Rapport de TIPE
Réécriture de graphe comme évaluation β-optimale du λ-calcul

Juliette PONSONNET, MPI, 2023-2024

Sommaire

1 Réseaux d'interaction

1.1 Combinateurs d'interaction

Dans ce TIPE, on s'intéresse à un cas particulier de réseaux d'interaction [4], les combinateurs d'interaction symétriques proposés par Lafont [3]. Le système de combinateurs d'interaction (symétriques ou non [6]) est un *système univsersel d'interaction*, donc on peut réduire l'étude de tout tel système à celui-ci (en particulier, les machines de Turing se réduisent à un système d'interaction).

Dans ce système, on utilise l'alphabet $\Sigma = \{\delta, \gamma, \varepsilon\}$ et les règles suivantes :

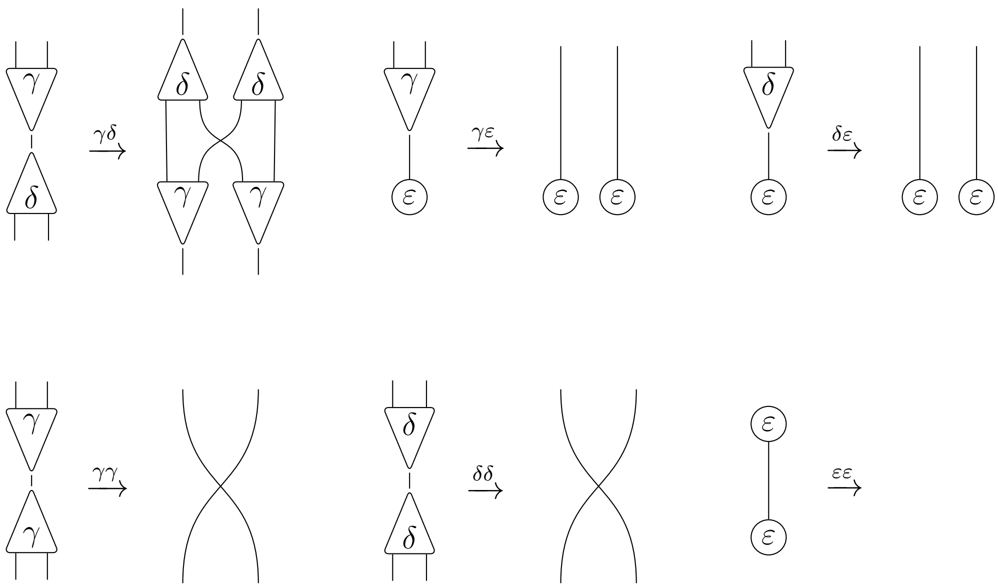

Figure 1: Les règles d'interaction symétrique

Les symboles δ et γ sont d'arité 2 tandis que ε est d'arité 0 (on tendra à l'ignorer dans la pratique, ε agit comme un ramasse-miettes, détruisant tout agent avec lequel il interagit). On appellera les règles $\delta\gamma$ et $\gamma\delta$ les **règles de commutation** et les règles $\delta\delta$ et $\gamma\gamma$ les **règles d'annihilation**. On peut démontrer que toutes ces règles sont essentielles à l'universalité (sauf $\delta\varepsilon$). Le système est donc minimal en ce sens. Une **paire active** est alors un couple d'agents susceptibles d'interagir en accord avec ces règles, c'est-à-dire dont les ports principaux sont connectés.

1.2 Réduction

1.2.1 Confluence

En remarquant qu'aucune des règles ne peut supprimer une paire active autre que la sienne, on peut montrer qu'étant donné un réseau v_0, si il existe deux paires actives distinctes p et p', alors le diagramme suivant commute :

$$
\begin{tikzcd}
 & v_0 \arrow[dl, "p"'] \arrow[dr, "p'"] & \\
v_1 \arrow[dr, "p'"'] & & v_1' \arrow[dl, "p"] \\
 & v_2 &
\end{tikzcd}
$$

Figure 2: Diagramme de confluence

On dira de v qu'il est réductible si il existe v_f sans paires actives tel que $v \longrightarrow^* v_f$, et on notera $v \Downarrow v_f$.

On déduit du diagramme de confluence que si v est un réseau réductible, alors toute réduction commençant en v est finie, de même longueur et aboutit en v_f (unicité du résultat). En particulier, l'ordre choisi d'exécution n'a aucune importance et on peut exécuter plusieurs réécritures simultanément.

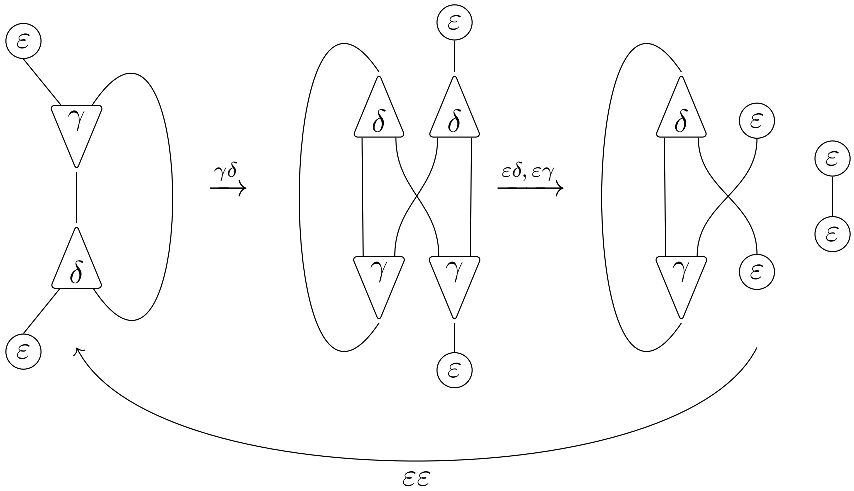

Figure 3: Un réseau dont la réduction ne termine pas

1.2.2 Algorithme à pile

Lafont [3] propose un algorithme d'exécution utilisant deux piles, simplifiable dans le cas symétrique à une pile, pour calculer un câblage auquel se réduit un réseau simplement en le traversant.

- Si on entre par un port auxiliaire, l'empiler et continuer en ressortant par le port principal.

- Si on entre par le port principal, dépiler un port et ressortir par ce port. Si ce n'est pas possible, l'algorithme termine.

Si on entre par un fil libre de v, on ressort par le fil libre correspondant dans le câblage ω tel que $v \Downarrow \omega$.

On adaptera cet algorithme dans le cas plus général de la réduction à une forme normale, en explorant les chemins obtenus en récrivant en direct lors de la traversée.

2 Application au λ-calcul

2.1 Conversion

On élargit ici l'alphabet à $\Sigma = \{\gamma, \delta_x^n \mid x \in \mathcal{V}, n \in \mathbb{N}\}$ où $\mathcal{V}$ est l'ensemble des variables. On définit un **pointeur** comme un couple (α, p) où α est un agent et p est un port (principal, gauche ou droit). On définit l'algorithme de conversion par induction structurelle sur les λ-termes sans variables libres. On suppose ainsi à chaque étape disposer pour chaque sous-terme d'une fonction $\mathcal{A}$ associant à chaque variable libre un pointeur. On notera φ la fonction associée cette procédure.

Les agents γ répliquent les règles du λ-calcul et les agents δ assurent la linéarité du système. Pour avoir le réseau final correspondant à un λ-terme T, on considère simplement $\bullet$-φ_T et on appelle $\bullet$ la racine.

On appellera **terme** le résultat d'une conversion, et on remarque qu'il ne reste plus d'arête pendante à la fin de la procédure.

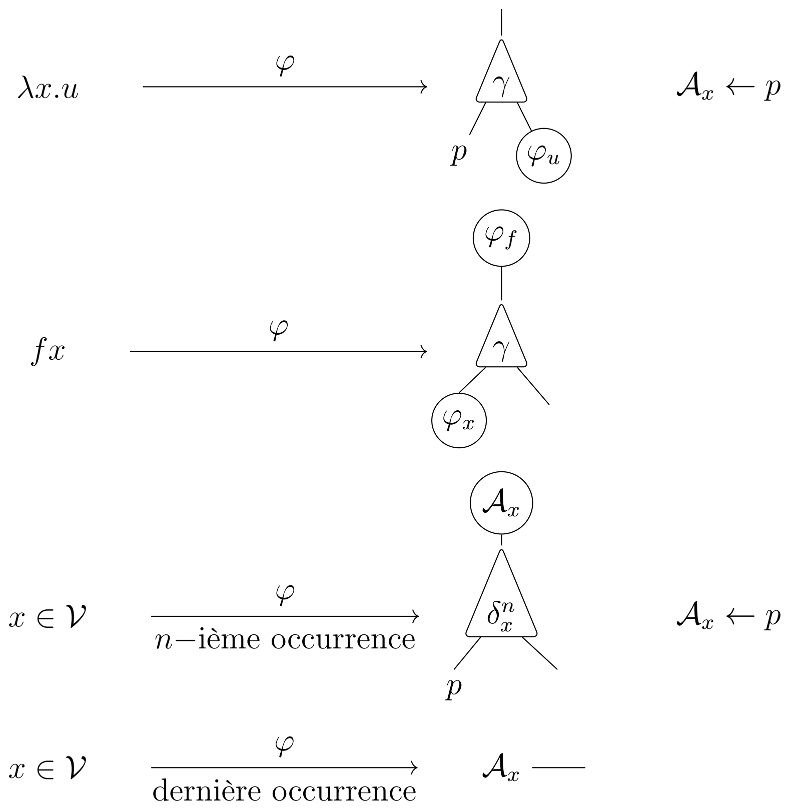

Figure 4: Conversion de λ-terme en réseau d'interaction symétrique

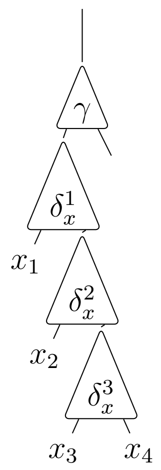

Figure 5: Linéarisation de la variable x

2.2 Optimalité

Avec l'algorithme de Lamping [5], on peut évaluer un terme converti de manière optimale, au sens du minimum de β-réductions réalisées au total. Ici on étudie une variante [7, 1] *sans oracle*, ce qui interdit l'évaluation de certains termes, mais qui permet d'implémenter l'algorithme de manière simplifiée (sans *book-keeping*) avec une variante de l'algorithme de Lafont sur le réseau décrit ci-dessus.

