Kindly Bent To Free Us

Gabriel Radanne Peter Thiemann

November 22, 2018

```
val Tls_lwt.of_t : Tls_lwt.Unix.t -> in * out
(* Turn a file descr into input/output channels *)
```

```
let fd : Tls_lwt.Unit.t = .....
let input, output = Tls_lwt.of_t fd
... (* read some things *)
let%lwt () = Lwt_io.close input in
...
let%lwt c = Lwt_io.write output "thing" in (*Oups*)
...
```

The default behavior is to close the underlying file description when a channel is closed.

```
val Tls_lwt.of_t : Tls_lwt.Unix.t -> in * out
(* Turn a file descr into input/output channels *)

let fd : Tls_lwt.Unit.t = .....
let input, output = Tls_lwt.of_t fd
... (* read some things *)
let%lwt () = Lwt_io.close input in
...
let%lwt c = Lwt_io.write output "thing" in (*Oups*)
...
```

The default behavior is to close the underlying file description when a channel is closed.

```
val Tls_lwt.of_t : Tls_lwt.Unix.t -> in * out
(* Turn a file descr into input/output channels *)


let fd : Tls_lwt.Unit.t = .....
let input, output = Tls_lwt.of_t fd
... (* read some things *)
let%lwt () = Lwt_io.close input in
...
let%lwt c = Lwt_io.write output "thing" in (*Oups*)
...
```

The default behavior is to close the underlying file description when a channel is closed.

```
val Tls_lwt.of_t : Tls_lwt.Unix.t -> in * out
(* Turn a file descr into input/output channels *)


let fd : Tls_lwt.Unit.t = .....
let input, output = Tls_lwt.of_t fd
... (* read some things *)
let%lwt () = Lwt_io.close input in
...
let%lwt c = Lwt_io.write output "thing" in (*Oups*)
...
```

The default behavior is to close the underlying file description when a channel is closed.

Many partial solutions

- Closures
- Monads
- Existential types
- ...

What we really need is to enforce linearity.

Many partial solutions

- Closures
- Monads
- Existential types
- ...

What we really need is to enforce linearity.

Many places in OCaml where enforcing linearity is useful:

- IO (File handle, channels, network connections, . . .)
- Protocols (With session types! Mirage libraries)
- One-shot continuations (effects!)
- Transient data-structures
- C-style “struct parsing”
- . . .

Which kind of linearity?

- Ownership approaches
- Capabilities and typestates
- Substructural type systems
- ...

Which kind of linearity?

- Ownership approaches
  Suitable to imperative languages (Rust, . . .).
- Capabilities and typestates
- Substructural type systems
- . . .

Which kind of linearity?

- Ownership approaches
- Capabilities and typestates
  Often use in Object-Oriented contexts (Wyvern, Plaid, Hopkins Objects Group, . . .).
- Substructural type systems
- . . .

Which kind of linearity?

- Ownership approaches
- Capabilities and typestates
- Substructural type systems
  Many variations, mostly in functional languages:
  - Inspired directly from linear logic (Linear Haskell, Walker, . . .)
  - Uniqueness (Clean)
  - Kinds (Alms, Clean, F°)
  - Constraints (Quill)
- . . .

Which kind of linearity?

- Ownership approaches
- Capabilities and typestates
- Substructural type systems
- . . .

  Mix of everything: Mezzo

Which kind of linearity?

- Ownership approaches
- Capabilities and typestates
- Substructural type systems
- ...

Goals:

- Complete and principal type inference
- Impure strict context
- Works well with type abstraction
- Play balls with various other ongoing works (Effects, Resource polymorphism, . . .)

Non Goals:

- Support every linear code pattern under the sun
- Design associated compiler optimisations/GC integration (yet)

Goals:

- Complete and principal type inference
- Impure strict context
- Works well with type abstraction
- Play balls with various other ongoing works (Effects, Resource polymorphism, . . .)

Non Goals:

- Support every linear code pattern under the sun
- Design associated compiler optimisations/GC integration (yet)

The Affe language

Types and Behaviors

In Affe, the behavior of a variable is determined by its type:

```
type channel : A (* channel is Affine! *)


let with_file s f =
  let c = open_channel s
  let c = f c in
  close_channel c
val with_file : string -> (channel -> channel)


let () =
  let r = ref None in
  with_file "thing"
    (fun c -> r := Some c ; c) (* ✘ No! *)
```

Types and Behaviors

In Affe, the behavior of a variable is determined by its type:

```
type channel : A (* channel is Affine! *)


let with_file s f =
  let c = open_channel s
  let c = f c in
  close_channel c
val with_file : string -> (channel -> channel)


let () =
  let r = ref None in
  with_file "thing"
    (fun c -> r := Some c ; c) (* ✘ No! *)
```

Types and Behaviors

In Affe, the behavior of a variable is determined by its type:

```
type channel : A (* channel is Affine! *)


let with_file s f =
  let c = open_channel s
  let c = f c in
  close_channel c
val with_file : string -> (channel -> channel)


let () =
  let r = ref None in
  with_file "thing"
    (fun c -> r := Some c ; c) (* ✘ No! *)
```

Inference at work

Infer unrestricted in case of duplication:

```
let f = fun c -> r := Some c ; c
val f : ('a : U) . 'a -> 'a
```

The kinds so far

So far, two kinds:

- **A** Affine types: can be used at most once
- **U** Unrestricted types

Additionally, we have:

$$\mathbf{U} \leq \mathbf{A}$$

Inference at work

What about closures?

```
let f = fun a -> fun b -> (a, b)
val f : 'a -> 'b -> 'a * 'b (* ? *)
```

Inference at work

What about closures?

```
let f = fun a -> fun b -> (a, b)
val f : ('a : 'k) => 'a -> 'b -{'k}> 'a * 'b
```

Inference at work

```
let app f x = f x
val app :
  'k1 < 'k2 =>
  ('a -{'k1}> 'b) -> 'a -{'k2}> 'b
```

Inference at work

What about more complicated cases ?

```
let compose f g x = f (g x)
val compose :
  ('b -{?}> 'a) -> ('c -{?}> 'b) -{?}> 'c -{?}> 'a
```

Inference at work

What about more complicated cases ?

```
let compose f g x = f (g x)
val compose :
  ('k2 < 'k) & ('k1 < 'k) & ('k1 < 'k3) =>
  ('b -{'k1}> 'a) -> ('c -{'k2}> 'b) -{'k3}>
  'c -{'k}> 'a
```

Inference at work

What about more complicated cases ?

```
let compose f g x = f (g x)
val compose :
  ('k1 < 'k) =>
  ('b -{'k1}> 'a) -> ('c -{'k}> 'b) -{'k1}>
  'c -{'k}> 'a
```

Closer look at type declarations

You can annotate the kinds on type declarations.

Vanilla OCaml references are fully unrestricted:

```
type ('a : U) ref : U = ...
```

We can also have constraints on kinds. The pair type operator:

```
type * : (k1 < k) & (k2 < k) => k1 -> k2 -> k
```

Closer look at type declarations

You can annotate the kinds on type declarations.

