Preuves de correction automatiques des programmes

Yuesubi

Numéro candidat : 00042

Sommaire

Motivation

Le problème des bogues

- Bogues omniprésents
- Conséquences dramatiques
  - Environnements critiques
  - Manipulation d'objets dangereux

Fig. 1. – Explosion de la fusée Ariane 5, à cause d'un dépassement d'entier

Les tests

La fonction $f : x \mapsto x^2$ est-elle égale à id ?

- $0 : f(0) = 0 = \mathrm{id}(0)$ ok
- $1 : f(1) = 1 = \mathrm{id}(1)$ ok
- $2 : f(2) = 4 \neq 2 = \mathrm{id}(2)$ faux

Les tests

La fonction $f : x \mapsto x^2$ est-elle égale à id ?

- $0 : f(0) = 0 = \mathrm{id}(0)$ ok
- $1 : f(1) = 1 = \mathrm{id}(1)$ ok
- $2 : f(2) = 4 \neq 2 = \mathrm{id}(2)$ faux

Problèmes

- Ne couvrent pas toutes les situations
  - Non exhaustif (espace d'entrée infini)

- Complexité de la création des tests
  - Création des valeurs (e.g. graphe)

Les preuves

$$\forall (x, y) \in \mathbb{R}^2, x + y = y + x$$

- Exhaustives : couverture totale
- Les propriété représentent des concepts abstraits
  - Documentation

Planification

Premières idées

Preuves de quoi ?

On code l'addition des entiers naturels :

```
type nat =
  | Zero
  | Succ of nat;;
```

```
let rec add x y =
  match x with
  | Zero -> y
  | Succ x' ->
      Succ (add x' y);;
```

Comment décrire l'effet de la fonction ?

Premières idées

Preuves de quoi ?

On code l’addition des entiers naturels :

```
type nat =
  | Zero
  | Succ of nat;;
```

```
let rec add x y =
  match x with
  | Zero -> y
  | Succ x' ->
      Succ (add x' y);;
```

Comment décrire l’effet de la fonction ?

On énonce plusieurs propriétés :

- $\forall n, \quad \text{add } n \text{ Zero} = n$
- $\forall n, \forall m, \quad \text{add } n\ m = \text{add } m\ n$
- $\forall n, \forall m, \forall p, \quad \text{add } n\ (\text{add } m\ p) = \text{add } (\text{add } n\ m)\ p$
- etc…

Premières idées

Choix du paradigme

Impératif

```
def add(a, b):
  while a != 0:
    a = pred(a) # a = a - 1
    b = Succ(b) # b = b + 1
  return b

add(2, 0)
```

→ recette de cuisine

Premières idées

Choix du paradigme

Impératif

```
def add(a, b):
  while a != 0:
    a = pred(a) # a = a - 1
    b = Succ(b) # b = b + 1
  return b

add(2, 0)
```

→ recette de cuisine

	Mémoire	Instruction
		`add(2, 0)`

↪	`a = 2`
	`b = 0`
		`while a != 0:`

↪	`a = 1`
	`b = 1`
		`while a != 0:`

↪	`a = 0`
	`b = 2`
		`while a != 0:`

↪	`a = 0`
	`b = 2`
		`return b`

Premières idées

Choix du paradigme

Fonctionnel pur

```
(* x + y *)
let rec add x y =
  match x with
  | Zero -> y
  | Succ x' ->
      (* 1 + (x' + y) *)
      Succ (add x' y)

add (Succ Succ Zero) Zero
```

→ formule

Premières idées

Choix du paradigme

Fonctionnel pur

```
(* x + y *)
let rec add x y =
  match x with
  | Zero -> y
  | Succ x' ->
      (* 1 + (x' + y) *)
      Succ (add x' y)
```

```
add (Succ Succ Zero) Zero
```

→ formule

```
   add (Succ Succ Zero) Zero
↪ match Succ Succ Zero with
   | Zero -> Zero
   | Succ x' ->
       Succ (add x' Zero)
↪ Succ (add (Succ Zero)
   Zero)
↪ Succ (match Succ Zero
   with ...)
↪ Succ Succ (add Zero Zero)
↪ Succ Succ (match Zero
   with ...)
↪ Succ Succ Zero
```

Premières idées

Choix du paradigme

Fonctionnel pur

```
(* x + y *)
let rec add x y =
  match x with
  | Zero -> y
  | Succ x' ->
      (* 1 + (x' + y) *)
      Succ (add x' y)
```

```
add (Succ Succ Zero) Zero
```

→ formule

```
  add (Succ Succ Zero) Zero
↪ match Succ Succ Zero with
  | Zero -> Zero
  | Succ x' ->
      Succ (add x' Zero)
↪ Succ (add (Succ Zero)
  Zero)
↪ Succ (match Succ Zero
  with ...)
↪ Succ Succ (add Zero Zero)
↪ Succ Succ (match Zero
  with ...)
↪ Succ Succ Zero
```

→ Langage purement fonctionnel choisi

Premières idées

Pour traiter une infinité de cas utilisation du théorème de récurrence.

- Initialisation : $P(\text{Zero})\ \text{vrai}$
- Hérédité :
  $\forall n : \text{nat}, P(n) \Rightarrow P(\text{Succ}\ n)$

Donc $\forall n \in \mathbb{N}, P(n)$.

Premières idées

Pour traiter une infinité de cas utilisation du théorème de récurrence.

- Initialisation : $P(\text{Zero})$ vrai
- Hérédité :
  $\forall n : \text{nat}, P(n) \Rightarrow P(\text{Succ } n)$

Donc $\forall n \in \mathbb{N}, P(n)$.

```
type nat =
  | Zero
  | Succ of nat;;
```

Premières idées

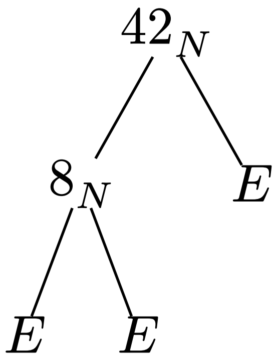
Pour traiter une infinité de cas utilisation du théorème de récurrence.

- Initialisation : $P(\text{Zero})$ vrai
- Hérédité :
  $\forall n : \text{nat}, P(n) \Rightarrow P(\text{Succ } n)$

Donc $\forall n \in \mathbb{N}, P(n)$.

```
type nat =
  | Zero
  | Succ of nat;;
```

```
type 'a tree =
  | E
  | N of 'a
      * 'a tree
      * 'a tree
```



Premières idées

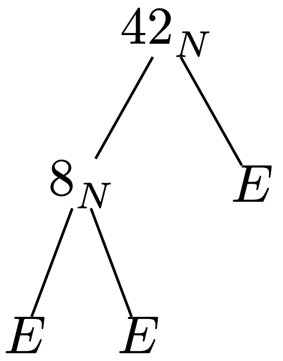
Pour traiter une infinité de cas utilisation du théorème de récurrence.

- Initialisation : $P(\text{Zero})$ vrai
- Hérédité :
  $\forall n : \text{nat}, P(n) \Rightarrow P(\text{Succ } n)$

Donc $\forall n \in \mathbb{N}, P(n)$.

```
type nat =
  | Zero
  | Succ of nat;;
```

```
type 'a tree =
  | E
  | N of 'a
      * 'a tree
      * 'a tree
```



Généralisation

- Vide : $P(E)$
- Noeud :
  $\forall \ell : \alpha, \forall g, d : \alpha \text{ tree},$
  $$\begin{cases} P(g) \\ P(d) \end{cases} \Rightarrow P(N(\ell, g, d))$$

Donc $\forall a : \alpha \text{ tree}, P(a)$.

Premières idées

Problématique

Comment générer des preuves de correction automatiquement dans un langage de programmation total et purement fonctionnel ?

Premières idées

Style des propriété

Égalités sémantiques

E.g. :

- `let y = 42 in y` ≡ `42`
- pour y quelconque,
  `(fun x -> x) y` ≡ `y`

Premières idées

Style des propriété

Égalités sémantiques

E.g. :

- `let y = 42 in y` ≡ `42`
- pour y quelconque, `(fun x -> x) y` ≡ `y`

Quantification

- universelle ∀ (sur les types)
- pas de mélange code / quantification
- pas de ∃

Premières idées

Style des propriété

Égalités sémantiques

E.g. :

- `let y = 42 in y` $\equiv$ `42`
- pour y quelconque, `(fun x -> x) y` $\equiv$ `y`

Quantification

- universelle $\forall$ (sur les types)
- pas de mélange code / quantification
- pas de $\exists$

- Forme générale : $\forall x_1 : \tau_1, \ldots, \forall x_n : \tau_n, E \equiv F$
- E.g. $\forall x : \text{nat}, \forall y : \text{nat}, \text{add } x\ y \equiv \text{add } y\ x$

Premières idées

Exemple preuve voulue

```
type nat =
  | Zero
  | Succ of nat;;

let rec add x y =
  match x with
  | Zero -> y
  | Succ x' ->
      Succ (add x' y);;
```

Propriété :

$$\forall x : \text{nat}, \text{add}\ x\ \text{Zero} \equiv x$$

Premières idées

Exemple preuve voulue

```
type nat =
 | Zero
 | Succ of nat;;

let rec add x y =
 match x with
 | Zero -> y
 | Succ x' ->
   Succ (add x' y);;
```

Propriété :

$$\forall x : \text{nat}, \text{add } x \text{ Zero} \equiv x$$

Preuve :

- Soit x : nat.
- Cas $x = \text{Zero}$:
  - Alors

    $\text{add } x \text{ Zero} \equiv \text{Zero} \equiv x.$
- Cas $\exists x' : \text{nat}, x = \text{Succ } x'$:
  - Soit un tel x'.
  - Alors

    $\text{add } x \text{ Zero}$

    $\equiv \text{Succ } (\text{add } x' \text{ Zero})$

    $\equiv \text{Succ } x'$

    $\equiv x.$

Le langage choisi

Une petite partie du OCaml

OCaml

- est fonctionnel
- est au programme

Une partie

- purement fonctionnelle (pas d'effets de bords)
- petite (moins de travail)

Le langage choisi

Définitions de types

Types inductifs

```
type bool =
  | False
  | True;;

type nat =
  | Zero
  | Succ of nat;;

type 'a list =
  | Nil
  | Cons of 'a * 'a list;;
```

Aucun type au départ

Le langage choisi

Définitions de types

Types inductifs

```
type bool =
  | False
  | True;;

type nat =
  | Zero
  | Succ of nat;;

type 'a list =
  | Nil
  | Cons of 'a * 'a list;;
```

Aucun type au départ

Les valeurs

Faux	Vrai
`False`	`True`

0	2	$n+1$
`Zero`	`Succ (Succ Zero)`	`Succ n`

Le langage choisi

Définitions de types

Types inductifs

```
type bool =
  | False
  | True;;

type nat =
  | Zero
  | Succ of nat;;

type 'a list =
  | Nil
  | Cons of 'a * 'a list;;
```

Aucun type au départ

Les valeurs

Faux	Vrai
`False`	`True`

0	2	$n + 1$
`Zero`	`Succ (Succ Zero)`	`Succ n`

Matchs (disjonctions de cas)

Fonctions & Appels

Déclarations

Le langage choisi

Exemple de code

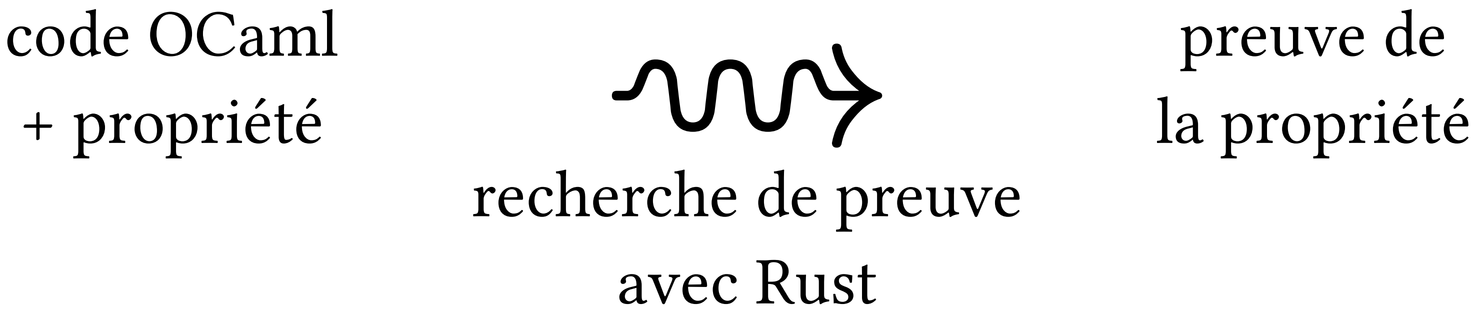
```
type nat =
  | Zero
  | Succ of nat;;

let add = fun x -> fun y ->
  match x with
  | Zero -> y
  | Succ x' -> Succ (add x' y);;

let mult = fun x -> fun y ->
  match x with
  | Zero -> Zero
  | Succ x' -> add y (mult x' y);;
```

Implémentation

Choix du langage

Rust

- types paramétrés complexes
- existence des matchs
- gestion des erreurs aisée



code OCaml
+ propriété

recherche de preuve
avec Rust

preuve de
la propriété

Parseur

Grammaire

$$\begin{aligned} \text{program} &::= \{\,\text{type def}\,\}, \\ &\ \{\,\text{statement}\,\} \end{aligned}$$

$$\begin{aligned} \text{expr} &::= \text{let in} \\ &\ \ \mid\ \ \text{lambda function} \\ &\ \ \mid\ \ \text{function call} \\ &\ \ \mid\ \ \text{match} \\ &\ \ \mid\ \ \text{litteral constr} \\ &\ \ \mid\ \ \text{litteral variable} \end{aligned}$$

$\vdots$

Parseur

Grammaire

program ::= { type def },
{ statement }

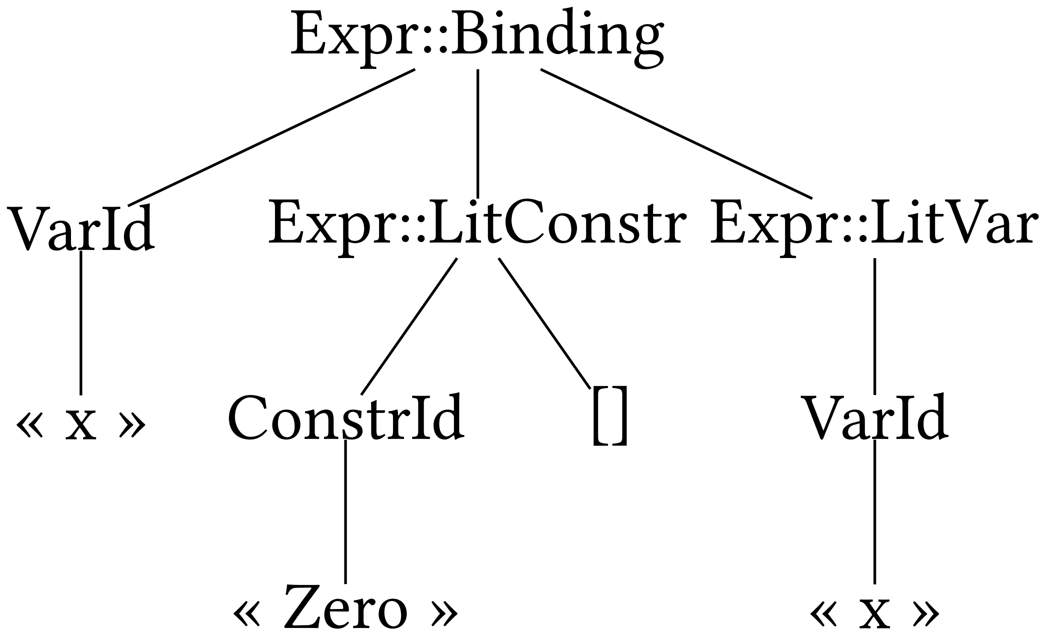
expr ::= let in
| lambda function
| function call
| match
| litteral constr
| litteral variable

⋮

Parseur

```
let x = Zero in x
```

⇩



Réécritures successives

Objectif

Réécritures successives

$$
\dfrac{
  \dfrac{
    \dfrac{}{
      \vdash \boxed{\begin{array}{l}\mathbf{let}\ \text{un} = \textsc{Succ}\ \textsc{Zero}\ \mathbf{in}\\ \textsc{Succ}\ \text{un}\end{array}} \equiv \boxed{\begin{array}{l}\mathbf{let}\ \text{un} = \textsc{Succ}\ \textsc{Zero}\ \mathbf{in}\\ \textsc{Succ}\ \text{un}\end{array}}
    }\;=
  }{
    \vdash \boxed{\begin{array}{l}\mathbf{let}\ \text{un} = \textsc{Succ}\ \textsc{Zero}\ \mathbf{in}\\ \textsc{Succ}\ \text{un}\end{array}} \equiv \boxed{\begin{array}{l}\mathbf{let}\ \text{zero} = \textsc{Zero}\ \mathbf{in}\\ \mathbf{let}\ \text{un} = \textsc{Succ}\ \text{zero}\ \mathbf{in}\\ \textsc{Succ}\ \text{un}\end{array}}
  }\;\text{var}_{\text{éval}}
}{
  \vdash \boxed{\textsc{Succ}\ \textsc{Succ}\ \textsc{Zero}} \equiv \boxed{\begin{array}{l}\mathbf{let}\ \text{zero} = \textsc{Zero}\ \mathbf{in}\\ \mathbf{let}\ \text{un} = \textsc{Succ}\ \text{zero}\ \mathbf{in}\\ \textsc{Succ}\ \text{un}\end{array}}
}\;\text{constr}_{\text{fact}}
$$

Réécritures successives

Indices de Brujin

Créer un alias (si $\mathcal{A}_1$ complexe)

$$\dfrac{\vdash \boxed{\begin{array}{l}\textbf{let}\ \mathrm{a} = \mathcal{A}_1\ \textbf{in}\\ \textsc{Constr}\,(\mathrm{a}, \ldots, \mathcal{A}_n)\end{array}} \equiv \boxed{\ldots}}{\vdash \boxed{\textsc{Constr}\,(\mathcal{A}_1, \ldots, \mathcal{A}_n)} \equiv \boxed{\ldots}}\ \mathrm{constr}_{\mathrm{fact}}$$

Réécritures successives

Indices de Brujin

Créer un alias (si $\mathcal{A}_1$ complexe)

$$\frac{\vdash \boxed{\begin{array}{l}\textbf{let}\ \mathrm{a} = \mathcal{A}_1\ \textbf{in}\\ \textsc{Constr}\,(\mathrm{a}, \ldots, \mathcal{A}_n)\end{array}} \equiv \boxed{\ldots}}{\vdash \boxed{\textsc{Constr}\,(\mathcal{A}_1, \ldots, \mathcal{A}_n)} \equiv \boxed{\ldots}}\,\mathrm{constr}_{\mathrm{fact}}$$

Problème si a déjà défini:

$$\frac{\vdash \boxed{\begin{array}{l}\textbf{let}\ \mathrm{a} = \mathcal{E}\ \textbf{in}\\ \textbf{let}\ \mathrm{a} = \mathcal{A}\ \textbf{in}\\ \textsc{Tuple}\,(\mathrm{a}, \mathrm{a})\end{array}} \equiv \boxed{\ldots}}{\vdash \boxed{\begin{array}{l}\textbf{let}\ \mathrm{a} = \mathcal{E}\ \textbf{in}\\ \textsc{Tuple}\,(\mathcal{A}, \mathrm{a})\end{array}} \equiv \boxed{\ldots}}\,\mathrm{constr}_{\mathrm{fact}}$$

Réécritures successives

Indices de Brujin

Créer un alias (si $\mathcal{A}_1$ complexe)

$$\dfrac{\vdash \boxed{\begin{array}{l}\textbf{let}\ \mathrm{a} = \mathcal{A}_1\ \textbf{in}\\ \textsc{Constr}\,(\mathrm{a}, \ldots, \mathcal{A}_n)\end{array}} \equiv \boxed{\ldots}}{\vdash \boxed{\textsc{Constr}\,(\mathcal{A}_1, \ldots, \mathcal{A}_n)} \equiv \boxed{\ldots}}\,\mathrm{constr}_{\mathrm{fact}}$$

Problème si a déjà défini:

$$\dfrac{\vdash \boxed{\begin{array}{l}\textbf{let}\ \mathrm{a} = \mathcal{E}\ \textbf{in}\\ \textbf{let}\ \mathrm{a} = \mathcal{A}\ \textbf{in}\\ \textsc{Tuple}\,(\mathrm{a}, \mathrm{a})\end{array}} \equiv \boxed{\ldots}}{\vdash \boxed{\begin{array}{l}\textbf{let}\ \mathrm{a} = \mathcal{E}\ \textbf{in}\\ \textsc{Tuple}\,(\mathcal{A}, \mathrm{a})\end{array}} \equiv \boxed{\ldots}}\,\mathrm{constr}_{\mathrm{fact}}$$

→ passage à des id relatifs

$$\begin{array}{c}\textbf{let}\ \mathrm{a} = \mathcal{E}\ \textbf{in}\\ \textbf{let}\ \mathrm{a} = \mathcal{A}\ \textbf{in}\\ \textsc{Tuple}_{42}\,(\mathrm{a}_{\uparrow 1}, \mathrm{a}_{\uparrow 2})\end{array}$$

Arbres de preuves

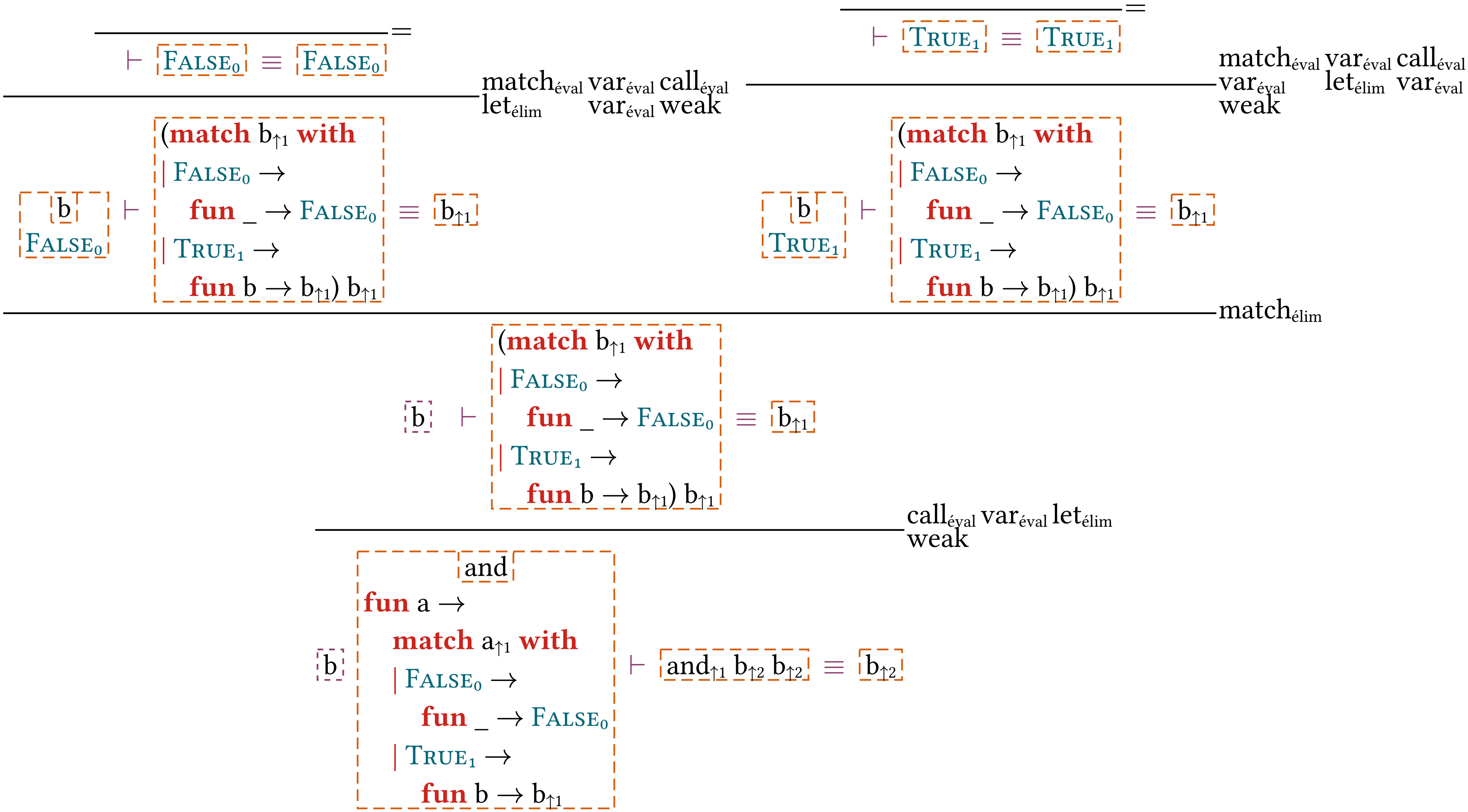
Les matchs

Disjonction de cas pour les `match`

$$\dfrac{\boxed{\begin{array}{c}\boxed{\mathrm{n}}\\ \textsc{Zero}_0\end{array}} \vdash \boxed{\begin{array}{l}\textbf{match}\ \mathrm{n}_{\uparrow 1}\ \textbf{with}\\ \mid \textsc{Zero}_0 \to \ldots\\ \mid \textsc{Succ}_1\ \mathrm{n}' \to \ldots\end{array}} \equiv \boxed{\ldots} \qquad \boxed{\mathrm{n}'}\ \boxed{\begin{array}{c}\boxed{\mathrm{n}}\\ \textsc{Succ}_1\ \mathrm{n}'_{\uparrow 1}\end{array}} \vdash \boxed{\begin{array}{l}\textbf{match}\ \mathrm{n}_{\uparrow 1}\ \textbf{with}\\ \mid \textsc{Zero}_0 \to \ldots\\ \mid \textsc{Succ}_1\ \mathrm{n}' \to \ldots\end{array}} \equiv \boxed{\ldots}}{\boxed{\mathrm{n}} \vdash \boxed{\begin{array}{l}\textbf{match}\ \mathrm{n}_{\uparrow 1}\ \textbf{with}\\ \mid \textsc{Zero}_0 \to \ldots\\ \mid \textsc{Succ}_1\ \mathrm{n}' \to \ldots\end{array}} \equiv \boxed{\ldots}}\ \mathrm{match}_{\text{élim}}$$