$(\lambda y.yy)(\lambda f.\lambda x.f(f\,x))$

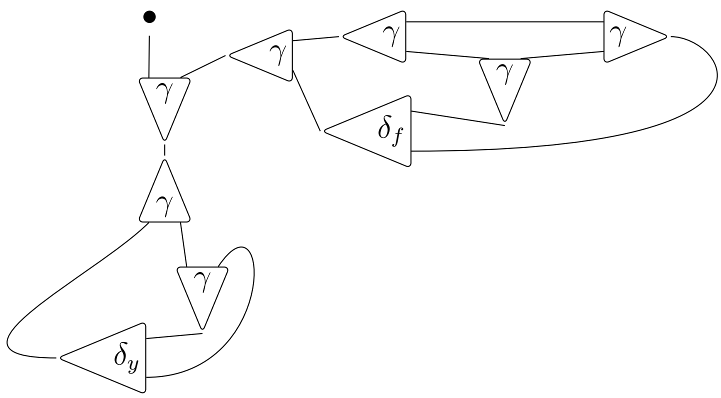

Figure 6: Un terme dont la réduction ne termine pas sans oracle

La propriété d'optimalité vient du partage rendu possible par les agents δ : chaque terme qui est évalué offre le maximum d'information en le partageant à tous ses «clones». Le graphe est alors un *graphe de partage* [1].

3 Machine virtuelle *à la* HVM

3.1 Choix des primitives

En acceptant le manque d'oracle, on peut étudier le système très simple à six règles et les programmer en dur.

Si on *polarise* le réseau (c'est-à-dire assigne un + ou − aux ports de sorte à ce que tout port connecté soit du même signe, et suivant des règles spécifiques à chaque type d'agent), on remarque que chaque agent joue en réalité deux rôles. Dans le but d'expliciter l'algorithme [2], on représentera séparément les agents *duaux* comme λ et @, qui sont les deux polarisations de γ, ainsi que ρ et σ qui sont les deux polarisations de δ.

On condense aussi les tours de γ qui peuvent servir à stocker des pseudo-structures de données avec un agent $n-$aire $\Gamma_n(\iota)$ où ι est un identifiant.

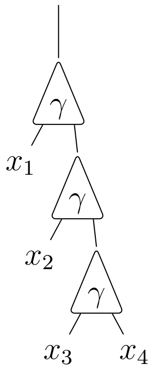

Figure 7: Structure composée $\Gamma_4(x)(x_1, x_2, x_3, x_4)$

3.2 Un langage avec pattern-matching

Étant munis de nouveaux agents d'arité quelconques et distinguables, on est libre de nantir notre système de nouvelles règles d'interaction de la forme $\Gamma_n(\iota)(x_1 \ldots x_n) \to f(x_1 \ldots x_n)$. On formule alors un langage `Exal` (pour *EXAmple Language*) permettant de décrire de telles règles.

```
Nats
    0 → Nil,
    n → Cons n (Nats (- n 1)).
Drop
    n Nil → Nil,
    0 xs  → xs,
    n (Cons x xs) → Drop (- n 1) xs.
Head
    (Cons x xs) → xs.
Main n → Head (Drop 100 (Nats n)).
```

Listing 1: Un exemple de programme `Exal`

Évaluer un programme `Exal` revient alors à réduire le réseau $\Gamma_0(\texttt{Main})$ (ou ici $\Gamma_1(\texttt{Main})(n)$ où n est un argument au programme).

4 Expériences

4.1 Entiers comme λ-termes fusibles

En λ-calcul, il est commun de représenter $n \in \mathbb{N}$ comme le terme qui à une fonction associe sa composée $n-$ième : $n \equiv \lambda f.\lambda x.\, f^n x$. Ainsi, on peut écrire des fonctions `successeur`, `addition` et `multiplication` :

$$\begin{aligned}\texttt{successeur} &\equiv \lambda n.\lambda f.\lambda x.\ f(n\,f\,x)\\ \texttt{addition} &\equiv \lambda n.\lambda m.\ n\,\texttt{successeur}\,m\\ \texttt{multiplication} &\equiv \lambda n.\lambda m.\ n\,(\texttt{addition}\,m)0\end{aligned}$$

De ce choix résultent des programmes très lents. En gardant le même esprit, on peut choisir de représenter un entier comme une liste finie de bits : $n \equiv \lambda\varepsilon.\lambda 1\lambda 0.\ \overline{n}_2\varepsilon$. Par exemple, on aura $6 \equiv \lambda\varepsilon.\lambda 1.\lambda 0.\ 011\varepsilon$. On peut récupérer l'ancienne utilisation d'application répétée mais en utilisant cette fois les capacités de fusion de notre machine virtuelle. En notant $\Delta \equiv \lambda f.\lambda x.\ f(fx)$, et en remarquant que Δ permet la fusion car elle duplique f, on peut écrire :

$$\texttt{app}\ n \equiv n\big(\lambda f.\lambda x.x\big)\big(\lambda m.\lambda f.\texttt{app} m(\Delta f)\big)\big(\lambda m.\lambda f.\lambda x.\texttt{app} m(\Delta f)(fx)\big) \quad (1)$$

$$\texttt{succ}\ n \equiv \lambda\varepsilon.\lambda 1.\lambda 0.\ n\varepsilon 1(\lambda p.0(\texttt{succ} p)) \quad (2)$$

On note que `app` et `succ` ne sont pas des λ-termes mais une règles de réécriture. On peut alors réaliser l'addition et la multiplication comme précédemment, mais grâce à la fusion, ces opérations s'exécutent en temps logarithmique.

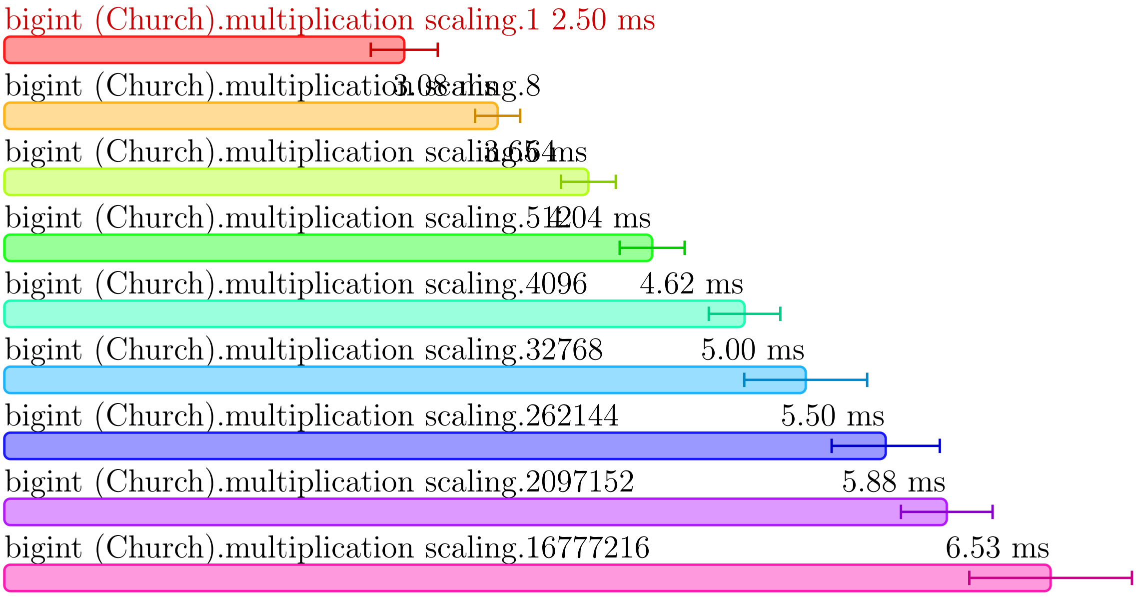

Figure 8: Multiplication d'entiers par incrémentation

4.2 Involutions

On considère ici $\varphi \equiv \lambda n.n\Delta$ le λ-terme qui à f associe f^{2^n}. On introduit ensuite les booléens :

$$\texttt{true} \equiv \lambda t.\lambda f.\ t \tag{3}$$
$$\texttt{false} \equiv \lambda t.\lambda f.\ f \tag{4}$$
$$\texttt{NOT} \equiv \lambda p.\lambda t.\lambda f.\ p\ f\ t \tag{5}$$

On peut vérifier que `NOT true` $\Downarrow$ `false` et `NOT false` $\Downarrow$ `true`. Lors de l'exécution de φn`NOT`, l'algorithme calcule directement que Δ`NOT` $\Downarrow$ `id`, et chaque appel suivant compose l'identité à elle-même. Cela résulte en un coût logarithmique (linéaire en n). En essayant ce même algorithme en `OCaml` ou `Haskell`, on obtient une complexité en $\Omega(2^n)$.