Vanilla OCaml references are fully unrestricted:

```
type ref : U -> U = ...
```

We can also have constraints on kinds. The pair type operator:

```
type * : (k1 < k) & (k2 < k) => k1 -> k2 -> k
```

Closer look at type declarations

You can annotate the kinds on type declarations.

Vanilla OCaml references are fully unrestricted:

```
type ref : U -> U = ...
```

We can also have constraints on kinds. The pair type operator:

```
type * : (k1 < k) & (k2 < k) => k1 -> k2 -> k
```

More interesting example

Mixing with abstraction:

```
module LinArray : sig
  type -'a w : A
  val create :
    ('a : U) . int -> 'a -> 'a w
  val set : 'a w -> int -{A}> 'a -{A}> 'a w

  type +'a r : U
  val freeze : 'a w -> 'a r
  val get : int -> 'a r -> 'a
end
```

The calculus

The calculus

Expressions

$$
\begin{aligned}
e ::= {} & c \mid x \mid (e\ e') \mid \lambda x.e \\
& \mid \texttt{let}\ x = e\ \texttt{in}\ e' \\
& \mid (K\ e) \mid \texttt{elim}_K\ e
\end{aligned}
$$

Type Expressions

$$
\begin{aligned}
\tau ::= {} & \alpha \mid \tau \xrightarrow{\mathsf{k}} \tau \mid (\tau^*)\ \mathrm{t} \\
k ::= {} & \kappa \mid \ell \in \mathcal{L}
\end{aligned}
$$

Constraints are only acceptable in schemes:

$$\sigma ::= \forall \kappa^* \forall (\alpha : k)^* . (C \Rightarrow \tau)$$
$$\theta ::= \forall \kappa^* . (C \Rightarrow k_i^* \to k)$$

The constraint language in schemes is limited to list of inequalities:

$$C ::= (k \leq k')^*$$

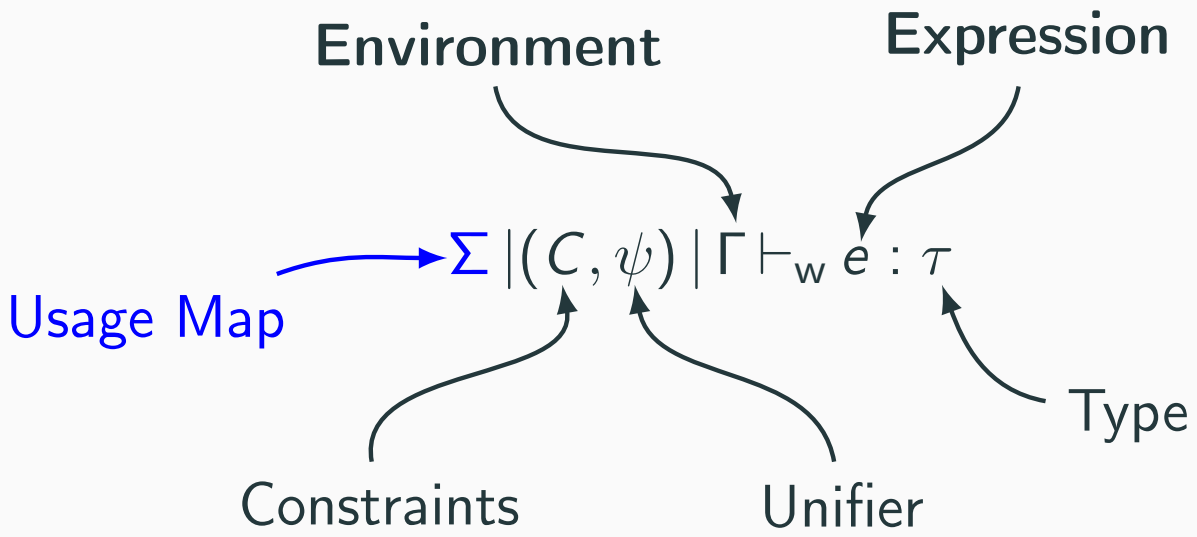
The HM(X) framework

HM(X) (Odersky et al., 1999) is a framework to build an HM type system (with inference) based on a given constraint system.

We provide two additions:

- A small extension of HM(X) that tracks kinds and linearity
- An appropriate constraint system

Typing


Environment
Expression
$\Sigma \mid (C, \psi) \mid \Gamma \vdash_{\mathsf{w}} e : \tau$
Usage Map
Type
Constraints
Unifier

Variables can be kind-polymorphic and all their instances might not have the same kinds.

$\Longrightarrow$ We must track the kinds of all use-sites for each variable.

Use maps (Σ) associates variables to multisets of kinds and are equipped with three operations:

$$\Sigma \cap \Sigma' \qquad \Sigma \cup \Sigma' \qquad \Sigma \leq k$$

Variables can be kind-polymorphic and all their instances might not have the same kinds.

$\Longrightarrow$ We must track the kinds of all use-sites for each variable.

Use maps (Σ) associates variables to multisets of kinds and are equipped with three operations:

$$\Sigma \cap \Sigma' \qquad \Sigma \cup \Sigma' \qquad \Sigma \leq k$$

When typechecking (e1, e2):

- $\Sigma_1 \,|(C_1, \psi_1)\,|\, \Gamma \vdash_{\mathsf{w}} e_1 : \tau_1$
- $\Sigma_2 \,|(C_2, \psi_2)\,|\, \Gamma \vdash_{\mathsf{w}} e_2 : \tau_2$
- Add $(\Sigma_1 \cap \Sigma_2 \leq \mathbf{U})$ to the constraints
- ...
- Return $\Sigma_1 \cup \Sigma_2$

When typechecking (e1, e2):

- $\Sigma_1 \,|(C_1, \psi_1)\,|\, \Gamma \vdash_{\mathsf{w}} e_1 : \tau_1$
- $\Sigma_2 \,|(C_2, \psi_2)\,|\, \Gamma \vdash_{\mathsf{w}} e_2 : \tau_2$
- Add $(\Sigma_1 \cap \Sigma_2 \leq \mathbf{U})$ to the constraints
- ...
- Return $\Sigma_1 \cup \Sigma_2$

When typechecking (e1, e2):

- $\Sigma_1 \,|(C_1, \psi_1)\,|\, \Gamma \vdash_{\mathsf{w}} e_1 : \tau_1$
- $\Sigma_2 \,|(C_2, \psi_2)\,|\, \Gamma \vdash_{\mathsf{w}} e_2 : \tau_2$
- Add $(\Sigma_1 \cap \Sigma_2 \leq \mathbf{U})$ to the constraints
- ...
- Return $\Sigma_1 \cup \Sigma_2$

When typechecking (e1, e2):

- $\Sigma_1 \,|(C_1, \psi_1)\,|\, \Gamma \vdash_{\mathsf{w}} e_1 : \tau_1$
- $\Sigma_2 \,|(C_2, \psi_2)\,|\, \Gamma \vdash_{\mathsf{w}} e_2 : \tau_2$
- Add $(\Sigma_1 \cap \Sigma_2 \leq \mathbf{U})$ to the constraints
- ...
- Return $\Sigma_1 \cup \Sigma_2$

When typechecking (e1, e2):

- $\Sigma_1 \,|(C_1, \psi_1)\,|\, \Gamma \vdash_{\mathsf{w}} e_1 : \tau_1$
- $\Sigma_2 \,|(C_2, \psi_2)\,|\, \Gamma \vdash_{\mathsf{w}} e_2 : \tau_2$
- Add $(\Sigma_1 \cap \Sigma_2 \leq \mathbf{U})$ to the constraints
- ...
- Return $\Sigma_1 \cup \Sigma_2$

Constraints

A slightly more general context: $\mathcal{C}_{\mathcal{L}}$ is the constraint system:

$$C ::= (\tau_1 \leq \tau_2) \mid (k_1 \leq k_2) \mid C_1 \wedge C_2 \mid \exists \alpha . C$$

where $k ::= \kappa \mid \ell \in \mathcal{L}$ and $(\mathcal{L}, \leq_{\mathcal{L}})$ is a complete lattice.