Arbres de preuves

Exemple : $\text{and}_{\uparrow 1}\ b_{\uparrow 2}\ b_{\uparrow 2} \equiv b_{\uparrow 2}$



$$\dfrac{}{\vdash \boxed{\textsc{False}_0} \equiv \boxed{\textsc{False}_0}}\ =$$

$$\dfrac{}{\vdash \boxed{\textsc{True}_1} \equiv \boxed{\textsc{True}_1}}\ =$$

match$_{\text{éval}}$ var$_{\text{éval}}$ call$_{\text{éval}}$ let$_{\text{élim}}$ var$_{\text{éval}}$ weak

$$\boxed{\begin{array}{c}\boxed{b}\\ \textsc{False}_0\end{array}} \vdash \boxed{\begin{array}{l}(\textbf{match}\ b_{\uparrow 1}\ \textbf{with}\\ \mid \textsc{False}_0 \to\\ \quad \textbf{fun}\ \_ \to \textsc{False}_0\\ \mid \textsc{True}_1 \to\\ \quad \textbf{fun}\ b \to b_{\uparrow 1})\ b_{\uparrow 1}\end{array}} \equiv \boxed{b_{\uparrow 1}}$$

match$_{\text{éval}}$ var$_{\text{éval}}$ call$_{\text{éval}}$ var$_{\text{éval}}$ let$_{\text{élim}}$ var$_{\text{éval}}$ weak

$$\boxed{\begin{array}{c}\boxed{b}\\ \textsc{True}_1\end{array}} \vdash \boxed{\begin{array}{l}(\textbf{match}\ b_{\uparrow 1}\ \textbf{with}\\ \mid \textsc{False}_0 \to\\ \quad \textbf{fun}\ \_ \to \textsc{False}_0\\ \mid \textsc{True}_1 \to\\ \quad \textbf{fun}\ b \to b_{\uparrow 1})\ b_{\uparrow 1}\end{array}} \equiv \boxed{b_{\uparrow 1}}$$

match$_{\text{élim}}$

$$\boxed{b} \vdash \boxed{\begin{array}{l}(\textbf{match}\ b_{\uparrow 1}\ \textbf{with}\\ \mid \textsc{False}_0 \to\\ \quad \textbf{fun}\ \_ \to \textsc{False}_0\\ \mid \textsc{True}_1 \to\\ \quad \textbf{fun}\ b \to b_{\uparrow 1})\ b_{\uparrow 1}\end{array}} \equiv \boxed{b_{\uparrow 1}}$$

call$_{\text{éval}}$ var$_{\text{éval}}$ let$_{\text{élim}}$ weak

$$\boxed{b}\ \boxed{\begin{array}{l}\qquad\quad \boxed{\text{and}}\\ \textbf{fun}\ a \to\\ \quad \textbf{match}\ a_{\uparrow 1}\ \textbf{with}\\ \quad \mid \textsc{False}_0 \to\\ \qquad \textbf{fun}\ \_ \to \textsc{False}_0\\ \quad \mid \textsc{True}_1 \to\\ \qquad \textbf{fun}\ b \to b_{\uparrow 1}\end{array}} \vdash \boxed{\text{and}_{\uparrow 1}\ b_{\uparrow 2}\ b_{\uparrow 2}} \equiv \boxed{b_{\uparrow 2}}$$

Conclusion

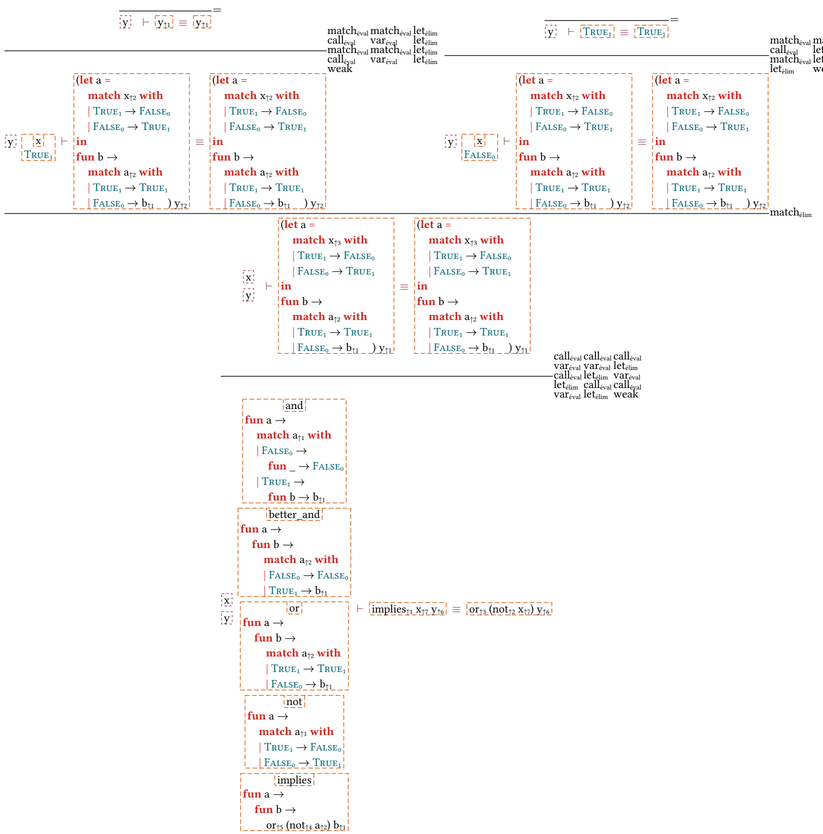
Résultat final

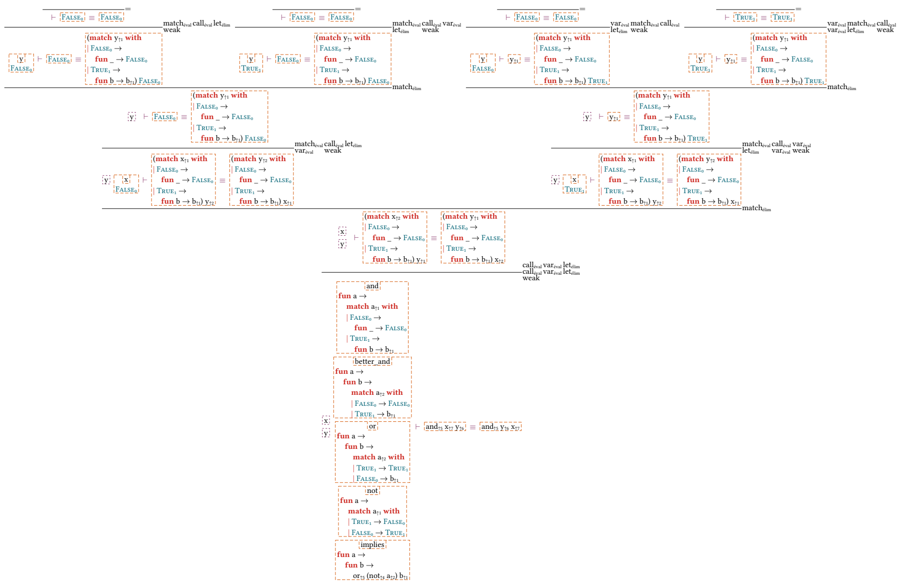
implies x y ≡ or (not x) y

$$(a \Rightarrow b \iff \neg a \lor b)$$

and x y ≡ and y x

$$(a \land b \iff b \land a)$$





→ 8 000 lignes de code en Rust

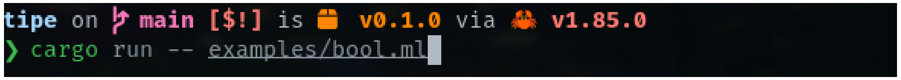
Annexe

Bilan

- ☑ Choisir un langage de programmation
- ☑ Construire un parseur
- ☑ Prouver des égalités sémantiques par réécritures
- ☑ Gestion des disjonction de cas
- ◧ Génération de contre-exemples
  - ☑ Inférence de types
  - ☐ Énumérer et tester des valeurs
- ☐ Induction structurelle.

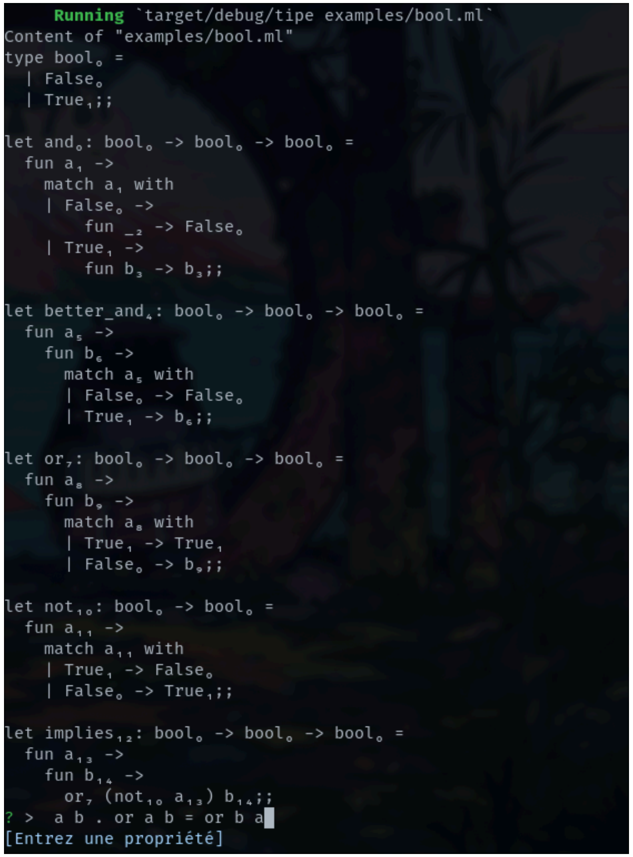
Explication facile, mais implémentation longue

Interface en invite de commandes (CLI)



```
tipe on  main [$!] is 📦 v0.1.0 via 🦀 v1.85.0
❯ cargo run -- examples/bool.ml
```

Fig. 2. – Lancement du programme

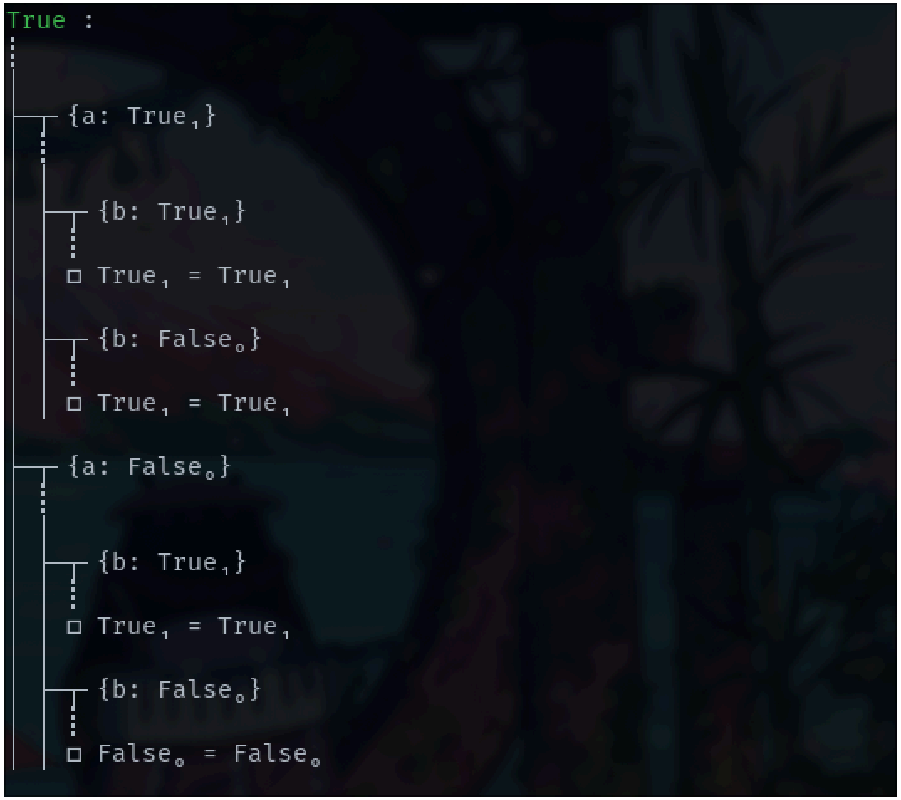


```
     Running `target/debug/tipe examples/bool.ml`
Content of "examples/bool.ml"
type bool₀ =
  | False₀
  | True₁;;

let and₀: bool₀ -> bool₀ -> bool₀ =
  fun a₁ ->
    match a₁ with
    | False₀ ->
        fun _₂ -> False₀
    | True₁ ->
        fun b₃ -> b₃;;

let better_and₄: bool₀ -> bool₀ -> bool₀ =
  fun a₅ ->
    fun b₆ ->
      match a₅ with
      | False₀ -> False₀
      | True₁ -> b₆;;

let or₇: bool₀ -> bool₀ -> bool₀ =
  fun a₈ ->
    fun b₉ ->
      match a₈ with
      | True₁ -> True₁
      | False₀ -> b₉;;

let not₁₀: bool₀ -> bool₀ =
  fun a₁₁ ->
    match a₁₁ with
    | True₁ -> False₀
    | False₀ -> True₁;;

let implies₁₂: bool₀ -> bool₀ -> bool₀ =
  fun a₁₃ ->
    fun b₁₄ ->
      or₇ (not₁₀ a₁₃) b₁₄;;
? >  a b . or a b = or b a
[Entrez une propriété]
```

Fig. 3. – Écrire une propriété

```
   or₇ b₁₆ a₁₅
Inferred type 'bool₀'
("call" ("call" ("lit var" ("leid" "or") ("var meta" 5
```

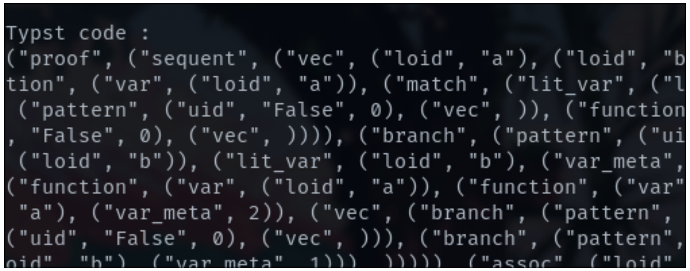
Fig. 4. – Inférence de type



```
True :
┆
├─┬ {a: True₁}
│ ┆
│ ├─┬ {b: True₁}
│ │ ┆
│ │ □ True₁ = True₁
│ │
│ ├─┬ {b: False₀}
│ │ ┆
│ │ □ True₁ = True₁
│
├─┬ {a: False₀}
│ ┆
│ ├─┬ {b: True₁}
│ │ ┆
│ │ □ True₁ = True₁
│ │
│ ├─┬ {b: False₀}
│ │ ┆
│ │ □ False₀ = False₀
```

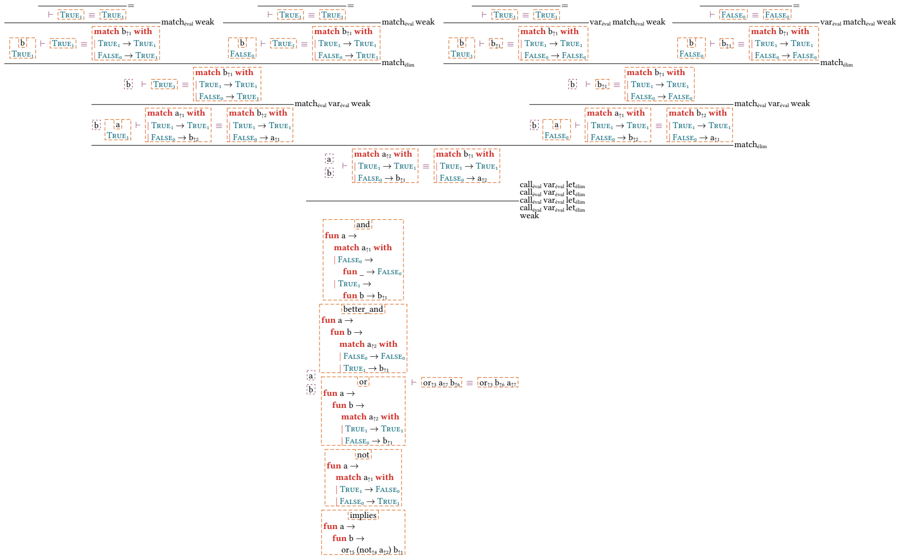
Fig. 5. – Schéma de la preuve

Interface en invite de commandes (CLI)



```
Typst code :
("proof", ("sequent", ("vec", ("loid", "a"), ("loid", "b
tion", ("var", ("loid", "a")), ("match", ("lit_var", ("l
 ("pattern", ("uid", "False", 0), ("vec", )), ("function
, "False", 0), ("vec", )))), ("branch", ("pattern", ("ui
 ("loid", "b")), ("lit_var", ("loid", "b"), ("var_meta",
("function", ("var", ("loid", "a")), ("function", ("var"
 "a"), ("var_meta", 2)), ("vec", ("branch", ("pattern",
("uid", "False", 0), ("vec", ))), ("branch", ("pattern",
oid", "b"), ("var_meta", 1))), ))))), ("assoc", ("loid",
```

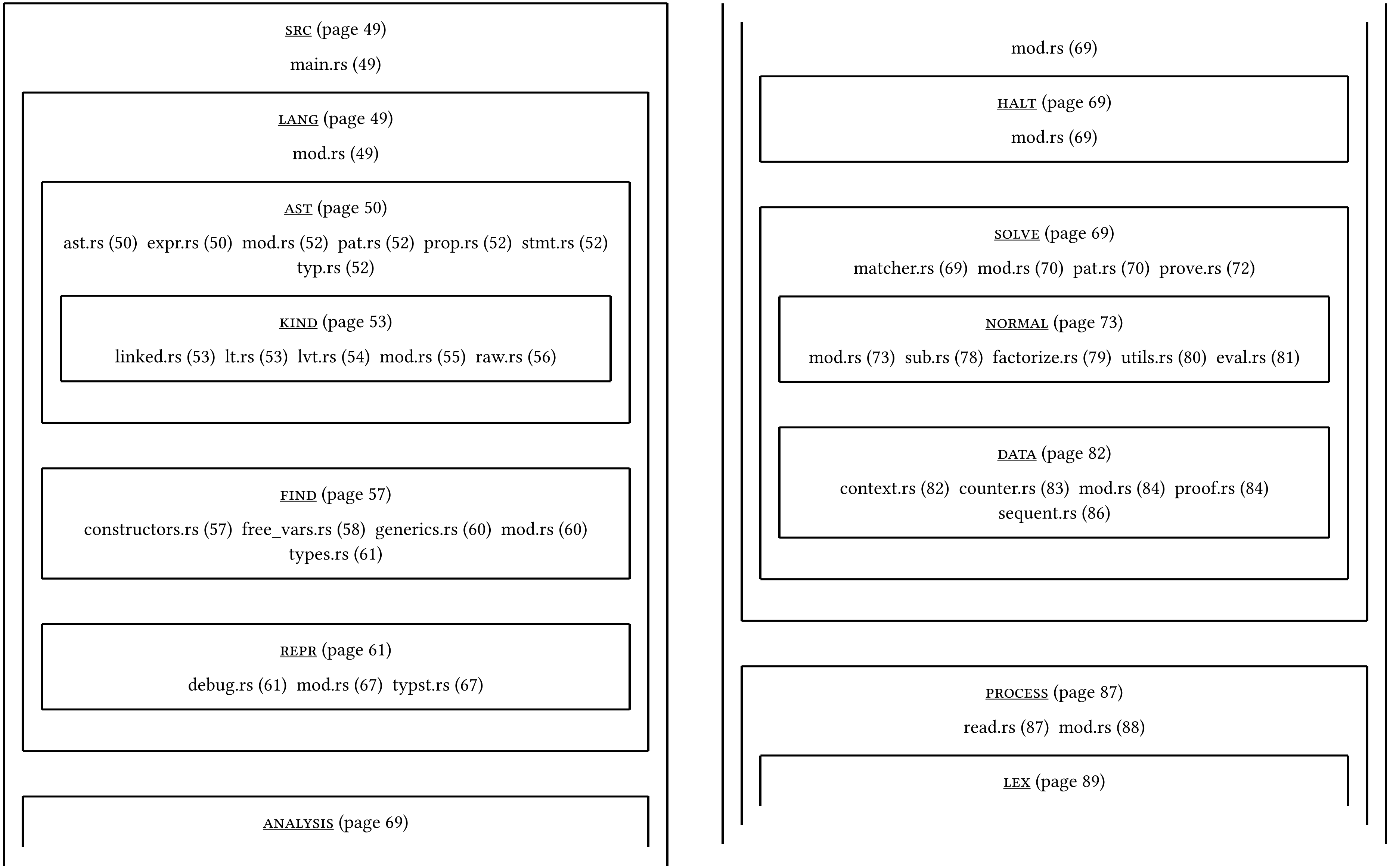
Fig. 6. – Code de représentation en typst



$\vdash$ TRUE$_1$ $\equiv$ TRUE$_1$
match$_{éval}$ weak
var$_{éval}$ match$_{éval}$ weak
$\vdash$ FALSE$_0$ $\equiv$ FALSE$_0$
match $b_{\uparrow 1}$ with
| TRUE$_1$ → TRUE$_1$
| FALSE$_0$ → TRUE$_1$
| FALSE$_0$ → FALSE$_0$
match$_{élim}$
match$_{éval}$ var$_{éval}$ weak
match $a_{\uparrow 1}$ with
match $b_{\uparrow 2}$ with
| FALSE$_0$ → $b_{\uparrow 2}$
| FALSE$_0$ → $a_{\uparrow 1}$
match $a_{\uparrow 2}$ with
| FALSE$_0$ → $b_{\uparrow 1}$
| FALSE$_0$ → $a_{\uparrow 2}$
call$_{éval}$ var$_{éval}$ let$_{élim}$
call$_{éval}$ var$_{éval}$ let$_{élim}$
call$_{éval}$ var$_{éval}$ let$_{élim}$
call$_{éval}$ var$_{éval}$ let$_{élim}$
weak
and
fun a →
match $a_{\uparrow 1}$ with
| FALSE$_0$ →
fun _ → FALSE$_0$
| TRUE$_1$ →
fun b → $b_{\uparrow 1}$
better_and
fun a →
fun b →
match $a_{\uparrow 2}$ with
| FALSE$_0$ → FALSE$_0$
| TRUE$_1$ → $b_{\uparrow 1}$
or
fun a →
fun b →
match $a_{\uparrow 2}$ with
| TRUE$_1$ → TRUE$_1$
| FALSE$_0$ → $b_{\uparrow 1}$
$\vdash$ $or_{\uparrow 3}$ $a_{\uparrow 7}$ $b_{\uparrow 6}$ $\equiv$ $or_{\uparrow 3}$ $b_{\uparrow 6}$ $a_{\uparrow 7}$
not
fun a →
match $a_{\uparrow 1}$ with
| TRUE$_1$ → FALSE$_0$
| FALSE$_0$ → TRUE$_1$
implies
fun a →
fun b →
$or_{\uparrow 5}$ ($not_{\uparrow 4}$ $a_{\uparrow 2}$) $b_{\uparrow 1}$

Fig. 7. – Représentation

Arborescence du code



src (page 49)

main.rs (49)

lang (page 49)

mod.rs (49)

ast (page 50)

ast.rs (50) expr.rs (50) mod.rs (52) pat.rs (52) prop.rs (52) stmt.rs (52) typ.rs (52)

kind (page 53)

linked.rs (53) lt.rs (53) lvt.rs (54) mod.rs (55) raw.rs (56)

find (page 57)

constructors.rs (57) free_vars.rs (58) generics.rs (60) mod.rs (60) types.rs (61)

repr (page 61)

debug.rs (61) mod.rs (67) typst.rs (67)

analysis (page 69)

mod.rs (69)

halt (page 69)

mod.rs (69)

solve (page 69)

matcher.rs (69) mod.rs (70) pat.rs (70) prove.rs (72)

normal (page 73)

mod.rs (73) sub.rs (78) factorize.rs (79) utils.rs (80) eval.rs (81)

data (page 82)

context.rs (82) counter.rs (83) mod.rs (84) proof.rs (84) sequent.rs (86)

process (page 87)

read.rs (87) mod.rs (88)

lex (page 89)

lexer.rs (89) mod.rs (92) token.rs (92)

LINK (page 93)

ast.rs (93) expr.rs (93) mod.rs (94) pat.rs (99) prop.rs (99) stmt.rs (100) typ.rs (100)

PARSE (page 102)

mod.rs (102) prop.rs (103) stmt.rs (104) typ.rs (104) expr.rs (109) pat.rs (111)

TYP (page 113)

ast.rs (113) convert.rs (113) prop.rs (114) stmt.rs (114) typ.rs (115) unify.rs (115) expr.rs (117) mod.rs (119) pat.rs (121)

UTILS (page 122)

id_gen.rs (122) mod.rs (122) scope_stack.rs (123) trace.rs (123)

LOOSE (page 124)

mod.rs (124) offset.rs (124)

LINK (page 126)

ast.rs (126) expr.rs (126) mod.rs (127) pat.rs (128) prop.rs (128) stmt.rs (129) typ.rs (129)

Annexe

Code

Dossier /src/

Fichier /src/main.rs

```rust
mod analysis;
mod lang;
mod process;

use std::io;
use colored::Colorize;
use clap::Parser;
use inquire::Text;

use lang::TypstRepr;
use process::read;


#[derive(clap::Parser)]
struct Cli {
    file_path: std::path::PathBuf
}

fn main() -> io::Result<()> {
    use process::{Linker, Typer};

    let args = Cli::parse();
    println!("Content of {:?}", args.file_path);
    let content = std::fs::read_to_string(args.file_path)
        .unwrap();

    let raw_ast = read::ast(content.chars());
    let mut linker = Linker::new();
    let linked_ast = linker.ast(raw_ast).unwrap();

    let mut typer = Typer::new();
    let ast = typer.type_ast(linked_ast).unwrap();
    println!("{:#?}", ast);

    loop {
        let input = Text::new("> ")
            .with_placeholder("x y. x = y")
            .with_help_message("Entrez une propriété")
            .prompt()
            .unwrap();

        let raw_prop = read::property(input.chars());
        let linked_prop = linker.property(raw_prop).unwrap();
        let prop = typer.type_property(linked_prop).unwrap();
        println!("{} {:#?}", "=>".blue().bold(), prop);
        println!("Inferred type '{:#?}'", prop.left.meta.typ);

        match analysis::prove(ast.clone(), prop) {
            Ok(proof) => {
                println!("{} :\n{}", "True".green(), proof);

                println!("Typst code :");
                println!("{}", proof.typst_repr())
            },
            Err(counter_ex) =>
                println!("{} :\n{}", "Could not deduce".red(),
counter_ex)
        }
    }
}
```

