Le chameau et le serpent rentrent dans un bar: vérification quasi-automatique de code OCaml en logique de séparation

Charlène Gros[1,2] Mário Pereira[2]

[1]Junia ISEN Lille, Lille, 59000, France – Tarides, Paris, 75005, France
[2]Nova School of Science and Technology, NOVA LINCS, Portugal

30 Janvier 2025, JFLA

Gospel, pour raisonner sur les programmes OCaml

Gospel[1] est un langage de spécification formelle pour OCaml

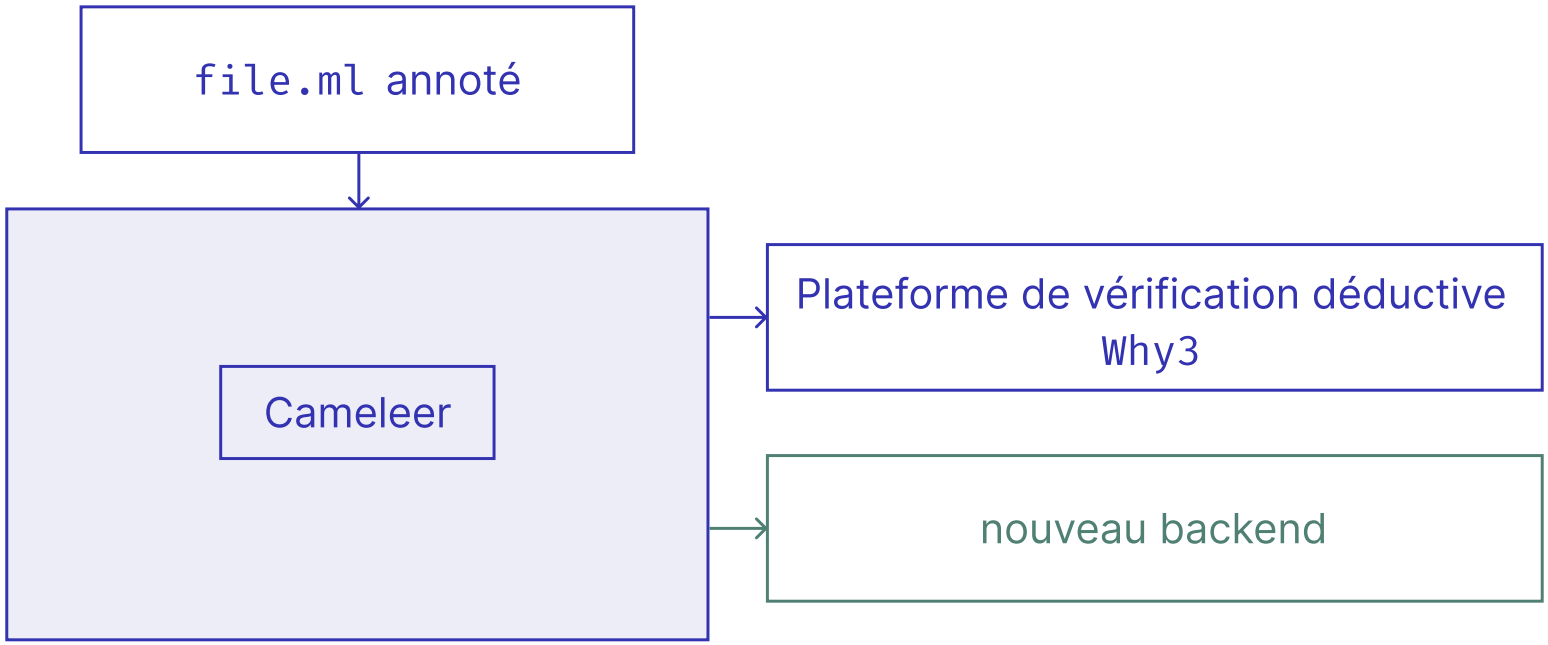
- apporte une sémantique formelle au langage,
- est inspiré de la logique de séparation,
- décrit les opérations en terme de contrats et de modèles symboliques.

À quoi ça sert ?

- documentation formelle,
- *runtime assertion checking* (Ortac),
- vérification déductive (Cameleer).