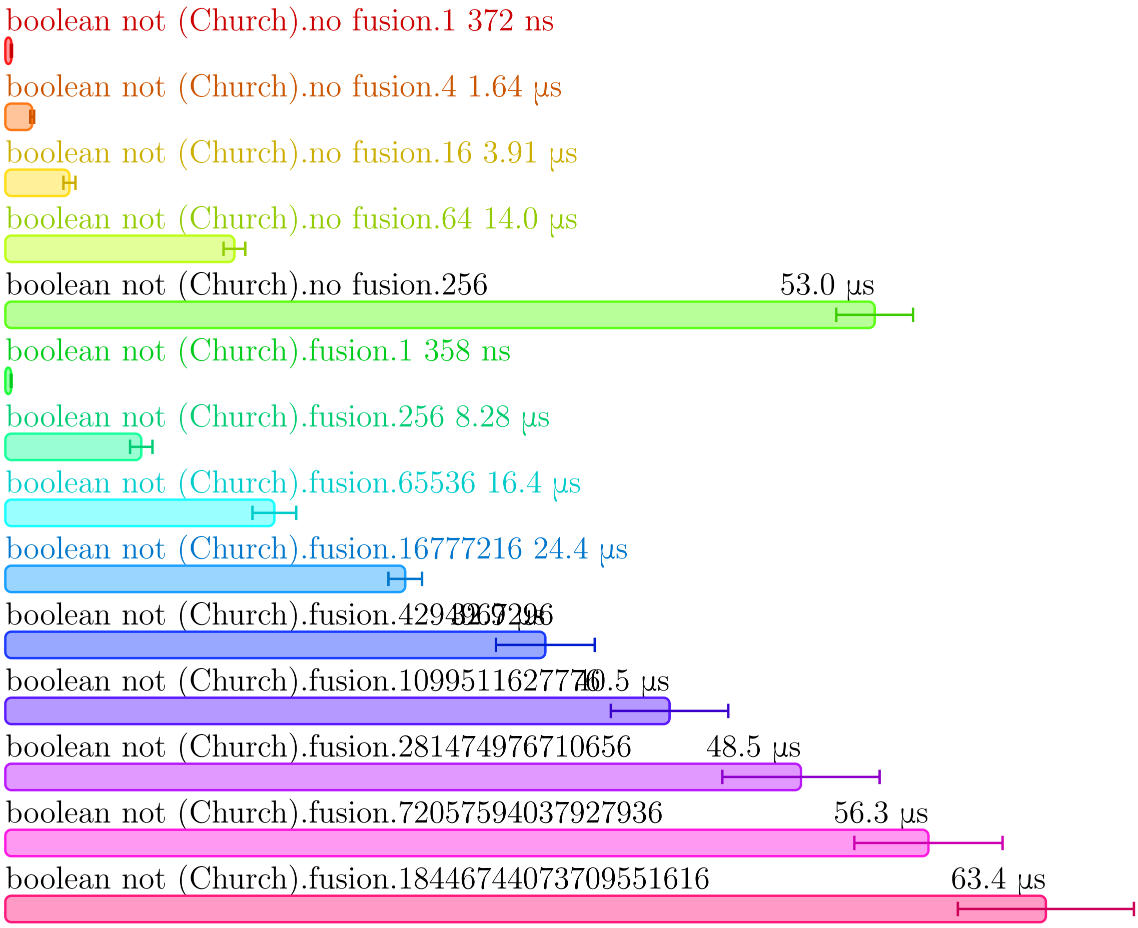

Figure 9: Fusion d'une involution - `NOT`

4.3 Flots de données

Comme illustré ci-dessus, la composition de fonctions *simplifie* les fonctions si elle le peut, c'est un mécanisme qui doit habituellement être ajouté à la main dans les compilateurs avec des règles de réécriture. On remarquera aussi que des règles comme

```
map f ∘ map g = map (f ∘ g)
fold f ∘ unfold g = refold f g
```

sont réalisées automatiquement. En mesurant le nombre de réécritures du graphe, on peut mesurer que l'algorithme suivant s'exécute avec un coût exactement affine :

```
Naturels 0 → Nil
Naturels n → Cons 1 (map successeur (Naturels (n - 1)))
```

Figure 10: Algorithme Naturels-Fusion

Celui-ci génère la liste $1 \dots n$, en incrémentant toute la queue de liste à chaque étape.

On peut réaliser l'algorithme 10 ainsi pour permettre la fusion :

$$\begin{aligned}
\texttt{Liste}\, a &: \forall r.\, r \to (a \to r \to r) \to r && (6)\\
\texttt{nil} &\equiv \lambda n.\lambda c.n && (7)\\
\texttt{cons} &\equiv \lambda a.\lambda \ell \lambda n.\lambda c.ca(\ell nc) && (8)\\
\texttt{map} &\equiv \lambda f.\lambda \ell.\lambda n.\lambda c.\ell n(\lambda v.\lambda r.c(fv)r) && (9)\\
\texttt{Naturels} &\equiv \lambda n.n\big(\lambda \ell.\texttt{cons}\, 1\, (\texttt{map succ } \ell)\big)\texttt{nil} && (10)
\end{aligned}$$

A Code source

Listings

A.1 Code Unison - Réduction de λ−termes

```
ability INet t a where
  delete : a ->{INet t a} ()
  link : Pointer a -> Pointer a ->{INet t a} ()
  root : '{INet t a} a
  unlink : Pointer a ->{INet t a} ()
  enter : Pointer a ->{INet t a} Pointer a
  read : a ->{INet t a} INode t a
  node : t ->{INet t a} a

type INet.INode t addr
  = INode t (Pointer addr) (Pointer addr) (Pointer addr)

structural type INet.Pointer a
  = MkPtr a INet.Port

structural type INet.Port
  = Main
  | Left
  | Right

type INet.run.RuntimeError a
  = InvalidAddress a
  | InvalidPointer (Pointer a)

INet.annihilate : addr -> addr ->{INet t addr} ()
INet.annihilate a b =
  use INet link
  use Port Left Right
  link (enter (a => Left)) (enter (b => Left))
  link (enter (a => Right)) (enter (b => Right))

INet.commute : addr -> addr ->{INet t addr} ()
INet.commute a b =
  use INet link node read
  use Port Left Right
  at = kind (read a)
  bt = kind (read b)
  p = node bt
  q = node bt
  r = node at
  s = node at
  link (r => Left) (p => Left)
  link (s => Left) (p => Right)
  link (r => Right) (q => Left)
  link (s => Right) (q => Right)
  link (p => Main) (enter (a => Left))
  link (q => Main) (enter (a => Right))
```

```
  link (r => Main) (enter (b => Left))
  link (s => Main) (enter (b => Right))

INet.enterPort : node → INet.Port →{INet t node} Pointer node
INet.enterPort node port = enter (MkPtr node port)

INet.INode.get : INet.Port → INode t addr → Pointer addr
INet.INode.get = cases
  Main, INode t l m r        → m
  Port.Left, INode t l m r   → l
  Port.Right, INode t l m r  → r

INet.INode.kind : INode t addr → t
INet.INode.kind = cases INode k _ _ _ → k

INet.INode.set : INet.Port → Pointer addr → INode t addr →
    INode t addr
INet.INode.set p v n = match (p, n) with
  (Main, INode t l m r)        → INode t l v r
  (Port.Left, INode t l m r)   → INode t v m r
  (Port.Right, INode t l m r)  → INode t l m v

(INet.Pointer.=>) : a → INet.Port → Pointer a
(INet.Pointer.=>) = MkPtr

INet.Pointer.ptrAddress : Pointer a → a
INet.Pointer.ptrAddress = cases MkPtr a p → a

INet.Pointer.ptrEq :
  (a →{g1} a →{g} Boolean) → Pointer a → Pointer a →{g1, g}
    Boolean
INet.Pointer.ptrEq eq = cases MkPtr a b, MkPtr c d → (eq a c) &&
    (b Port.== d)

INet.Pointer.ptrPort : Pointer a → INet.Port
INet.Pointer.ptrPort = cases MkPtr _ p → p

(INet.Port.==) : INet.Port → INet.Port → Boolean
a INet.Port.== b = match (a, b) with
  (Main, Main)               → true
  (Port.Left, Port.Left)     → true
  (Port.Right, Port.Right)   → true
  _                          → false

INet.reduce : '{INet t a} Nat
INet.reduce = do
  result do
    withInitialValue ([], []) do
      next = enter (!INet.root => Port.Left)
```

```
    step next

INet.reduce.rewritePair :
  Pointer a
  → Pointer a
  →{Counter, Store ([Pointer a], [INet.Port]), INet t a} Pointer
   a
INet.reduce.rewritePair a b =
  use List +:
  use Store modify
  (warp, exited) = Store.get
  match (a, b) with
    (MkPtr next Main, MkPtr prev Main) →
      (port, exited') =
        (do lib.base.data.List.uncons exited |> Optional.toAbort)
          |> Abort.toBug
      Store.put (warp, exited')
      back = enter (prev => port)
      rewrite (===) prev next
      Counter.increment
      enter back
    (MkPtr next Main, _) →
      modify (Tuple.mapLeft ((+:) (next => Port.Right)))
      enter (next => Port.Left)
    (MkPtr next port, _) →
      modify (Tuple.mapRight ((+:) port))
      enter (next => Main)

INet.reduce.step :
  Pointer a →{Counter, Store ([Pointer a], [INet.Port]), INet t a
   } ()
INet.reduce.step next =
  use INet.reduce step
  (warp, exited) = Store.get
  match (!INet.root === ptrAddress next, warp) with
    (true, [])         → ()
    (true, w +: warp') →
      Store.put (warp', exited)
      next' = enter w
      rewritePair next' (enter next') |> step
    (false, _)         → rewritePair next (enter next) |> step

INet.rewrite : (t → t → Boolean) → addr → addr →{INet t addr}
    ()
INet.rewrite eq a b =
  use INet delete read
  if eq (kind (read a)) (kind (read b)) then annihilate a b else
   commute a b
  unlinkNode a
```

```
  unlinkNode b
  delete a
  delete b

INet.run.asNatMap : '{g, INet t Nat} r →{g, Throw (RuntimeError
   Nat)} r
INet.run.asNatMap requests =
  use INet link
  use INode set
  use Nat +
  use NatMap adjust
  impl n m = cases
    { pure } → pure
    { INet.root () → resume } → handle resume 0 with impl n m
    { INet.node k → resume } →
      handle resume n
      with
        impl
          (n + 1)
          (m
            |> NatMap.insert
                 n (INode k (n => Port.Left) (n => Main) (n =>
   Port.Right)))
    { INet.read a → resume } →
      node = note (InvalidAddress a) (NatMap.get a m)
      handle resume node with impl n m
    { unlink x → resume } →
      action = do
        use Nat ==
        y = enter x
        if ptrEq (==) x (enter y) then
          link x x
          link y y
        else ()
        !resume
      handle !action with impl n m
    { INet.delete a → resume } →
      handle !resume with impl n (NatMap.delete a m)
    { enter ptr@(MkPtr a p) → resume } →
      node = note (InvalidPointer ptr) (NatMap.get a m)
      handle resume (INode.get p node) with impl n m
    { link x@(MkPtr a p) y@(MkPtr b q) → resume } →
      m' = m |> adjust (set p y) a |> adjust (set q x) b
      handle !resume with impl n m'
  handle !requests with impl 0 NatMap.empty

INet.streamNodes : '{Ask u, Stream (SNode t u), INet t a} ()
INet.streamNodes =
  do
```

```
    use Port Left Right
    name port node addr =
      use Store put
      m = Store.get
      next = INode.get port node
      match Map.get (addr => port) m with
        None    →
          n = ask
          put (Map.insert next n m)
          n
        Some n  →
          put (Map.delete (addr => port) m)
          n
    aux u seen =
      if Set.contains u seen then seen
      else
        use INode get
        node = INet.read u
        SNode
          (kind node) (name Left node u) (name Main node u) (name Right node u)
          |> emit
        seen
          |> Set.insert u
          |> aux (ptrAddress (get Left node))
          |> aux (ptrAddress (get Main node))
          |> aux (ptrAddress (get Right node))
    rootCC = withInitialValue Map.empty do aux !INet.root Set.empty
    ()

INet.unlinkNode : node →{INet t node} ()
INet.unlinkNode node = lib.base.abilities.Each.run do
  port = base.abilities.Each.each [Main, Port.Left, Port.Right]
  unlink (MkPtr node port)
```