Respect, among other things:

$$\frac{\ell \leq_{\mathcal{L}} \ell'}{\vdash_{\mathsf{e}} (\ell \leq \ell')} \qquad \vdash_{\mathsf{e}} (k \leq \ell^{\top}) \qquad \vdash_{\mathsf{e}} (\ell^{\bot} \leq k)$$

Constraints

A slightly more general context: $\mathcal{C}_{\mathcal{L}}$ is the constraint system:

$$C ::= (\tau_1 \leq \tau_2) \mid (k_1 \leq k_2) \mid C_1 \wedge C_2 \mid \exists \alpha . C$$

where $k ::= \kappa \mid \ell \in \mathcal{L}$ and $(\mathcal{L}, \leq_{\mathcal{L}})$ is a complete lattice.

Respect, among other things:

$$\frac{\ell \leq_{\mathcal{L}} \ell'}{\vdash_{\mathsf{e}} (\ell \leq \ell')} \qquad \vdash_{\mathsf{e}} (k \leq \ell^{\top}) \qquad \vdash_{\mathsf{e}} (\ell^{\bot} \leq k)$$

Example : $\lambda f.\lambda x.((f\ x), x)$

Raw constraints:

$$(\alpha_f : \kappa_f)(\alpha_x : \kappa_x)\ldots$$

$$(\alpha_f \leq \gamma \xrightarrow{\kappa_1} \beta) \wedge (\gamma \leq \alpha_x) \wedge (\beta * \alpha_x \leq \alpha_r) \wedge (\kappa_x \leq \mathbf{U})$$

We unify the types and discover new constraints:

$$\alpha_r = (\gamma \xrightarrow{\kappa_3} \beta) \xrightarrow{\kappa_2} \gamma \xrightarrow{\kappa_1} \beta * \gamma$$

$$(\kappa_x \leq \mathbf{U}) \wedge (\kappa_\gamma \leq \kappa_x) \wedge (\kappa_x \leq \kappa_r) \wedge (\kappa_\beta \leq \kappa_r) \wedge (\kappa_3 \leq \kappa_f) \wedge (\kappa_f \leq \kappa_1)$$

Example : $\lambda f.\lambda x.((f\ x), x)$

Raw constraints:

$$(\alpha_f : \kappa_f)(\alpha_x : \kappa_x)\ldots$$

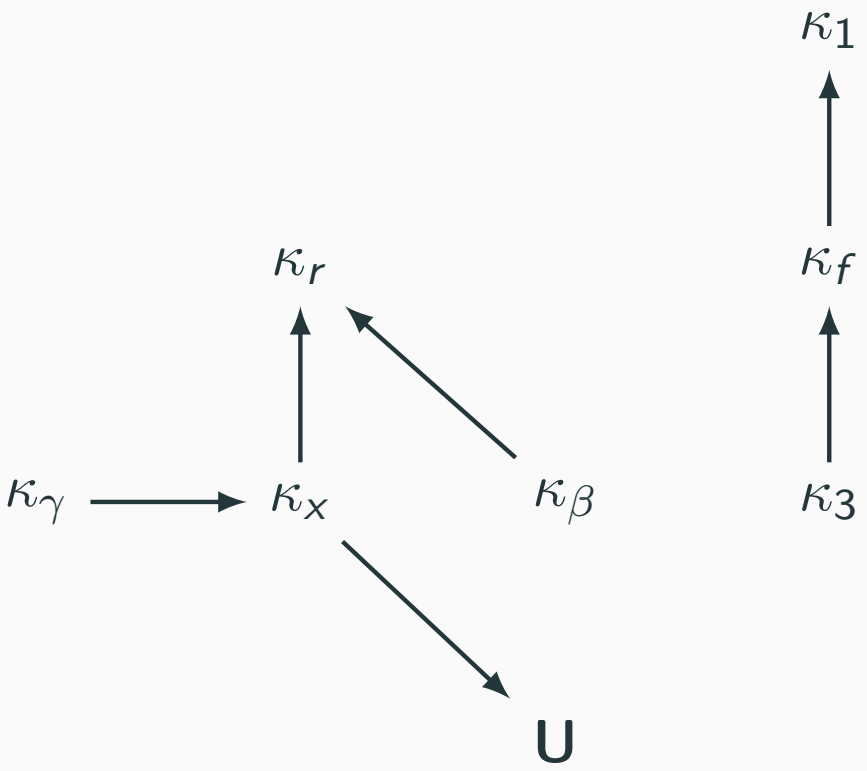
$$(\alpha_f \leq \gamma \xrightarrow{\kappa_1} \beta) \wedge (\gamma \leq \alpha_x) \wedge (\beta * \alpha_x \leq \alpha_r) \wedge (\kappa_x \leq \mathbf{U})$$

We unify the types and discover new constraints:

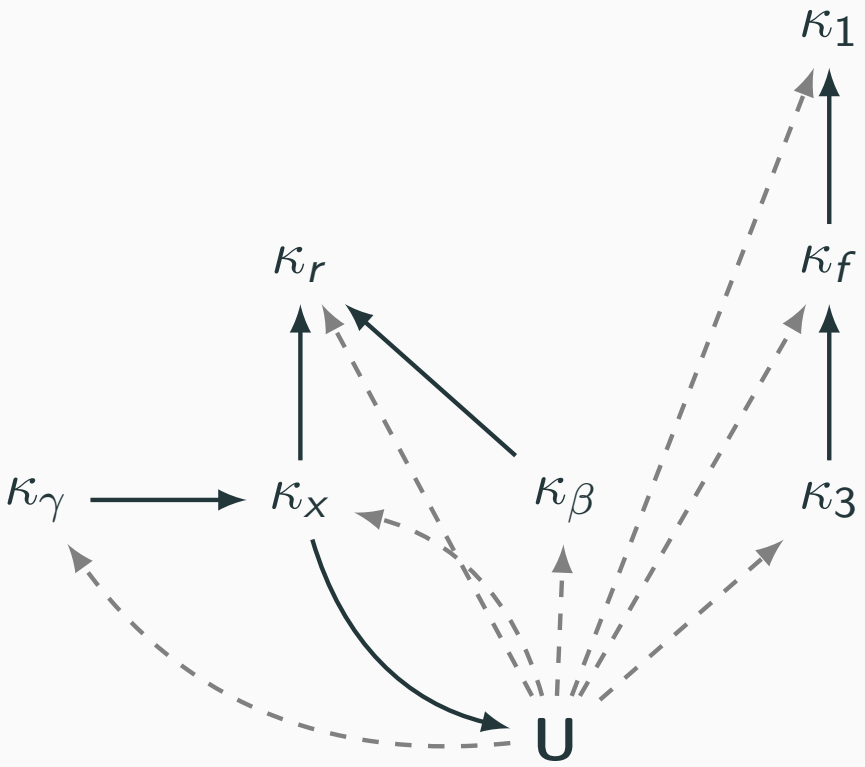
$$\alpha_r = (\gamma \xrightarrow{\kappa_3} \beta) \xrightarrow{\kappa_2} \gamma \xrightarrow{\kappa_1} \beta * \gamma$$