[1] `https://ocaml-gospel.github.io/gospel/`

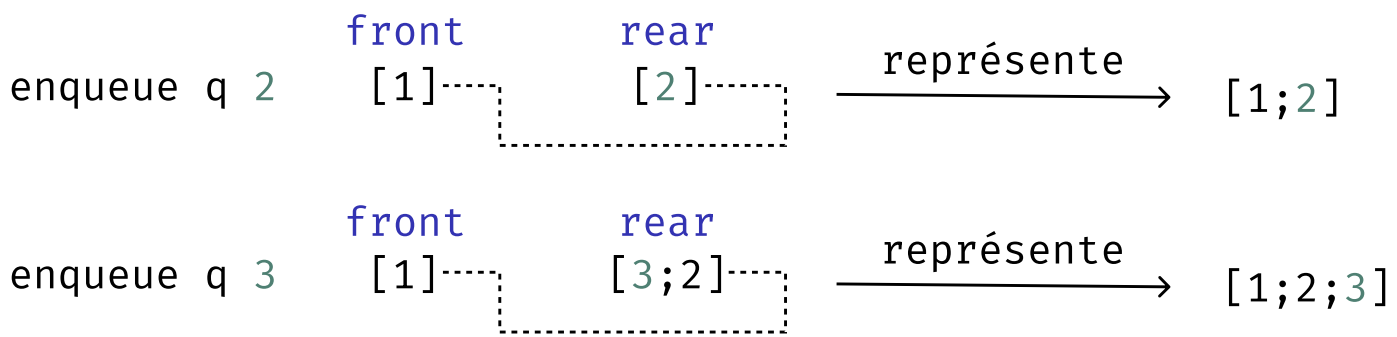
Vérification déductive en Cameleer



queue.ml

OCaml

```
type 'a queue = {
    mutable front: 'a list;
    mutable rear:  'a list;
}

let enqueue q e =
  match q.front with
  | [] -> q.front <- [e]
  | _  -> q.rear <- e :: q.rear
```



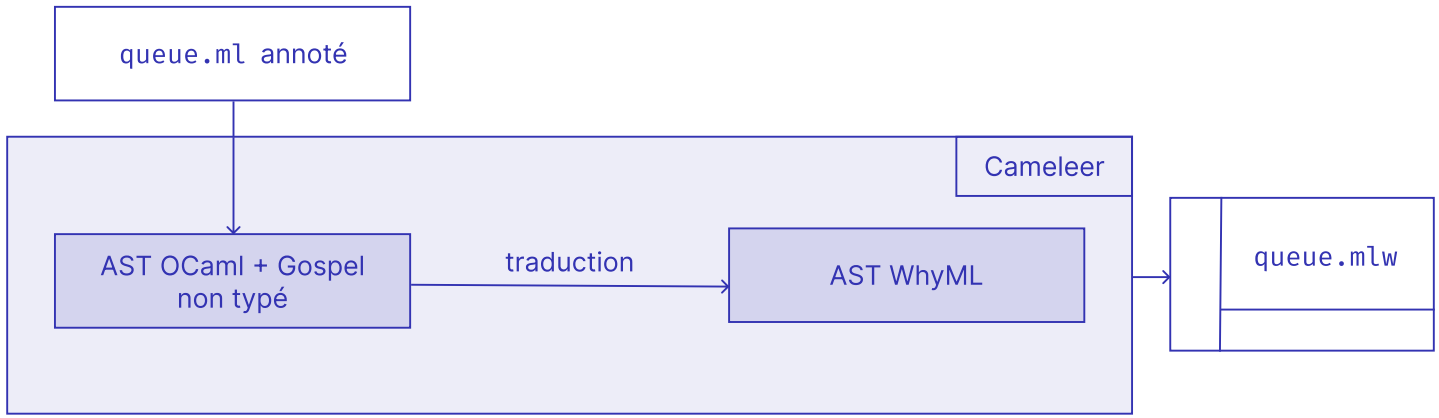
queue.ml spécifié en Gospel

OCaml + Gospel

```
type 'a queue = {
    mutable front: 'a list;
    mutable rear:  'a list;
    mutable view:  'a list; [@ghost]
}

(*@ invariant front = [] -> rear = []
    invariant view = front ++ List.rev rear *)

(*@ function length (q: 'a queue) : integer = List.length q.view *)

let enqueue q e =
  q.view <- q.view @ [e];
  match q.front with
  | [] -> q.front <- [e]
  | _  -> q.rear <- e :: q.rear
(*@ enqueue q e
    ensures length q = length (old q) + 1
    ensures q.view = (old q.view) @ (e :: []) *)
```