Listing 2: Réseaux d'interaction et réduction

```
type Representation.Lambda t
  = App (Representation.Lambda t) (Representation.Lambda t)
  | Abs t (Representation.Lambda t)
  | Var t

type Representation.SNode t u
  = SNode t u u u

Representation.iso.INetToLambda : '{Abort, Ask u, INet Text a} Lambda u
Representation.iso.INetToLambda =
  do
```

```
    use Port Left
    term :
      Pointer a
      → [(Pointer a, u)]
      → [INet.Port]
      →{Abort, Ask u, INet Text a} Lambda u
    term p vars dupExit =
      use List +:
      use Port Right
      node = INet.read (ptrAddress p)
      match (Text.take 1 (kind node), ptrPort p) with
        ("\955", Main) →
          name = ask
          body =
            term
              (enter (ptrAddress p => Right))
              ((ptrAddress p => Left, name) +: vars)
              dupExit
          Abs name body
        ("\955", Left) →
          name = find! ((===) p << at1) vars |> at2
          Var name
        ("\955", Right) →
          arg = term (enter (ptrAddress p => Left)) vars dupExit
          fun = term (enter (ptrAddress p => Main)) vars dupExit
          App fun arg
        ("\948", Main) →
          (exit, dupExit') =
            lib.base.data.List.uncons dupExit |> Optional.toAbort
          term (enter (ptrAddress p => exit)) vars dupExit'
        ("\948", port) →
          term (enter (ptrAddress p => Main)) vars (port +:
   dupExit)
        _ → abort
    term (enter (!INet.root => Left)) [] []

Representation.iso.INetToStaticNodes : '{g, INet t Nat} () →{g} [
    SNode t Nat]
Representation.iso.INetToStaticNodes inet = withNaturals do
  Throw.toBug do
    asNatMap do
      !inet
      lib.base.data.Stream.toList streamNodes

Representation.iso.lambdaToStaticNodes : Lambda t →{Throw t} [
    SNode Text Nat]
Representation.iso.lambdaToStaticNodes lam =
  use Map insert
  use Nat + -
```

```
  use Store modify
  use Text ++
  count v = cases
    Var u   → if u === v then 1 else 0
    App f x → count v f + count v x
    Abs u x → if u === v then 0 else count v x
  aux = cases
    Abs v t →
      self = ask
      var = ask
      m = Store.get
      modify (insert v (var, count v t))
      body = aux t
      emit (SNode "λ" var self body)
      Store.put m
      self
    App f x →
      self = ask
      fid = aux f
      xid = aux x
      emit (SNode "λ" xid fid self)
      self
    Var t   →
      self = match Map.get t Store.get with
        None          → throw t
        Some (var, 1) → var
        Some (var, n) →
          dup1 = ask
          dup2 = ask
          emit (SNode ("δ" ++ Nat.toText ask) dup1 var dup2)
          modify (insert t (dup2, n - 1))
          dup1
      self
  withNaturals do
    (<<) lib.base.data.List.reverse lib.base.data.Stream.toList do
      withInitialValue Map.empty do
        root = ask
        term = aux lam
        emit (SNode "λ" term root root)

Representation.iso.lambdaToText : Lambda Text → Text
Representation.iso.lambdaToText = cases
  Var t → t
  App f x →
    "("
      Text.++ Representation.iso.lambdaToText f
      Text.++ " "
      Text.++ Representation.iso.lambdaToText x
      Text.++ ")"
```

```
  Abs t u →
    "λ" Text.++ t Text.++ " " Text.++ Representation.iso.lambdaToText u

Representation.iso.staticNodesToINet : [SNode t u] →{INet t u} ()
Representation.iso.staticNodesToINet snodes =
  tryLink name addr = match Map.get name Store.get with
    Some p → INet.link p addr
    None   → Store.modify (Map.insert name addr)
  withInitialValue Map.empty do
    lib.base.abilities.Each.run do
      (SNode k l m r) = base.abilities.Each.each snodes
      n = INet.node k
      tryLink l (n => Port.Left)
      tryLink m (n => Main)
      tryLink r (n => Port.Right)
      ()

Representation.iso.staticNodesToText :
  (i1 →{g1} Text) → (i →{g} Text) → [SNode i1 i] →{g1, g} Text
Representation.iso.staticNodesToText f g =
  use Text ++
  lib.base.Text.join "\n"
    << (lib.base.data.List.map cases
         SNode k l m r → "<" ++ f k ++ "> " ++ g l ++ " " ++ g m ++ " " ++ g r)

Representation.iso.textToLambda : Text →{Abort} Lambda Text
Representation.iso.textToLambda t =
  use Char ==
  use Text uncons
  expect i t = match uncons t with
    None             → abort
    Some (c, rest) → if c == i then rest else abort
  identifier t = match uncons t with
    None              → ("", t)
    Some (?\s, rest) → ("", t)
    Some (?), rest)  → ("", t)
    Some (i, rest)   →
      (dent, rest') = identifier rest
      (Text.cons i dent, rest')
  term t = match uncons t with
    None              → abort
    Some (?λ, rest) →
      (x, r1) = identifier rest
      r2 = expect ?\s r1
      let
        (u, r3) = term r2
```

```
        (Abs x u, r3)
    Some (?(, rest) →
      (t1, r1) = term rest
      r2 = expect ?\s r1
      let
        (t2, r3) = term r2
        r4 = expect ?) r3
        (App t1 t2, r4)
    _               → first Var (identifier t)
  match term t with
    (lam, "")    → lam
    (lam, rest) → lam

Representation.iso.textToStaticNodes :
  (Text →{g1} o) → (Text →{g} t) → [Text] →{g1, g, Abort} [
   SNode o t]
Representation.iso.textToStaticNodes f g =
  use Pattern capture
  use patterns char
  pat =
    Pattern.join
      [ literal "<"
      , capture (many (notChars ">"))
      , literal ">"
      , many (char whitespace)
      , capture (many (notChars " "))
      , many (char whitespace)
      , capture (many (notChars " "))
      , many (char whitespace)
      , capture (many (notChars " "))
      , many (char whitespace)
      , eof
      ]
  lib.base.data.List.map
    (Pattern.run pat >> (cases
      Some ([k, l, m, r], "") → SNode (f k) (g l) (g m) (g r)
      x                       → abort))

Representation.staticNodesToDotVis : Text → [SNode Text Nat] →
   Text
Representation.staticNodesToDotVis name nodes =
  withInitialValue Map.empty do
    DotVis.toText Digraph name do
      use Nat ==
      use Port Left Right
      use Text ++
      setNodeAttribute "shape" "triangle"
      lib.base.abilities.Each.run do
        (SNode k l m r) = base.abilities.Each.each nodes
```

```
        label = if m == 0 then "" else k
        u = node.simple label
        dir = cases
          Main   → "n"
          Left   → "sw"
          Right  → "se"
        (::) d = cases NodeRef x → NodeRef (x ++ ":" ++ dir d)
        doEdge name portU =
          match Map.get name Store.get with
            None  → Store.modify (Map.insert name (u => portU))
            Some (MkPtr v portV) →
              dotVis.edge (portU :: u) (portV :: v) do _attribute "dir" "none"
        doEdge l Left
        doEdge m Main
        doEdge r Right
```

Listing 3: Représentations et conversions

```
eval : Text →{Abort} (Nat, Text)
eval = textToLambda >> evalLambda >> Tuple.mapRight lambdaToText


evalLambda : Lambda Text →{Abort} (Nat, Lambda Text)
evalLambda lam =
  throwToAbort = Optional.toAbort << Throw.toOptional
  throwToAbort do
    withNaturals do
      asNatMap do
        snodes = throwToAbort do lambdaToStaticNodes lam
        staticNodesToINet snodes
        rwts = !INet.reduce
        red = Ask.map! (Char.toText << natToLetter) INetToLambda
        (rwts, red)
```