Dossier /src/lang/

Fichier /src/lang/mod.rs

```rust
mod ast;
mod repr;
mod find;

pub use ast::*;
pub use find::*;
pub use repr::*;
```

Dossier /src/lang/ast/

Fichier /src/lang/ast/ast.rs

```
use super::{ Statement, TypeDef };
use super::kind::Kind;


#[derive(Clone)]
pub struct AST<K: Kind> {
    pub type_defs: Vec<TypeDef<K>>,
    pub statements: Vec<Statement<K>>,
}
```

Fichier /src/lang/ast/expr.rs

```
use super::{ Pattern, Var };
use super::kind::Kind;


#[derive(Clone, PartialEq, Eq)]
pub struct Expr<K: Kind> {
    pub data: ExprData<K>,
    pub meta: K::ExprMeta
}

#[derive(Clone, PartialEq, Eq)]
pub enum ExprData<K: Kind> {
    Binding {
        var: Var<K>,
        val: Box<Expr<K>>,
        body: Box<Expr<K>>,
    },
    Function {
        input: Var<K>,
        body: Box<Expr<K>>,
    },
    Call {
        caller: Box<Expr<K>>,
        arg: Box<Expr<K>>,
    },
    Match {
        expr: Box<Expr<K>>,
        cases: Vec<MatchBranch<K>>,
    },
    LitVar {
        id: K::VariableId,
        meta: K::LitVarMeta,
    },
    LitConstructor {
        id: K::ConstructorId,
        args: Vec<Expr<K>>,
    },
}


#[derive(Clone, PartialEq, Eq)]
pub struct MatchBranch<K: Kind> {
    pub pattern: Pattern<K>,
    pub body: Expr<K>,
}


use ExprData::*;

impl<K: Kind> Expr<K> {
    pub fn map<F>(self, f: &F) -> Self
    where
        F: Fn(Self) -> Self,
    {
        let expr = f(self);

        let data = match expr.data {
            LitVar { id, meta } =>
                LitVar { id, meta },

            LitConstructor { id, args } =>
                LitConstructor {
                    id,
                    args: args.into_iter()
                        .map(|arg| arg.map(f))
                        .collect()
                },
            Binding { var, val, body } =>
                Binding {
                    var,
                    val: Box::new(val.map(f)),
                    body: Box::new(body.map(f))
                },

            Function { input, body } =>
                Function {
                    input,
                    body: Box::new(body.map(f))
                },

            Call { caller, arg } =>
                Call {
                    caller: Box::new(caller.map(f)),
                    arg: Box::new(arg.map(f))
                },

            Match { expr, cases } =>
                Match {
                    expr: Box::new(expr.map(f)),
                    cases: cases.into_iter()
                        .map(|MatchBranch { pattern, body }|
                            MatchBranch { pattern, body: body.map(f) }
                        )
                        .collect()
                }
        };

        Expr { data, meta: expr.meta }
    }

    /// Folds an `Expr<K>` tree by using `combine` on all direct
subexpressions
    /// and using `base_case` to specify special behavior.
    pub fn fold<B, F, G>(&self, default: B, base_case: &F, combine: &G)
-> B
    where
        B: Clone,
        F: Fn(&Self) -> Option<B>,
```

```
        G: Fn(B, B) -> B,
    {
        base_case(self).unwrap_or_else(||
            match &self.data {
                LitVar { .. } =>
                    default,

                LitConstructor { args, .. } =>
                    args.iter()
                        .map(|arg|
                            arg.fold(default.clone(), base_case,
combine)
                        )
                        .fold(default.clone(), combine),

                Binding { val, body, .. } =>
                    combine(
                        val.fold(default.clone(), base_case, combine),
                        body.fold(default, base_case, combine)
                    ),

                Function { body, .. } =>
                    body.fold(default, base_case, combine),

                Call { caller, arg } =>
                    combine(
                        caller.fold(default.clone(), base_case,
combine),
                        arg.fold(default, base_case, combine)
                    ),

                Match { expr, cases } =>
                    cases.iter()
                        .map(|MatchBranch { body, .. }|
                            body.fold(default.clone(), base_case,
combine)
                        )
                        .fold(
                            expr.fold(default.clone(), base_case,
combine),
                            combine
                        )
                }
            )
        }
}
```

Fichier /src/lang/ast/mod.rs

```
pub mod ast;
pub mod expr;
pub mod kind;
pub mod pat;
pub mod prop;
pub mod stmt;
pub mod typ;

pub use ast::*;
pub use expr::*;
pub use kind::*;
pub use pat::*;
pub use prop::*;
pub use stmt::*;
pub use typ::*;
```

Fichier /src/lang/ast/pat.rs

```
use super::kind::Kind;


#[derive(Clone, PartialEq, Eq)]
pub struct Pattern<K: Kind> {
    pub data: PatternData<K>,
    pub meta: K::PatternMeta
}

#[derive(Clone, PartialEq, Eq)]
pub enum PatternData<K: Kind> {
    Var(Var<K>),
    Constructor {
        id: K::ConstructorId,
        args: Vec<Pattern<K>>,
    }
}

#[derive(Clone, PartialEq, Eq)]
pub struct Var<K: Kind> {
    pub id: K::VariableId,
    pub meta: K::PatternMeta
}
```

Fichier /src/lang/ast/prop.rs

```
use super::{Expr, Var};
use super::kind::Kind;


#[derive(Clone)]
pub struct Property<K: Kind> {
    pub vars: Vec<Var<K>>,
    pub left: Expr<K>,
    pub right: Expr<K>
}
```

Fichier /src/lang/ast/stmt.rs

```
use super::{Var, Expr};
use super::kind::Kind;


#[derive(Clone)]
pub struct Statement<K: Kind> {
    pub var: Var<K>,
    pub val: Expr<K>,
}
```

Fichier /src/lang/ast/typ.rs

```
use super::kind::Kind;


#[derive(Clone, PartialEq, Eq)]
pub struct TypeDef<K: Kind> {
    pub id: K::TypeId,
    pub arg_ids: Vec<K::GenericId>,
    pub typ: TypeDefType<K>,
}


#[derive(Clone, PartialEq, Eq)]
pub enum TypeDefType<K: Kind> {
    Type(Type<K>),
    TypeSum(Vec<TypeSumBranch<K>>),
}


#[derive(Clone, PartialEq, Eq, Hash)]
pub enum Type<K: Kind> {
    Generic {
        id: K::GenericId
    },
    Specialization{
        args: Vec<Type<K>>,
        typ: K::TypeId,
    },
    Function {
        input: Box<Type<K>>,
        output: Box<Type<K>>,
    },
}


#[derive(Clone, PartialEq, Eq)]
pub struct TypeSumBranch<K: Kind> {
    pub constructor_id: K::ConstructorId,
    pub args: Vec<Type<K>>,
}
```

Dossier /src/lang/ast/kind/

Fichier /src/lang/ast/kind/linked.rs

```
use crate::lang::ast::{Expr, ExprData, Pattern, PatternData};

use super::*;


#[derive(Debug, Clone, PartialEq, Eq)]
/// Linked
pub struct Linked {}

impl Kind for Linked {
    type VariableId = UId;
    type ConstructorId = UId;
    type TypeId = UId;
    type GenericId = UId;

    type LitVarMeta = ();
    type ExprMeta = ();
    type PatternMeta = ();
}


impl From<ExprData<Linked>> for Expr<Linked> {
    fn from(data: ExprData<Linked>) -> Self {
        Expr { data, meta: () }
    }
}


impl From<PatternData<Linked>> for Pattern<Linked> {
    fn from(data: PatternData<Linked>) -> Self {
        Pattern { data, meta: () }
    }
}
```

Fichier /src/lang/ast/kind/lt.rs

```
use super::*;


#[derive(Debug, Clone, PartialEq, Eq, Hash)]
/// Linked & typed
pub struct LT {}

impl Kind for LT {
    type VariableId = UId;
    type ConstructorId = UId;
    type TypeId = UId;
    type GenericId = UId;

    type LitVarMeta = ();
    type ExprMeta = LTNodeMeta;
    type PatternMeta = LTNodeMeta;
}


#[derive(Clone, Eq)]
```

```
pub struct LTNodeMeta {
    pub typ: Type<LT>
}


impl PartialEq for LTNodeMeta {
    fn eq(&self, _: &Self) -> bool { true }
}


impl Meta for LTNodeMeta {
    fn type_repr(&self) -> String {
        format!("{:?}", self.typ)
    }
}


impl From<Type<LT>> for LTNodeMeta {
    fn from(typ: Type<LT>) -> Self {
        LTNodeMeta { typ }
    }
}


impl Debug for LTNodeMeta {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "{:?}", self.typ)
    }
}
```

Fichier /src/lang/ast/kind/lvt.rs

```
use super::*;
use super::lt::LTNodeMeta;


#[derive(Debug, Clone, PartialEq, Eq, Hash)]
/// Loose variables & typed
pub struct LVT {}

impl Kind for LVT {
    type VariableId = LoId;
    type ConstructorId = UId;
    type TypeId = UId;
    type GenericId = UId;

    type LitVarMeta = LVTVarMeta;
    type ExprMeta = LTNodeMeta;
    type PatternMeta = LTNodeMeta;
}


/// Loose id
#[derive(Clone, Eq)]
pub struct LoId {
    pub name: String,
}


pub type DistToDecl = usize;

#[derive(Eq, Clone)]
pub struct LVTVarMeta {
    pub dist_to_decl: DistToDecl,
    pub is_recursive: bool,
}


impl Id for LoId {}


impl PartialEq for LoId {
    fn eq(&self, _: &Self) -> bool {
        true
    }
}


impl Debug for LoId {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "{}", self.name)
    }
}
```

```
impl Meta for LVTVarMeta {}


impl PartialEq for LVTVarMeta {
    fn eq(&self, other: &Self) -> bool {
        self.dist_to_decl == other.dist_to_decl
    }
}


impl Debug for LVTVarMeta {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        if self.is_recursive {
            write!(f, "ᴿ")?;
        }
        write!(f, "↑")?;
        subscript(f, self.dist_to_decl)
    }
}
```

Fichier /src/lang/ast/kind/mod.rs

```
mod linked;
mod raw;
mod lt;
mod lvt;

pub use linked::Linked;
pub use raw::Raw;
pub use lt::{LT, LTNodeMeta};
pub use lvt::{LVT, LoId, DistToDecl, LVTVarMeta};

use std::{fmt::Debug, hash::Hash};

use super::Type;


pub trait Kind : Debug {
    type VariableId: Id;
    type ConstructorId: Id;
    type TypeId: Id;
    type GenericId: Id;

    type LitVarMeta: Meta;
    type ExprMeta: Meta;
    type PatternMeta: Meta;
}


pub trait Meta: Debug + Clone + Eq {
    fn type_repr(&self) -> String {
        String::new()
    }
}

impl Meta for () {}


pub trait Id: Debug + Clone + Eq { }


pub trait RawId: Id + Hash {
    fn name(&self) -> String;
}


/// Unique id
#[derive(Clone, Eq)]
pub struct UId {
    pub id: usize,
    pub name: String,
}


impl Id for UId { }


impl RawId for UId {
    fn name(&self) -> String {
        self.name.clone()
    }
}
```

```
impl UId {
    pub fn unnamed(id: usize) -> UId {
        UId { id, name: id.to_string() }
    }
}

impl From<(usize, &str)> for UId {
    fn from((id, str_name): (usize, &str)) -> Self {
        UId { id, name: String::from(str_name) }
    }
}

impl Debug for UId {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "{}", self.name)?;
        subscript(f, self.id)
    }
}

fn subscript(f: &mut std::fmt::Formatter<'_>, id: usize) ->
std::fmt::Result {
    for digit in id.to_string().chars() {
        write!(f, "{}", match digit {
            '0' => "₀",
            '1' => "₁",
            '2' => "₂",
            '3' => "₃",
            '4' => "₄",
            '5' => "₅",
            '6' => "₆",
            '7' => "₇",
            '8' => "₈",
            '9' => "₉",
            _ => "₋"
        })?
    }
    Ok(())
}

impl ToString for UId {
    fn to_string(&self) -> String {
        self.name.clone()
    }
}

impl PartialEq for UId {
    fn eq(&self, other: &Self) -> bool {
        self.id == other.id
    }
}

impl std::hash::Hash for UId {
    fn hash<H: std::hash::Hasher>(&self, state: &mut H) {
        self.id.hash(state)
    }
}
```

Fichier /src/lang/ast/kind/raw.rs

```
use crate::lang::ast::{Expr, ExprData, Pattern, PatternData};

use super::*;


#[derive(Debug, Clone, PartialEq, Eq)]
pub struct Raw {}
impl Kind for Raw {
    type VariableId = String;
    type ConstructorId = String;
    type TypeId = String;
    type GenericId = String;

    type LitVarMeta = ();
    type ExprMeta = ();
    type PatternMeta = ();
}


impl Id for String {}


impl RawId for String {
    fn name(&self) -> String {
        self.clone()
    }
}


impl From<ExprData<Raw>> for Expr<Raw> {
    fn from(data: ExprData<Raw>) -> Self {
        Expr { data, meta: () }
    }
}


impl From<PatternData<Raw>> for Pattern<Raw> {
    fn from(data: PatternData<Raw>) -> Self {
        Pattern { data, meta: () }
    }
}
```

Dossier /src/lang/find/

Fichier /src/lang/find/constructors.rs

```
use std::{collections::HashSet, hash::Hash};
use crate::lang::*;


pub trait FindConstructors<K: Kind>
    where K::ConstructorId: Hash
{
    fn constructors(&self) -> HashSet<K::ConstructorId>;
}


impl<K: Kind> FindConstructors<K> for Expr<K>
    where K::ConstructorId: Hash
{
    fn constructors(&self) -> HashSet<K::ConstructorId> {
        self.data.constructors()
    }
}

impl<K: Kind> FindConstructors<K> for ExprData<K>
    where K::ConstructorId: Hash
{
    fn constructors(&self) -> HashSet<K::ConstructorId> {
        use ExprData::*;

        match self {
            Binding { val, body, .. } =>
                body.constructors()
                    .union(&val.constructors())
                    .cloned()
                    .collect(),

            Function { body, .. } =>
                body.constructors(),

            Call { caller, arg } =>
                (*caller).constructors()
                    .union(&(*arg).constructors())
                    .cloned()
                    .collect(),

            Match { expr, cases } =>
                (*expr).constructors()
                    .into_iter()
                    .chain(
                        cases.into_iter()
                            .flat_map(|case|
                                case.constructors().into_iter()
                            )
                    )
                    .collect(),

            LitVar { .. } =>
                HashSet::new(),

            LitConstructor { id, args } =>
                HashSet::from([id.clone()]).into_iter()
                    .chain(
                        args.into_iter()
                            .flat_map(|arg|
                                arg.constructors().into_iter()
                            )
                    )
                    .collect()
        }
    }
}


impl<K: Kind> FindConstructors<K> for MatchBranch<K>
where
    K::ConstructorId: Hash
{
    fn constructors(&self) -> HashSet<<K as Kind>::ConstructorId> {
        self.body.constructors()
            .union(&self.pattern.constructors())
            .cloned()
            .collect()
    }
}


impl<K: Kind> FindConstructors<K> for Pattern<K>
where
    K::ConstructorId: Hash
{
    fn constructors(&self) -> HashSet<K::ConstructorId> {
        match &self.data {
            PatternData::Var(_) =>
                HashSet::new(),

            PatternData::Constructor { id, args } =>
                HashSet::from([id.clone()]).into_iter()
                    .chain(
                        args.into_iter()
                            .flat_map(|arg|
                                arg.constructors().into_iter()
                            )
                    )
                    .collect()
        }
    }
}
```

Fichier /src/lang/find/free_vars.rs

```
use std::{collections::HashSet, hash::Hash};

use crate::lang::*;


pub trait FindFreeVars<K: Kind>
where
    K::VariableId: Hash
{
    fn free_vars(&self) -> HashSet<K::VariableId>;
}


impl<K: Kind> FindFreeVars<K> for AST<K>
where
    K::VariableId: Hash
{
    fn free_vars(&self) -> HashSet<K::VariableId> {
        self.statements
            .iter()
            .flat_map(|stmt| stmt.free_vars().into_iter())
            .collect()
    }
}


impl<K: Kind> FindFreeVars<K> for Expr<K>
where
    K::VariableId: Hash
{
    fn free_vars(&self) -> HashSet<K::VariableId> {
        self.data.free_vars()
    }
}


impl<K: Kind> FindFreeVars<K> for ExprData<K>
where
```

```
    K::VariableId: Hash
{
    fn free_vars(&self) -> HashSet<K::VariableId> {
        use ExprData::*;

        match self {
            Binding { var, val, body } =>
                body.free_vars()
                    .difference(&var.free_vars())
                    .cloned()
                    .collect::<HashSet<_>>()
                    .union(&val.free_vars())
                    .cloned()
                    .collect(),

            Function { input, body, .. } =>
                body.free_vars()
                    .difference(&input.free_vars())
                    .cloned()
                    .collect(),

            Call { caller, arg } =>
                (*caller)
                    .free_vars()
                    .union(&(*arg).free_vars())
                    .cloned()
                    .collect(),

            Match { expr, cases } =>
                (*expr)
                    .free_vars()
                    .into_iter()
                    .chain(
                        cases.into_iter()
                            .flat_map(|case|
                                case.free_vars().into_iter()
                            )
                    )
                    .collect(),

            LitVar { id, .. } =>
                HashSet::from([id.clone()]),

            LitConstructor { args, .. } =>
                args.into_iter()
                    .flat_map(|arg| arg.free_vars().into_iter())
                    .collect()
        }
    }
}


impl<K: Kind> FindFreeVars<K> for MatchBranch<K>
where
    K::VariableId: Hash
{
    fn free_vars(&self) -> HashSet<K::VariableId> {
        self.body.free_vars()
            .difference(&self.pattern.free_vars())
            .cloned()
            .collect()
    }
}


impl<K: Kind> FindFreeVars<K> for Pattern<K>
where
    K::VariableId: Hash
{
    fn free_vars(&self) -> HashSet<K::VariableId> {
        match &self.data {
            PatternData::Var(var) =>
                var.free_vars(),
            PatternData::Constructor { args, .. } =>
                args.into_iter()
                    .flat_map(|arg|
                        arg.free_vars().into_iter()
                    )
                    .collect()
        }
    }
}
```

```
impl<K: Kind> FindFreeVars<K> for Var<K>
where
    K::VariableId: Hash
{
    fn free_vars(&self) -> HashSet<K::VariableId> {
        HashSet::from([self.id.clone()])
    }
}


impl<K: Kind> FindFreeVars<K> for Statement<K>
where
    K::VariableId: Hash
{
    fn free_vars(&self) -> HashSet<K::VariableId> {
        self.val.free_vars()
            .into_iter()
            .chain(self.var.free_vars().into_iter())
            .collect()
    }
}
```

Fichier /src/lang/find/generics.rs

```
use std::{collections::HashSet, hash::Hash};
use crate::lang::*;


pub trait FindGenerics<K: Kind> {
    fn generics(&self) -> HashSet<K::GenericId>;
}


impl<K: Kind> FindGenerics<K> for AST<K>
where
    K::GenericId: Hash,
{
    fn generics(&self) -> HashSet<K::GenericId> {
        self.type_defs.iter()
            .flat_map(|type_def| type_def.generics())
            .collect()
    }
}

impl<K: Kind> FindGenerics<K> for TypeDef<K>
where
    K::GenericId: Hash,
{
    fn generics(&self) -> HashSet<K::GenericId> {
        self.arg_ids.clone()
            .into_iter()
            .collect()
    }
}

impl<K: Kind> FindGenerics<K> for Type<K>
where
    K::GenericId: Hash,
{
    fn generics(&self) -> HashSet<K::GenericId> {
        use Type::*;
        match self {
            Generic { id } => HashSet::from([id.clone()]),
            Specialization { args, .. } =>
                args.into_iter()
                    .flat_map(|arg|
                        arg.generics().into_iter()
                    )
                    .collect(),
            Function { input, output } =>
                input.generics()
                    .into_iter()
                    .chain(output.generics().into_iter())
                    .collect()
        }
    }
}
```

Fichier /src/lang/find/mod.rs

```
mod constructors;
mod free_vars;
mod generics;
mod types;

pub use constructors::FindConstructors;
pub use free_vars::FindFreeVars;
pub use generics::FindGenerics;
pub use types::FindTypes;
```

Fichier /src/lang/find/types.rs

```
use std::{collections::HashSet, hash::Hash};
use crate::lang::*;


pub trait FindTypes<K: Kind> {
    fn types(&self) -> HashSet<K::TypeId>;
}


impl<K: Kind> FindTypes<K> for AST<K>
where
    K::TypeId: Hash
{
    fn types(&self) -> HashSet<K::TypeId> {
        self.type_defs.iter()
            .flat_map(|type_def| type_def.types())
            .collect()
    }
}


impl<K: Kind> FindTypes<K> for TypeDef<K>
where
    K::TypeId: Hash
{
    fn types(&self) -> HashSet<K::TypeId> {
        HashSet::from([self.id.clone()])
    }
}
```