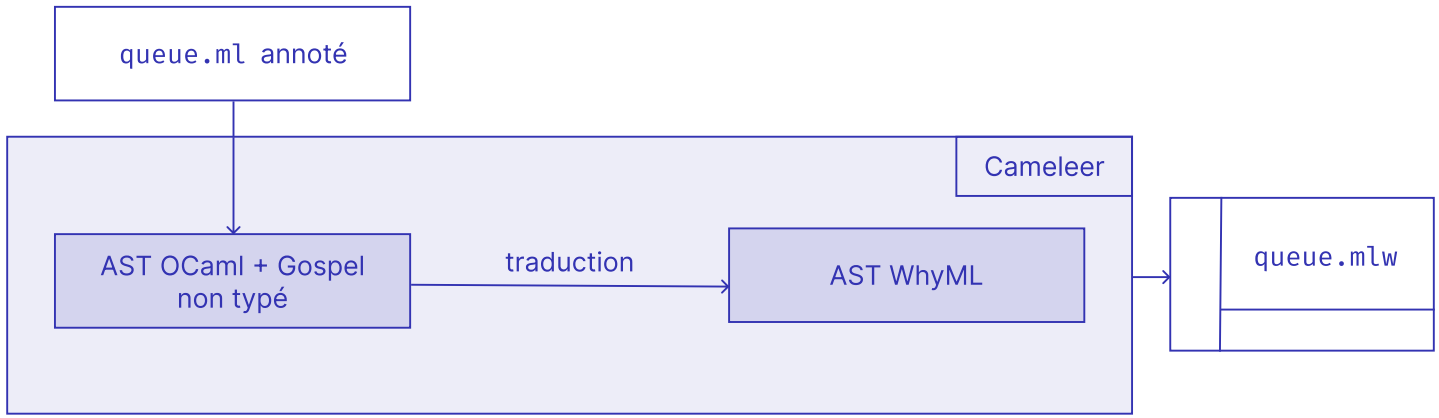
Backend Why3 et ses limites

Cameleer traduit le programme OCaml annoté en WhyML.



Backend Why3 et ses limites

Cameleer traduit le programme OCaml annoté en WhyML.



Cependant, Why3 contraint l'utilisation des données mutables.

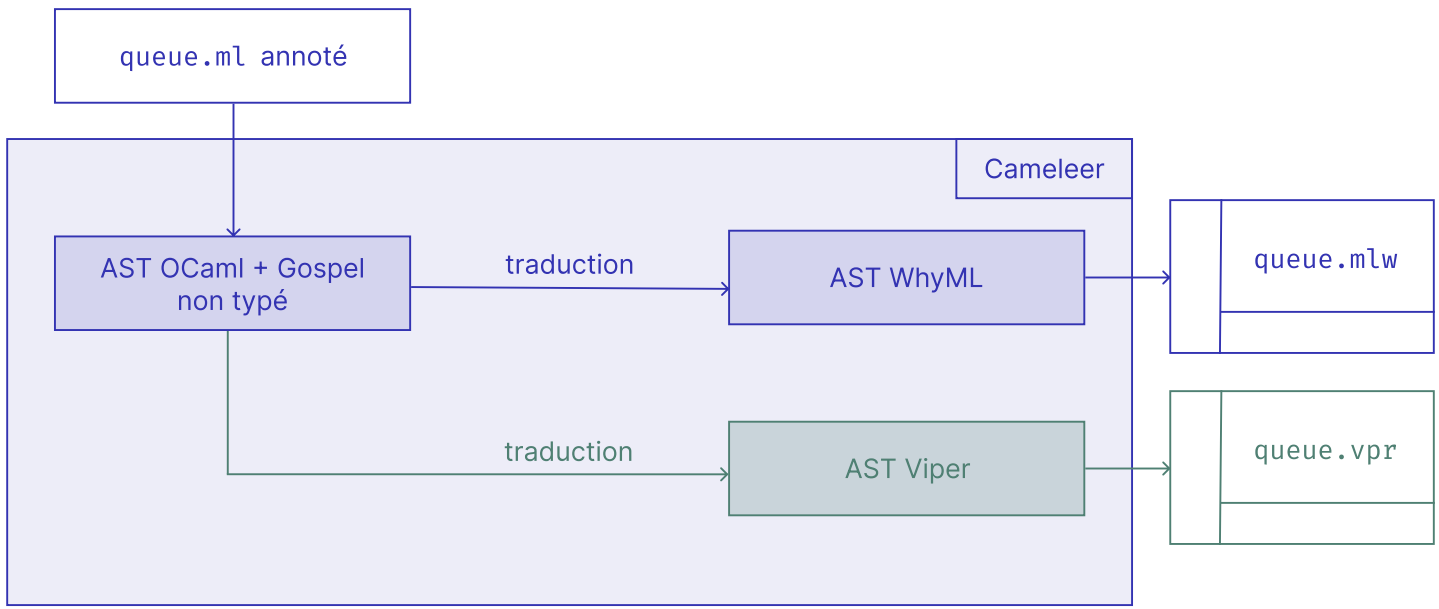
Sa gestion de l'*aliasing* empêche d'utiliser une donnée mutable