Listing 4: Évaluation de termes

```
ability utils.Counter where increment : {utils.Counter} ()

utils.Counter.result : '{g, Counter} () →{g} Nat
utils.Counter.result requests =
  use Nat +
  impl count = cases
    { pure }                          →
      pure
      count
    { Counter.increment → resume } → handle !resume with impl (count + 1)
  handle !requests with impl 0

```

```
utils.lettersToNat : Text ->{Abort} Nat
utils.lettersToNat = Text.head >> Optional.toAbort >> letterToNat

utils.letterToNat : Char -> Nat
utils.letterToNat =
  use Nat -
  Char.toNat >> (n -> n - 97)

utils.natToLetter : Nat ->{Abort} Char
utils.natToLetter =
  use Nat +
  (n -> n + 97) >> Char.fromNat >> Optional.toAbort

test> utils.natToLetter.tests.a =
  use Char ==
  lib.base.test.check ((Abort.toBug do natToLetter 0) == ?a)

utils.natToLetters : Nat ->{Abort} Text
utils.natToLetters = natToLetter >> Char.toText

test> utils.tests.isoNatLetter = lib.base.test.verify do
  c = !base.abilities.Random.char.ascii.lower
  ensure (Some c === (toOptional! do natToLetter (letterToNat c)))

utils.withNaturals : '{g, Ask Nat} t ->{g} t
utils.withNaturals x = withInitialValue 0 do
  next = do
    use Nat +
    n = Store.get
    Store.put (n + 1)
    n
  provide' next x
```

Listing 5: Fonctions utiles

A.2 Code Haskell - Machine virtuelle et performance

```haskell
module Types where

import Control.DeepSeq
import Data.Data
import GHC.Conc
import GHC.Generics
import GHC.TypeLits
import System.Random (randomIO)

type Identifier = Int

newtype NodeRef = MkRef {getRef :: TVar Node}
  deriving (Eq, Typeable)

instance NFData NodeRef where rnf x = seq x ()

-- NOTE(Maxime): unlawful
instance Data NodeRef where
  dataTypeOf _ = mkIntType "NodeRef"
  toConstr a = mkConstr (dataTypeOf a) "NodeRef" [] Data.Data.Prefix
  gunfold _ _ _ = undefined

newNodeRef :: Node -> STM NodeRef
newNodeRef = fmap MkRef . newTVar

newNodeRefIO :: Node -> IO NodeRef
newNodeRefIO = fmap MkRef . newTVarIO

readNodeRef :: NodeRef -> STM Node
readNodeRef = readTVar . getRef

readNodeRefIO :: NodeRef -> IO Node
readNodeRefIO = readTVarIO . getRef

writeNodeRef :: NodeRef -> Node -> STM ()
writeNodeRef = writeTVar . getRef

data Node
  = Superposition Identifier (NodeRef, NodeRef)
  | Duplication Identifier NodeRef (NodeRef, NodeRef)
  | Duplicated NodeRef
  | IntegerValue Int
  | Lambda NodeRef NodeRef
  | Variable (Maybe NodeRef)
  | Application NodeRef NodeRef
  | Constructor Identifier [NodeRef]
```

```
  | Operator Char NodeRef NodeRef
  deriving (Eq, Generic, Data, NFData)

showNode :: Node → String
showNode = show . toConstr

createDup :: Identifier → NodeRef → STM (NodeRef, NodeRef,
    NodeRef)
createDup ι α = do
  δ₁ ← newNodeRef (Variable Nothing)
  δ₂ ← newNodeRef (Variable Nothing)
  ρ ← newNodeRef (Duplication ι α (δ₁, δ₂))
  writeNodeRef δ₁ (Duplicated ρ)
  writeNodeRef δ₂ (Duplicated ρ)
  pure (ρ, δ₁, δ₂)

duplicationOf :: Node → IO (NodeRef, NodeRef, NodeRef)
duplicationOf ν = do
  α ← newNodeRefIO ν
  ι ← randomIO
  atomically do createDup ι α

nDuplicates :: Nat → NodeRef → IO [NodeRef]
nDuplicates 0 ____ = pure []
nDuplicates 1 node = pure [node]
nDuplicates n node = do
  ι ← randomIO
  (latestClone : rest) ← nDuplicates (n - 1) node
  (_, δ₁, δ₂) ← atomically do createDup ι latestClone
  pure (δ₁ : δ₂ : rest)

lambdaHelper :: (NodeRef → STM NodeRef) → STM NodeRef
lambdaHelper body = do
  α ← newNodeRef (Variable Nothing)
  ν ← body α
  newNodeRef (Lambda α ν)
```

Listing 6: Définitions et types

```
module Parser
  ( startScope,
    expr,
    pattern,
    Parser,
  )
where

import Control.Monad
import Control.Monad.Trans
import Data.Char
```

```haskell
import Data.Foldable
import Data.Functor
import Data.Hashable
import qualified Data.IntMap as IntMap
import Data.Map
import GHC.Conc
import GHC.Generics
import Runtime (Patterns)
import Text.Parsec
import Types

data Scope = Scope {scope :: !(Map String NodeRef), iotas :: [
    Identifier], patterns :: [String]}
  deriving (Generic)

startScope :: Scope
startScope = Scope mempty [0 ..] []

modifyScope :: (Map String NodeRef → Map String NodeRef) → Scope
     → Scope
modifyScope f s = s {scope = f (scope s)}

modifyIotas :: ([Identifier] → [Identifier]) → Scope → Scope
modifyIotas f s = s {iotas = f (iotas s)}

modifyPatterns :: ([String] → [String]) → Scope → Scope
modifyPatterns f s = s {patterns = f (patterns s)}

type Parser = ParsecT String Scope STM

identifierChars :: [Char]
identifierChars = ['a' .. 'z']

(.:) :: (b → c) → (a1 → a2 → b) → a1 → a2 → c
(.:) = (.) . (.)

constructorName :: Parser Int
constructorName = hash .: (:) <$> oneOf (toUpper <$>
    identifierChars) <*> many (oneOf identifierChars)

integer :: Parser Node
integer = try do
  spaces
  i ← read @Int <$> many1 (oneOf ['0' .. '9'])
  spaces
  pure (IntegerValue i)

expr :: Parser NodeRef
expr = expr'List <|> expr'
```

```
  where
    letParser = try do
      spaces
      void (string "let")
      spaces
      name ← many1 (oneOf identifierChars)
      spaces
      void (string "=")
      spaces
      value ← expr
      spaces
      modifyState (modifyScope (insert name value))
      void (char ',')
      spaces
      expr
    dupParser = try do
      spaces
      void (string "dup")
      spaces
      name1 ← many1 (oneOf identifierChars)
      spaces
      name2 ← many1 (oneOf identifierChars)
      spaces
      void (string "=")
      spaces
      value ← expr
      spaces
      iota ← getState <&> iotas <&> head
      modifyState (modifyIotas tail)
      (_, δ1, δ2) ← lift (createDup iota value)
      modifyState (modifyScope (insert name2 δ2 . insert name1 δ1)
 )
      void (char ',')
      spaces
      expr
    identifier = try do
      spaces
      name ← many1 (oneOf identifierChars)
      spaces
      getState <&> scope <&> (! name)
    expr'List = try do
      spaces
      (x : xs) ← (:) <$> expr' <*> many1 expr'
      spaces
      foldlM ((lift . newNodeRef) .: Application) x xs
    constructor = try do
      spaces
      name ← constructorName
      spaces
```

```
      arguments ← many expr'
      lift (newNodeRef (Constructor name arguments))
    operator = try do
      spaces
      op ← oneOf "+-*/%"
      spaces
      lhs ← expr'
      spaces
      rhs ← expr'
      spaces
      lift (newNodeRef (Operator op lhs rhs))
    lambda = try do
      spaces
      void (oneOf "\\λ")
      argname ← many1 (oneOf identifierChars)
      spaces
      arg ← lift (newNodeRef (Variable Nothing))
      modifyState (modifyScope (insert argname arg))
      body ← expr
      spaces
      lift (newNodeRef (Lambda arg body))
    integer' = lift . newNodeRef =<< integer
    exprParen = spaces *> char '(' *> spaces *> expr <* spaces <* char ')' <* spaces
    expr' =
      exprParen
        <|> integer'
        <|> letParser
        <|> dupParser
        <|> constructor
        <|> operator
        <|> lambda
        <|> identifier

pattern :: Parser Patterns
pattern =
  Data.Foldable.foldl' (flip $ uncurry IntMap.insert) mempty
    <$> patternParser
    `sepBy` char '.'
  where
    patternParser :: Parser (Int, [([Node], [NodeRef] → STM NodeRef)])
    patternParser = do
      spaces
      name ← constructorName
      spaces
      rest ← flip sepBy (char ',') do
        modifyState (modifyPatterns (const []))
        singleConstructor
```

```haskell
    pure (name, rest)

  singleConstructor = do
    spaces
    entries ← many entry
    spaces
    void (string "→")
    spaces
    -- Saves input to feed to the expr parser *later*
    input ← getInput
    -- Simulates running the parser but discards the result, preventing failure
    _body ← expr
    currentPatterns ← getState <&> patterns <&> reverse
    let -- The parser cannot actually fail, as it is ran (successfully) before on the same input
        unwrap (Right x) = x; unwrap _ = error "unwrap"
        bodyF xs =
          let addToScope = (modifyState . modifyScope) .: insert
              -- Injects all the necessary variables (provided later)
              -- in the parsing environment of the body, simulating passing them as arguments
              inject = zipWithM_ addToScope currentPatterns xs
           in unwrap <$> runParserT (inject *> expr) startScope "pattern" input
    pure (entries, bodyF)

  nested = do
    spaces *> optional (char '(') *> spaces
    name ← constructorName
    spaces
    rest ← many entry >>= lift . traverse newNodeRef
    spaces *>optional (char ')') *> spaces
    pure (Constructor name rest)

  entry =
    try $
      nested
        <|> (integer <* modifyState (modifyPatterns ("" :)))
        <|> do
          spaces
          name ← many1 (oneOf identifierChars)
          spaces
          modifyState (modifyPatterns (name :))
          pure (Variable Nothing)
```