$$(\kappa_x \leq \mathbf{U}) \wedge (\kappa_\gamma \leq \kappa_x) \wedge (\kappa_x \leq \kappa_r) \wedge (\kappa_\beta \leq \kappa_r) \wedge (\kappa_3 \leq \kappa_f) \wedge (\kappa_f \leq \kappa_1)$$

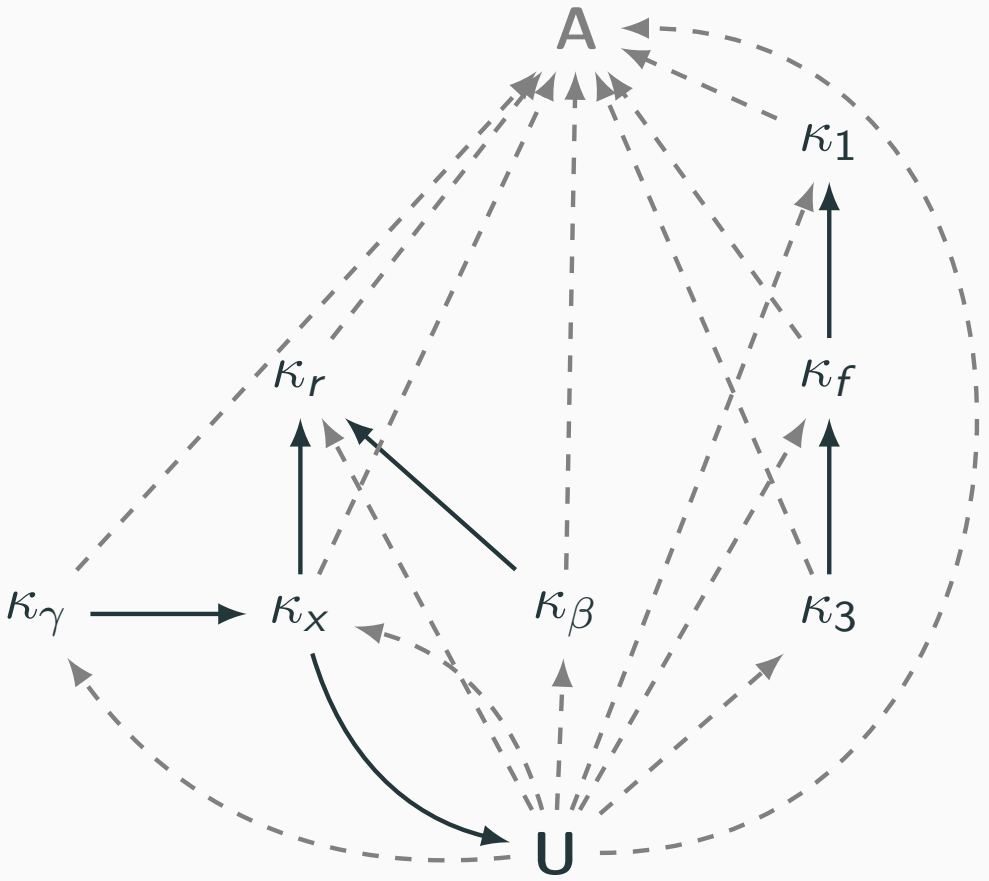
$$(\gamma : \kappa_\gamma)(\beta : \kappa_\beta).\ (\gamma \xrightarrow{\kappa_3} \beta) \xrightarrow{\kappa_2} \gamma \xrightarrow{\kappa_1} \beta * \gamma$$



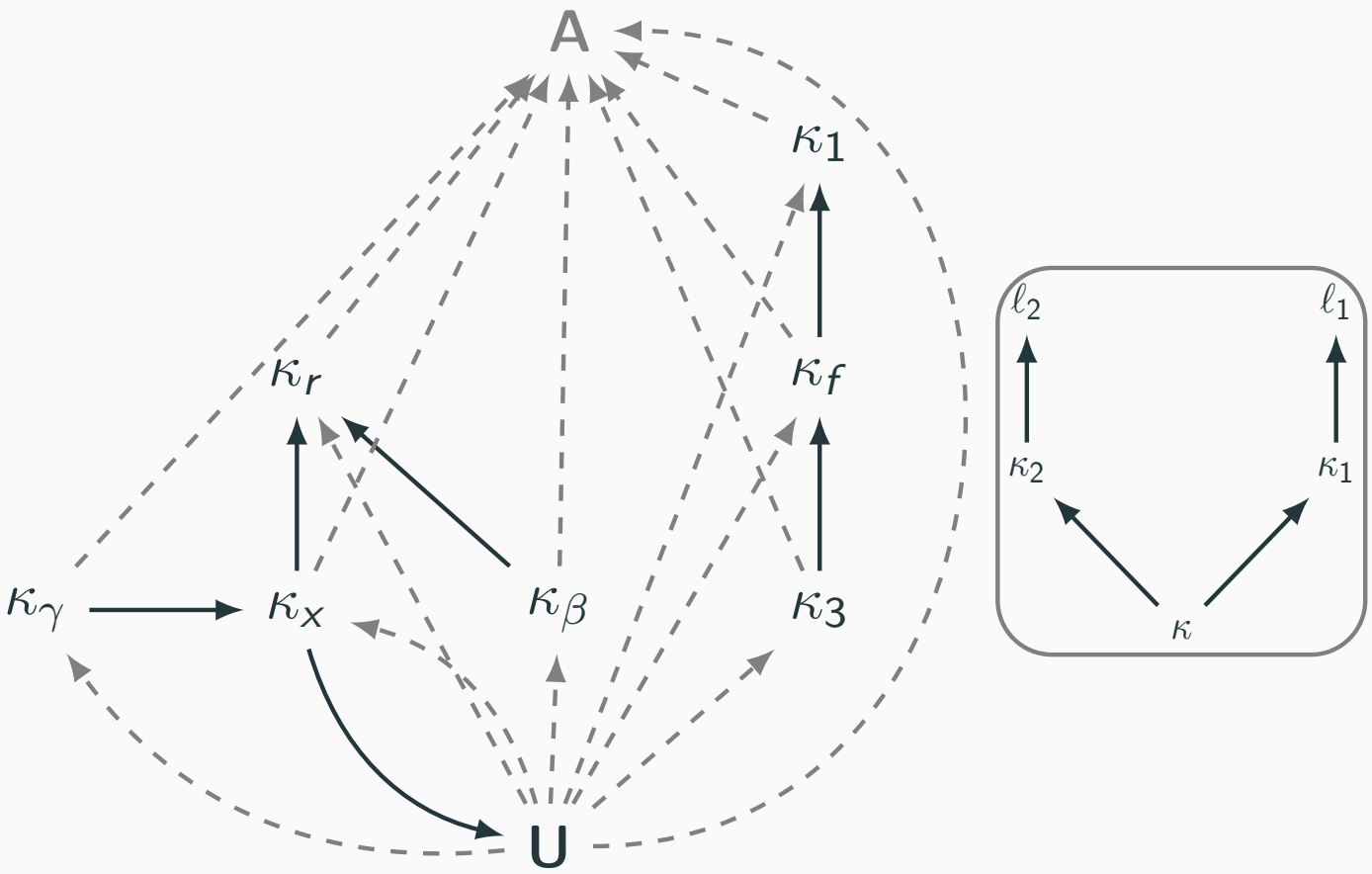
$$(\gamma : \kappa_\gamma)(\beta : \kappa_\beta).\ (\gamma \xrightarrow{\kappa_3} \beta) \xrightarrow{\kappa_2} \gamma \xrightarrow{\kappa_1} \beta * \gamma$$