- dans une structure de données récursive,
- ou dans un type abstrait.

Notre objectif: ajouter un backend complémentaire

Comment la logique de séparation surmonte les limites de Why3 ?

- Raisonne sur des éléments disjoints de la mémoire grâce à la conjonction séparante ($*$).
- Peut décrire une hiérarchie arbitraire de structures de données récursives.
- Lorsqu'un nœud est modifié à un endroit, la logique garantit que les autres parties de la structure restent inchangées.



Viper[2] en quelques mots

Le langage intermédiaire Viper est

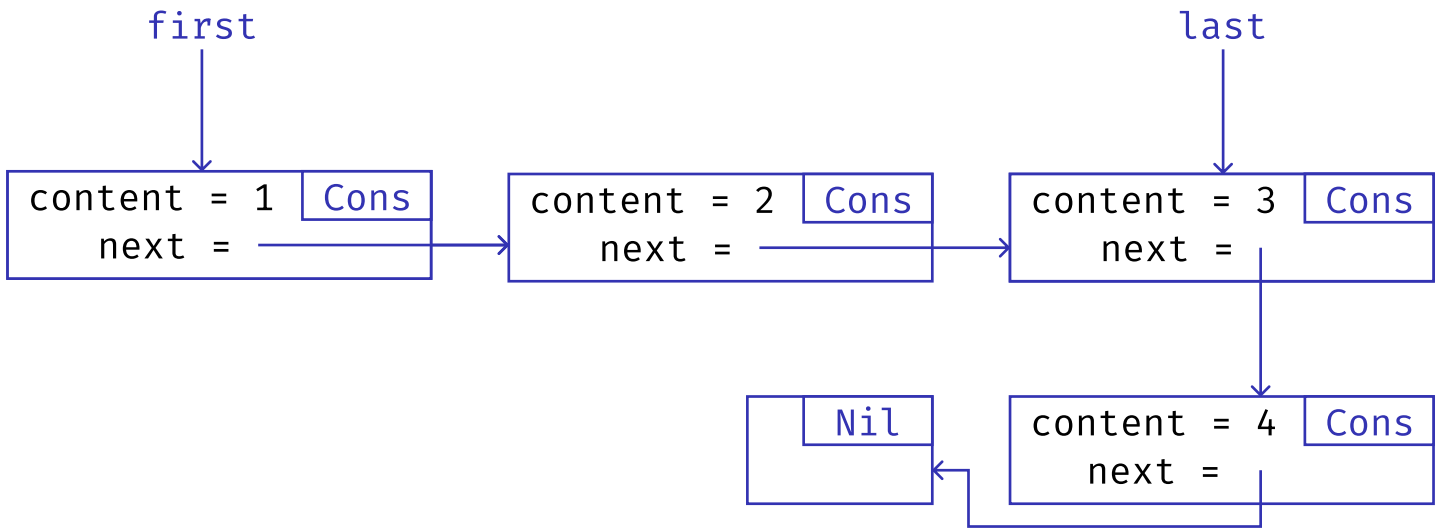
- un outil de vérification automatique,
- basé sur les *Implicit Dynamic Frames* (variante de la logique de séparation),
- déjà ciblé par de nombreux langages (Python: Nagini, Rust: Prusti, Go: Gobra, etc.).