Dossier /src/lang/repr/

Fichier /src/lang/repr/debug.rs

```
use std::fmt::Debug;

use crate::lang::{kind::Meta, *};


trait PadDebug {
    fn fmt_with_padding(&self, f: &mut std::fmt::Formatter<'_>, pad:
usize)
        -> std::fmt::Result;
}


impl<K: Kind> Debug for AST<K> {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        self.fmt_with_padding(f, 0)
    }
}

impl<K: Kind> PadDebug for AST<K> {
    fn fmt_with_padding(&self, f: &mut std::fmt::Formatter<'_>, pad:
usize)
        -> std::fmt::Result
    {
        for (i, type_def) in self.type_defs.iter().enumerate() {
            if i != 0 {
                if f.alternate() {
                    write!(f, "\n\n")?;
                    write_pad(f, pad)?;
                } else {
                    write!(f, " ")?;
                }
            }
            type_def.fmt_with_padding(f, pad)?;
        }

        for statement in &self.statements {
            if f.alternate() {
                write!(f, "\n\n")?;
                write_pad(f, pad)?;
            } else {
                write!(f, " ")?;
            }
            statement.fmt_with_padding(f, pad)?;
        }

        Ok(())
    }
}


impl<K: Kind> Debug for TypeDef<K> {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        self.fmt_with_padding(f, 0)
    }
}

impl<K: Kind> PadDebug for TypeDef<K> {
    fn fmt_with_padding(&self, f: &mut std::fmt::Formatter<'_>, pad:
usize)
        -> std::fmt::Result
    {
        write!(f, "type ")?;
        for arg in &self.arg_ids {
            write!(f, "{:?} ", arg)?;
        }

        write!(f, "{:?} =", self.id)?;
        if f.alternate() && self.typ.is_type_sum() {
            write!(f, "\n")?;
            write_pad(f, pad + 1)?;
            write!(f, "| ")?;
        } else {
            write!(f, " ")?;
        }
        self.typ.fmt_with_padding(f, pad + 1)?;
        write!(f, ";;")
    }
}


impl<K: Kind> Debug for TypeDefType<K> {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        self.fmt_with_padding(f, 0)
    }
```

```
}

impl<K: Kind> TypeDefType<K> {
    fn is_type_sum(&self) -> bool {
        use TypeDefType::*;

        match self {
            Type(_) => false,
            TypeSum(_) => true
        }
    }
}

impl<K: Kind> PadDebug for TypeDefType<K> {
    fn fmt_with_padding(&self, f: &mut std::fmt::Formatter<'_>, pad:
usize)
        -> std::fmt::Result
    {
        use TypeDefType::*;

        match self {
            Type(typ) => write!(f, "{:?}", typ),
            TypeSum(branches) => {
                for (i, branch) in branches.iter().enumerate() {
                    if i != 0 {
                        if f.alternate() {
                            write!(f, "\n")?;
                            write_pad(f, pad)?;
                            write!(f, "| ")?;
                        } else {
                            write!(f, " | ")?;
                        }
                    }
                    write!(f, "{:?}", branch)?;
                }
                Ok(())
            }
        }
    }
}
```

```
impl<K: Kind> Debug for TypeSumBranch<K> {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "{:?}", self.constructor_id)?;

        if self.args.len() > 0 {
            write!(f, " of ")?;
        }

        for (i, arg) in self.args.iter().enumerate() {
            if i != 0 {
                write!(f, " * ")?;
            }
            write!(f, "{:?}", arg)?;
        }
        Ok(())
    }
}


impl<K: Kind> Debug for Type<K> {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        use Type::*;

        match self {
            Generic { id } =>
                write!(f, "{:?}", id),

            Function { input, output } => {
                let parenthesize_input =
                    matches!(input.as_ref(), Function { .. });
                if parenthesize_input {
                    write!(f, "(")?;
                }
                write!(f, "{:?}", *input)?;
                if parenthesize_input {
                    write!(f, ")")?;
                }

                write!(f, " -> {:?}", *output)
            },
```

```
            Specialization { args, typ } => {
                args.iter()
                    .map(|arg| {
                        let parenthesize =
                            matches!(arg, Function { .. })
                            || matches!(arg, Specialization { args, .. }
if !args.is_empty());
                        if parenthesize {
                            write!(f, "(")?;
                        }
                        write!(f, "{:?} ", arg)?;
                        if parenthesize {
                            write!(f, ")")?;
                        }
                        Ok(())
                    })
                    .collect::<std::fmt::Result>()?;
                write!(f, "{:?}", typ)
            }
        }
    }
}


impl<K: Kind> Debug for Statement<K> {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        self.fmt_with_padding(f, 0)
    }
}

impl<K: Kind> PadDebug for Statement<K> {
    fn fmt_with_padding(&self, f: &mut std::fmt::Formatter<'_>, pad:
usize)
        -> std::fmt::Result
    {
        write!(f, "let {:?}", self.var)?;

        let type_repr = self.val.meta.type_repr();
        if f.alternate() && !type_repr.is_empty() {
            write!(f, ": {}", type_repr)?;
        }

        write!(f, " =")?;
        if f.alternate() && self.val.is_complex() {
            write!(f, "\n")?;
            write_pad(f, pad + 1)?;
        } else {
            write!(f, " ")?;
        }
        self.val.fmt_with_padding(f, pad + 1)?;
        write!(f, ";;")
    }
}


impl<K: Kind> Debug for Expr<K> {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        self.fmt_with_padding(f, 0)
    }
}

impl<K: Kind> Expr<K> {
    fn is_complex(&self) -> bool {
        use ExprData::*;

        match self.data {
            LitVar { .. }
            | LitConstructor { .. } => false,
            _ => true
        }
    }

    fn is_call(&self) -> bool {
        match self.data {
            ExprData::Call { .. } => true,
            _ => false
        }
    }
}

impl<K: Kind> PadDebug for Expr<K> {
    fn fmt_with_padding(&self, f: &mut std::fmt::Formatter<'_>, pad:
usize)
        -> std::fmt::Result
    {
        self.data.fmt_with_padding(f, pad)
        // ?; write!(f, " <{:?}>", self.meta)
    }
}


impl<K: Kind> Debug for ExprData<K> {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        self.fmt_with_padding(f, 0)
    }
}

impl<K: Kind> PadDebug for ExprData<K> {
    fn fmt_with_padding(&self, f: &mut std::fmt::Formatter<'_>, pad:
usize)
        -> std::fmt::Result
    {
        use ExprData::*;

        match self {
            Binding { var, val, body } => {
                write!(f, "let {:?}", var)?;

                let type_repr = val.meta.type_repr();
                if f.alternate() && !type_repr.is_empty() {
                    write!(f, ": {}", type_repr)?;
                }

                write!(f, " =")?;
                if f.alternate() {
                    if val.is_complex() {
                        write!(f, "\n")?;
                        write_pad(f, pad + 1)?;
                        val.fmt_with_padding(f, pad + 1)?;
                        write!(f, "\n")?;
                        write_pad(f, pad)?;
                        write!(f, "in\n")?;
        } else {
            write!(f, " ")?;
            val.fmt_with_padding(f, pad + 1)?;
            write!(f, " in\n")?;
        }
        write_pad(f, pad)?;
    } else {
        write!(f, " ")?;
        val.fmt_with_padding(f, pad + 1)?;
        write!(f, " in ")?;
    }
    body.fmt_with_padding(f, pad)
},

Function { input, body, .. } => {
    write!(f, "fun {:?} ->", input)?;

    if f.alternate() && body.is_complex() {
        write!(f, "\n")?;
        write_pad(f, pad + 1)?;
    } else {
        write!(f, " ")?;
    }

    body.fmt_with_padding(f, pad + 1)
},

Call { caller, arg } => {
    if caller.is_complex() && !caller.is_call() {
        write!(f, "(")?;
        caller.fmt_with_padding(f, pad)?;
        write!(f, ")")?;
    } else {
        caller.fmt_with_padding(f, pad)?;
    }

    if arg.is_complex() {
        write!(f, " (")?;
        arg.fmt_with_padding(f, pad + 1)?;
        write!(f, ")")
    } else {
        write!(f, " ")?;
        arg.fmt_with_padding(f, pad + 1)
    }


}

Match { expr, cases } => {
    write!(f, "match ")?;
    expr.fmt_with_padding(f, pad + 1)?;
    write!(f, " with")?;
    for case in cases {
        if f.alternate() {
            write!(f, "\n")?;
            write_pad(f, pad)?;
        } else {
            write!(f, " ")?;
        }
        write!(f, "| ")?;
        case.fmt_with_padding(f, pad)?;
    }
    Ok(())
},

LitVar { id, meta } => {
    // TODO: write this more nicely
    write!(f, "{:?}", id)?;
    if format!("{:?}", meta) != "()" {
        write!(f, "{:?}", meta)?;
    }
    Ok(())
},

LitConstructor { id, args } => {
    write!(f, "{:?}", id)?;

    if args.len() > 0 {
        write!(f, "(")?;
        args[0].fmt_with_padding(f, pad + 1)?;
        for arg in args.iter().skip(1) {
            write!(f, ", ")?;
```

```
                        arg.fmt_with_padding(f, pad + 1)?;
                    }
                    write!(f, ")")?;
                }
                Ok(())
            }
        }
    }
}


impl<K: Kind> Debug for MatchBranch<K> {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        self.fmt_with_padding(f, 0)
    }
}

impl<K: Kind> PadDebug for MatchBranch<K> {
    fn fmt_with_padding(&self, f: &mut std::fmt::Formatter<'_>, pad:
usize)
        -> std::fmt::Result
    {
        write!(f, "{:?} ->", self.pattern.data)?;
        if f.alternate() && self.body.is_complex() {
            write!(f, "\n")?;
            write_pad(f, pad + 2)?;
        } else {
            write!(f, " ")?;
        }
        self.body.fmt_with_padding(f, pad + 2)
    }
}


impl<K: Kind> Debug for Pattern<K> {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "{:?}", self.data)
    }
}

impl<K: Kind> Debug for PatternData<K> {
```

```
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match self {
            PatternData::Var(var) =>
                write!(f, "{:?}", var),
            PatternData::Constructor { id, args } => {
                write!(f, "{:?}", id)?;

                if args.len() > 0 {
                    write!(f, "(")?;
                    write!(f, "{:?}", args[0])?;
                    for arg in args.iter().skip(1) {
                        write!(f, ", {:?}", arg)?;
                    }
                    write!(f, ")")?;
                }
                Ok(())
            }
        }
    }
}


impl<K: Kind> Debug for Var<K> {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "{:?}", self.id)
    }
}


impl<K: Kind> Debug for Property<K> {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        self.fmt_with_padding(f, 0)
    }
}
impl<K: Kind> PadDebug for Property<K> {
    fn fmt_with_padding(&self, f: &mut std::fmt::Formatter<'_>, pad:
usize)
        -> std::fmt::Result
    {
        for (i, var) in self.vars.iter().enumerate() {
            if i != 0 {
                write!(f, " ")?;
            }
            write!(f, "{:?}", var)?;
        }
        write!(f, ".")?;

        if f.alternate() && (self.left.is_complex() ||
self.right.is_complex()) {
            write!(f, "\n")?;
            write_pad(f, pad + 1)?;
            self.left.fmt_with_padding(f, pad + 1)?;
            write!(f, "\n")?;
            write_pad(f, pad)?;
            write!(f, "≡\n")?;
            write_pad(f, pad + 1)?;
            self.right.fmt_with_padding(f, pad + 1)
        } else {
            write!(f, " ")?;
            self.left.fmt_with_padding(f, pad + 1)?;
            write!(f, " ≡ ")?;
            self.right.fmt_with_padding(f, pad + 1)
        }
    }
}


fn write_pad(f: &mut std::fmt::Formatter<'_>, pad: usize) ->
std::fmt::Result {
    write!(f, "{}", "  ".repeat(pad))
}
```

Fichier /src/lang/repr/mod.rs

```
mod debug;
mod typst;

pub use typst::TypstRepr;
```

Fichier /src/lang/repr/typst.rs

```
use std::collections::VecDeque;

use crate::lang::*;


pub trait TypstRepr {
    fn typst_repr(&self) -> String;
}


impl<T: TypstRepr> TypstRepr for VecDeque<T> {
    fn typst_repr(&self) -> String {
        format!(
            "({})",
            self.iter()
                .map(|elem| elem.typst_repr())
                .fold(
                    String::from("\"vec\", "),
                    |acc, elem| String::from(acc + &elem + ", ")
                )
        )
    }
}


impl<T: TypstRepr> TypstRepr for Vec<T> {
    fn typst_repr(&self) -> String {
        format!(
            "({})",
            self.iter()
                .map(|elem| elem.typst_repr())
                .fold(
                    String::from("\"vec\", "),
                    |acc, elem| String::from(acc + &elem + ", ")
                )
        )
    }
}


impl TypstRepr for Expr<LVT> {
    fn typst_repr(&self) -> String {
        self.data.typst_repr()
    }
}
```

```
impl TypstRepr for ExprData<LVT> {
    fn typst_repr(&self) -> String {
        use ExprData::*;

        match &self {
            LitVar { id, meta } =>
                format!(
                    "(\"lit_var\", {}, {})",
                    id.typst_repr(),
                    meta.typst_repr()
                ),

            LitConstructor { id, args } =>
                format!(
                    "(\"lit_constr\", {}, {})",
                    id.typst_repr(),
                    args.typst_repr()
                ),

            Function { input, body, .. } =>
                format!(
                    "(\"function\", {}, {})",
                    input.typst_repr(),
                    body.typst_repr()
                ),

            Call { caller, arg } =>
                format!(
                    "(\"call\", {}, {})",
                    caller.typst_repr(),
                    arg.typst_repr()
                ),

            Match { expr, cases } =>
                format!(
                    "(\"match\", {}, {})",
                    expr.typst_repr(),
                    cases.typst_repr()
                ),

            Binding { var, val, body } =>
                format!(
                    "(\"binding\", {}, {}, {})",
                    var.typst_repr(),
                    val.typst_repr(),
                    body.typst_repr()
                )
        }.to_string()
    }
}


impl TypstRepr for MatchBranch<LVT> {
    fn typst_repr(&self) -> String {
        format!(
            "(\"branch\", {}, {})",
            self.pattern.typst_repr(),
            self.body.typst_repr()
        ).to_string()
    }
}


impl TypstRepr for Pattern<LVT> {
    fn typst_repr(&self) -> String {
        self.data.typst_repr()
    }
}


impl TypstRepr for PatternData<LVT> {
    fn typst_repr(&self) -> String {
        use PatternData as PD;

        match &self {
            PD::Var(var) =>
                var.typst_repr(),
            PD::Constructor { id, args } =>
                format!(
                    "(\"pattern\", {}, {})",
                    id.typst_repr(),
                    args.typst_repr()
                )
        }.to_string()
    }
}


impl TypstRepr for Var<LVT> {
    fn typst_repr(&self) -> String {
        format!(
            "(\"var\", {})",
            self.id.typst_repr(),
        ).to_string()
    }
}


impl TypstRepr for UId {
    fn typst_repr(&self) -> String {
        format!(
            "(\"uid\", \"{}\", {})",
            self.name,
            self.id
        ).to_string()
    }
}


impl TypstRepr for LoId {
    fn typst_repr(&self) -> String {
        format!(
            "(\"loid\", \"{}\")",
            self.name,
        ).to_string()
    }
}


impl TypstRepr for LVTVarMeta {
    fn typst_repr(&self) -> String {
        format!(
            "(\"var_meta\", {})",
            self.dist_to_decl,
        ).to_string()
    }
}
```

Dossier /src/analysis/

Fichier /src/analysis/mod.rs

```
mod halt;
mod solve;

pub use solve::prove;
```

Dossier /src/analysis/halt/

Fichier /src/analysis/halt/mod.rs

```
// TODO: find a variant
```

Dossier /src/analysis/solve/

Fichier /src/analysis/solve/matcher.rs

```
use crate::lang::*;

use super::data::Association;


pub struct FoundMatch {
    pub associations: Vec<Option<Association>>,
    pub expr: Expr<LVT>,
    pub cases: Vec<MatchBranch<LVT>>
}


pub fn find_match(expr: Expr<LVT>) -> Option<FoundMatch> {
    find_match_aux(Vec::new(), expr)
}


fn find_match_aux(mut associations: Vec<Option<Association>>, expr:
Expr<LVT>)
    -> Option<FoundMatch>
{
    use ExprData::*;

    match expr.data {
        Binding { var, val, body } => {
            let mut value_associations = associations.clone();
            value_associations.push(None);
            find_match_aux(value_associations, (*val).clone())
                .or_else(|| {
                    associations.push(Some(Association { id: var.id,
value: *val}));
                    find_match_aux(associations, *body)
                })
        }

        Function { body, .. } => {
            associations.push(None);
            find_match_aux(associations, *body)
        },

        Call { caller, arg } =>
            find_match_aux(associations.clone(), *caller)
                .or_else(|| find_match_aux(associations, *arg)),

        Match { expr, cases } =>
            Some(FoundMatch { associations, expr: *expr, cases }),

        LitVar { .. } => None,

        LitConstructor { args, .. } =>
            args.into_iter()
                .fold(
                    None,
                    |opt_res, arg|
                        opt_res.or_else(||
                            find_match_aux(associations.clone(), arg)
                        )
                )
    }
}
```

Fichier /src/analysis/solve/mod.rs

```
mod data;
mod normal;
mod prove;

// TODO: maybe move
mod matcher;
mod pat;

pub use prove::prove;
```

Fichier /src/analysis/solve/pat.rs

```
use crate::lang::*;

use super::data::Context;


pub struct Assumption {
    pub used_dist: DistToDecl,
    pub unlinked_value: Expr<LVT>
}


pub fn make_into(
    context: &Context,
    pat: &Pattern<LVT>,
    expr: &Expr<LVT>,
) -> Option<Vec<Assumption>>
{
    use PatternData as PD;

    match (&pat.data, &context.shallow_resolve(expr).data) {
        (PD::Var(Var { .. }), _) =>
            Some(Vec::new()),

        (_, ExprData::LitVar { meta, .. })
        if context.is_free_id(meta.dist_to_decl) =>
            Some(vec![Assumption {
                used_dist: meta.dist_to_decl,
                unlinked_value: make_fitting(pat.clone())
            }]),

        (PD::Constructor { id: pat_id, args: pat_args },
            ExprData::LitConstructor { id: lit_id, args: lit_args  })
        if pat_id == lit_id =>
        {
            let assumptions = pat_args.iter()
                .zip(lit_args.iter())
                .map(|(pat, expr)|
                    make_into(context, pat, expr)
                )
                .collect::<Option<Vec<_>>>()?
                .into_iter()
                .flatten()
                .collect();

            // TODO: resolve conflicts

            Some(assumptions)
        }

        _ => None
    }
}


fn make_fitting(
    pat: Pattern<LVT>,
) -> Expr<LVT>
{
    use PatternData as PD;
    use ExprData::*;

    match pat.data {
        PD::Var(Var { id, meta }) =>
            Expr {
                data: LitVar {
                    id: id,
                    meta: LVTVarMeta {
                        dist_to_decl: 0,
                        is_recursive: false
                    }
                }.into(),
                meta: meta.into()
            },
```

```
        PD::Constructor { id, args } => {
            let args = args.into_iter()
                .map(|arg| make_fitting(arg))
                .collect();

            Expr {
                data: LitConstructor { id: id, args },
                meta: pat.meta.into()
            }
        }
    }
}


/*
#[cfg(test)]
mod tests {
    use crate::lang::read;
    use super::*;

    #[test]
    fn test_fit() {
        let any = read::pattern("_".chars());
        let id = read::expr("fun (x) -> x".chars());

        let dbl_succ = read::pattern("Succ (Succ (n'))".chars());
        let one = read::expr("Succ (Zero)".chars());
        let tree = read::expr("Succ (Succ (Succ (Zero)))".chars());

        // assert_eq!(
        //     fit(&any, &id),
        //     Some(HashMap::from([
        //         (String::from("_"), id.clone())
        //     ]))
        // );
        // assert_eq!(
        //     fit(&any, &one),
        //     Some(HashMap::from([
        //         (String::from("_"), one.clone())
        //     ]))
        // );
        // assert_eq!(
        //     fit(&any, &tree),
        //     Some(HashMap::from([
        //         (String::from("_"), tree.clone())
        //     ]))
        // );
        //
        // assert_eq!(fit(&dbl_succ, &id), None);
        // assert_eq!(fit(&dbl_succ, &one), None);
        // assert_eq!(
        //     fit(&dbl_succ, &tree),
        //     Some(HashMap::from([
        //         (String::from("n'"), one.clone())
        //     ]))
        // );
    }


    #[test]
    fn test_make_into() {
        let succ = read::pattern("Succ (n')".chars());
        let id = read::expr("fun (x) -> x".chars());
        // let x = read::expr("x".chars());
        // let succ_x = read::expr("Succ (x)".chars());

        //assert_eq!(make_into(&succ, &id), None);
        // TODO: test when structural equality works
        // assert_eq!(make_into(&succ, &x), None);
        // assert_eq!(make_into(&succ, &succ_x), None);
    }
}*/
```

Fichier /src/analysis/solve/prove.rs

```
use crate::lang::*;
use crate::process::LooseLinker;

use super::data::*;

use super::{matcher, pat};

pub fn prove(lt_ast: AST<LT>, lt_property: Property<LT>)
    -> Result<Proof, CounterEx>
{
    let mut loose_linker = LooseLinker::new();
    let ast = loose_linker.ast(lt_ast);
    let property = loose_linker.property(lt_property);
    println!("{}", property.left.typst_repr());

    let hypotheses: Vec<_> = ast.statements.into_iter()
        .map(|stmt|
            Association { id: stmt.var.id, value: stmt.val }
        )
        .collect();

    let sequent = Sequent::from_property(hypotheses, property);
    let prover = Prover { type_defs: ast.type_defs };
    println!("{}", sequent.typst_repr());
    prover.prove(sequent)
}


#[derive(Clone)]
struct Prover {
    type_defs: Vec<TypeDef<LVT>>
}


impl Prover {
    pub fn prove(&self, sequent: Sequent) -> Result<Proof, CounterEx> {
        use super::normal::normalized;
        let prev_sequent = sequent.clone();
        let (trans, sequent) = normalized(sequent);

        let rule = match matcher::find_match(sequent.left.clone())
            .or_else(|| matcher::find_match(sequent.right.clone()))
        {
            Some(matcher::FoundMatch {
                associations,
                expr: match_expr,
                cases,
            }) => {
                // TODO : check if on branch is not a sub pattern of
above cases
                let choices = cases.iter()
                    .filter_map(|case|
                        self.try_make_choice(
                            sequent.clone(),
                            &associations,
                            &match_expr,
                            case
                        )
                    )
                    .collect::<Result<Vec<_>, _>>()?;

                if choices.len() > 0 {
                    Rule::MatchElimination(choices)
                } else {
                    unreachable!("Expected an exhaustive match")
                }
            },

            None if sequent.left == sequent.right =>
                Rule::Equal,

            None => {
                println!("{:#?}", sequent);
                todo!("Try to find counter examples")
                // Err(CounterEx {
                //     vals: self.assumptions,
                //     left: final_left,
                //     right: final_right
                // })
            }
        };

        if trans.is_empty() {
            Ok(Proof { sequent, rule })
        } else {
            Ok(Proof {
                sequent: prev_sequent,
                rule: Rule::Transform(
                    trans,
                    Box::new(Proof { sequent, rule })
                )
            })
        }
    }

    fn try_make_choice(
        &self,
        mut sequent: Sequent,
        associations: &Vec<Option<Association>>,
        match_expr: &Expr<LVT>,
        case: &MatchBranch<LVT>,
    )
        -> Option<Result<Proof, CounterEx>>
    {
        let assumptions = pat::make_into(
            &Context::from_sequent(&sequent)
                .with_opt_associations(associations),
            &case.pattern,
            match_expr,
        )?;

        for assumption in assumptions {
            let origin_dist = assumption.used_dist - associations.len();
            sequent.use_free_id(origin_dist, assumption.unlinked_value)
        }

        Some(self.prove(sequent))
    }
}
```

Dossier /src/analysis/solve/normal/

Fichier /src/analysis/solve/normal/mod.rs

```
mod factorize;
mod eval;
mod utils;

use std::collections::VecDeque;
use utils::is_id_used;

use crate::lang::*;

use super::data::{Association, Context, Sequent, TransformRule};


pub fn normalized(sequent: Sequent) -> (Vec<TransformRule>, Sequent) {
    let context = Context::from_sequent(&sequent);
    let mut trans = vec![];

    let mut left = sequent.left.clone();
    loop {
        let (new_trans, new_left) =
            normalize_pass(&context, left.clone());
        left = new_left;

        if new_trans.is_empty() {
            break;
        }
        trans.extend(new_trans);
    }

    let mut right = sequent.right.clone();
    loop {
        let (new_trans, new_right) =
            normalize_pass(&context, right.clone());
        right = new_right;

        if new_trans.is_empty() {
            break;
        }
        trans.extend(new_trans);
    }

    let hypotheses_len = sequent.hypotheses.len();
    let mut rev_new_hypotheses = Vec::with_capacity(hypotheses_len);

    for (dist, hyp) in sequent.hypotheses.into_iter().rev().enumerate()
{
        let is_used = is_id_used(dist + 1, &left)
            || is_id_used(dist + 1, &right)
            || rev_new_hypotheses.iter()
                .any(|(counter_dist, Association { value, .. })|
                    is_id_used(dist + 1 - counter_dist, value)
                );

        if is_used {
            rev_new_hypotheses.push((dist, hyp));
        }
    }

    let hypotheses: VecDeque<_> = rev_new_hypotheses.into_iter()
        .enumerate()
        .map(|(new_dist, (old_dist, hyp))| {
            let offset = new_dist as isize - old_dist as isize;
            let Association { id, value } = hyp;

            left.move_declaration_dist(old_dist + 1, new_dist + 1);
            right.move_declaration_dist(old_dist + 1, new_dist + 1);

            Association { id, value: value.all_moved(0, offset) }
        })
        .rev()
        .collect();

    let global_offset = hypotheses.len() as isize - hypotheses_len as
isize;
    left = left.all_moved(hypotheses_len, global_offset);
    right = right.all_moved(hypotheses_len, global_offset);

    if global_offset != 0 {
        trans.push(TransformRule::Weakening)
    }

    (trans, Sequent {
        free_ids: sequent.free_ids,
        hypotheses,
        left,
        right
    })
}
```

```
fn normalize_pass(context: &Context, expr: Expr<LVT>)
    -> (Vec<TransformRule>, Expr<LVT>)
{
    use ExprData::*;
    use TransformRule::*;

    match expr.data {
        LitVar { id, meta } =>
            match context.get_value(meta.dist_to_decl) {
                Some(value) if utils::is_elementary_expr(&value)
                    => (vec![SubstituteBasicValue], value.clone()),
                _ => (Vec::new(), Expr {
                    data: LitVar { id, meta },
                    meta: expr.meta
                })
            },

        LitConstructor { id, args } =>
            factorize::constructor(expr.meta, id, args),

        Call { caller, arg }
        if matches!(
            &context.shallow_resolve(caller.as_ref()).data,
            Function { .. }
        ) =>
            match context.shallow_resolve(caller.as_ref()).data.clone()
{
                Function { input, body } =>
                    (vec![EvalCall], Expr {
                        data: Binding {
                            var: input,
                            val: Box::new(arg.all_moved(0, 1)),
                            body
                        },
                        meta: expr.meta
                    }),
                _ => unreachable!()
            },
```

```
Match { expr, cases }
if eval::can_a_branch_be_chosen(context, expr.as_ref(), &cases)
=>

    (vec![EvalMatch],
        eval::chose_branch(context, *expr, cases)),

Binding { body, .. }
if !utils::is_id_used(1, &*body) =>
    (vec![DeleteUnusedBinding],
        body.all_moved(0, -1)),

// Take binding out of other binding's value
// take binding out of match branch
// take binding out of function body
// take binding out of call
// delete duplicate bindings
// check if bindings are available in hypotheses
// sort bindings
// sort functions

Binding { var, val, body } => {
    let (val_trans, val) = normalize_pass(
        &context.with_opt_associations(&vec![None]),
        *val
    );

    let (body_trans, body) = normalize_pass(
        &context.with_association(&Association {
            id: var.id.clone(),
            value: val.clone()
        }),
        *body
    );

    (
        val_trans.into_iter()
            .chain(body_trans.into_iter())
            .collect(),
        Expr {
            data: Binding {
                var,
                val: Box::new(val),
                body: Box::new(body)
            },
            meta: expr.meta
        }
    )
},

Function { input, body } => {
    let (trans, body) = normalize_pass(
        &context.with_opt_associations(&vec![None]),
        *body
    );
    (trans, Expr {
        data: Function { input, body: Box::new(body) },
        meta: expr.meta
    })
},

Call { caller, arg } => {
    let (caller_trans, caller) =
        normalize_pass(context, *caller);

    let (arg_trans, arg) =
        normalize_pass(context, *arg);

    (
        caller_trans.into_iter()
            .chain(arg_trans.into_iter())
            .collect(),
        Expr {
            data: Call {
                caller: Box::new(caller),
                arg: Box::new(arg)
            },
            meta: expr.meta
        }
    )
},
```

```
        Match { expr: patterne, cases } => {
            let (trans_patterne, patterne) =
                normalize_pass(context, *patterne);

            let (trans_cases_vec, cases): (Vec<_>, _) =
                cases
                .into_iter()
                .map(|MatchBranch { pattern, body }| {
                    let (trans_body, body) =
                        normalize_pass(
                            &context.with_opt_associations(
                                &vec![None; pattern.induced_offset()]
                            ),
                            body
                        );
                    (trans_body, MatchBranch { pattern, body })
                })
                .unzip();

            (
                trans_patterne.into_iter()
                    .chain(trans_cases_vec.into_iter().flatten())
                    .collect(),
                Expr {
                    data: Match {
                        expr: Box::new(patterne),
                        cases
                    },
                    meta: expr.meta
                }
            )
        }
    }
}


#[cfg(test)]
mod tests {
    use crate::process::expr_read_link_type_loosen;

    use super::*;
```

```
fn assert_normalize_pass_eq(
    constructors_id_arity: Vec<(String, usize)>,
    free_variable_ids: Vec<String>,
    original: &str,
    normal: &str
) {
    assert_eq!(
        normalize_pass(
            &Context::new(&[].into()),
            expr_read_link_type_loosen(
                constructors_id_arity.clone(),
                free_variable_ids.clone(),
                original
            )
        ).1,
        expr_read_link_type_loosen(
            constructors_id_arity,
            free_variable_ids,
            normal
        )
    );
}


#[test]
fn test_normalize_pass() {
    assert_normalize_pass_eq(
        vec![],
        vec!["x".to_string()],
        "x",
        "x"
    );

    assert_normalize_pass_eq(
        vec![
            ("Zero".to_string(), 0),
            ("Tuple".to_string(), 2)
        ],
        vec!["x".to_string()],
        "Tuple(x, Tuple(Zero, Zero)))",
        "let y = Tuple(Zero, Zero) in Tuple (x, y)"
    );

    assert_normalize_pass_eq(
        vec![],
        vec!["x".to_string()],
        "(fun u -> u) x",
        "let u = x in u"
    );
}


fn assert_normalized_eq(original: &str, normal: &str) {
    // let mut linker = Linker::new();

    todo!()
    // assert_eq!(
    //     normalized(
    //         linker.expr(expr(original.chars())).unwrap(),
    //         &mut linker
    //     ).link(),
    //     expr(normal.chars()).link()
    // );
}


// #[test]
// fn test_normalizing() {
//     assert_normalized_eq(
//         "(fun (a) -> fun (b) -> b) x y",
//         "y"
//     );

//     assert_normalized_eq(
//         "fun (b) -> let c = False in b",
//         "fun (b) -> b"
//     );

//     assert_normalized_eq(
//         "(fun (a) -> a) x",
    //         "x"
    //     );

    //     assert_normalized_eq(
    //         "Tuple (
    //             let y = z in
    //                 let x = y in
    //                 x,
    //             let b = a in
    //                 b
    //         )",
    //         "
    //         Tuple (z, a)
    //         "
    //     );
    // }
}
```