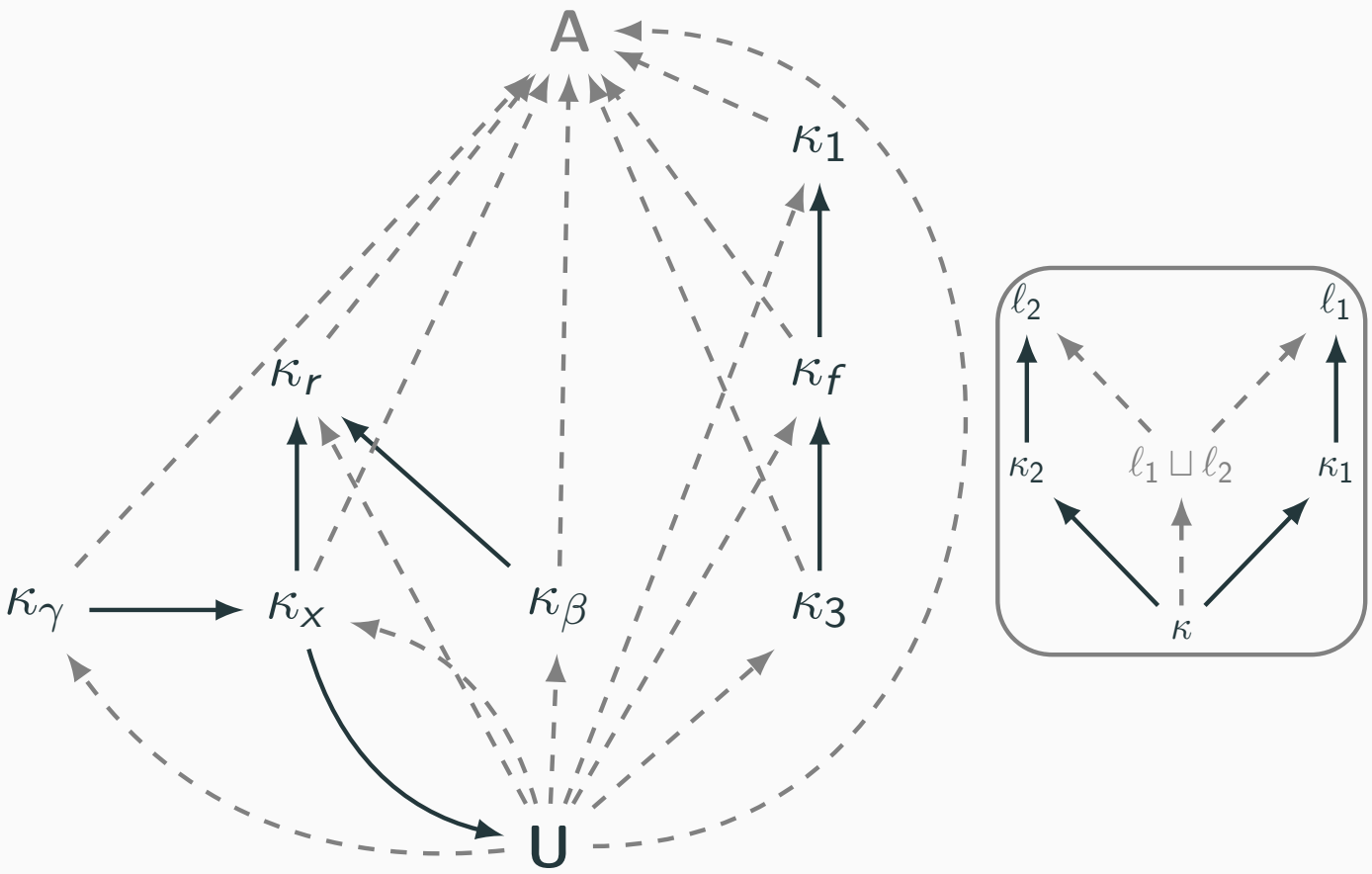
$$(\gamma : \kappa_\gamma)(\beta : \kappa_\beta).\ (\gamma \xrightarrow{\kappa_3} \beta) \xrightarrow{\kappa_2} \gamma \xrightarrow{\kappa_1} \beta * \gamma$$