Listing 7: Parser de Exal

```haskell
module Runtime where

import Control.Applicative
import Control.Monad
import Data.Bitraversable
import Data.Functor
import Data.IntMap.Strict
import Data.Maybe
import GHC.Conc
import System.Random (randomIO)
import Types

type Patterns = IntMap [([Node], [NodeRef] → STM NodeRef)]

evaluate :: Patterns → NodeRef → IO Node
evaluate pat = evaluate'
  where
    evaluate' root =
      readNodeRefIO root >>= \case
        Variable (Just α) → evaluate' α
        Duplicated ρ → do
          Duplication ι v (δ1, δ2) ← readNodeRefIO ρ
          β ← evaluate' v
          atomically do writeNodeRef v β
          unless (root `elem` [δ1, δ2]) (error "INCOHERENT")
          case β of
            Constructor μ xs →
              atomically do
                (_, δ1s, δ2s) ← unzip3 <$> traverse (createDup ι) xs
                writeNodeRef δ1 (Constructor μ δ1s)
                writeNodeRef δ2 (Constructor μ δ2s)
                pure (δ1s <> δ2s)
              *> evaluate' root
            Lambda arg body →
              atomically do
                arg'1 ← newNodeRef (Variable Nothing)
                arg'2 ← newNodeRef (Variable Nothing)
                (_, body'1, body'2) ← createDup ι body
                σ ← newNodeRef (Superposition ι (arg'1, arg'2))
                writeNodeRef arg (Variable (Just σ))
                writeNodeRef δ1 (Lambda arg'1 body'1)
                writeNodeRef δ2 (Lambda arg'2 body'2)
                *> evaluate' root
            Superposition ι' (σ1, σ2)
              | ι == ι' → do
                  atomically do
                    writeNodeRef δ1 =<< readNodeRef σ1
                    writeNodeRef δ2 =<< readNodeRef σ2
```

```haskell
                *> evaluate' root
          | otherwise → do
              (ι1, ι2) ← bisequence (randomIO, randomIO)
              atomically do
                (_, δ1σ1, δ2σ1) ← createDup ι1 σ1
                (_, δ1σ2, δ2σ2) ← createDup ι2 σ2
                writeNodeRef δ1 (Superposition ι1 (δ1σ1, δ1σ2))
                writeNodeRef δ2 (Superposition ι2 (δ2σ1, δ2σ2))
              evaluate' root
        n@IntegerValue {} → do
          atomically do
            writeNodeRef δ1 n
            writeNodeRef δ2 n
            *> evaluate' root
        -- NOTE(Maxime): already in nf
        _ → error "invariant broken"
    Application φ α →
      readNodeRefIO φ >>= \case
        Lambda arg body →
          atomically do
            writeNodeRef arg (Variable (Just α))
            *> evaluate' body
        Superposition ι (σ1, σ2) →
          atomically do
            (_, α1, α2) ← createDup ι α
            σ1' ← newNodeRef (Application σ1 α1)
            σ2' ← newNodeRef (Application σ2 α2)
            writeNodeRef root (Superposition ι (σ1', σ2'))
            *> evaluate' root
        f → do
          ψ ← evaluate' φ
          atomically do writeNodeRef φ ψ
          when (f == ψ) (error ("impossible to evaluate' " <> showNode f))
          evaluate' root
    Operator c x y
      | c `elem` "+-*/%" → do
          Just op ←
            pure $ Prelude.lookup c [('+', (+)), ('-', (-)), ('*', (*)), ('/', quot), ('%', rem)]
          (,) <$> evaluate' x <*> evaluate' y >>= \case
            (IntegerValue a, IntegerValue b) → pure (IntegerValue (a `op` b))
            _ → error "called operator on non-integers"
    Operator '=' x y → do
      (,) <$> evaluate' x <*> evaluate' y >>= \case
        (IntegerValue a, IntegerValue b) →
```

```haskell
              atomically $
                readNodeRef
                  =<< if a == b
                    then lambdaHelper \t → lambdaHelper \_ →
   pure t
                    else lambdaHelper \_ → lambdaHelper pure
            _ → error "called operator on non-integers"
        Constructor ι xs
          | ι `member` pat →
              do
                let matchAndGenerate (ys, pattern) =
                      fmap pattern . concatJust
                        <$> zipWithM matches xs ys
                newRef ←
                  fmap (head . catMaybes)
                    . mapM matchAndGenerate
                    $ pat ! ι
                atomically (writeNodeRef root =<< readNodeRef =<<
   newRef)
                *> evaluate' root
        node → pure node

    concatJust = Prelude.foldl @[] (liftA2 @Maybe (++)) (Just [])
    matches x (Variable Nothing) = pure (Just [x])
    matches x' (Constructor υ ys) = do
      evaluate' x' >>= \case
        Constructor τ xs'
          | υ == τ →
              fmap concatJust
                . zipWithM matches xs'
                =<< traverse evaluate' ys
        _ → pure Nothing
    matches x' y = do
      x ← evaluate' x'
      xr ← newNodeRefIO x
      pure (guard (x == y) $> [xr])
```