Fichier /src/analysis/solve/normal/sub.rs

```
use std::collections::HashMap;
use crate::lang::*;

use ExprData::*;


pub fn substituted(
    expr: Expr<Linked>,
    to_substitute: &HashMap<UId, Expr<Linked>>
) -> Expr<Linked>
{
    match expr.data {
        LitVar { id, meta } =>
            match to_substitute.get(&id) {
                None => LitVar { id, meta },
                Some(val) => val.data.clone()
            },

        LitConstructor { id, args } =>
            LitConstructor {
                id,
                args: args
                    .into_iter()
                    .map(|expr| substituted(expr, to_substitute))
                    .collect()
            },

        Binding { var, val, body } =>
            Binding {
                var,
                val: Box::new(substituted(*val, to_substitute)),
                body: Box::new(substituted(*body, to_substitute))
            },

        Function { input, body } =>
            Function {
                input,
                body: Box::new(substituted(*body, to_substitute))
            },

        Call { caller, arg } =>
            Call {
                caller: Box::new(substituted(*caller, to_substitute)),
                arg: Box::new(substituted(*arg, to_substitute))
            },

        Match { expr, cases } =>
            Match {
                expr: Box::new(substituted(*expr, to_substitute)),
                cases: cases
                    .into_iter()
                    .map(|MatchBranch { pattern, body }|
                        MatchBranch {
                            pattern,
                            body: substituted(body, to_substitute)
                        }
                    )
                    .collect()
            }
    }.into()
}


#[cfg(test)]
```

```
mod tests {
    use super::*;

    use crate::process::read::expr;


    // #[test]
    fn test_substituted() {
    }
}
```

Fichier /src/analysis/solve/normal/factorize.rs

```
use crate::{analysis::solve::data::{Association, TransformRule},
lang::*};
use super::utils;
use ExprData::*;


pub fn constructor(meta: LTNodeMeta, id: UId, args: Vec<Expr<LVT>>)
    -> (Vec<TransformRule>, Expr<LVT>)
{
    let new_ids: Vec<_> = args.iter()
        .map(|arg|
            if utils::is_elementary_expr(arg) {
                None
            } else {
                Some(utils::random_loid())
            }
        )
        .collect();

    let new_associations: Vec<_> = new_ids.iter()
        .cloned()
        .enumerate()
        .filter_map(|(i, id_opt)|
            id_opt.map(|id| (i, id))
        )
        .enumerate()
        .collect::<Vec<_>>()
        .into_iter()
        .rev()
        .map(|(local_offset, (i, id))|
            Association {
                id,
                value: args[i].clone()
                    .all_moved(0, (local_offset + 1) as isize)
            }
        )
        .collect();

    let global_offset = new_associations.len();
    let mut new_decl_dist = global_offset;
    let args = new_ids.into_iter()
        .enumerate()
        .map(|(i, opt_id)|
            match opt_id {
                None => args[i].clone()
                    .all_moved(0, global_offset as isize),
                Some(id) => {
                    let dist_to_decl = new_decl_dist;
                    new_decl_dist -= 1;
                    Expr {
                        data: LitVar {
                            id,
                            meta: LVTVarMeta {
                                dist_to_decl,
                                is_recursive: false
                            }
                        },
                        meta: args[i].meta.clone()
                    }
                }
            }
        )
        .collect();

    let trans = if new_associations.is_empty() {
        vec![]
    } else {
        vec![TransformRule::FactorizeConstr]
    };

    (trans, utils::fold_associations(
        new_associations.into_iter(),
        Expr {
            data: LitConstructor { id, args },
            meta: meta
        }
    ))
}
```

Fichier /src/analysis/solve/normal/utils.rs

```
use crate::{analysis::solve::data::Association, lang::*};


pub fn is_elementary_expr(expr: &Expr<LVT>) -> bool {
    use ExprData::*;
    match &expr.data {
        LitVar { .. } => true,
        LitConstructor { args, .. }
            if args.is_empty() => true,
        _ => false
    }
}


pub fn fold_associations<I>(associations: I, tail: Expr<LVT>) ->
Expr<LVT>
    where I: DoubleEndedIterator<Item = Association>
{
    associations
        .rev()
        .fold(
            tail,
            |result, Association { id, value }| {
                let meta = result.meta.clone();
                Expr {
                    data: ExprData::Binding {
                        var: Var { id, meta: value.meta.clone() },
                        val: Box::new(value),
                        body: Box::new(result)
                    },
                    meta
                }
            }
        )
}


pub fn random_loid() -> LoId {
    LoId {
        name: String::from("todo")
    }
}


pub fn is_id_used(id: DistToDecl, expr: &Expr<LVT>) -> bool {
    use ExprData::*;

    match &expr.data {
        LitVar { meta, .. } =>
            id == meta.dist_to_decl,

        LitConstructor { args, .. } =>
            args.iter()
                .any(|arg| is_id_used(id, arg)),

        Call { caller, arg } =>
            is_id_used(id, caller.as_ref())
            || is_id_used(id, arg.as_ref()),

        Binding { val, body, .. } =>
            is_id_used(id + 1, val.as_ref())
            || is_id_used(id + 1, body.as_ref()),

        Function { body, .. } =>
            is_id_used(id + 1, body.as_ref()),

        Match { expr, cases } =>
            is_id_used(id, expr.as_ref())
            || cases.iter()
                .any(|MatchBranch { pattern, body }|
                    is_id_used(id + pattern.induced_offset(), body)
                )
    }
}
```

Fichier /src/analysis/solve/normal/eval.rs

```
use crate::{analysis::solve::data::{Association, Context}, lang::*};

use ExprData::*;

use super::utils;


pub fn can_a_branch_be_chosen(
    context: &Context,
    expr: &Expr<LVT>,
    branches: &Vec<MatchBranch<LVT>>
) -> bool
{
    branches.into_iter()
        .any(|MatchBranch { pattern, .. }|
            fit(context, pattern, expr).is_some()
        )
}


pub fn chose_branch(
    context: &Context,
    expr: Expr<LVT>,
    branches: Vec<MatchBranch<LVT>>
) -> Expr<LVT>
{
    let (associations, body) = branches.into_iter()
        .filter_map(|MatchBranch { pattern, body }|
            fit(context, &pattern, &expr)
                .map(|assocs| (assocs, body))
        )
        .next()
        .unwrap();
    // todo!("maybe the offsets of the associations are not taken into
account (try add x y = add y x)");

    let associations = associations.into_iter()
        .enumerate()
        .map(|(i, Association { id, value })|
            Association {
                id,
                value: value.all_moved(0, i as isize + 1)
            }
        )
        .collect::<Vec<_>>();

    let body = body.all_moved(
        0,
        associations.len() as isize
    );

    utils::fold_associations(associations.into_iter(), body)
}


fn fit(context: &Context, pattern: &Pattern<LVT>, expr: &Expr<LVT>)
    -> Option<Vec<Association>>
{
    use PatternData as PD;

    match &pattern.data {
        PD::Var(Var { id, .. }) =>
            Some(vec![Association { id: id.clone(), value:
expr.clone() }]),

        PD::Constructor { id: pat_id, args: pat_args } =>
            match &context.shallow_resolve(expr).data {
                LitConstructor { id: expr_id, args: expr_args }
                if pat_id == expr_id =>
                    Some(
                        pat_args.into_iter()
                            .zip(expr_args.into_iter())
                            .map(|(pat, arg)|
                                fit(context, pat, arg)
                            )
                            .collect::<Option<Vec<_>>>()?
                            .into_iter()
                            .flat_map(|associations|
                                associations.into_iter()
                            )
                            .collect()
                    ),

                _ => None
            }
    }
}
```

Dossier /src/analysis/solve/data/

Fichier /src/analysis/solve/data/context.rs

```
use std::collections::VecDeque;

use crate::lang::*;
use super::sequent::*;


#[derive(Clone)]
pub struct Context<'a> {
    free_ids: &'a VecDeque<LoId>,
    associations: Vec<Option<&'a Association>>,
}


impl<'a> Context<'a> {
    pub fn new(free_ids: &'a VecDeque<LoId>) -> Self {
        Context {
            free_ids,
            associations: Vec::new()
        }
    }

    pub fn from_sequent(sequent: &'a Sequent) -> Self {
        Context {
            free_ids: &sequent.free_ids,
            associations: sequent.hypotheses.iter()
                .map(|association| Some(association))
                .collect(),
        }
    }

    pub fn shallow_resolve(&self, expr: &Expr<LVT>) -> Expr<LVT> {
        use ExprData::*;
```

```
    match &expr.data {
        LitVar { meta, .. }
        if !meta.is_recursive =>
            self.get_value(meta.dist_to_decl)
                .unwrap_or_else(|| expr.clone()),

        _ => expr.clone()
    }
}

pub fn is_free_id(&self, dist_to_decl: DistToDecl) -> bool {
    let max_binding_dist = self.associations.len();
    dist_to_decl > max_binding_dist
}

pub fn get_id(&self, dist_to_decl: DistToDecl)
    -> Option<LoId>
{
    let max_binding_dist = self.associations.len();
    let max_free_id_dist = max_binding_dist + self.free_ids.len();

    if dist_to_decl == 0 {
        None
    } else if dist_to_decl <= max_binding_dist {
        self.associations[max_binding_dist - dist_to_decl]
            .map(|assoc| assoc.id.clone())
    } else if dist_to_decl <= max_free_id_dist {
        Some(self.free_ids[max_free_id_dist - dist_to_decl].clone())
    } else {
        None
    }
}

pub fn get_value(&self, dist_to_decl: DistToDecl)
    -> Option<Expr<LVT>>
{
    let max_binding_dist = self.associations.len();
    let max_free_id_dist = max_binding_dist + self.free_ids.len();

    if dist_to_decl == 0 {
            None
        } else if dist_to_decl <= max_binding_dist {
            let index = max_binding_dist - dist_to_decl;
            let opt_value = self.associations[index]
                .map(|assoc| &assoc.value);

            match opt_value {
                Some(Expr { data: ExprData::LitVar { meta, .. }, .. })
                if !meta.is_recursive =>
                    Context {
                        free_ids: self.free_ids,
                        associations: self.associations[0..(index - 1)].to_vec()
                    }.get_value(meta.dist_to_decl),
                _ => None
            }.or_else(|| opt_value.cloned())
                .map(|value|
                    value.all_moved(0, dist_to_decl as isize - 1)
                )
        } else if dist_to_decl <= max_free_id_dist {
            None
        } else {
            None
        }
    }

    pub fn with_opt_associations(&self, associations: &'a Vec<Option<Association>>) -> Self {
        Context {
            free_ids: self.free_ids,
            associations: self.associations.iter()
                .cloned()
                .chain(
                    associations.iter()
                        .map(|assoc| assoc.as_ref())
                )
                .collect()
        }
    }

    pub fn with_associations(&self, associations: &'a Vec<Association>) -> Self {
        Context {
            free_ids: self.free_ids,
            associations: self.associations.iter()
                .cloned()
                .chain(associations.iter()
                    .map(|assoc| Some(assoc))
                )
                .collect()
        }
    }

    pub fn with_association(&self, association: &'a Association) -> Self {
        Context {
            free_ids: self.free_ids,
            associations: self.associations.iter()
                .cloned()
                .chain([Some(association)])
                .collect()
        }
    }
}
```

Fichier /src/analysis/solve/data/counter.rs

```
use std::{collections::HashMap, fmt::Display};

use crate::lang::*;


#[derive(Debug)]
pub struct CounterEx {
    pub vals: HashMap<UId, Expr<Linked>>,
    pub left: Expr<Linked>,
    pub right: Expr<Linked>,
}


impl Display for CounterEx {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        f.debug_map()
            .entries(self.vals.iter())
            .finish()?;
        write!(f, "\n")?;

        writeln!(f, "{:?} = {:?}", self.left, self.right)
    }
}
```

Fichier /src/analysis/solve/data/mod.rs

```
mod context;
mod proof;
mod sequent;
mod counter;

pub use context::*;
pub use proof::*;
pub use sequent::*;
pub use counter::*;
```

Fichier /src/analysis/solve/data/proof.rs

```
use std::fmt::Display;

use crate::lang::*;
use super::sequent::*;


#[derive(Debug)]
pub enum Rule {
    Equal,
    Transform(Vec<TransformRule>, Box<Proof>),
    Hypothesis(Box<Proof>),
    MatchElimination(Vec<Proof>)
}


#[derive(Debug)]
pub enum TransformRule {
    SubstituteBasicValue,
    EvalCall,
    EvalMatch,
    DeleteUnusedBinding,
    FactorizeConstr,
    Weakening
}


#[derive(Debug)]
pub struct Proof {
    pub sequent: Sequent,
    pub rule: Rule
}


impl TypstRepr for Proof {
    fn typst_repr(&self) -> String {
        format!(
            "(\"proof\", {}, {})",
            self.sequent.typst_repr(),
            self.rule.typst_repr()
        )
    }
}


impl TypstRepr for Rule {
    fn typst_repr(&self) -> String {
        use Rule::*;

        match &self {
            Equal => format!("(\"rule\", \"equal\")"),
            Hypothesis(_) => todo!(),
            Transform(trans_rule, proof) =>
                format!(
                    "(\"rule\", \"trans\", {}, {})",
                    trans_rule.typst_repr(),
                    proof.typst_repr()
                ),
            MatchElimination(cases) =>
                format!(
                    "(\"rule\", \"match_e\", {})",
                    cases.typst_repr()
                )
        }
    }
}
```

```
impl TypstRepr for TransformRule {
    fn typst_repr(&self) -> String {
        use TransformRule::*;
        String::from(match self {
            SubstituteBasicValue => "\"sub_basic_value\"",
            EvalCall => "\"eval_call\"",
            EvalMatch => "\"eval_match\"",
            DeleteUnusedBinding => "\"del_bind\"",
            FactorizeConstr => "\"fact_constr\"",
            Weakening => "\"weak\"",
        })
    }
}


impl Display for Proof {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        self.fmt_with_depth(f, 0)
    }
}


impl Proof {
    fn fmt_with_depth(&self, f: &mut std::fmt::Formatter<'_>, depth:
usize)
        -> std::fmt::Result
    {
        use Rule::*;

        match &self.rule {
            Equal => {
                print_tab(f, depth)?;
                writeln!(f, "□ {:?} = {:?}", self.sequent.left,
self.sequent.right)
            },

            Hypothesis(next) => {
                print_tab(f, depth)?;
                writeln!(f, "┝ ...")?;
                next.fmt_with_depth(f, depth)
            },

            Transform(_, next) => {
                print_tab(f, depth)?;
                writeln!(f, "┇")?;
                next.fmt_with_depth(f, depth)
            },

            MatchElimination(cases) => {
                for case in cases {
                    print_tab(f, depth + 1)?;
                    write!(f, "\n")?;

                    print_tab(f, depth)?;
                    write!(f, "├─┬ ")?;
                    f.debug_map()
                        .entries(
                            case.sequent.hypotheses.iter()
                                .map(|Association { id, value }|
                                    (id, value)
                                )
                        )
                        .finish()?;
                    write!(f, "\n")?;

                    case.fmt_with_depth(f, depth + 1)?;
                }

                Ok(())
            }
        }
    }
}


fn print_tab(f: &mut std::fmt::Formatter<'_>, size: usize) ->
std::fmt::Result {
    for _ in 0..size {
        write!(f, "│ ")?
    }
    Ok(())
}
```

Fichier /src/analysis/solve/data/sequent.rs

```
use std::collections::VecDeque;

use crate::lang::*;


#[derive(Debug, Clone)]
pub struct Association {
    pub id: LoId,
    pub value: Expr<LVT>
}


#[derive(Debug, Clone)]
pub struct Sequent {
    pub free_ids: VecDeque<LoId>,
    pub hypotheses: VecDeque<Association>,
    pub left: Expr<LVT>,
    pub right: Expr<LVT>
}


impl Sequent {
    pub fn from_property(hypotheses: Vec<Association>, property:
Property<LVT>)
        -> Sequent
    {
        let free_ids: VecDeque<_> = property.vars.into_iter()
            .map(|var| var.id)
            .collect();

        let mut left = property.left;
        let mut right = property.right;

        move_free_ids_above_hypotheses(hypotheses.len(), free_ids.len(),
&mut left);
        move_free_ids_above_hypotheses(hypotheses.len(), free_ids.len(),
&mut right);

        Sequent {
            free_ids,
            hypotheses: hypotheses.into(),
            left,
            right,
        }
    }

    pub fn use_free_id(&mut self, free_id: DistToDecl, unliked_value:
Expr<LVT>)
    {
        let free_ids_bottom = self.hypotheses.len() + 1;
        self.left.move_declaration_dist(free_id, free_ids_bottom);
        self.right.move_declaration_dist(free_id, free_ids_bottom);

        let free_id_index = self.hypotheses.len() + self.free_ids.len()
- free_id;
        let id = self.free_ids.remove(free_id_index).unwrap();

        let value = self.link_value_located_at_top(unliked_value);
        self.hypotheses.push_front(Association { id, value });
    }

    fn link_value_located_at_top(&mut self, unliked_value: Expr<LVT>)
        -> Expr<LVT>
    {
        use ExprData::*;

        let data = match unliked_value.data {
            LitVar { id, meta } => {
                self.free_ids.push_front(id.clone());
                LitVar {
                    id,
                    meta: LVTVarMeta {
                        dist_to_decl: self.free_ids.len(),
                        is_recursive: meta.is_recursive
                    }
                }
            },

            LitConstructor { id, args } =>
```

```
                LitConstructor {
                    id,
                    args: args.into_iter()
                        .map(|arg|
                            self.link_value_located_at_top(arg)
                        )
                        .collect()
                },

            _ => unreachable!("expected a value produced by
pat::make_into")
        };

        Expr { data, meta: unliked_value.meta }
    }
}


fn move_free_ids_above_hypotheses(
    hypotheses_len: usize,
    free_ids_len: usize,
    expr: &mut Expr<LVT>
) {
    let move_dist = hypotheses_len + free_ids_len;
    for _ in 0..free_ids_len {
        expr.move_declaration_dist(1, move_dist);
    }
}


impl TypstRepr for Sequent {
    fn typst_repr(&self) -> String {
        format!(
            "(\"sequent\", {}, {}, {}, {})",
            self.free_ids.typst_repr(),
            self.hypotheses.typst_repr(),
            self.left.typst_repr(),
            self.right.typst_repr()
        ).to_string()
    }
}
```

```
impl TypstRepr for Association {
    fn typst_repr(&self) -> String {
        format!(
            "(\"assoc\", {}, {})",
            self.id.typst_repr(),
            self.value.typst_repr(),
        ).to_string()
    }
}
```

Dossier /src/process/

Fichier /src/process/read.rs

```
use crate::lang::*;
use super::lex::*;
use super::parse::*;


pub fn lex_and_unwrap<I: IntoIterator<Item = char>>(input: I) -> impl
Iterator<Item = Token> {
    lex(input)
        .map(|res_tok|
            match res_tok {
                Ok(tok) => tok,
                Err(UnrecognizedToken { line, column, text }) =>
                    panic!(
                        "Lexing error at {}:{} : unrecognized token
'{}'.",
                        line, column,
                        text
                    )
            }
        )
}


fn ast_node_unwrap<N>(node: Result<N, ParsingErr>) -> N {
    use ParsingErr::*;

    match node {
        Ok(node) => node,

        Err(UnexpectedToken { expected, got }) => {
            panic!(
                "Parsing error at {}:{} : expected '{}' got '{}'.",
                got.line, got.column, expected, got.data
            );
        }
    }
}


pub fn ast<I: IntoIterator<Item = char>>(input: I) -> AST<Raw> {
    ast_node_unwrap(Parser::new(lex_and_unwrap(input)).ast())
}

pub fn type_def<I: IntoIterator<Item = char>>(input: I) -> TypeDef<Raw>
{
    ast_node_unwrap(Parser::new(lex_and_unwrap(input)).type_def())
}

pub fn expr<I: IntoIterator<Item = char>>(input: I) -> Expr<Raw> {
    ast_node_unwrap(Parser::new(lex_and_unwrap(input)).expr())
}

pub fn property<I: IntoIterator<Item = char>>(input: I) -> Property<Raw>
{
    ast_node_unwrap(Parser::new(lex_and_unwrap(input)).property())
}

pub fn pattern<I: IntoIterator<Item = char>>(input: I) -> Pattern<Raw> {
    ast_node_unwrap(Parser::new(lex_and_unwrap(input)).pattern())
}

pub fn var<I: IntoIterator<Item = char>>(input: I) -> Var<Raw> {
    ast_node_unwrap(Parser::new(lex_and_unwrap(input)).variable())
}
```

Fichier /src/process/mod.rs

```
mod link;
mod loose;
pub mod read;
mod lex;
mod parse;
mod typ;
mod utils;

use utils::IdGenerator;
pub use loose::LooseLinker;
pub use link::{ Linker, LinkingErr, LinkableKind };
pub use typ::Typer;


use crate::lang::*;


pub fn expr_read_link_type_loosen(
    constructors_id_arity: Vec<(String, usize)>,
    free_variable_ids: Vec<String>,
    expr: &str,
) -> Expr<LVT>
{
    let mut linker = Linker::new();

    for (constr, _) in &constructors_id_arity {
        linker.constructor_id_create(constr.clone()).unwrap();
    }
    for free_var_id in &free_variable_ids {
        linker.var_id_create(free_var_id.clone()).unwrap();
    }

    let raw = read::expr(expr.chars());
    let linked = linker.expr(raw)
        .unwrap();

    let constructors_id_arity = constructors_id_arity.into_iter()
        .map(|(id, arity)|
            (linker.constructor_id_get(id).unwrap(), arity)
        )
        .collect();
    let free_variable_ids = free_variable_ids.into_iter()
        .map(|id| linker.var_id_get(id).unwrap())
        .collect();
    let typed = linked.type_alone(constructors_id_arity,
free_variable_ids);
    typed.loose_link_alone()
}
```

Dossier /src/process/lex/

Fichier /src/process/lex/lexer.rs

```
use std::iter::{Iterator, Peekable};
use super::token::*;


// TODO: fix broken line & column error messages
pub fn lex<I: IntoIterator<Item = char>>(input: I)
    -> impl Iterator<Item = Result<Token, UnrecognizedToken>>
{
    Lexer::new(input.into_iter())
}


#[derive(Debug)]
pub struct  UnrecognizedToken {
    pub line: usize,
    pub column: usize,
    pub text: String
}


struct Lexer<I: Iterator<Item = char>> {
    line: usize,
    column: usize,
    input: Peekable<I>,
}


impl<I: Iterator<Item = char>> Iterator for Lexer<I> {
    type Item = Result<Token, UnrecognizedToken>;

    fn next(&mut self) -> Option<Self::Item> {
        self.consume_eventual_whitespaces();
        self.consume_token()
            .or(Some(Ok(Token { line: 0, column: 0, data:
TokenData::EOF })))
    }
}


impl<I: Iterator<Item = char>> Lexer<I> {

    fn new(iter: I) -> Lexer<I> {
        Lexer {
            line: 1,
            column: 1,
            input: iter.into_iter().peekable()
        }
    }

    fn peek(&mut self) -> Option<char> {
        self.input.peek().map(|&c| c)
    }

    fn next_alt(&mut self) -> Option<char> {
        let c_opt = self.input.next();

        if let Some(c) = c_opt {
            if c == '\n' {
                self.line += 1;
                self.column = 1;
            } else {
                self.column += 1;
            }
        }

        c_opt
    }

    fn consume_eventual_whitespaces(&mut self) {
        while self.peek().is_some_and(|c| c.is_ascii_whitespace()) {
            self.next_alt();
        }
    }
```

```
  fn consume_token(&mut self) -> Option<Result<Token,
UnrecognizedToken>> {
    use TokenData::*;

    let data = match self.peek()? {
      '&' => self.single_char(AMP),
      '.' => self.single_char(DOT),
      ',' => self.single_char(COMMA),
      '*' => self.single_char(STAR),
      ':' => self.single_char(COLUMN),
      '=' => self.single_char(EQUAL),
      '|' => self.single_char(PIPE),
      '#' => self.single_char(HASHTAG),
      '(' => self.single_char(LPAREN),
      ')' => self.single_char(RPAREN),
      ';' => self.consume(";;").and(Ok(DBLSEMICOL)),
      '-' => self.consume("->").and(Ok(ARROW)),
      _ => self.word()
    };

    Some(data.map(|d| Token {
      line: self.line,
      column: self.column,
      data: d
    }))
  }

  fn single_char(&mut self, data: TokenData) -> Result<TokenData,
UnrecognizedToken> {
    self.next_alt();
    Ok(data)
  }

  fn consume(&mut self, expected: &str) -> Result<(),
UnrecognizedToken> {
    let mut unrecognized = UnrecognizedToken {
      line: self.line,
      column: self.column,
      text: String::new()
    };

    for ch in expected.chars() {
      if self.peek().is_none() {
        return Err(unrecognized);
      }
      unrecognized.text.push(self.peek().unwrap());

      if ch != self.peek().unwrap() {
        return Err(unrecognized);
      }

      self.next_alt();
    }

    Ok(())
  }

  fn word(&mut self) -> Result<TokenData, UnrecognizedToken> {
    assert!(self.peek().is_some());
    let ch = self.peek().unwrap();

    if ch == '\'' {
      self.generic()
    } else if ch.is_ascii_lowercase() || ch == '_' {
      self.variable_of_keyword()
    } else if ch.is_ascii_uppercase() {
      self.constructor()
    } else {
      Err(UnrecognizedToken {
        line: self.line, column: self.column,
        text: String::from(ch)
      })
    }
  }

  fn generic(&mut self) -> Result<TokenData, UnrecognizedToken> {
    assert!(self.peek().is_some());
    assert_eq!(self.peek().unwrap(), '\'');

    let mut unrecognized = UnrecognizedToken {
      line: self.line, column: self.column,
      text: String::new()
```

```
        };

        unrecognized.text.push('\'');
        self.next_alt();

        if self.peek().is_none() {
            return Err(unrecognized);
        }

        let ch = self.peek().unwrap();
        unrecognized.text.push(ch);
        if !(ch.is_ascii_lowercase() || ch == '_') {
            return Err(unrecognized)
        }
        self.next_alt();

        while self.peek()
            .filter(|&ch|
                ch.is_ascii_lowercase()
                || ch.is_ascii_digit()
                || ch == '_' )
            .is_some()
        {
            unrecognized.text.push(self.peek().unwrap());
        }

        Ok(TokenData::GenericIdent(unrecognized.text))
    }

    fn variable_of_keyword(&mut self) -> Result<TokenData,
UnrecognizedToken> {
        assert!(self.peek().is_some());
        let ch = self.peek().unwrap();
        assert!(ch.is_ascii_lowercase() || ch == '_');

        let mut unrecognized = UnrecognizedToken {
            line: self.line, column: self.column,
            text: String::new()
        };

        unrecognized.text.push(ch);
        self.next_alt();

        while self.peek()
            .filter(|&ch|
                ch.is_ascii_lowercase()
                || ch.is_ascii_digit()
                || ch == '_'
                || ch == '\'')
            .is_some()
        {
            unrecognized.text.push(self.peek().unwrap());
            self.next_alt();
        }

        use TokenData::*;

        Ok(match unrecognized.text.as_str() {
            "let" => LET,
            "in" => IN,
            "fun" => FUN,
            "match" => MATCH,
            "with" => WITH,
            "type" => TYPE,
            "of" => OF,
            _ => SnakeIdent(unrecognized.text),
        })
    }

    fn constructor(&mut self) -> Result<TokenData, UnrecognizedToken> {
        assert!(self.peek().is_some());
        let ch = self.peek().unwrap();
        assert!(ch.is_ascii_uppercase());

        let mut unrecognized = UnrecognizedToken {
            line: self.line, column: self.column,
            text: String::new()
        };

        unrecognized.text.push(ch);
        self.next_alt();
        while self.peek()
            .filter(|ch| ch.is_ascii_alphanumeric())
            .is_some()
        {
            unrecognized.text.push(self.peek().unwrap());
            self.next_alt();
        }

        Ok(TokenData::PascalIdent(unrecognized.text))
    }

}
```