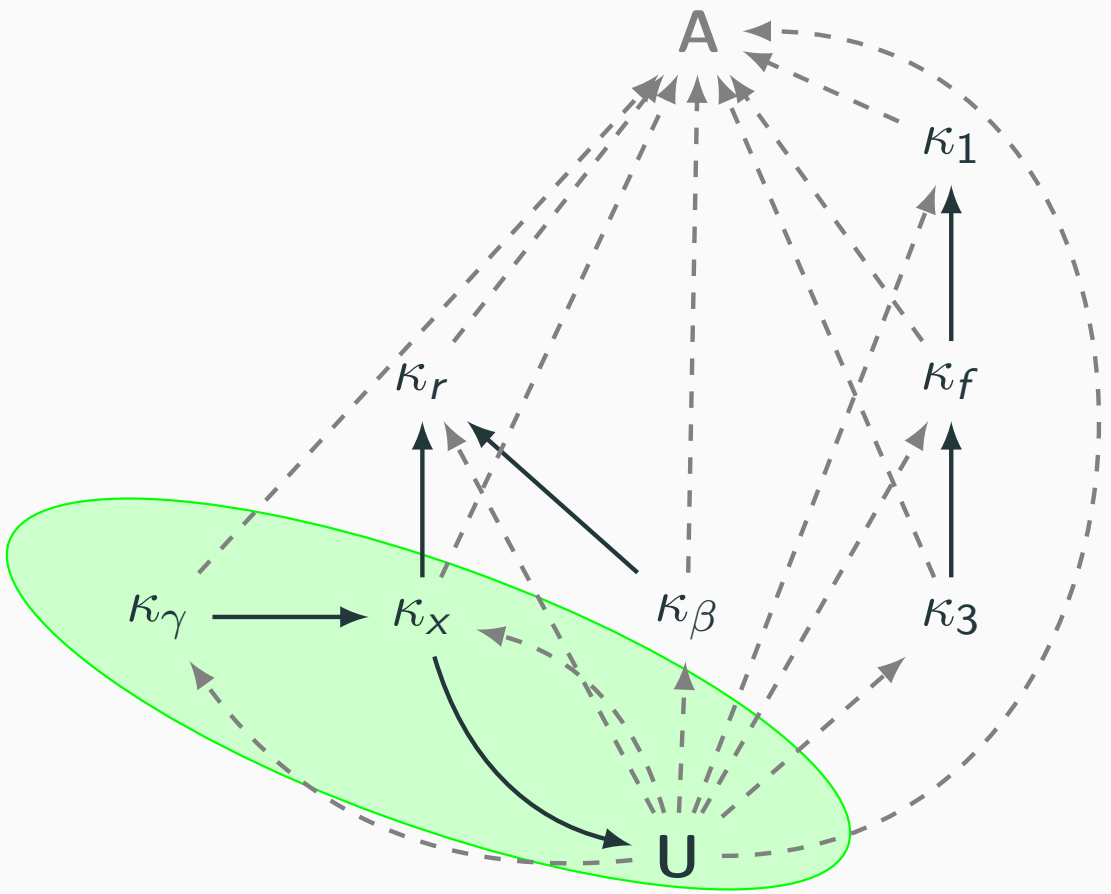
$$(\gamma : \kappa_\gamma)(\beta : \kappa_\beta).\ (\gamma \xrightarrow{\kappa_3} \beta) \xrightarrow{\kappa_2} \gamma \xrightarrow{\kappa_1} \beta * \gamma$$



$$(\gamma : \kappa_\gamma)(\beta : \kappa_\beta).\ (\gamma \xrightarrow{\kappa_3} \beta) \xrightarrow{\kappa_2} \gamma \xrightarrow{\kappa_1} \beta * \gamma$$

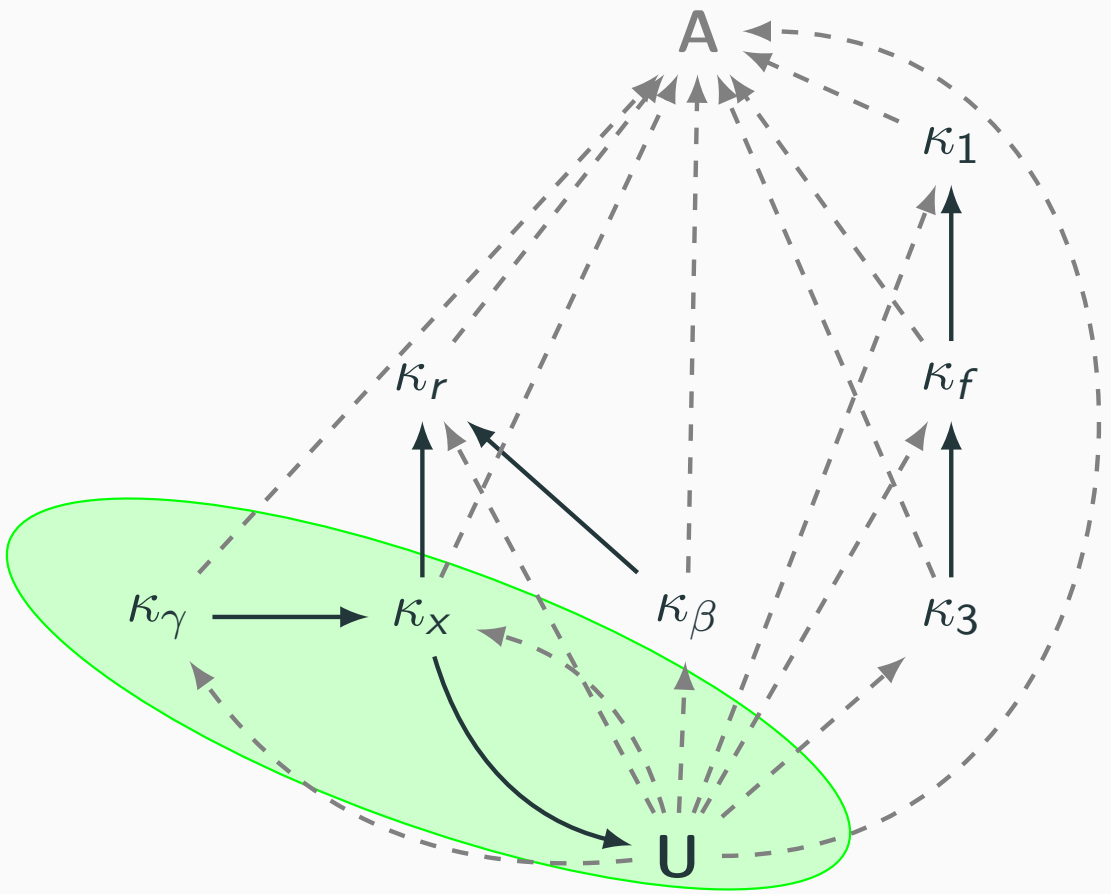


$$(\gamma : \kappa_\gamma)(\beta : \kappa_\beta).\ (\gamma \xrightarrow{\kappa_3} \beta) \xrightarrow{\kappa_2} \gamma \xrightarrow{\kappa_1} \beta * \gamma$$