[2] `https://viper.ethz.ch`

Running example : la structure de “segment de cellules”

Caractéristiques:

- `first` le début du segment,
- `last` la fin du segment qui est exclue,
- ici le segment de cellules contient `[1;2]`.



Définition des ressources mutables et types algébriques

Définition d'un `record` OCaml:

```
type cell =
  | Nil
  | Cons of {
      mutable content : int;
      mutable next : cell
    }
```

Définition des ressources mutables et types algébriques

Définition d'un `record` OCaml:

```
type cell =
  | Nil
  | Cons of {
      mutable content : int;
      mutable next : cell
    }
```

En Viper, cela se traduirait en:

```
adt Cell {
  Nil()
  Cons(cell: Ref)
}

field content : Int
field next    : Cell
```

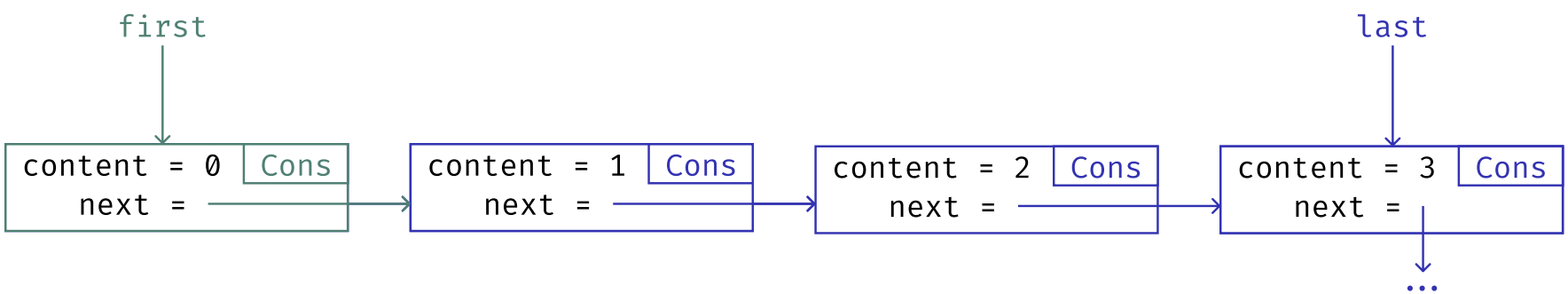
Quelques précisions concernant Viper.

- Le mot-clé `adt` introduit un type algébrique.
- Les données mutables sont déclarées globalement avec le mot-clé `field`.
- Le type `Ref` est une référence ayant accès à tous les `fields` déclarés.

Définition de prepend en OCaml

La fonction `prepend` ajoute une cellule en tête du segment.

```
type cellSeg = {
  mutable first : cell;
  mutable last  : cell
}

let prepend (first: cell) (last: cell) (x: int): cellSeg =
  let new_first : cell =
    Cons { content = x; next = first} in
  { first = new_first; last }
```



Définition de prepend en Viper

```
predicate CellSeg (first: Cell,
  v: Seq[Int], last: Cell)
{
  v == Seq[Int]() ?
    last == first
  : first.isCons &&
    let c == (first.cell) in
    acc (c.content) &&
    acc (c.next) &&
    c.content == v[0] &&
    CellSeg(c.next, v[1 ..], last)
}
```

```
method prepend(first: Cell,
  last: Cell, x: Int, v: Seq[Int])
  returns (new_first: Cell)
  requires CellSeg(first, v,  last)
  ensures  CellSeg(new_first,
           Seq(x) ++ v, last) {
  unfold CellSeg(first, v, last)
  var t: Ref; t := new(content, next)
  t.content := x
  t.next := first
  new_first := Cons(t)
  fold CellSeg(first, v, last)
  fold CellSeg(new_first,
    Seq(x) ++ v, last) }
```

Davantage d'éléments sont requis en Viper mais absents en Gospel.