Fichier /src/process/lex/mod.rs

```
pub mod lexer;
pub mod token;

pub use token::{Token, TokenData};
pub use lexer::{lex, UnrecognizedToken};
```

Fichier /src/process/lex/token.rs

```
#[derive(Debug, PartialEq, Clone)]
pub enum TokenData {
    AMP, DOT,
    ARROW, COMMA, STAR,
    COLUMN, DBLSEMICOL, EQUAL,
    PIPE, HASHTAG,
    LPAREN, RPAREN,

    LET, IN, FUN,
    MATCH, WITH,
    TYPE, OF,

    EOF,

    SnakeIdent(String),
    PascalIdent(String),
    GenericIdent(String),
}


#[derive(Debug, Clone)]
pub struct Token {
    pub line: usize,
    pub column: usize,
    pub data: TokenData
}


impl std::fmt::Display for TokenData {

    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        use TokenData::*;

        write!(f, "{}",
            match self {
                DOT        => ".",
                AMP        => "&",
                ARROW      => "->",
                COMMA      => ",",
                STAR       => "*",
                COLUMN     => ":",
                DBLSEMICOL => ";;",
                EQUAL      => "=",
                PIPE       => "|",
                HASHTAG    => "#",
                LPAREN     => "(",
                RPAREN     => ")",

                LET   => "let",
                IN    => "in",
                FUN   => "fun",
                MATCH => "match",
                WITH  => "with",
                TYPE  => "type",
                OF    => "of",

                EOF => "",

                SnakeIdent(ident)  => ident,
                PascalIdent(ident)   => ident,
                GenericIdent(ident) => ident,
            }
        )
    }

}
```

Dossier /src/process/link/

Fichier /src/process/link/ast.rs

```
use crate::lang::{kind::RawId, *};

use super::{LinkResult, LinkableKind, Linker};


impl<K: LinkableKind> Linker<K>
where
    K::VariableId: RawId,
    K::ConstructorId: RawId,
    K::TypeId: RawId,
    K::GenericId: RawId,
{
    pub fn ast(&mut self, raw: AST<K>) -> LinkResult<K, AST<Linked>> {
        let type_defs = self.type_defs(raw.type_defs)?;
        let statements = self.statements(raw.statements)?;

        Ok(AST {
            type_defs,
            statements,
        })
    }
}
```

Fichier /src/process/link/expr.rs

```
use super::*;


use TraceNode::*;


impl<K: LinkableKind> Linker<K>
where
    K::VariableId: RawId,
    K::ConstructorId: RawId,
    K::TypeId: RawId,
    K::GenericId: RawId,
{
    pub fn expr(&mut self, raw: Expr<K>) -> LinkResult<K, Expr<Linked>>
{
        use ExprData::*;

        Ok(match raw.data {
            Binding { var: raw_var, val: raw_val, body: raw_body } => {
                let (var, val, body);
                id_scope!(self.var_ids, {
                    trace!(self.trace, BindingVar, {
                        var = self.variable(raw_var)?;
                    });

                    // Done after registering var for recursive reasons
                    trace!(self.trace, BindingVal, {
                        val = Box::new(self.expr(*raw_val)?);
                    });

                    trace!(self.trace, BindingBody, {
                            body = Box::new(self.expr(*raw_body)?);
                    });
                });

                Binding { var, val, body }
            }

            Function { input: raw_input, body: raw_body } => {
                let (input, body);

                id_scope!(self.var_ids, {
                    id_scope!(self.generic_ids, {
                        trace!(self.trace, FunctionInput, {
                            input = self.variable(raw_input)?;
                        });

                        trace!(self.trace, FunctionBody, {
                            body = Box::new(self.expr(*raw_body)?);
                        });
                    });
                });
                Function { input, body }
            },

            Match { expr: raw_expr, cases: raw_cases } => {
                let (expr, cases);
                trace!(self.trace, MatchExpr, {
                    expr = Box::new(self.expr(*raw_expr)?);
                });

                trace!(self.trace, MatchCases, {
                    cases = raw_cases
                        .into_iter()
                        .map(|b| self.match_branch(b))
                        .collect::<LinkResult<_, Vec<_>>>()?;
                });

                Match { expr, cases }
            },

            Call { caller: raw_caller, arg: raw_arg } => {
                let (caller, arg);

                trace!(self.trace, CallCaller, {
                    caller = Box::new(self.expr(*raw_caller)?);
                });

                trace!(self.trace, CallArg, {
                    arg = Box::new(self.expr(*raw_arg)?);
                });

                Call { caller, arg }
            },

            LitVar { id: raw_id, .. } => {
                let id;
                trace!(self.trace, LitVarId, {
                    id = self.var_id_get(raw_id)?;
                });
                LitVar { id, meta: () }
            },

            LitConstructor { id: raw_id, args: raw_args } => {
                let (id, args);

                trace!(self.trace, LitConstructorId, {
                    id = self.constructor_id_get(raw_id)?;
                });

                trace!(self.trace, LitConstructorArgs, {
                    args = raw_args
                        .into_iter()
                        .map(|e| self.expr(e))
                        .collect::<LinkResult<_, Vec<_>>>()?;
                });

                LitConstructor { id, args }
            }
        }.into())
    }

    fn match_branch(&mut self, raw: MatchBranch<K>)
        -> LinkResult<K, MatchBranch<Linked>>
    {
        let (pattern, body);
        id_scope!(self.var_ids, {
            id_scope!(self.generic_ids, {
                trace!(self.trace, MatchBranchPattern, {
                    pattern = self.pattern(raw.pattern)?;
                });

                trace!(self.trace, MatchBranchBody, {
                    body = self.expr(raw.body)?;
                });
            });
        });

        Ok(MatchBranch { pattern, body })
    }
}
```

Fichier /src/process/link/mod.rs

```
use std::collections::HashMap;
```

```rust
use crate::lang::{kind::RawId, *};

use super::{
    IdGenerator,
    utils::*
};

macro_rules! id_scope {
    ($stack: expr, $code: block) => {
        {
            $stack.push_empty_scope();
            $code;
            $stack.pop_scope();
        }
    };
}

macro_rules! trace {
    ($tracer: expr, $node: ident, $code: block) => {
        {
            $tracer.enter($node);
            $code;
            $tracer.exit();
        }
    };
}

mod ast;
mod expr;
mod pat;
mod prop;
mod stmt;
mod typ;


pub trait LinkableKind: Kind
where
    Self::VariableId: RawId,
    Self::ConstructorId: RawId,
    Self::TypeId: RawId,
    Self::GenericId: RawId,
{}


impl<K: Kind> LinkableKind for K
where
    Self::VariableId: RawId,
    Self::ConstructorId: RawId,
    Self::TypeId: RawId,
    Self::GenericId: RawId,
{}


impl<K: LinkableKind> Expr<K>
where
    K::VariableId: RawId,
    K::ConstructorId: RawId,
    K::TypeId: RawId,
    K::GenericId: RawId,
{
    pub fn link_alone(
        self,
        constructor_ids: Vec<K::ConstructorId>,
        free_variable_ids: Vec<K::VariableId>
    ) -> Expr<Linked>
    {
        let mut linker = Linker::new();

        for constr in constructor_ids {
            linker.constructor_id_create(constr).unwrap();
        }
        for free_var_id in free_variable_ids {
            linker.var_id_create(free_var_id).unwrap();
        }

        linker.expr(self)
            .unwrap()
    }
}
```

```
#[derive(Clone)]
pub struct Linker<K: LinkableKind>
where
    K::VariableId: RawId,
    K::ConstructorId: RawId,
    K::TypeId: RawId,
    K::GenericId: RawId,
{
    var_ids: ScopeStack<K::VariableId, usize>,
    generic_ids: ScopeStack<K::GenericId, usize>,
    constructor_ids: HashMap<K::ConstructorId, usize>,
    type_ids: HashMap<K::TypeId, usize>,

    var_counter: IdGenerator,
    generic_counter: IdGenerator,
    constructor_counter: IdGenerator,
    type_counter: IdGenerator,

    trace: Trace,
}


#[derive(Debug)]
pub enum LinkingErr<K: LinkableKind>
where
    K::VariableId: RawId,
    K::ConstructorId: RawId,
    K::TypeId: RawId,
    K::GenericId: RawId,
{
    VariableIdAlreadyInScope {
        id: K::VariableId,
        trace: Trace,
    },
    UndefinedVariableId {
        id: K::VariableId,
        trace: Trace,
    },

    ConstructorIdAlreadyInScope {
        id: K::ConstructorId,
        trace: Trace,
    },
    UndefinedConstructorId {
        id: K::ConstructorId,
        trace: Trace,
    },

    TypeIdAlreadyInScope {
        id: K::TypeId,
        trace: Trace,
    },
    UndefinedTypeId {
        id: K::TypeId,
        trace: Trace,
    },

    GenericIdAlreadyInScope {
        id: K::GenericId,
        trace: Trace,
    },
    UndefinedGenericId {
        id: K::GenericId,
        trace: Trace,
    },
}

type LinkResult<K, T> = Result<T, LinkingErr<K>>;



impl<K: LinkableKind> Linker<K>
where
    K::VariableId: RawId,
    K::ConstructorId: RawId,
    K::TypeId: RawId,
    K::GenericId: RawId,
{
    pub fn new() -> Linker<K> {
        let mut var_ids = ScopeStack::new();
        let mut generic_ids = ScopeStack::new();
        var_ids.push_empty_scope();
        generic_ids.push_empty_scope();

        let constructor_ids = HashMap::new();
        let type_ids = HashMap::new();

        Linker {
            var_ids,
            generic_ids,
            constructor_ids,
            type_ids,

            var_counter: IdGenerator::new(),
            generic_counter: IdGenerator::new(),
            constructor_counter: IdGenerator::new(),
            type_counter: IdGenerator::new(),

            trace: Trace::new(),
        }
    }

    pub fn var_id_get(&mut self, key: K::VariableId) -> LinkResult<K,
UId> {
        match self.var_ids.get(&key) {
            Some(&val) => Ok(UId {
                id: val,
                name: key.name()
            }),
            None => Err(LinkingErr::UndefinedVariableId {
                id: key,
                trace: self.trace.clone()
            })
        }
    }

    pub fn var_id_create(&mut self, key: K::VariableId) -> LinkResult<K,
UId> {
        match self.var_ids.get(&key) {
            None => {
                let val = self.var_counter.gen();
                self.var_ids.insert(key.clone(), val);
                Ok(UId { id: val, name: key.name() })
            }
            Some(_) => Err(LinkingErr::VariableIdAlreadyInScope {
                id: key,
                trace: self.trace.clone()
            })
        }
    }

    fn generic_id_get_or_create(&mut self, key: K::GenericId) -> UId {
        let id = match self.generic_ids.get(&key) {
            Some(&val) => val,
            None => {
                let val = self.generic_counter.gen();
                self.generic_ids.insert(key.clone(), val);
                val
            }
        };
        UId { id, name: key.name() }
    }

    fn generic_id_create(&mut self, key: K::GenericId) -> LinkResult<K,
UId> {
        match self.generic_ids.get(&key) {
            None => {
                let val = self.generic_counter.gen();
                self.generic_ids.insert(key.clone(), val);
                Ok(UId { id: val, name: key.name() })
            }
            Some(_) => Err(LinkingErr::GenericIdAlreadyInScope {
                id: key,
                trace: self.trace.clone()
            })
        }
    }

    fn type_id_get(&self, key: K::TypeId) -> LinkResult<K, UId> {
        match self.type_ids.get(&key) {
            Some(&val) => Ok(UId {
                id: val,
                name: key.name()
            }),
            None => Err(LinkingErr::UndefinedTypeId {
                id: key,
                trace: self.trace.clone()
            })
        }
    }

    fn type_id_create(&mut self, key: K::TypeId) -> LinkResult<K, UId> {
        match self.type_ids.get(&key) {
            None => {
                let val = self.type_counter.gen();
                self.type_ids.insert(key.clone(), val);
                Ok(UId { id: val, name: key.name() })
            }
            Some(_) => Err(LinkingErr::TypeIdAlreadyInScope {
                id: key,
                trace: self.trace.clone()
            })
        }
    }

    pub fn constructor_id_get(&mut self, key: K::ConstructorId)
        -> LinkResult<K, UId>
    {
        match self.constructor_ids.get(&key) {
            Some(&val) => Ok(UId {
                id: val,
                name: key.name()
            }),
            None => Err(LinkingErr::UndefinedConstructorId {
                id: key,
                trace: self.trace.clone()
            })
        }
    }

    pub fn constructor_id_create(&mut self, key: K::ConstructorId)
        -> LinkResult<K, UId>
    {
        match self.constructor_ids.get(&key) {
            None => {
```

```
                let val = self.constructor_counter.gen();
                self.constructor_ids.insert(key.clone(), val);
                Ok(UId { id: val, name: key.name() })
            }
            Some(_) => Err(LinkingErr::ConstructorIdAlreadyInScope {
                id: key,
                trace: self.trace.clone()
            })
        }
    }

    // pub fn create_unassociated(&mut self) -> UId {
    //     UId::unnamed(self.var_counter.gen())
    // }

    // fn create_just_bellow(&mut self, key: K::Id) -> LinkResult<K,
UId> {
    //     match self.var_ids.get(&key) {
    //         None => {
    //             let val = self.var_counter.gen();
    //             self.var_ids.insert_bellow(1, key.clone(), val);
    //             Ok(UId { id: val, name: key.to_string() })
    //         }
    //         Some(_) => Err(LinkingErr::IdAlreadyInScope {
    //             id: key.to_string(),
    //             trace: self.trace.clone()
    //         })
    //     }
    // }
}


// impl Linker<Linked>
// {
//     pub fn reserve_id(&mut self, key: UId) -> LinkResult<()> {
//         match self.var_ids.get(&key) {
//             None => {
//                 // TODO: handle error
//                 self.var_counter.reserve(key.id).unwrap();
//                 self.var_ids.insert(key.clone(), key.id);
//                 Ok(())
//              }
//              Some(_) => Err(LinkingErr::IdAlreadyInScope {
//                  id: key.to_string(),
//                  trace: self.trace.clone()
//              })
//          }
//      }
// }
```

Fichier /src/process/link/pat.rs

```
use super::*;


use TraceNode::*;


impl<K: LinkableKind> Linker<K>
where
    K::VariableId: RawId,
    K::ConstructorId: RawId,
    K::TypeId: RawId,
    K::GenericId: RawId,
{
    pub fn pattern(&mut self, raw: Pattern<K>) -> LinkResult<K,
Pattern<Linked>> {
        use PatternData::*;

        Ok(match raw.data {
            Var(raw_var) => {
                let var;
                trace!(self.trace, PatternVar, {
                    var = self.variable(raw_var)?;
                });
                Var(var).into()
            },

            Constructor { id: raw_id, args: raw_args } => {
                let (id, args);
                trace!(self.trace, ConstructorId, {
                    id = self.constructor_id_get(raw_id)?;
                });
                trace!(self.trace, ConstructorArgs, {
                    args = raw_args
                        .into_iter()
                        .map(|p| self.pattern(p))
                        .collect::<LinkResult<_,
Vec<Pattern<Linked>>>>()?;
                });

                Constructor { id, args }.into()
            }
        })
    }

    pub fn variable(&mut self, raw: Var<K>) -> LinkResult<K,
Var<Linked>> {
        let id;

        trace!(self.trace, VarId, {
            id = self.var_id_create(raw.id)?;
        });

        Ok(Var { id, meta: () })
    }
}
```

Fichier /src/process/link/prop.rs

```
use crate::lang::kind::RawId;
use crate::lang::*;

use super::LinkableKind;
use super::{Linker, LinkResult, TraceNode};

use TraceNode::*;


impl<K: LinkableKind> Linker<K>
where
    K::VariableId: RawId,
    K::ConstructorId: RawId,
    K::TypeId: RawId,
    K::GenericId: RawId,
{
    pub fn property(&mut self, raw: Property<K>) -> LinkResult<K,
Property<Linked>> {
        let (vars, left, right);

        id_scope!(self.var_ids, {
            trace!(self.trace, PropertyVars, {
                vars = raw.vars
                    .into_iter()
                    .map(|var| self.variable(var))
                    .collect::<LinkResult<_, _>>()?;
            });

            trace!(self.trace, PropertyLeft, {
                left = self.expr(raw.left)?;
            });
            trace!(self.trace, PropertyRight, {
                right = self.expr(raw.right)?;
            });
        });

        Ok(Property { vars, left, right })
    }
}
```

Fichier /src/process/link/stmt.rs

```
use crate::lang::kind::RawId;
use crate::lang::*;

use super::LinkableKind;
use super::{Linker, LinkResult, TraceNode};


use TraceNode::*;


impl<K: LinkableKind> Linker<K>
where
    K::VariableId: RawId,
    K::ConstructorId: RawId,
    K::TypeId: RawId,
    K::GenericId: RawId,
{
    pub fn statements(&mut self, raw: Vec<Statement<K>>)
        -> LinkResult<K, Vec<Statement<Linked>>>
    {
        raw.into_iter()
            .map(|s| self.statement(s))
            .collect()
    }

    fn statement(&mut self, raw: Statement<K>)
        -> LinkResult<K, Statement<Linked>>
    {
        let (var, val);
        trace!(self.trace, StatementVar, {
            var = self.variable(raw.var)?;
        });

        // Done after registering var for recursive reasons
        id_scope!(self.var_ids, {
            trace!(self.trace, StatementVal, {
                val = self.expr(raw.val)?;
            });
        });

        Ok(Statement { var, val })
    }
}
```

Fichier /src/process/link/typ.rs

```
use super::*;


use TraceNode::*;


impl<K: LinkableKind> Linker<K>
where
    K::VariableId: RawId,
    K::ConstructorId: RawId,
    K::TypeId: RawId,
    K::GenericId: RawId,
{
    pub fn type_defs(&mut self, raw: Vec<TypeDef<K>>)
```

```
        -> LinkResult<K, Vec<TypeDef<Linked>>>
    {
        raw.into_iter()
            .map(|td| self.type_def(td))
            .collect()
    }

    fn type_def(&mut self, raw: TypeDef<K>) -> LinkResult<K,
TypeDef<Linked>> {
        let (id, arg_ids, typ);

        trace!(self.trace, TypeDefId, {
            id = self.type_id_create(raw.id)?;
        });

        id_scope!(self.generic_ids, {
            trace!(self.trace, TypeDefArgIds, {
                arg_ids = raw.arg_ids
                    .into_iter()
                    .map(|raw_id| self.generic_id_create(raw_id))
                    .collect::<LinkResult<_, Vec<_>>>()?;
            });

            trace!(self.trace, TypeDefType, {
                use crate::lang::TypeDefType::*;
                typ = match raw.typ {
                    Type(raw_typ) => Type(self.typ(raw_typ)?),
                    TypeSum(raw_branches) =>
                        TypeSum(self.type_sum_branches(raw_branches)?)
                };
            });
        });

        Ok(TypeDef { id, arg_ids, typ })
    }

    fn type_sum_branches(&mut self, raw: Vec<TypeSumBranch<K>>)
        -> LinkResult<K, Vec<TypeSumBranch<Linked>>>
    {
        raw.into_iter()
            .map(|raw_branch| self.type_sum_branch(raw_branch))
```

```
            .collect()
    }

    fn type_sum_branch(&mut self, raw: TypeSumBranch<K>)
        -> LinkResult<K, TypeSumBranch<Linked>>
    {
        let (constructor_id, args);

        trace!(self.trace, TypeSumBranchConstructorId, {
            constructor_id =
self.constructor_id_create(raw.constructor_id)?;
        });

        trace!(self.trace, TypeSumBranchArgs, {
            args = raw.args
                .into_iter()
                .map(|t| self.typ(t))
                .collect::<LinkResult<_, Vec<_>>>()?;
        });

        Ok(TypeSumBranch { constructor_id, args })
    }

    pub fn typ(&mut self, raw: Type<K>) -> LinkResult<K, Type<Linked>> {
        use Type::*;

        Ok(match raw {
            Generic { id: raw_id } => {
                let id;
                trace!(self.trace, GenericId, {
                    id = self.generic_id_get_or_create(raw_id);
                });
                Generic { id }
            },

            Specialization { args: raw_args, typ: raw_type } => {
                let (args, typ);

                trace!(self.trace, SpecializationArgs, {
                    args = raw_args
                        .into_iter()
                        .map(|t| self.typ(t))
                        .collect::<LinkResult<_, Vec<Type<Linked>>>>()?;
                });

                trace!(self.trace, SpecializationType, {
                    typ = self.type_id_get(raw_type)?;
                });

                Specialization { args, typ }
            }

            Function { input: raw_input, output: raw_output } => {
                let (input, output);

                trace!(self.trace, TypeFunctionInput, {
                    input = Box::new(self.typ(*raw_input)?);
                });
                trace!(self.trace, TypeFunctionOutput, {
                    output = Box::new(self.typ(*raw_output)?);
                });

                Function { input, output }
            }
        })
    }
}


#[cfg(test)]
mod tests {
    use super::super::super::read;
    use super::*;


    #[test]
    fn test_type_def() {
        use read::type_def;

        let mut linker = Linker::new();

        assert!(linker.type_def(type_def(
            "type bool = False | True;;".chars()
        )).is_ok());

        assert!(linker.type_def(type_def(
            "type bool = False | True;;".chars()
        )).is_err());

        assert!(linker.type_def(type_def(
            "type alias = bool;;".chars()
        )).is_ok());

        assert!(linker.type_def(type_def(
            "type 'a option = None | Some of 'a;;".chars()
        )).is_ok());
        assert!(linker.type_def(type_def(
            "type 'a 'a wrong_option = 'a option;;".chars()
        )).is_err());
    }
}
```

Dossier /src/process/parse/

Fichier /src/process/parse/mod.rs

```
use std::iter::{Iterator, Peekable};

use crate::lang::*;
use super::lex::token::{Token, TokenData};

mod expr;
mod pat;
mod prop;
mod stmt;
mod typ;


use TokenData::*;


#[derive(Debug)]
pub enum ParsingErr {
    UnexpectedToken {
        expected: String,
        got: Token
    },
}

use ParsingErr::*;
type ParseResult<T> = Result<T, ParsingErr>;


pub struct Parser<I: Iterator<Item = Token>>
{
    input: Peekable<I>,
}


impl<I: Iterator<Item = Token>> Parser<I> {
    pub fn new(iter: I) -> Self {
        Parser {
            input: iter.peekable(),
        }
    }

    fn peek(&mut self) -> ParseResult<&Token> {
        self.input.peek()
            .ok_or_else(|| unreachable!("At the very least EOF is
expected"))
    }

    fn next(&mut self) -> ParseResult<Token> {
        self.input.next()
            .ok_or_else(|| unreachable!("At the very least EOF is
expected"))
    }

    pub fn ast(&mut self) -> ParseResult<AST<Raw>> {
        let type_defs = self.type_defs()?;
        let statements = self.statements()?;
        Ok(AST { type_defs, statements })
    }

    fn consume(&mut self, token_data: TokenData) -> ParseResult<()> {
        let token = self.next()?;
        if token.data == token_data {
            Ok(())
        } else {
            Err(UnexpectedToken {
                // TODO: cleaner printing
                expected: String::from(format!("{:?}", token_data)),
                got: token
            })
        }
    }

    fn consume_if_exists(&mut self, token_data: TokenData) {
        let exists = self.peek()
            .map_or(
                false,
                |token| token.data == token_data
            );

        if exists {
            self.next().unwrap();
        }
    }
}


#[cfg(test)]
mod tests {
    use super::{Parser, Token};
    use crate::process::read;

    pub fn str_to_parser<'a>(text: &'a str) -> Parser<impl Iterator<Item
= Token> + 'a> {
        Parser::new(read::lex_and_unwrap(text.chars()))
    }
}
```