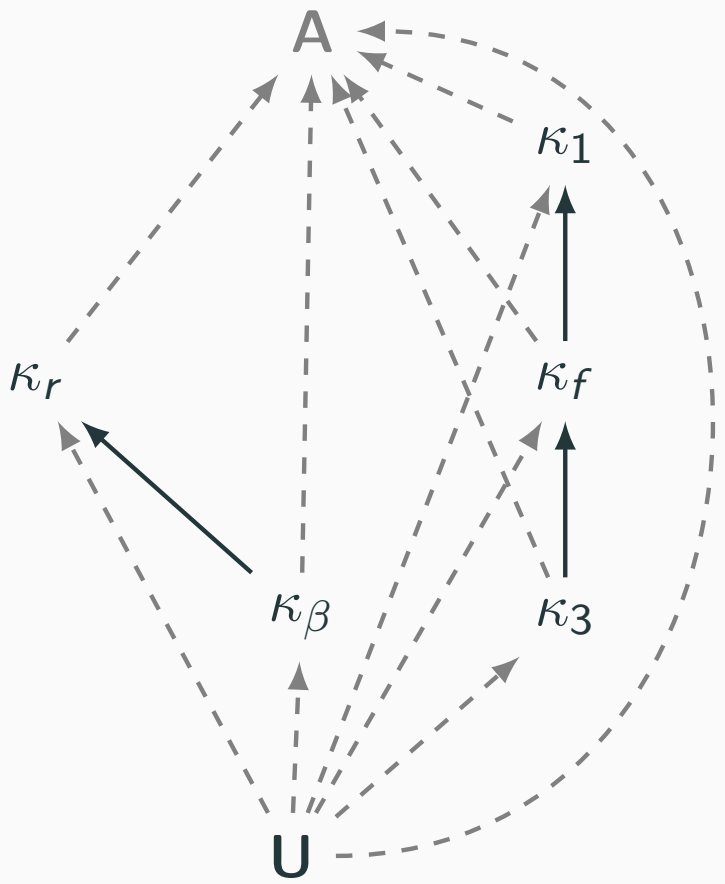
$$(\gamma : \kappa_\gamma)(\beta : \kappa_\beta).\ (\gamma \xrightarrow{\kappa_3} \beta) \xrightarrow{\kappa_2} \gamma \xrightarrow{\kappa_1} \beta * \gamma$$

$$\kappa_\gamma = \kappa_x = \mathbf{U}$$



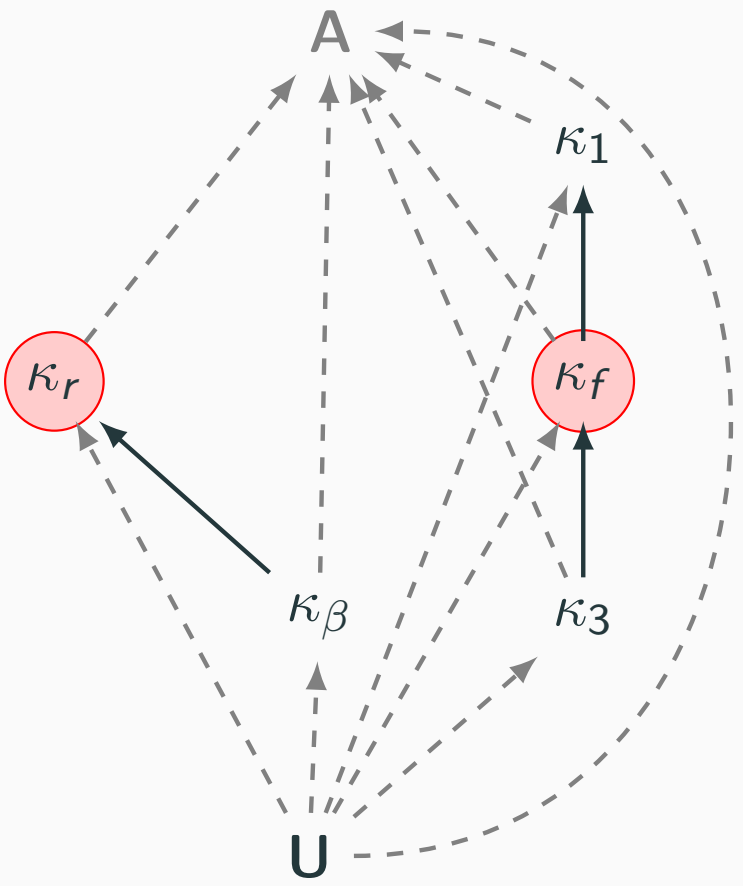
$$(\gamma : \kappa_\gamma)(\beta : \kappa_\beta).\ (\gamma \xrightarrow{\kappa_3} \beta) \xrightarrow{\kappa_2} \gamma \xrightarrow{\kappa_1} \beta * \gamma$$

$$\kappa_\gamma = \kappa_x = \mathbf{U}$$



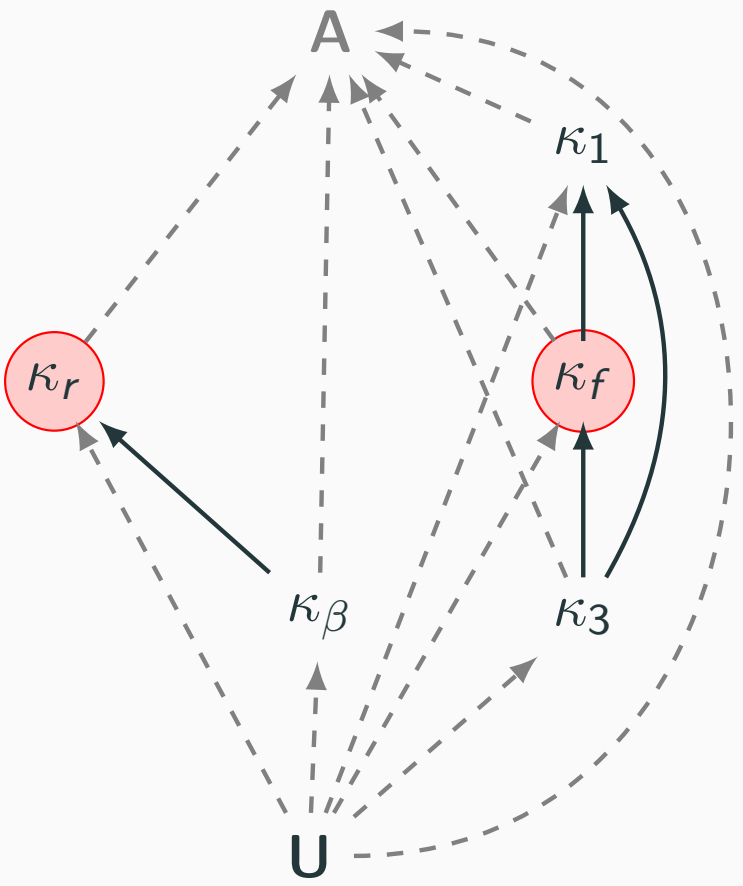
$$(\gamma : \kappa_\gamma)(\beta : \kappa_\beta).\ (\gamma \xrightarrow{\kappa_3} \beta) \xrightarrow{\kappa_2} \gamma \xrightarrow{\kappa_1} \beta * \gamma$$

$$\kappa_\gamma = \kappa_x = \mathbf{U}$$



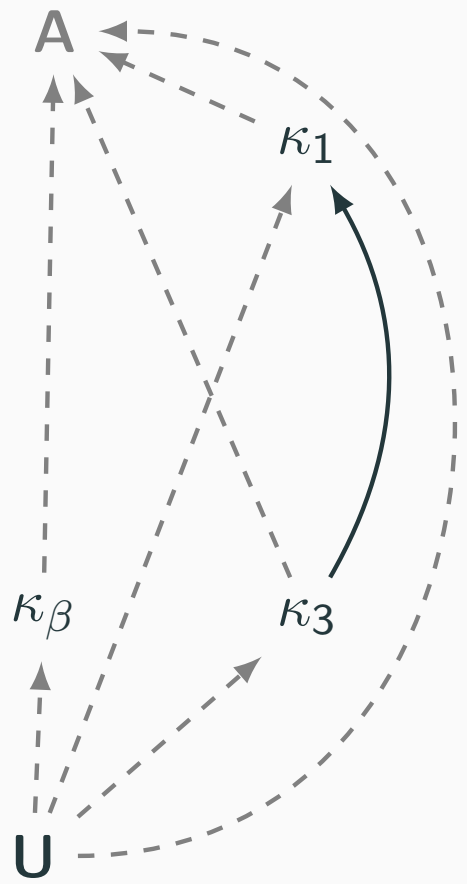
$$(\gamma : \kappa_\gamma)(\beta : \kappa_\beta).\ (\gamma \xrightarrow{\kappa_3} \beta) \xrightarrow{\kappa_2} \gamma \xrightarrow{\kappa_1} \beta * \gamma$$