- Définition d'un prédicat de représentation qui exprime:
  - la bonne formation de la structure de données,
  - la possession des champs avec acc.

Définition de prepend en Viper

```
predicate CellSeg (first: Cell,
  v: Seq[Int], last: Cell)
{
  v == Seq[Int]() ?
    last == first
  : first.isCons &&
    let c == (first.cell) in
    acc (c.content) &&
    acc (c.next) &&
    c.content == v[0] &&
    CellSeg(c.next, v[1 ..], last)
}
```

```
method prepend(first: Cell,
  last: Cell, x: Int, v: Seq[Int])
  returns (new_first: Cell)
  requires CellSeg(first, v,  last)
  ensures  CellSeg(new_first,
           Seq(x) ++ v, last) {
  unfold CellSeg(first, v, last)
  var t: Ref; t := new(content, next)
  t.content := x
  t.next := first
  new_first := Cons(t)
  fold CellSeg(first, v, last)
  fold CellSeg(new_first,
    Seq(x) ++ v, last) }
```

Davantage d'éléments sont requis en Viper mais absents en Gospel.

- Définition d'un prédicat de représentation qui exprime:
  - la bonne formation de la structure de données,
  - la possession des champs avec `acc`.
- Les opérations "origami" sur les prédicats avec `fold` et `unfold`.
  - `unfold` déplie le prédicat et fournit l'*ownership*,
  - `fold` le replie et vérifie la bonne formation et rends l'*ownership*.

Ajouts dans Gospel

Prédicat de représentation et *ownership*:

- utilisation du mot-clé `predicate`,
- opérateur "$\rightsquigarrow$" pour demander l'accès (se traduit en `acc...`),

```
(*@ predicate cellSeg (first: cell)
    (v: int sequence)
    (last: cell) =
  if v = empty then last = first
  else
    let Cons c = first in
    c ⇝ {content  next} &&&
    c.content = get v 0 &&&
    cellSeg c.next (tl v) last *)

let prepend (first: cell)
    (last: cell) (x: int): cellSeg =
  (*@ unfold cellSeg first v last *)
  let new_first : cell =
    Cons { content = x; next = first} in
  (*@ fold cellSeg first v last *)
  (*@ fold cellSeg new_first
        ((singleton x) ++ v) last *)
  { first = new_first; last }
  (*@ new_first = prepend first last
        x [v: int sequence]
      requires cellSeg first v last
      ensures  cellSeg new_first
        ((singleton x) ++ v) last *)
```

Ajouts dans Gospel

Prédicat de représentation et *ownership*:

- utilisation du mot-clé `predicate`,
- opérateur "$\rightsquigarrow$" pour demander l'accès (se traduit en `acc...`),

Opérations "origami":

- ajout des opérations `fold` et `unfold`,
- appels à ces opérations entre les lignes OCaml.

```
(*@ predicate cellSeg (first: cell)
    (v: int sequence)
    (last: cell) =
  if v = empty then last = first
  else
    let Cons c = first in
    c ⇝ {content  next} &&&
    c.content = get v 0 &&&
    cellSeg c.next (tl v) last *)

let prepend (first: cell)
    (last: cell) (x: int): cellSeg =
  (*@ unfold cellSeg first v last  *)
  let new_first : cell =
    Cons { content = x; next = first} in
  (*@ fold cellSeg first v last  *)
  (*@ fold cellSeg new_first
        ((singleton x) ++ v) last  *)
  { first = new_first; last }
  (*@ new_first = prepend first last
        x [v: int sequence]
      requires cellSeg first v last
      ensures  cellSeg new_first
        ((singleton x) ++ v) last *)
```

Conclusion

Réalisation:

- extension du langage Gospel,
- ajout du *backend* Viper supportant la logique de séparation,
- possibilité de prouver formellement davantage de programmes OCaml,
- étude de cas : module `Queue` de OCaml.

Cette première version du *backend* vers Viper supporte

- les structures utilisant les entiers et modélisées par des séquences.

