

```
{letrec {[name rhs]} body}
```

est équivalent à

```
{let {[name {mk-rec {lambda {name} rhs}}]} body}
```

et en réécrivant mk-rec

```
{let {[name {{lambda {body-proc}
               {let {[fX {lambda {f}
                           {let {[name {lambda {x} {{f f} x}}]}
                             {body-proc name}}}]}
                 {fX fX}}}
             {lambda {name} rhs}}]}
  body}
```

Implémentation de la récursion

```
{letrec {[fac
          {lambda {n}
            {if {zero? n}
                1
                {* n {fac {- n 1}}}}}]}
  {fac 6}}
```

→

```
(λfac.fac (λf.λx.f (f (f (f (f (f x))))))((λbody-proc.(λfX.fX fX)(λfX.(λf.body-proc
f)(λx.fX fX x)))(λfac.λn.(λn.n (λ_.λx.λy.y) (λx.λy.x)) n (λ_.λf.λx.f x) (λ_.(λn.λm.n
((λn.λm.n (λn.λf.λx.f (n f x)) m) m) (λf.λx.x)) n (fac ((λn.λm.m ((λ_shift.λn.(λp.p
(λx.λy.x))(n _shift ((λx.λy.λsel.sel x y)(λf.λx.x) (λf.λx.x))))(λp.(λx.λy.λsel.sel x
y)((λp.p (λx.λy.y)) p) ((λn.λf.λx.f (n f x))((λp.p (λx.λy.y)) p)))) n) n (λf.λx.f
x)))) _))
```