Listing 8: Interprète

```haskell
module Tests where

import Data.Bits
import Data.Foldable
import Data.Functor
import Data.IntMap.Strict
import GHC.Conc
import GHC.TypeLits
import Parser
import Runtime (Patterns, evaluate)
import System.Random (randomIO)
```

```haskell
import Test.QuickCheck
import Test.QuickCheck.Monadic
import Text.Parsec (eof, runParserT)
import Types
import Test.Tasty.Bench

prop_vie_est_belle :: Bool
prop_vie_est_belle = True

prop_id_on_int :: Int → Property
prop_id_on_int i = monadicIO $ run do
  let expected = IntegerValue i
  value ← newNodeRefIO expected
  lambda ← atomically do lambdaHelper pure
  root ← newNodeRefIO (Application lambda value)
  result ← evaluate mempty root
  pure (result == expected)

prop_dup_id :: Int → Property
prop_dup_id i = monadicIO $ run do
  let expected = IntegerValue i
  input ← newNodeRefIO expected
  lambda ← atomically do lambdaHelper pure
  (_, clone1, _) ← atomically do createDup 0 lambda
  root ← newNodeRefIO (Application clone1 input)
  result ← evaluate mempty root
  pure (result == expected)

prop_dup_cons :: Identifier → Property
prop_dup_cons i = monadicIO $ run do
  lab ← randomIO
  let expected = Constructor i []
  input ← newNodeRefIO expected
  (_, out1, out2) ← atomically do createDup lab input
  res1 ← evaluate mempty out1
  res2 ← evaluate mempty out2
  pure (expected == res1 && expected == res2)

prop_not :: NodeRef → Int → IO Bool
prop_not f p = do
  dummie1 ← newNodeRefIO (IntegerValue 0)
  dummie2 ← newNodeRefIO (IntegerValue 1)
  partial ← newNodeRefIO (Application f dummie1)
  root ← newNodeRefIO (Application partial dummie2)
  result ← evaluate mempty root
  pure (result == IntegerValue p)

prop_not_composition_naive :: Nat → IO Node
prop_not_composition_naive n = do
```

```haskell
  true ← atomically do lambdaHelper \t → lambdaHelper \_ → pure t
  notF ← atomically do
    lambdaHelper \p → lambdaHelper \t → lambdaHelper \f → do
      partial ← newNodeRef (Application p f)
      newNodeRef (Application partial t)
  nots ← nDuplicates n notF
  result ← atomically do foldlM ((newNodeRef .) . flip Application) true nots
  evaluate mempty result

prop_not_composition :: Nat → IO Node
prop_not_composition n = do
  true ← atomically do lambdaHelper \t → lambdaHelper \_ → pure t
  notF ← atomically do
    lambdaHelper \p → lambdaHelper \t → lambdaHelper \f → do
      partial ← newNodeRef (Application p f)
      newNodeRef (Application partial t)
  let ff =
        [ ([IntegerValue 0, Variable Nothing], pure . (!! 1)),
          ( [Variable Nothing, Variable Nothing],
            \case
              [m, f] → lambdaHelper \x → do
                m' ← newNodeRef . Operator '-' m =<< newNodeRef (IntegerValue 1)
                φ ← newNodeRef (Constructor 0x0 [m', f])
                (_, φ1, φ2) ← createDup 0 φ
                partial ← newNodeRef (Application φ1 x)
                newNodeRef (Application φ2 partial)
              _ → undefined
          )
        ]
  m ← newNodeRefIO (IntegerValue (fromEnum n))
  finalF ← newNodeRefIO (Constructor 0x0 [m, notF])
  result ← newNodeRefIO (Application finalF true)
  evaluate (singleton 0x0 ff) result

prop_op :: Int → Int → Property
prop_op a' b' = monadicIO $ run do
  a ← newNodeRefIO (IntegerValue a')
  b ← newNodeRefIO (IntegerValue b')
  (_, a1, a2) ← atomically do createDup 0 a
  partial ← newNodeRefIO (Operator '+' a1 b)
  root' ← newNodeRefIO (Operator '*' partial a2)
  (_, root, _) ← atomically do createDup 1 root'
  result ← evaluate mempty root
  pure (result == IntegerValue ((a' + b') * a'))
```

```
prop_fib :: Nat → Property
prop_fib i = monadicIO $ run do
  let fibName = 0x0
      fibF =
        [ ([IntegerValue 0], const (newNodeRef (IntegerValue 1))),
          ([IntegerValue 1], const (newNodeRef (IntegerValue 1))),
          ( [Variable Nothing],
            \(head → n) → do
              (_, n1, n2) ← createDup 0x1 n
              n1' ←
                newNodeRef . Operator '-' n1
                  =<< newNodeRef (IntegerValue 1)
              n2' ←
                newNodeRef . Operator '-' n2
                  =<< newNodeRef (IntegerValue 2)
              a ← newNodeRef (Constructor fibName [n1'])
              b ← newNodeRef (Constructor fibName [n2'])
              newNodeRef (Operator '+' a b)
          )
        ]
  iNode ← newNodeRefIO (IntegerValue (fromEnum i))
  root ← newNodeRefIO (Constructor fibName [iNode])
  result ← evaluate (singleton fibName fibF) root
  let expected = IntegerValue (fib (fromEnum i))
  pure (result == expected)
  where
    fib = (fibs !!)
    fibs = 1 : scanl (+) 1 fibs

-- BigInts
bigIntPresets :: Patterns
bigIntPresets =
  let any' = Variable Nothing
   in fromList
        [ -- End
          (0x0, [([], const do lambdaHelper \e → lambdaHelper \_ → lambdaHelper (const (pure e)))]),
          -- B0
          (0x1, [([any'], \(head → p) → lambdaHelper \_ → lambdaHelper \o → lambdaHelper (const do newNodeRef (Application o p)))]),
          -- B1
          (0x2, [([any'], \(head → p) → lambdaHelper \_ → lambdaHelper \_ → lambdaHelper \i → do newNodeRef (Application i p))]),
          -- Inc
          ( 0x3,
            [ ( [any'],
                \(head → n) → lambdaHelper \ex → lambdaHelper \
```

```
ox → lambdaHelper \ix → do
                  part1 ← newNodeRef (Application n ex)
                  part2 ← newNodeRef (Application part1 ix)
                  i ← lambdaHelper \p → do
                    ip ← newNodeRef (Constructor 0x3 [p])
                    newNodeRef (Application ox ip)
                  newNodeRef (Application part2 i)
              )
            ]
          ),
          -- App
          ( 0x4,
            [ ( [any', any', any'],
                \case
                  [n, f, x] → do
                    e ← lambdaHelper \_ → lambdaHelper pure
                    let φ h = lambdaHelper \z → do
                          (_, f1, f2) ← createDup 0x4 h
                          part ← newNodeRef (Application f1 z)
                          newNodeRef (Application f2 part)
                    o ← lambdaHelper \p → lambdaHelper \g → lambdaHelper \y → do
                      φ1 ← φ g
                      newNodeRef (Constructor 0x4 [p, φ1, y])
                    i ← lambdaHelper \p → lambdaHelper \g → lambdaHelper \y → do
                      (_, g1, g2) ← createDup 0x4 g
                      φ1 ← φ g1
                      gy ← newNodeRef (Application g2 y)
                      newNodeRef (Constructor 0x4 [p, φ1, gy])
                    newNodeRef (Application n e)
                      >>= newNodeRef . flip Application o
                      >>= newNodeRef . flip Application i
                      >>= newNodeRef . flip Application f
                      >>= newNodeRef . flip Application x
                  _ → undefined
              )
            ]
          ),
          -- Add
          ( 0x5,
            [ ( [any', any'],
                \case
                  [a, b] → do
                    inc ← lambdaHelper \x → newNodeRef (Constructor 0x3 [x])
                    newNodeRef (Constructor 0x4 [a, inc, b])
                  _ → undefined
              )
```

```
          ]
        ),
        -- FromInt
        ( 0x6,
          [ ([IntegerValue 0, Variable Nothing], const (newNodeRef (Constructor 0x0 []))),
            ( [any', any'],
              \case
                [s, i] → do
                  one ← newNodeRef (IntegerValue 1)
                  (_, two1, two2) ←
                    createDup 0x6
                      =<< newNodeRef (IntegerValue 2)
                  (_, i1, i2) ← createDup 0x6 i
                  s1 ← newNodeRef (Operator '-' s one)
                  bit' ← newNodeRef (Operator '%' i1 two1)
                  rest ← newNodeRef (Operator '/' i2 two2)
                  newNodeRef (Constructor 0x7 [bit', s1, rest])
                _ → undefined
            )
          ]
        ),
        -- FromIntUtil
        ( 0x7,
          [ ( [IntegerValue 0, any', any'],
              \case
                [_, s, i] →
                  newNodeRef . Constructor 0x1 . pure
                    =<< newNodeRef (Constructor 0x6 [s, i])
                _ → undefined
            ),
            ( [IntegerValue 1, any', any'],
              \case
                [_, s, i] →
                  newNodeRef . Constructor 0x2 . pure
                    =<< newNodeRef (Constructor 0x6 [s, i])
                _ → undefined
            ),
            ([any', any', any'], error "here")
          ]
        ),
        -- ToInt
        ( 0x8,
          [ ( [any'],
              \(head → n) → do
                e ← newNodeRef (IntegerValue 0)
                one ← newNodeRef (IntegerValue 1)
                (_, two1, two2) ← createDup 0x8 =<< newNodeRef (IntegerValue 2)
```

```
                  o ← lambdaHelper \p →
                    newNodeRef . Operator '*' two1
                      =<< newNodeRef (Constructor 0x8 [p])
                  i ← lambdaHelper \p →
                    newNodeRef . Operator '+' one
                      =<< newNodeRef . Operator '*' two2
                      =<< newNodeRef (Constructor 0x8 [p])
                  newNodeRef (Application n e)
                    >>= newNodeRef . flip Application o
                    >>= newNodeRef . flip Application i
              )
            ]
          ),
          ( 0x9,
            [ ( [any', any'],
                \case
                  [a, b] → do
                    (_, b1, b') ← createDup 0 b
                    (_, b2, b) ← createDup 1 b'
                    e ← newNodeRef (Constructor 0x0 [])
                    o ← lambdaHelper \p → do
                      rest ← newNodeRef (Constructor 0x9 [p, b1])
                      newNodeRef (Constructor 0x1 [rest])
                    i ← lambdaHelper \p → do
                      rest ← newNodeRef (Constructor 0x9 [p, b2])
                      rest' ← newNodeRef (Constructor 0x1 [rest])
                      newNodeRef (Constructor 0x5 [rest', b])
                    newNodeRef (Application a e)
                      >>= newNodeRef . flip Application o
                      >>= newNodeRef . flip Application i
                  _ → undefined
              )
            ]
          )
        ]

prop_bigint_iso :: Nat → Property
prop_bigint_iso n = monadicIO $ run do
  let expected = IntegerValue (fromEnum n)
  input ← newNodeRefIO expected
  size' ← newNodeRefIO (IntegerValue (finiteBitSize @Int 0))
  scott ← newNodeRefIO (Constructor 0x6 [size', input])
  unscott ← newNodeRefIO (Constructor 0x8 [scott])
  result ← evaluate bigIntPresets unscott
  pure (result == expected)

prop_bigint_add :: Nat → Nat → IO Bool
prop_bigint_add a b = do
  let expected = IntegerValue (fromEnum (a + b))
```

```haskell
  (_, size'1, size'2) ← atomically do
    createDup 0x0 =<< newNodeRef (IntegerValue (finiteBitSize @Int 0 * 2))
  a' ← newNodeRefIO (IntegerValue (fromEnum a))
  b' ← newNodeRefIO (IntegerValue (fromEnum b))
  scottA ← newNodeRefIO (Constructor 0x6 [size'1, a'])
  scottB ← newNodeRefIO (Constructor 0x6 [size'2, b'])
  scottC ← newNodeRefIO (Constructor 0x5 [scottA, scottB])
  root ← newNodeRefIO (Constructor 0x8 [scottC])
  result ← evaluate bigIntPresets root
  pure (result == expected)

prop_bigint_mul :: Nat → Nat → IO Bool
prop_bigint_mul a b = do
  let expected = IntegerValue (fromEnum (a * b))
  (_, size'1, size'2) ← atomically do
    createDup 0x0 =<< newNodeRef (IntegerValue (finiteBitSize @Int 0 * 2))
  a' ← newNodeRefIO (IntegerValue (fromEnum a))
  b' ← newNodeRefIO (IntegerValue (fromEnum b))
  scottA ← newNodeRefIO (Constructor 0x6 [size'1, a'])
  scottB ← newNodeRefIO (Constructor 0x6 [size'2, b'])
  scottC ← newNodeRefIO (Constructor 0x9 [scottA, scottB])
  root ← newNodeRefIO (Constructor 0x8 [scottC])
  result ← evaluate bigIntPresets root
  pure (result == expected)

prop_should_parse :: Parser a → String → IO Bool
prop_should_parse p s =
  atomically (runParserT (p <* eof) startScope "test" s) <&> \case
    Left _ → False
    Right _ → True

prop_parse_and_run :: String → String → IO Node
prop_parse_and_run pat src = do
  pat' ← atomically (runParserT (pattern <* eof) startScope "test" pat)
  src' ← atomically (runParserT (expr <* eof) startScope "test" src)
  case (pat', src') of
    (Right patterns, Right ref) → evaluate patterns ref
    _ → error "No Parse!"

prop_parse_and_check :: String → String → (Node → Bool) → IO Bool
prop_parse_and_check pat src predicate = predicate <$> prop_parse_and_run pat src

prop_list_map :: [Nat] → Property
```

```haskell
prop_list_map xs = not (Prelude.null xs) && length xs < 10 ⟹ monadicIO $ run do
  result ←
    prop_parse_and_run "Head (Cons a as) → a. Map f (Cons a as) → Cons (f a) (Map f as), f (Nil) → Nil" $
      "Head (Map (λx + x 1) (" <> toListString xs <> "))"
  pure (result == IntegerValue (fromEnum (head xs + 1)))
  where
    toListString [] = "Nil"
    toListString (a : as) = "Cons " <> show a <> "(" <> toListString as <> ")"

prop_bench_program :: String → [Int] → String → Benchmark
prop_bench_program name ns p = bgroup name
  [bench (show n) (nfAppIO (prop_parse_and_run p) ("Main " <> show n)) | n ← ns]
```