Fichier /src/process/parse/prop.rs

```
use super::*;


impl<I: Iterator<Item = Token>> Parser<I> {
    pub fn property(&mut self) -> ParseResult<Property<Raw>> {
        let mut vars = Vec::new();


        loop {
            match self.peek()?.data {
                TokenData::SnakeIdent { .. } => {
                    vars.push(self.variable()?);
                },
                _ => break
            }
        }

        self.consume(TokenData::DOT)?;
        let left = self.expr()?;
        self.consume(TokenData::EQUAL)?;
        let right = self.expr()?;

        Ok(Property { vars, left, right })
    }
}
```

Fichier /src/process/parse/stmt.rs

```
use super::*;


impl<I: Iterator<Item = Token>> Parser<I> {
    pub fn statements(&mut self) -> ParseResult<Vec<Statement<Raw>>> {
        let mut res = Vec::new();

        loop {
            match self.peek()? {
                Token { data: TokenData::LET, .. } =>
                    res.push(self.statement()?),
                _ => break
            }
        }

        Ok(res)
    }

    pub fn statement(&mut self) -> ParseResult<Statement<Raw>> {
        self.consume(TokenData::LET)?;
        let variable = self.variable()?;
        self.consume(TokenData::EQUAL)?;
        let val = self.expr()?;
        self.consume(TokenData::DBLSEMICOL)?;

        Ok(Statement { var: variable, val })
    }
}
```

Fichier /src/process/parse/typ.rs

```
use super::*;


impl<I: Iterator<Item = Token>> Parser<I> {
    pub fn type_defs(&mut self) -> ParseResult<Vec<TypeDef<Raw>>> {
        let mut res = Vec::new();

        loop {
            match self.peek()?.data {
                TokenData::TYPE => res.push(self.type_def()?),
                _ => break
            }
        }

        Ok(res)
    }

    pub fn type_def(&mut self) -> ParseResult<TypeDef<Raw>> {
        self.consume(TokenData::TYPE)?;

        let mut argument_ids = Vec::new();

        // This is actually not the way OCaml parses type parameters if
their
        // are more then two of them. But this way of doing it is easier
to
        // parse.
        loop {
            match &self.peek()?.data {
                TokenData::GenericIdent(ident) => {
                    argument_ids.push(ident.clone());
                    self.next()?;
                },
                _ => break
            }
        }

        let id = match &self.peek()?.data {
            TokenData::SnakeIdent(ident) => ident.clone(),
            _ => return Err(UnexpectedToken {
                expected: String::from("type identifier"),
                got: self.next()?
            })
        };
        self.next()?;

        self.consume(TokenData::EQUAL)?;

        let typ = match self.peek()?.data {
            TokenData::PIPE | TokenData::PascalIdent(_)
                => TypeDefType::TypeSum(self.type_sum_branches()?),
            _ => TypeDefType::Type(self.typ()?)
        };

        self.consume(TokenData::DBLSEMICOL)?;

        return Ok(TypeDef { arg_ids: argument_ids, id, typ });
    }

    pub fn type_sum_branches(&mut self) ->
ParseResult<Vec<TypeSumBranch<Raw>>> {
        let mut branches = Vec::new();

        loop {
            match self.peek()?.data {
                TokenData::PIPE | TokenData::PascalIdent(_) =>
                    branches.push(self.type_sum_branch()?),
                _ => break
            }
        }

        Ok(branches)
    }

    pub fn type_sum_branch(&mut self) -> ParseResult<TypeSumBranch<Raw>>
{
        self.consume_if_exists(TokenData::PIPE);

        let id = match &self.peek()?.data {
            TokenData::PascalIdent(ident) => ident.clone(),
            _ => return Err(UnexpectedToken {
                expected: String::from("constructor identifier"),
                got: self.next()?
            })
        };
        self.next()?;

        let mut args = Vec::new();

        if self.peek()?.data ==  TokenData::OF {
            self.consume(TokenData::OF)?;

            args.push(self.short_type()?);

            loop {
                match self.peek()?.data {
                    TokenData::STAR => {
                        self.consume(TokenData::STAR)?;
                        args.push(self.short_type()?);
                    },
                    _ => break
                }
            }
        }

        Ok(TypeSumBranch {
            constructor_id: id,
            args
        })
    }

    pub fn typ(&mut self) -> ParseResult<Type<Raw>> {
        let t = self.short_type()?;

        if self.peek()?.data != TokenData::ARROW {
            return Ok(t);
        }
        self.consume(TokenData::ARROW)?;

        Ok(Type::Function {
            input: Box::new(t),
            output: Box::new(self.typ()?)
        })
    }

    pub fn short_type(&mut self) -> ParseResult<Type<Raw>> {
        let mut args = Vec::new();

        loop {
            match &self.peek()?.data {
                TokenData::GenericIdent(id) => {
                    args.push(Type::Generic { id: id.clone() });
                    self.next()?;
                },
                TokenData::SnakeIdent(id) => {
                    args.push(Type::Specialization {
                        args: Vec::new(),
                        typ: id.clone()
                    });
                    self.next()?;
                },
                TokenData::LPAREN => {
                    self.consume(TokenData::LPAREN)?;
                    args.push(self.typ()?);
                    self.consume(TokenData::RPAREN)?;
                }
                _ => break
            }
        }

        match args.pop() {
            Some(Type::Specialization {
                args: special_args,
                typ
            }) =>
                if special_args.is_empty() {
                    Ok(Type::Specialization { args, typ })
                } else {
                    Err(UnexpectedToken {
                        expected: String::from("type identifier"),
                        got: self.next()?
                    })
                },

            Some(not_specialization) if args.is_empty() =>
                Ok(not_specialization),

            _ => Err(UnexpectedToken {
                expected: String::from("generic, type identifier or
parenthesized type"),
                got: self.next()?
            })
        }
    }
}


#[cfg(test)]
mod tests {
    use super::super::tests::str_to_parser;
    use super::*;

    use Type::*;


    #[test]
    fn test_type_def() -> ParseResult<()> {
        use TypeDefType::*;

        assert_eq!(
            str_to_parser("type bool = | True | False;;").type_def()?,
            TypeDef {
                id: String::from("bool"),
                arg_ids: vec![],
                typ: TypeSum(vec![
                    TypeSumBranch {
                        constructor_id: String::from("True"),
                        args: vec![]
                    },
                    TypeSumBranch {
                        constructor_id: String::from("False"),
                        args: vec![]
                    }
                ])
            }
        );

        assert_eq!(
            str_to_parser("type alt_bool = unit option;;").type_def()?,
            TypeDef {
                id: String::from("alt_bool"),
                arg_ids: vec![],
                typ: Type(Specialization {
                    args: vec![Specialization {
                        args: vec![],
                        typ: String::from("unit")
                    }],
                    typ: String::from("option")
                })
            }
        );

        assert_eq!(
            str_to_parser("type 'a list = Nil | Cons of 'a *'a
list;;").type_def()?,
            TypeDef {
                id: String::from("list"),
                arg_ids: vec![String::from("'a")],
                typ: TypeSum(vec![
                    TypeSumBranch {
                        constructor_id: String::from("Nil"),
                        args: vec![]
                    },
                    TypeSumBranch {
                        constructor_id: String::from("Cons"),
```

```
                        args: vec![
                            Generic { id: String::from("'a") },
                            Specialization {
                                args: vec![Generic { id:
String::from("'a") }],
                                typ: String::from("list")
                            }
                        ]
                    }
                ])
            }
        );

        Ok(())
    }

    #[test]
    fn test_type_sum_branch() -> ParseResult<()> {
        assert_eq!(
            str_to_parser("Zero").type_sum_branch()?,
            TypeSumBranch {
                constructor_id: String::from("Zero"),
                args: Vec::new()
            }
        );

        assert_eq!(
            str_to_parser("Fun of ('a -> 'b)").type_sum_branch()?,
            TypeSumBranch {
                constructor_id: String::from("Fun"),
                args: vec![Function {
                    input: Box::new(Generic { id: String::from("'a") }),
                    output: Box::new(Generic { id: String::from("'b") })
                }]
            }
        );

        assert_eq!(
            str_to_parser("Some of 'a").type_sum_branch()?,
            TypeSumBranch {
                constructor_id: String::from("Some"),
                args: vec![Generic { id: String::from("'a") }]
            }
        );

        assert_eq!(
            str_to_parser("Cons of 'a * 'a list").type_sum_branch()?,
            TypeSumBranch {
                constructor_id: String::from("Cons"),
                args: vec![
                    Generic { id: String::from("'a") },
                    Specialization {
                        args: vec![Generic { id: String::from("'a") }],
                        typ: String::from("list")
                    }
                ]
            }
        );

        Ok(())
    }

    #[test]
    fn test_typ() -> ParseResult<()> {
        assert_eq!(
            str_to_parser("('a -> 'b) -> ('b -> 'a)").typ()?,
            Function {
                input: Box::new(Function {
                    input: Box::new(Generic { id: String::from("'a") }),
                    output: Box::new(Generic { id: String::from("'b") })
                }),
                output: Box::new(Function {
                    input: Box::new(Generic { id: String::from("'b") }),
                    output: Box::new(Generic { id: String::from("'a") })
                })
            }
        );

        assert_eq!(
            str_to_parser("'a list -> (unit -> 'a) option").typ()?,
            Function {
                input: Box::new(Specialization {
```

```
                    args: vec![Generic { id: String::from("'a") }],
                    typ: String::from("list")
                }),
                output: Box::new(Specialization {
                    args: vec![Function {
                        input: Box::new(Specialization {
                            args: Vec::new(),
                            typ: String::from("unit")
                        }),
                        output: Box::new(Generic {
                            id: String::from("'a")
                        })
                    }],
                    typ: String::from("option")
                })
            }
        );

        Ok(())
    }

    #[test]
    fn test_short_type() -> ParseResult<()> {
        assert_eq!(
            str_to_parser("bool").short_type()?,
            Specialization { args: Vec::new(), typ:
String::from("bool") }
        );
        assert_eq!(
            str_to_parser("nat -> nat").short_type()?,
            Specialization { args: Vec::new(), typ:
String::from("nat") }
        );

        assert_eq!(
            str_to_parser("'a").short_type()?,
            Generic { id: String::from("'a") }
        );
        assert_eq!(
            str_to_parser("'a -> 'a").short_type()?,
            Generic { id: String::from("'a") }
        );

        assert_eq!(
            str_to_parser("('a -> 'a)").short_type()?,
            Function {
                input: Box::new(Generic { id: String::from("'a") }),
                output: Box::new(Generic { id: String::from("'a") })
            }
        );

        Ok(())
    }
}
```

Fichier /src/process/parse/expr.rs

```
use super::*;


impl<I: Iterator<Item = Token>> Parser<I> {
    pub fn expr(&mut self) -> ParseResult<Expr<Raw>> {
        let e = match self.peek()?.data {
            TokenData::LET => self.let_in()?,
            TokenData::FUN => self.function()?,
            TokenData::MATCH => self.match_with()?,
            TokenData::LPAREN => self.scope()?,
            TokenData::PascalIdent(_) => self.literal_constructor()?,
            TokenData::SnakeIdent(_) => self.literal_variable()?,
            _ => return Err(UnexpectedToken {
                expected: String::from("expression"),
                got: self.next()?
            })
        };

        let mut args = Vec::new();

        loop {
            let arg = match &self.peek()?.data {
                TokenData::LPAREN => self.scope()?,
                TokenData::SnakeIdent(id) => {
                    let lit = ExprData::LitVar {
                        id: id.clone(),
                        meta: ()
                    };
                    self.next()?;
                    lit.into()
                },
                TokenData::PascalIdent(id) => {
                    let lit = ExprData::LitConstructor {
                        id: id.clone(),
                        args: Vec::new()
                    };
                    self.next()?;
                    lit.into()
                },
                _ => break
            };

            args.push(arg);
        }

        Ok(args
            .into_iter()
            .fold(
                e,
                |caller, argument|
                ExprData::Call {
                    caller: Box::new(caller),
                    arg: Box::new(argument)
                }.into()
            )
        )
    }

    pub fn let_in(&mut self) -> ParseResult<Expr<Raw>> {
        self.consume(TokenData::LET)?;
        let variable = self.variable()?;
        self.consume(TokenData::EQUAL)?;
        let assignment = Box::new(self.expr()?);
        self.consume(TokenData::IN)?;
        let body = Box::new(self.expr()?);

        Ok(ExprData::Binding { var: variable, val: assignment,
body }.into())
```

```
    }

    pub fn function(&mut self) -> ParseResult<Expr<Raw>> {
        self.consume(TokenData::FUN)?;

        let input = match &self.peek()?.data {
            TokenData::LPAREN => {
                self.consume(TokenData::LPAREN)?;
                let input = self.variable()?;
                self.consume(TokenData::RPAREN)?;
                input
            },
            _ => self.untyped_variable()?
        };

        let _out_type_annotation = match &self.peek()?.data {
            TokenData::COLUMN => {
                self.consume(TokenData::COLUMN)?;
                Some(self.short_type()?)
            },
            _ => None
        };

        self.consume(TokenData::ARROW)?;
        let body = Box::new(self.expr()?);

        Ok(ExprData::Function { input, body }.into())
    }

    pub fn match_with(&mut self) -> ParseResult<Expr<Raw>> {
        self.consume(TokenData::MATCH)?;
        let expr = Box::new(self.expr()?);
        self.consume(TokenData::WITH)?;

        let mut cases = Vec::new();
        self.consume_if_exists(TokenData::PIPE);
        cases.push(self.match_branch()?);

        loop {
            if self.peek()?.data == TokenData::PIPE {
                self.consume(TokenData::PIPE)?;
                cases.push(self.match_branch()?);
            } else {
                break;
            }
        }

        Ok(ExprData::Match { expr, cases }.into())
    }

    pub fn match_branch(&mut self) -> ParseResult<MatchBranch<Raw>> {
        // TODO: allow multiple patterns
        let pattern = self.pattern()?;
        self.consume(TokenData::ARROW)?;
        let body = self.expr()?;
        Ok(MatchBranch { pattern, body })
    }

    pub fn scope(&mut self) -> ParseResult<Expr<Raw>> {
        self.consume(TokenData::LPAREN)?;
        let expr = self.expr()?;
        self.consume(TokenData::RPAREN)?;
        Ok(expr)
    }

    pub fn literal_constructor(&mut self) -> ParseResult<Expr<Raw>> {
        let id = match &self.peek()?.data {
            TokenData::PascalIdent(id) => id.clone(),
            _ => return Err(UnexpectedToken {
                expected: String::from("constructor identifier"),
                got: self.next()?
            })
        };
        self.next()?;

        if self.peek()?.data != TokenData::LPAREN {
            return Ok(ExprData::LitConstructor { id, args:
Vec::new() }.into());
        }
        self.consume(TokenData::LPAREN)?;

        let mut argument = Vec::new();
        argument.push(self.expr()?);

        loop {
            match &self.peek()?.data {
                TokenData::RPAREN => break,
                _ => {
                    self.consume(TokenData::COMMA)?;
                    argument.push(self.expr()?);
                }
            }
        }

        self.consume(TokenData::RPAREN)?;
        Ok(ExprData::LitConstructor { id, args: argument }.into())
    }

    pub fn literal_variable(&mut self) -> ParseResult<Expr<Raw>> {
        let lit = match &self.peek()?.data {
            TokenData::SnakeIdent(id) =>
                ExprData::LitVar { id: id.clone(), meta: () },
            _ => return Err(UnexpectedToken {
                expected: String::from("variable identifier"),
                got: self.next()?
            })
        };

        self.next()?;
        Ok(lit.into())
    }
}
```

Fichier /src/process/parse/pat.rs

```
use super::*;


impl<I: Iterator<Item = Token>> Parser<I> {
    pub fn pattern(&mut self) -> ParseResult<Pattern<Raw>> {
        Ok(match &self.peek()?.data {
            SnakeIdent(_) =>
                PatternData::Var(self.variable()?).into(),
            PascalIdent(_) =>
                self.pattern_constructor()?,
            _ => return  Err(UnexpectedToken {
                expected: String::from("pattern"),
                got: self.next()?
            })
        })
    }

    pub fn pattern_constructor(&mut self) -> ParseResult<Pattern<Raw>> {
        let id = match &self.peek()?.data {
            PascalIdent(id) => id.clone(),
            _ => return Err(UnexpectedToken {
                expected: String::from("constructor identifier"),
                got: self.next()?
            })
        };
        self.next()?;

        let mut arguments = Vec::new();

        match &self.peek()?.data {
            LPAREN => {
                self.consume(LPAREN)?;
                arguments.push(self.pattern()?);

                while self.peek()?.data == COMMA {
                    self.consume(COMMA)?;
                    arguments.push(self.pattern()?);
                }

                self.consume(RPAREN)?;
            },

            SnakeIdent(_) | PascalIdent(_) =>
                arguments.push(self.pattern()?),

            _ => ()
        }

        Ok(PatternData::Constructor { id, args: arguments }.into())
    }
    pub fn variable(&mut self) -> ParseResult<Var<Raw>> {
        let id = self.untyped_variable()?.id;

        let _type_annotation = match &self.peek()?.data {
            COLUMN => {
                self.consume(COLUMN)?;
                Some(self.typ()?)
            },
            _ => None
        };

        Ok(Var { id, meta: () })
    }

    pub fn untyped_variable(&mut self) -> ParseResult<Var<Raw>> {
        let id = match &self.peek()?.data {
            SnakeIdent(id) => id.clone(),
            _ => return Err(UnexpectedToken {
                expected: String::from("variable identifier"),
                got: self.next()?
            })
        };
        self.next()?;

        Ok(Var { id, meta: () })
    }
}


#[cfg(test)]
mod tests {
    use super::super::tests::str_to_parser;
    use super::*;

    use PatternData::{Constructor, Var as PVar};
    use Type::*;


    #[test]
    fn test_pattern_constructor() -> ParseResult<()> {
        assert_eq!(
            str_to_parser("Zero").pattern_constructor()?,
            Constructor { id: String::from("Zero"), args: vec!
[] }.into()
        );

        assert_eq!(
            str_to_parser("Tuple (Succ n, n')").pattern_constructor()?,
            Constructor {
                id: String::from("Tuple"),
                args: vec![
                    Constructor {
                        id: String::from("Succ"),
                        args: vec![
                            PVar(Var { id: String::from("n"), meta:
() }).into()
                        ]
                    }.into(),
                    PVar(Var { id: String::from("n'"), meta:
() }).into()
                ]
            }.into()
        );

        Ok(())
    }


    #[test]
    fn test_variable() -> ParseResult<()> {
        assert_eq!(
            str_to_parser("_").variable()?,
            Var { id: String::from("_"), meta: () }
        );

        assert_eq!(
            str_to_parser("n: nat").variable()?,
            Var {
                id: String::from("n"),
                meta: ()
            }
        );

        assert_eq!(
            str_to_parser("n: nat -> n").variable()?,
            Var {
                id: String::from("n"),
                meta: ()
            }
        );

        Ok(())
    }
}
```

Dossier /src/process/typ/

Fichier /src/process/typ/ast.rs

```
use super::*;


impl Typer {
    pub fn type_ast(&mut self, ast: AST<Linked>) ->
TypingResult<AST<LT>> {
        let ast = self.register_ast(ast)?;
        Ok(self.unify_ast(ast))
    }

    fn register_ast(&mut self, ast: AST<Linked>) ->
TypingResult<AST<LT>> {
        let type_defs = ast.type_defs.into_iter()
            .map(|type_def| {
                self.register_type_def(type_def.clone());
                type_def.into()
            })
            .collect();
        let statements = ast.statements.into_iter()
            .map(|statement| self.register_statement(statement))
            .collect::<TypingResult<_>>()?;

        Ok(AST { type_defs, statements })
    }

    fn unify_ast(&mut self, ast: AST<LT>) -> AST<LT> {
        let statements = ast.statements.into_iter()
            .map(|statement| self.unify_statement(statement))
            .collect();
        AST { type_defs: ast.type_defs, statements }
    }
}
```

Fichier /src/process/typ/convert.rs

```
use crate::lang::*;


impl From<TypeDef<Linked>> for TypeDef<LT> {
    fn from(linked: TypeDef<Linked>) -> Self {
        TypeDef {
            id: linked.id,
            arg_ids: linked.arg_ids,
            typ: linked.typ.into()
        }
    }
}

impl From<TypeDefType<Linked>> for TypeDefType<LT> {
    fn from(linked: TypeDefType<Linked>) -> Self {
        use TypeDefType::*;
        match linked {
            Type(typ) => Type(typ.into()),
            TypeSum(branches)  => TypeSum(
                branches.into_iter()
                    .map(|branch| branch.into())
                    .collect()
            )
        }
    }
}

impl From<TypeSumBranch<Linked>> for TypeSumBranch<LT> {
    fn from(linked: TypeSumBranch<Linked>) -> Self {
        TypeSumBranch {
            constructor_id: linked.constructor_id,
            args: linked.args.into_iter()
```

```
                .map(|arg| arg.into())
                .collect()
        }
    }
}

impl From<Type<Linked>> for Type<LT> {
    fn from(linked: Type<Linked>) -> Self {
        use Type::*;
        match linked {
            Generic { id } => Generic { id },
            Specialization { args, typ } =>
                Specialization {
                    args: args.into_iter()
                        .map(|arg| arg.into())
                        .collect(),
                    typ
                },
            Function { input, output } =>
                Function {
                    input: Box::new((*input).into()),
                    output: Box::new((*output).into())
                }
        }
    }
}
```

Fichier /src/process/typ/prop.rs

```
use super::*;


impl Typer {
    pub fn type_property(&mut self, property: Property<Linked>)
        -> TypingResult<Property<LT>>
    {
        let property = self.register_property(property)?;
        Ok(self.unify_property(property))
    }

    pub fn register_property(&mut self, property: Property<Linked>)
        -> TypingResult<Property<LT>>
    {
        let vars = property.vars.into_iter()
            .map(|var| self.register_variable(var))
            .collect();
        let left = self.register_expr(property.left)?;
        let right = self.register_expr(property.right)?;
        self.unify(&left.meta.typ, &right.meta.typ)?;
        Ok(Property { vars, left, right })
    }

    pub fn unify_property(&mut self, property: Property<LT>) ->
Property<LT> {
        let vars = property.vars.into_iter()
            .map(|var| self.unify_variable(var))
            .collect();
        let left = self.unify_expr(property.left);
        let right = self.unify_expr(property.right);
        Property { vars, left, right }
    }
}
```

Fichier /src/process/typ/stmt.rs

```
use super::*;


impl Typer {
    pub fn type_statement(&mut self, statement: Statement<Linked>)
        -> TypingResult<Statement<LT>>
    {
        let statement = self.register_statement(statement)?;
        Ok(self.unify_statement(statement))
    }

    pub fn register_statement(&mut self, statement: Statement<Linked>)
        -> TypingResult<Statement<LT>>
    {
        let var = self.register_variable(statement.var);
        let val = self.register_expr(statement.val)?;
        self.unify(&var.meta.typ, &val.meta.typ)?;
        Ok(Statement { var, val })
    }

    pub fn unify_statement(&mut self, statement: Statement<LT>) ->
Statement<LT> {
        let var = statement.var;
        let val = self.unify_expr(statement.val);
        Statement { var, val }
    }
}
```

Fichier /src/process/typ/typ.rs

```
use super::*;


impl Typer {
    pub fn register_type_def(&mut self, type_def: TypeDef<Linked>) {
        let typ = Type::Specialization {
            args: type_def.arg_ids.into_iter()
                .map(|id| Type::Generic { id })
                .collect(),
            typ: type_def.id
        };

        use TypeDefType as TDF;
        match type_def.typ {
            TDF::Type(alias) => todo!(),
            TDF::TypeSum(branches) =>
                for TypeSumBranch { constructor_id, args } in branches {
                    self.constructor_types.insert(
                        constructor_id,
                        ConstructorTypes {
                            typ: typ.clone(),
                            arg_types: args.into_iter()
                                .map(|arg| arg.into())
                                .collect()
                        }
                    );
                }
        }
    }
}
```

Fichier /src/process/typ/unify.rs

```
use std::collections::HashMap;

use crate::lang::*;
use super::{TypingErr, TypingResult};


pub struct Unifier {
    nodes: HashMap<Type<LT>, Node>
}


#[derive(Debug)]
enum Node {
    Child {
        parent: Type<LT>
    },
    Root {
        typ: Type<LT>,
    }
}


impl Unifier {
    pub fn new() -> Unifier {
        Unifier {
            nodes: HashMap::new()
        }
    }

    fn register_type(&mut self, typ: &Type<LT>) {
        if !self.nodes.contains_key(typ) {
            self.nodes.insert(
                typ.clone(),
                Node::Root {
                    typ: typ.clone(),
                }
            );
        }

        use Type::*;
        match typ {
            Specialization { args, .. } =>
                args.into_iter()
                    .for_each(|arg| self.register_type(arg)),
            Function { input, output } => {
                self.register_type(input);
                self.register_type(output)
            },
            Generic { .. } => ()
        }
    }

    pub fn unify(&mut self, type_a: &Type<LT>, type_b: &Type<LT>)
        -> TypingResult<()>
    {
        let type_a = self.find(type_a.clone());
        let type_b = self.find(type_b.clone());

        if type_a == type_b {
            return Ok(());
        }

        use Type::*;
        match (&type_a, &type_b) {
            (Specialization { args: args_a, typ: type_a },
                Specialization { args: args_b, typ: type_b })
            if type_a == type_b && args_a.len() == args_b.len() =>
            {
                args_a.into_iter()
                    .zip(args_b.into_iter())
                    .map(|(arg_a, arg_b)|
                        self.unify(arg_a, arg_b)
                    )
                    .collect::<TypingResult<Vec<_>>>()?;
                Ok(())
            }

            // TODO: handle cases where aliases are made with type defs

            (Function { input: input_a, output: output_a },
                Function { input: input_b, output: output_b }) =>
            {
                self.unify(input_a, input_b)?;
                self.unify(output_a, output_b)
            },
```

```
            (Generic { id }, other)
            | (other, Generic { id }) =>
            {
                self.nodes.insert(
                    Generic { id: id.clone() },
                    Node::Child { parent: other.clone() }
                );
                self.register_type(other);
                Ok(())
            }

            _ => Err(TypingErr::CantUnify {
                type_a: type_a.clone(),
                type_b: type_b.clone()
            })
        }
    }

    pub fn find(&mut self, typ: Type<LT>) -> Type<LT> {
        use Node::*;
        let typ = match self.nodes.get(&typ) {
            Some(Child { parent }) =>
                return self.find(parent.clone()),
            Some(Root { typ: root }) =>
                root.clone(),
            _ => {
                self.nodes.insert(
                    typ.clone(),
                    Node::Root { typ: typ.clone() }
                );
                typ.clone()
            }
        };

        use Type::*;
        match typ {
            Generic { id } =>
                Generic { id },

            Function { input, output } =>
```

```
                Function {
                    input: Box::new(self.find(*input)),
                    output: Box::new(self.find(*output))
                },

            Specialization { args, typ } =>
                Specialization {
                    args: args.into_iter()
                        .map(|arg| self.find(arg))
                        .collect(),
                    typ: typ.clone()
                }
        }
    }
}
```