Avec l'annotation explicite pour

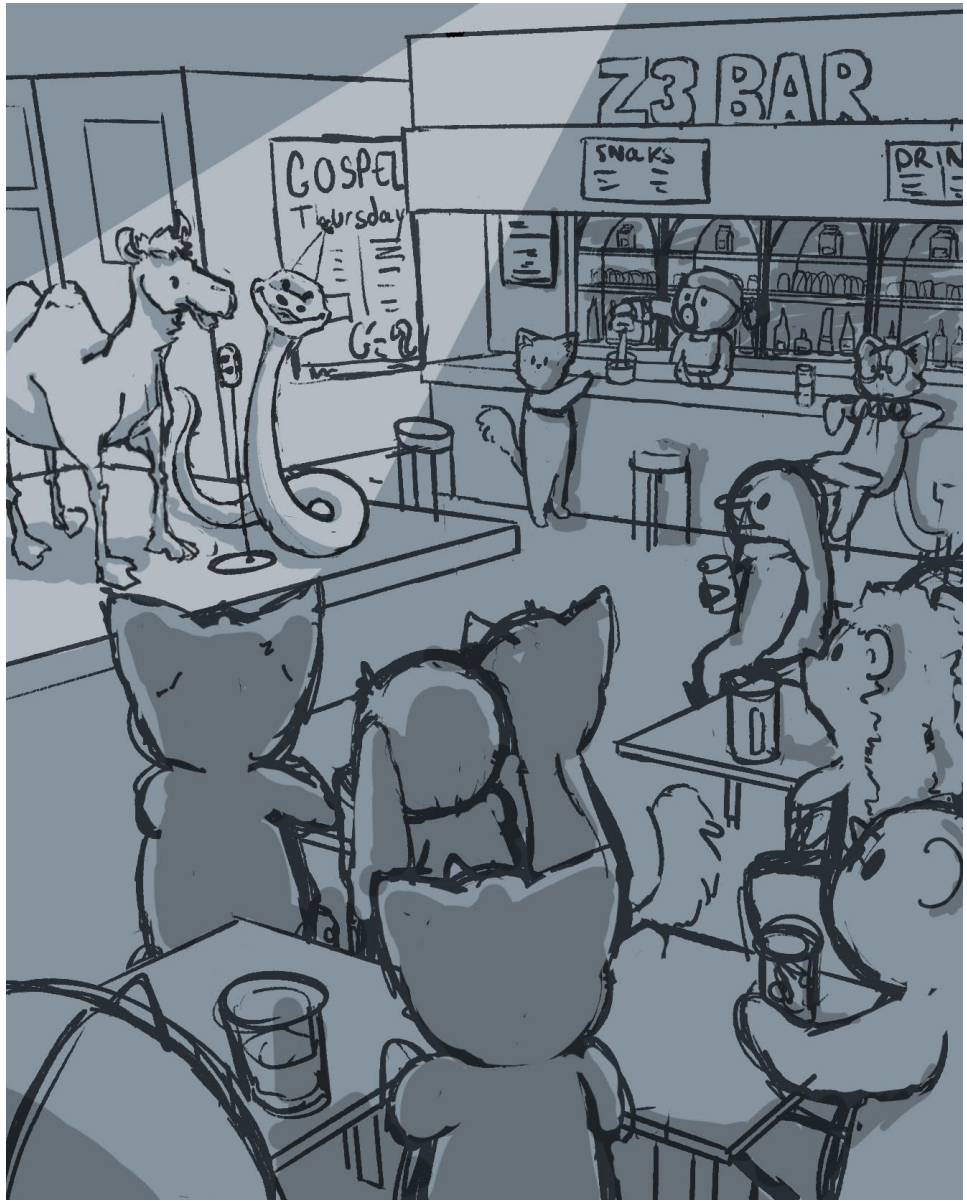
- les types en OCaml et Gospel,
- l'accessibilité aux champs avec le nouvel opérateur "$\rightsquigarrow$",
- le pliage et dépliage des prédicats de représentation.

Perspectives futures

Ajouter davantage d'études de cas et d'exemples.

Implémenter les autres types et structures de contrôle.

Ajouter l'inférence de type.



Versions utilisées

- `ocaml 4.14.2` (but works on further versions).
- cameleer:
  `ocaml-gospel/cameleer:master`
- gospel:
  `mariojppereira/gospel:implementations_gospel`.
- gospel2viper:
  `ocaml-gospel/gospel2viper:main`.