Listing 9: Définition des tests

```haskell
{-# LANGUAGE ImportQualifiedPost #-}

module Main where

import GHC.TypeLits
import Parser qualified
import Test.QuickCheck
import Test.QuickCheck.Monadic
import Test.Tasty.Bench
import Test.Tasty.QuickCheck
import Tests
import Types

main :: IO ()
main =
  defaultMain
    [ testProperty "tautology" prop_vie_est_belle,
      bgroup
        "parsing"
        [ testProperties "basic expressions" $
            fmap (monadicIO . run . prop_should_parse Parser.expr)
              <$> [ ("num literal", "1"),
                    ("constructor", "Hello 1 2 3"),
                    ("application", "hi 1 2 y"),
                    ("parentheses", "a b (test 1 2 3)"),
                    ("parentheses", "(test 1 2) a"),
                    ("nesting app", "hi (hello world) (And (you nesting))"),
                    ("let binding", "let a = 2, 3"),
                    ("dup binding", "dup a b = 1, 3"),
                    ("lam binding", "λx λy f x y"),
```

```
                    ("prefixbinop", "+ 1 (* 2 (/ 3 4))")
                  ],
        testProperties "patterns" $
          fmap (monadicIO . run . prop_should_parse Parser.pattern)
              <$> [ ("single name", "X → 0"),
                    ("single argument", "X a → a"),
                    ("many arguments", "X a b → a"),
                    ("many cases", "X 0 b → 1, c d → d"),
                    ("many patterns", "X 0 b → 1, c d → d. Y a → a"),
                    ("nested patterns", "X (Y a b) c → c"),
                    ("nested complicated", "Map f (Cons a as) → Cons (f a) (Map f as), f (Nil) → Nil")
                  ],
        testProperties "runs straightforward expressions" $
          let uncurry' f (a, b) = f "" a b
           in fmap (monadicIO . run . uncurry' prop_parse_and_check)
                  <$> [ ("literal", ("3", (== IntegerValue 3))),
                        ("let binding", ("let x = 3, x", (== IntegerValue 3))),
                        ("dup binding", ("dup x y = + 3 3, * x y", (== IntegerValue 36))),
                        ("identity", ("let id = λx x, id 3", (== IntegerValue 3))),
                        ("true", ("dup true t = λx λy x, true 3 4", (== IntegerValue 3))),
                        ( "not",
                          ( "dup true t = λx λy x,"
                              <> "let not = λp λx λy p y x, (not true) 3 4",
                            (== IntegerValue 4)
                          )
                        )
                      ],
        testProperties "call destructor patterns" $
          let uncurry3 f (a, b, c) = f a b c
           in fmap (monadicIO . run . uncurry3 prop_parse_and_check)
                  <$> [ ("literal", ("X → 3", "X", (== IntegerValue 3))),
                        ("simple rewrite", ("F x → + x 1", "F 3", (== IntegerValue 4))),
                        ("many arguments", ("F a b c → + a (* b c)", "F 1 2 3", (== IntegerValue 7))),
                        ("integer arguments", ("F 0 a → a", "F 0 1", (== IntegerValue 1))),
                        ("integer arguments", ("F 0 a → a, a b →
```

```haskell
 a", "F 0 1", (== IntegerValue 1))),
                      ("exact matches", ("F 1 → 0, 0 → 1", "+ (F 1) (F 0)", (== IntegerValue 1))),
                      ("mixed matches", ("F 0 → 0, a → a", "F 18", (== IntegerValue 18))),
                      ("recursive identity", ("F 0 → 0, n → + 1 (F (- n 1))", "F 18", (== IntegerValue 18))),
                      ("fibonacci", ("F 0 → 1, 1 → 1, n → + ( F (- n 1)) (F (- n 2))", "F 8", (== IntegerValue 34))),
                      ("nested destruction", ("F x (C a b) y → b", "F 0 (C 1 2) 3", (== IntegerValue 2)))
                    ]
      ],
    bgroup
      "interpreter correctness"
      [ testProperty "identity" prop_id_on_int,
        bgroup
          "duplication"
          [ testProperty "duplication of the identity" prop_dup_id,
            testProperty "duplication of a constructor" prop_dup_cons
          ],
        bgroup
          "operations"
          [ testProperty "basic operators" prop_op
          ],
        bgroup
          "constructors"
          [ testProperty
              "fib function"
              ( \n → n >= 0 && n <= 20 ⟹ prop_fib (toEnum n)
              ),
            testProperty "list map" (prop_list_map . fmap ( toEnum . abs))
          ]
      ],
    bgroup
      "boolean not (Church)"
      [ testProperty
          "not composition correctness"
          $ property \n → n >= 0 ⟹ monadicIO $ run do
            newF ← newNodeRefIO =<< prop_not_composition_naive (toEnum n)
            prop_not newF (n `mod` 2),
        testProperty
          "not 2^n composition correctness"
          $ property \n → n >= 0 ⟹ monadicIO $ run do
            newF ← newNodeRefIO =<< prop_not_composition (
```

```
toEnum n)
           prop_not newF (fromEnum (n == 0)),
       bgroup
         "no fusion"
         do
           i ← [0, 2 :: Int .. 9]
           let n = 2 ^ i
           pure $
             bench (show n) $
               nfAppIO prop_not_composition_naive n,
       bgroup
         "fusion"
         do
           i ← [0, 8 .. 64]
           pure $ bench (show @Nat (2 ^ i)) $ nfAppIO
prop_not_composition i
     ],
   bgroup
     "bigint (Scott)"
     [ testProperty
         "fromInt & toInt reciprocal correctness"
         \n → n >= 0 ⟹ prop_bigint_iso (toEnum n),
       testProperty
         "addition correctness"
         \a b → (((monadicIO . run) .) . prop_bigint_add) (
toEnum (abs a)) (toEnum (abs b)),
       bgroup
         "addition scaling"
         do
           i ← [0 :: Nat, 3 .. 24]
           let n = 2 ^ i
           pure $
             bench (show @Nat n) $
               nfAppIO (\a → prop_bigint_add a a) n,
       bgroup
         "multiplication scaling"
         do
           i ← [0 :: Nat, 3 .. 24]
           let n = 2 ^ i
           pure $
             bench (show @Nat n) $
               nfAppIO (\a → prop_bigint_mul a a) n
     ],
   prop_bench_program
     "laziness (head of lists)"
     [2 ^ i | i ← [10 :: Int, 20 .. 40]]
     "Nats 0 → Nil, n → Cons n (Nats (- n 1)). Head (Cons a
as) → a. Main n → Head (Nats n)",
   prop_bench_program
```

```
        "laziness (dropping elements)"
        [2 ^ i | i ← [10 :: Int, 20 .. 40]]
        $ "Nats 0 → Nil, n → Cons n (Nats (- n 1)). Head (Cons a as) → a. "
          <> "Drop n Nil → Nil, 0 x → x, n (Cons a as) → Drop (- n 1) as. "
          <> "Main n → Head (Drop 100 (Nats n))"
    ]
```

Listing 10: Fichier principal

A.3 Code Exal - Exécutable sur la machine virtuelle

```
One → λx x.
Succ → λn λsucc λzero (succ (n succ zero)).
Nil →  λcons λnil nil.
Map → λf λl λcons λnil ((l λx λxs (cons (f x) xs)) nil).
Cons → λe λl λcons λnil (cons e) (l cons nil).
Range → λn (n λl (Cons) (One) ((Map) (Succ) l)) (Nil).
```

Listing 11: Naturels Fusion

```
-- Bitstring := E | O Bitstring | I Bitstring
-- RadixTree := Nil | Radix BitString RadixTree RadixTree

-- Prefix extrait le plus long prefixe commun de deux mots
Prefix p (I x) (I y) → Prefix (I p) x y,
Prefix p (O x) (O y) → Prefix (O p) x y.

-- Insertion dans un RadixTree
Insert
    -- on a garanti que v et w n'ont plus de prefixe commun
    (Prefix c v w) (Radix u n0 n1) →
        Insert (Step s)
       (Insert (Step t)
       (Radix c n0 n1)),
    -- on cree les branches necessaires
    (Step (O x)) (Radix c n0 n1) → Radix c (Insert x n0) n1,
    (Step (I x)) (Radix c n0 n1) → Radix c n0 (Insert x n1),
    (Step E) r → r,
    -- cas de base, l'arbre est vide
    w Nil → Radix w Nil Nil,
    -- calcule le prefixe puis insere les restes
    w (Radix u n0 n1) → Insert (Prefix E w u) (Radix u n0 n1).
```

Listing 12: Insertion dans un arbre radix

B Bibliographie

[1] Andrea Asperti and Stefano Guerrini. *The optimal implementation of functional programming languages*. Jan. 1998. ISBN: 0 521 62112 7.

[2] Higher Order Company. *GitHub - HigherOrderCO/HVM: A massively parallel, optimal functional runtime in Rust — github.com*. `https://github.com/HigherOrderCO/HVM`. 2022.

[3] Yves Lafont. "Interaction Combinators". In: *Inf. Comput.* 137 (1997), pp. 69–101. URL: `https://api.semanticscholar.org/CorpusID:16385844`.

[4] Yves Lafont. "Interaction nets". In: *ACM-SIGACT Symposium on Principles of Programming Languages*. 1989. URL: `https://api.semanticscholar.org/CorpusID:263897938`.

[5] John Lamping. *An algorithm for optimal lambda calculus reduction | Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages — dl.acm.org*. `https://dl.acm.org/doi/10.1145/96709.96711`. 1990.

[6] Damiano Mazza. *Observational Equivalence and Full Abstraction in the Symmetric Interaction Combinators — arxiv.org*. `https://arxiv.org/abs/0906.0380`. [Accessed 30-05-2024]. 2009.

[7] Victor Taelin. *GitHub - VictorTaelin/abstract-algorithm: Optimal evaluator of λ-calculus terms. — github.com*. `https://github.com/VictorTaelin/abstract-algorithm`. 2018.