Fichier /src/process/typ/expr.rs

```
use super::*;


impl Typer {
    pub fn register_expr(&mut self, expr: Expr<Linked>) ->
TypingResult<Expr<LT>> {
        use ExprData::*;

        Ok(match expr.data {
            Binding { var, val, body } => {
                let var = self.register_variable(var);
                let val = self.register_expr(*val)?;
                self.unify(&val.meta.typ, &var.meta.typ)?;

                let body = self.register_expr(*body)?;
                let body_type = body.meta.typ.clone();

                Expr {
                    data: Binding {
                        var,
                        val: Box::new(val),
                        body: Box::new(body)
                    },
                    meta: body_type.into()
                }
            }

            Function { input, body } => {
                let input = self.register_variable(input);
                let output_type = self.new_type();

                let body = self.register_expr(*body)?;
                self.unify(&output_type, &body.meta.typ)?;

                Expr {
                    data: Function {
                        input: input.clone(),
                        body: Box::new(body)
                    },
                    meta: Type::Function {
                        input: Box::new(input.meta.typ),
                        output: Box::new(output_type)
                    }.into()
                }
            },

            Match { expr, cases } => {
                let expr = self.register_expr(*expr)?;

                let cases = cases.into_iter()
                    .map(|branch| self.register_match_branch(branch))
                    .collect::<TypingResult<Vec<_>>>()?;

                let typ = cases.get(0)
                    .expect("A match should have at least one branch")
                    .body.meta.typ
                    .clone();

                cases.iter()
                    .map(|MatchBranch { pattern, body }| {
                        self.unify(&expr.meta.typ, &pattern.meta.typ)?;
                        self.unify(&typ, &body.meta.typ)
                    })
                    .collect::<TypingResult<Vec<_>>>()?;

                Expr {
                    data: Match {
                        expr: Box::new(expr),
                        cases
                    },
                    meta: typ.into()
                }
            },

            Call { caller, arg } => {
                let caller = self.register_expr(*caller)?;
                let arg = self.register_expr(*arg)?;

                let result_typ = self.new_type();
                self.unify(
                    &caller.meta.typ,
                    &Type::Function {
                        input: Box::new(arg.meta.typ.clone()),
                        output: Box::new(result_typ.clone())
                    }
                )?;

                Expr {
                    data: Call {
                        caller: Box::new(caller),
                        arg: Box::new(arg)
                    },
                    meta: result_typ.into()
                }
            },

            LitVar { id, meta } => {
                // TODO: find a way to not kill polymorphism
                let typ = self.var_types.get(&id)
                    .expect("Expected variable to have been declared")
                    .clone();
                Expr {
                    data: LitVar { id, meta },
                    meta: typ.into()
                }
            },
```

```
            LitConstructor { id, args } => {
                let ConstructorTypes { typ, arg_types } =
                    self.instantiate_constructor_types(&id);

                let args = args.into_iter()
                    .map(|arg| self.register_expr(arg))
                    .collect::<TypingResult<Vec<_>>>()?;

                args.iter()
                    .zip(arg_types.iter())
                    .map(|(arg, typ)|
                        self.unify(&arg.meta.typ, &typ)
                    )
                    .collect::<TypingResult<Vec<_>>>()?;

                Expr {
                    data: LitConstructor { id, args },
                    meta: typ.into()
                }
            }
        })
    }

    pub fn unify_expr(&mut self, expr: Expr<LT>) -> Expr<LT> {
        use ExprData::*;

        let data = match expr.data {
            Binding { var, val, body } => {
                let var = self.unify_variable(var);
                let val = self.unify_expr(*val);
                let body = self.unify_expr(*body);
                Binding { var, val: Box::new(val), body:
Box::new(body) }
            }

            Function { input, body, .. } => {
                let input = self.unify_variable(input);
                let body = self.unify_expr(*body);
                Function { input, body: Box::new(body) }
            },
            Match { expr, cases } => {
                let expr = self.unify_expr(*expr);

                let cases = cases.into_iter()
                    .map(|branch| self.unify_match_branch(branch))
                    .collect();

                Match { expr: Box::new(expr), cases }
            },

            Call { caller, arg } => {
                let caller = self.unify_expr(*caller);
                let arg = self.unify_expr(*arg);
                Call {
                    caller: Box::new(caller),
                    arg: Box::new(arg)
                }
            },

            LitVar { id, meta } =>
                LitVar { id, meta },

            LitConstructor { id, args } => {
                let args = args.into_iter()
                    .map(|arg| self.unify_expr(arg))
                    .collect();
                LitConstructor { id, args }
            }
        };

        Expr {
            data,
            meta: self.find_current_best_type(expr.meta.typ).into()
        }
    }

    fn register_match_branch(&mut self, branch: MatchBranch<Linked>)
        -> TypingResult<MatchBranch<LT>>
    {
        Ok(MatchBranch {
            pattern: self.register_pattern(branch.pattern)?,
            body: self.register_expr(branch.body)?
        })
    }

    fn unify_match_branch(&mut self, branch: MatchBranch<LT>)
        -> MatchBranch<LT>
    {
        MatchBranch {
            pattern: self.unify_pattern(branch.pattern),
            body: self.unify_expr(branch.body)
        }
    }
}
```

Fichier /src/process/typ/mod.rs

```
mod ast;
mod convert;
mod expr;
mod pat;
mod prop;
mod stmt;
mod typ;
mod unify;

use std::collections::HashMap;

use crate::lang::*;
use super::IdGenerator;
use unify::Unifier;


impl Expr<Linked> {
    pub fn type_alone(
        self,
        constructors_id_arity: Vec<(UId, usize)>,
        free_variable_ids: Vec<UId>,
    ) -> Expr<LT>
    {
        let mut typer = Typer::new();

        for id in free_variable_ids {
            typer.register_variable(Var { id, meta: () });
        }
        for (constr, arity) in constructors_id_arity {
            let arg_types: Vec<_> = (0..arity).into_iter()
                        .map(|_| typer.new_type())
                        .collect();

            let typ = Type::Specialization {
                args: arg_types.clone(),
                typ: UId {
                    id: typer.generic_counter.gen(),
                    name: String::from("todo")
                }
            };

            typer.constructor_types.insert(
                constr.clone(),
                ConstructorTypes { typ, arg_types }
            );
        }

        let registered = typer.register_expr(self).unwrap();
        typer.unify_expr(registered)
    }
}


pub struct Typer {
    constructor_types: HashMap<UId, ConstructorTypes>,

    generic_counter: IdGenerator,
    unifier: Unifier,

    var_types: HashMap<UId, Type<LT>>
}


#[derive(Clone)]
struct ConstructorTypes {
    typ: Type<LT>,
    arg_types: Vec<Type<LT>>
}
```

```
impl ConstructorTypes {
    pub fn beta_rename(self, counter: &mut IdGenerator) ->
ConstructorTypes {
        let ids = self.typ.generics();
        let dictionary = ids.into_iter()
            .map(|id| (id, UId::unnamed(counter.gen())))
            .collect();

        let typ = Self::rename_generics_of_type(self.typ, &dictionary);
        let arg_types = self.arg_types.into_iter()
            .map(|arg_type| Self::rename_generics_of_type(arg_type,
&dictionary))
            .collect();

        ConstructorTypes { typ, arg_types }
    }

    fn rename_generics_of_type(typ: Type<LT>, dictionary: &HashMap<UId,
UId>) -> Type<LT> {
        use Type::*;

        match typ {
            Generic { id } =>
                Generic {
                    id: dictionary.get(&id)
                        .unwrap()
                        .clone()
                },

            Specialization { args, typ } =>
                Specialization {
                    args: args.into_iter()
                        .map(|arg|
                            Self::rename_generics_of_type(arg,
dictionary)
                        )
                        .collect(),
                    typ
                },
            Function { input, output } =>
                Function {
                    input:
Box::new(Self::rename_generics_of_type(*input, dictionary)),
                    output:
Box::new(Self::rename_generics_of_type(*output, dictionary))
                }
        }
    }
}


#[derive(Debug)]
pub enum TypingErr {
    CantUnify {
        type_a: Type<LT>,
        type_b: Type<LT>
    }
}

type TypingResult<T> = Result<T, TypingErr>;


impl Typer {
    pub fn new() -> Typer {
        Typer {
            constructor_types: HashMap::new(),
            generic_counter: IdGenerator::new(),
            unifier: Unifier::new(),
            var_types: HashMap::new()
        }
    }

    fn instantiate_constructor_types(&mut self, id: &UId) ->
ConstructorTypes {
        self.constructor_types.get(id)
            .expect("constructor should be registered")
            .clone()
            .beta_rename(&mut self.generic_counter)
    }

    fn new_type(&mut self) -> Type<LT> {
        Type::Generic {
            id: UId::unnamed(self.generic_counter.gen())
        }
    }

    fn unify(&mut self, type_a: &Type<LT>, type_b: &Type<LT>) ->
TypingResult<()> {
        self.unifier.unify(type_a, type_b)
    }

    fn find_current_best_type(&mut self, typ: Type<LT>) -> Type<LT> {
        self.unifier.find(typ)
    }
}
```

Fichier /src/process/typ/pat.rs

```
use super::*;


impl Typer {
    pub fn register_pattern(&mut self, pattern: Pattern<Linked>)
        -> TypingResult<Pattern<LT>>
    {
        use PatternData as PD;

        Ok(match pattern.data {
            PD::Var(var) => {
                let var = self.register_variable(var);
                Pattern {
                    data: PD::Var(var.clone()),
                    meta: var.meta.typ.into()
                }
            },

            PD::Constructor { id, args } => {
                let ConstructorTypes { typ, arg_types } =
                    self.instantiate_constructor_types(&id);

                let args = args.into_iter()
                    .map(|arg| self.register_pattern(arg))
                    .collect::<TypingResult<Vec<_>>>()?;
                args.iter()
                    .zip(arg_types.iter())
                    .map(|(arg, typ)|
                        self.unify(&arg.meta.typ, &typ)
                    )
                    .collect::<TypingResult<Vec<_>>>()?;

                Pattern {
                    data: PD::Constructor { id, args },
                    meta: typ.into()
                }
            }
        })
    }

    pub fn unify_pattern(&mut self, pattern: Pattern<LT>) -> Pattern<LT>
    {
        use PatternData as PD;

        let data = match pattern.data {
            PD::Var(var) =>
                PD::Var(var),

            PD::Constructor { id, args } => {
                let args = args.into_iter()
                    .map(|arg| self.unify_pattern(arg))
                    .collect();
                PD::Constructor { id, args }
            }
        };

        Pattern {
            data,
            meta: self.find_current_best_type(pattern.meta.typ).into()
        }
    }

    pub fn register_variable(&mut self, var: Var<Linked>) -> Var<LT> {
        let typ = self.new_type();
        self.var_types.insert(var.id.clone(), typ.clone());
        Var { id: var.id, meta: typ.into() }
    }

    pub fn unify_variable(&mut self, var: Var<LT>) -> Var<LT> {
        let typ = self.find_current_best_type(var.meta.typ);
        Var { id: var.id, meta: typ.into() }
    }
}
```

Dossier /src/process/utils/

Fichier /src/process/utils/id_gen.rs

```
#[derive(Debug, Clone)]
pub struct IdGenerator {
    counter: usize
}

impl From<usize> for IdGenerator {
    fn from(counter_start: usize) -> Self {
        IdGenerator { counter: counter_start  }
    }
}

impl IdGenerator {
    pub fn new() -> IdGenerator {
        IdGenerator {
            counter: 0
        }
    }

    pub fn gen(&mut self) -> usize {
        let c = self.counter;
        self.counter += 1;
        c
    }
}
```

Fichier /src/process/utils/mod.rs

```
mod id_gen;
mod scope_stack;
mod trace;
```

```
pub use id_gen::*;
pub use scope_stack::*;
pub use trace::*;
```

Fichier /src/process/utils/scope_stack.rs

```
use std::collections::HashMap;
use std::hash::Hash;


#[derive(Debug, Clone)]
pub struct ScopeStack<K, V> {
    stack: Vec<HashMap<K, V>>
}


impl<K: Eq + Hash, V: Clone> ScopeStack<K, V> {
    pub fn new() -> Self {
        ScopeStack {
            stack: Vec::new(),
        }
    }

    pub fn push_empty_scope(&mut self) {
        self.stack.push(HashMap::new());
    }

    pub fn pop_scope(&mut self) -> Option<HashMap<K, V>> {
        self.stack.pop()
    }

    pub fn insert(&mut self, key: K, val: V) {
        self.stack.last_mut()
            .expect("stack should not be empty")
            .insert(key, val);
    }

    // TODO: find a solution to this horrible complexity, what even is
the point
    // of the hashmap ?
    pub fn get(&self, key: &K) -> Option<&V> {
        for scope in self.stack.iter().rev() {
            if let Some(val) = scope.get(key) {
                return Some(val);
            }
        }

        None
    }
}
```

Fichier /src/process/utils/trace.rs

```
#[derive(Clone, Debug)]
pub enum TraceNode {
    BindingVar,
    BindingVal,
    BindingBody,
    FunctionInput,
    FunctionOutType,
    FunctionBody,
    CallCaller,
    CallArg,
    MatchExpr,
    MatchCases,
    LitVarId,
    LitConstructorId,
    LitConstructorArgs,

    MatchBranchPattern,
    MatchBranchBody,

    PatternVar,
    ConstructorId,
    ConstructorArgs,

    VarId,
    VarTyp,

    PropertyVars,
    PropertyLeft,
    PropertyRight,

    StatementVar,
    StatementVal,
    TypeDefId,
    TypeDefArgIds,
    TypeDefType,

    GenericId,
    SpecializationArgs,
    SpecializationType,
    TypeFunctionInput,
    TypeFunctionOutput,

    TypeSumBranchConstructorId,
    TypeSumBranchArgs,
}


#[derive(Clone, Debug)]
pub struct Trace {
    trace: Vec<TraceNode>
}

impl Trace {
    pub fn new() -> Trace {
        Trace { trace: Vec::new() }
    }

    pub fn enter(&mut self, node: TraceNode) {
        self.trace.push(node);
    }

    pub fn exit(&mut self) {
        self.trace.pop();
    }
}
```

Dossier /src/process/loose/

Fichier /src/process/loose/mod.rs

```
mod link;
mod offset;

pub use link::*;
```

Fichier /src/process/loose/offset.rs

```
use crate::lang::*;

use ExprData::*;


impl Expr<LVT> {
    pub fn move_declaration_dist(&mut self, src: DistToDecl, dest:
DistToDecl) {
        match &mut self.data {
            Binding { val, body, .. } => {
                val.move_declaration_dist(src + 1, dest + 1);
                body.move_declaration_dist(src + 1, dest + 1);
            },

            Function { body, .. } =>
                body.move_declaration_dist(src + 1, dest + 1),

            Call { caller, arg } => {
                caller.move_declaration_dist(src, dest);
                arg.move_declaration_dist(src, dest);
            },

            Match { expr, cases } => {
                expr.move_declaration_dist(src, dest);

                for case in cases {
                    let offset = case.pattern.induced_offset();
                    case.body.move_declaration_dist(src + offset, dest +
offset);
                }
            },

            LitVar { meta, .. } =>
                if meta.dist_to_decl == src {
                    meta.dist_to_decl = dest;
                } else if src < meta.dist_to_decl && meta.dist_to_decl
<= dest {
                    meta.dist_to_decl -= 1;
                } else if dest <= meta.dist_to_decl && meta.dist_to_decl
< src {
```

```
                    meta.dist_to_decl += 1;
                },

            LitConstructor { args, .. } =>
                for arg in args {
                    arg.move_declaration_dist(src, dest);
                }
        }
    }

    pub fn all_moved(self, strict_above: DistToDecl, amount: isize) ->
Expr<LVT> {
        let data = match self.data {
            LitVar { id, meta } =>
                LitVar {
                    id,
                    meta: LVTVarMeta {
                        dist_to_decl: if meta.dist_to_decl >
strict_above {
                            meta.dist_to_decl
                                .checked_add_signed(amount)
                                .unwrap()
                        } else {
                            meta.dist_to_decl
                        },
                        is_recursive: meta.is_recursive
                    }
                },

            LitConstructor { id, args } =>
                LitConstructor {
                    id,
                    args: args.into_iter()
                        .map(|arg| arg.all_moved(strict_above, amount))
                        .collect()
                },

            Binding { var, val, body } =>
                Binding {
                    var,
                    val: Box::new(val.all_moved(strict_above + 1,
amount)),
                    body: Box::new(body.all_moved(strict_above + 1,
amount))
                },

            Match { expr, cases } =>
                Match {
                    expr: Box::new(expr.all_moved(strict_above,
amount)),
                    cases: cases.into_iter()
                        .map(|MatchBranch { pattern, body }| {
                            let offset = pattern.induced_offset();
                            MatchBranch {
                                pattern: pattern,
                                body: body.all_moved(
                                    strict_above + offset,
                                    amount
                                )
                            }
                        })
                        .collect()
                },

            Call { caller, arg } =>
                Call {
                    caller: Box::new(caller.all_moved(strict_above,
amount)),
                    arg: Box::new(arg.all_moved(strict_above, amount))
                },

            Function { input, body } =>
                Function {
                    input,
                    body: Box::new(
                        body.all_moved(strict_above + 1, amount)
                    )
                }
        };

        Expr { data, meta: self.meta }
    }
```

```
}


impl Pattern<LVT> {
    pub fn induced_offset(&self) -> usize {
        use PatternData as PD;

        match &self.data {
            PD::Var(_) => 1,
            PD::Constructor { args, .. } =>
                args.iter()
                    .map(|arg| arg.induced_offset())
                    .sum()
        }
    }
}
```

Dossier /src/process/loose/link/

Fichier /src/process/loose/link/ast.rs

```
use crate::lang::*;

use super::LooseLinker;


impl LooseLinker {
    pub fn ast(&mut self, lt: AST<LT>) -> AST<LVT> {
        AST {
            type_defs: lt.type_defs.into_iter()
                .map(|type_def| type_def.into())
                .collect(),
            statements: self.statements(lt.statements),
        }
    }
}
```

Fichier /src/process/loose/link/expr.rs

```
use crate::lang::*;

use super::LooseLinker;


impl LooseLinker {
    pub fn expr(&mut self, lt: Expr<LT>) -> Expr<LVT> {
        use ExprData::*;

        let data = match lt.data {
            Binding { var: lt_var, val: lt_val, body: lt_body } => {
                let (var, val, body);

                id_scope!(self, {
                    var = self.variable(lt_var.clone());

                    // Done after var for recursive reasons
                    recursive_detection_scope!(self, lt_var.id, {
                        val = self.expr(*lt_val);
                    });

                    body = self.expr(*lt_body);
                });

                Binding { var, val: Box::new(val), body:
Box::new(body) }
            },

            Function { input: lt_input, body: lt_body } => {
                let (input, body);
                id_scope!(self, {
                    input = self.variable(lt_input);
                    body = self.expr(*lt_body);
                });

                Function { input, body: Box::new(body) }
            },

            Call { caller: lt_caller, arg: lt_arg } => {
                let caller = self.expr(*lt_caller);
                let arg = self.expr(*lt_arg);
                Call { caller: Box::new(caller), arg: Box::new(arg) }
            },

            Match { expr: lt_expr, cases: lt_cases } => {
                let expr = self.expr(*lt_expr);
                let cases = lt_cases.into_iter()
```

```
                    .map(|branch| self.match_branch(branch))
                    .collect();
                Match { expr: Box::new(expr), cases }
            },

            LitVar { id: lt_id, .. } => {
                LitVar {
                    id: LoId {
                        name: lt_id.name.clone()
                    },
                    meta: LVTVarMeta {
                        dist_to_decl: self.dist_to_decl(&lt_id),
                        is_recursive:
self.current_recursive_ids.contains(&lt_id)
                    }
                }
            },

            LitConstructor { id: lt_id, args: lt_args } => {
                LitConstructor {
                    id: lt_id,
                    args: lt_args.into_iter()
                        .map(|arg| self.expr(arg))
                        .collect()
                }
            }
        };

        Expr { data, meta: lt.meta }
    }

    fn match_branch(&mut self, lt: MatchBranch<LT>) -> MatchBranch<LVT>
{
        let (pattern, body);

        id_scope!(self, {
            pattern = self.pattern(lt.pattern);
            body = self.expr(lt.body);
        });

        MatchBranch { pattern, body }
    }
}
```

Fichier /src/process/loose/link/mod.rs

```
macro_rules! id_scope {
    ($loose_linker: expr, $code: block) => {
        {

($loose_linker).scope_positions.push(($loose_linker).current_position());
            $code;
            ($loose_linker).scope_positions.pop();
        }
    };
}


macro_rules! recursive_detection_scope {
    ($loose_linker: expr, $id: expr, $code: block) => {
        {
            ($loose_linker).current_recursive_ids.push($id);
            $code;
            ($loose_linker).current_recursive_ids.pop();
        }
    }
}


mod ast;
mod expr;
mod pat;
mod prop;
mod stmt;
mod typ;

use std::{collections::HashMap, ops::AddAssign};
use crate::lang::*;


impl Expr<LT> {
    pub fn loose_link_alone(self) -> Expr<LVT> {
        let mut ll = LooseLinker::new();
        for id in self.free_vars() {
            ll.register(id);
        }

        ll.expr(self)
    }
}


pub struct LooseLinker {
    scope_positions: Vec<usize>,
    current_recursive_ids: Vec<UId>,
    id_positions: HashMap<UId, usize>
}


impl LooseLinker {
    pub fn new() -> LooseLinker {
        LooseLinker {
            scope_positions: Vec::from([0]),
            current_recursive_ids: Vec::new(),
            id_positions: HashMap::new()
        }
    }

    fn current_position(&self) -> usize {
        *self.scope_positions.get(self.scope_positions.len() - 1)
            .expect("at least one scope should be available")
    }

    fn register(&mut self, id: UId) {
        self.id_positions.insert(id, self.current_position());
        let index = self.scope_positions.len() - 1;
        self.scope_positions.get_mut(index)
            .expect("at least one scope should be available")
            .add_assign(1);
    }

    fn dist_to_decl(&self, id: &UId) -> DistToDecl {
        let id_pos = self.id_positions.get(id)
            .expect(format!("variable '{}' should have been declared",
id.name).as_str());
        self.current_position() - id_pos
    }
}
```

Fichier /src/process/loose/link/pat.rs

```
use crate::lang::*;

use super::LooseLinker;


impl LooseLinker {
    pub fn pattern(&mut self, lt: Pattern<LT>) -> Pattern<LVT> {
        use PatternData as PD;

        let data = match lt.data {
            PD::Var(var) =>
                PD::Var(self.variable(var)),

            PD::Constructor { id, args } =>
                PD::Constructor {
                    id,
                    args: args.into_iter()
                        .map(|arg| self.pattern(arg))
                        .collect()
                }
        };

        Pattern { data, meta: lt.meta }
    }

    pub fn variable(&mut self, lt: Var<LT>) -> Var<LVT> {
        self.register(lt.id.clone());
        Var {
            id: LoId { name: lt.id.name },
            meta: lt.meta
        }
    }
}
```

Fichier /src/process/loose/link/prop.rs

```
use crate::lang::*;
use super::LooseLinker;


impl LooseLinker {
    pub fn property(&mut self, lt: Property<LT>) -> Property<LVT> {
        let (vars, left, right);

        id_scope!(self, {
            vars = lt.vars.into_iter()
                .map(|var| self.variable(var))
                .collect();

            left = self.expr(lt.left);
            right = self.expr(lt.right);
        });

        Property { vars, left, right }
    }
}
```

Fichier /src/process/loose/link/stmt.rs

```
use crate::lang::{kind::LVT, Statement, LT};

use super::LooseLinker;


impl LooseLinker {
    pub fn statements(&mut self, lt: Vec<Statement<LT>>)
        -> Vec<Statement<LVT>>
    {
        lt.into_iter()
            .map(|stmt| self.statement(stmt))
            .collect()
    }

    fn statement(&mut self, lt: Statement<LT>) -> Statement<LVT> {
        let var = self.variable(lt.var.clone());

        // Done after var for recursive reasons
        let val;
        recursive_detection_scope!(self, lt.var.id, {
            id_scope!(self, {
                val = self.expr(lt.val);
            });
        });

        Statement { var, val }
    }
}
```

Fichier /src/process/loose/link/typ.rs

```
use crate::lang::{kind::LVT, *};


impl From<TypeDef<LT>> for TypeDef<LVT> {
    fn from(lt: TypeDef<LT>) -> Self {
        TypeDef {
            id: lt.id,
            arg_ids: lt.arg_ids,
            typ: lt.typ.into()
        }
    }
}


impl From<TypeDefType<LT>> for TypeDefType<LVT> {
    fn from(lt: TypeDefType<LT>) -> Self {
        use TypeDefType::*;

        match lt {
            Type(typ) => Type(typ.into()),
            TypeSum(branches) =>
                TypeSum(
                    branches.into_iter()
                        .map(|branch| branch.into())
                        .collect()
                )
        }
    }
}


impl From<TypeSumBranch<LT>> for TypeSumBranch<LVT> {
    fn from(lt: TypeSumBranch<LT>) -> Self {
        TypeSumBranch {
            constructor_id: lt.constructor_id,
            args: lt.args.into_iter()
                .map(|arg| arg.into())
                .collect()
        }
    }
}


impl From<Type<LT>> for Type<LVT> {
    fn from(lt: Type<LT>) -> Self {
        use Type::*;

        match lt {
            Generic { id } =>
                Generic { id },

            Function { input, output } =>
                Function {
                    input: Box::new((*input).into()),
                    output: Box::new((*output).into())
                },

            Specialization { args, typ } =>
                Specialization {
                    args: args.into_iter()
                        .map(|arg| arg.into())
                        .collect(),
                    typ
                }
        }
    }
}
```

– Fin –