$$\kappa_\gamma = \kappa_x = \mathbf{U}$$



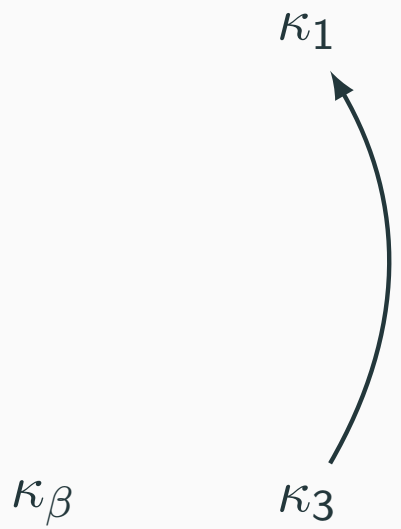
$$(\gamma : \kappa_\gamma)(\beta : \kappa_\beta).\ (\gamma \xrightarrow{\kappa_3} \beta) \xrightarrow{\kappa_2} \gamma \xrightarrow{\kappa_1} \beta * \gamma$$

$$\kappa_\gamma = \kappa_x = \mathsf{U}$$



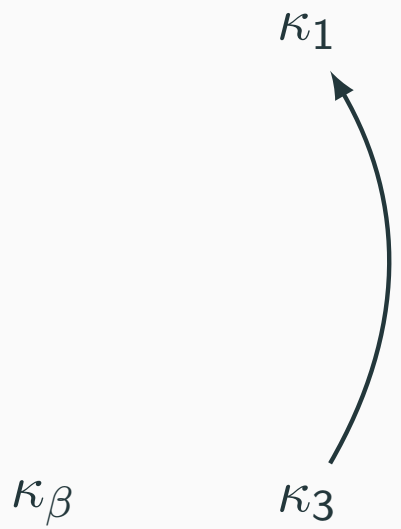
$$(\gamma : \kappa_\gamma)(\beta : \kappa_\beta).\ (\gamma \xrightarrow{\kappa_3} \beta) \xrightarrow{\kappa_2} \gamma \xrightarrow{\kappa_1} \beta * \gamma$$

$$\kappa_\gamma = \kappa_x = \mathsf{U}$$



$$(\gamma : \kappa_\gamma)(\beta : \kappa_\beta).\ (\gamma \xrightarrow{\kappa_3} \beta) \xrightarrow{\kappa_2} \gamma \xrightarrow{\kappa_1} \beta * \gamma$$

$$\kappa_\gamma = \kappa_x = \mathbf{U} \wedge \kappa_3 \leq \kappa_1$$



Normalization is complete and principal.

$$\lambda f.\lambda x.((f\ x), x) :$$

$$\forall \kappa_\beta \kappa_1 \kappa_2 \kappa_3 (\gamma : \mathbf{U})(\beta : \kappa_\beta).\ (\kappa_3 \leq \kappa_1) \Rightarrow (\gamma \xrightarrow{\kappa_3} \beta) \xrightarrow{\kappa_2} \gamma \xrightarrow{\kappa_1} \beta * \gamma$$

Well known simplifications on constraints:

- Replace variable in positive position by their lower bound
- Replace variable in negative position by their upper bound

Well known simplifications on constraints:

- Replace variable in positive position by their lower bound
- Replace variable in negative position by their upper bound

$$\forall \kappa_\beta \kappa_1 \kappa_2 \kappa_3 (\gamma : \mathsf{U})(\beta : \kappa_\beta).(\kappa_3 \leq \kappa_1) \Rightarrow (\gamma \xrightarrow{\kappa_3} \beta) \xrightarrow{\kappa_2} \gamma \xrightarrow{\kappa_1} \beta * \gamma$$

Well known simplifications on constraints:

- Replace variable in positive position by their lower bound
- Replace variable in negative position by their upper bound

$$\forall \kappa_\beta \kappa_1 \kappa_3 (\gamma : \mathbf{U})(\beta : \kappa_\beta).(\kappa_3 \leq \kappa_1) \Rightarrow (\gamma \xrightarrow{\kappa_3} \beta) \rightarrow \gamma \xrightarrow{\kappa_1} \beta * \gamma$$

Well known simplifications on constraints:

- Replace variable in positive position by their lower bound
- Replace variable in negative position by their upper bound

$$\forall \kappa_\beta \kappa(\gamma : \mathbf{U})(\beta : \kappa_\beta).(\gamma \xrightarrow{\kappa} \beta) \to \gamma \xrightarrow{\kappa} \beta * \gamma$$

Well known simplifications on constraints:

- Replace variable in positive position by their lower bound
- Replace variable in negative position by their upper bound

$$\forall \kappa_\beta \kappa(\gamma : \mathbf{U})(\beta : \kappa_\beta).(\gamma \xrightarrow{\kappa} \beta) \to \gamma \xrightarrow{\kappa} \beta * \gamma$$

$\Longrightarrow$ Unfinished, need to investigate principality

Constraints

The only requirement is that $\ell^{\perp} = \mathbf{U}$.

- **A** doesn't appear in the typing rules.
  It only comes from the buitins and/or the type declarations.
- The lattice doesn't have to be finite.
- The constraint language can be expanded further.

Constraints

The only requirement is that $\ell^{\perp} = \mathbf{U}$.

- **A** doesn't appear in the typing rules.
  It only comes from the buitins and/or the type declarations.
- The lattice doesn't have to be finite.
- The constraint language can be expanded further.

Going further

Current area of work

1. Richer type system
2. Modules
3. Borrowing
4. Prototype cool APIs with it

Constraints in a similar style have been applied to:

- (Relaxed) value restriction
- GADTs
- Rows
- Type elaboration
- ...

Modules

Several distinct problematic:

- Type abstraction
- Linear/affine values in modules
- Functors
- Separate compilation

Modules

Several distinct problematic:

- Type abstraction ✔
  Can declare unrestricted types and expose them as Affine.
- Linear/affine values in modules
- Functors
- Separate compilation

Modules

Several distinct problematic:

- Type abstraction
- Linear/affine values in modules

  Behave like tuples: take the LUB of the kinds of the exposed values.

  What about values that are not exposed? They don't matter!
- Functors
- Separate compilation

Modules

Several distinct problematic:

- Type abstraction
- Linear/affine values in modules
- Functors

  What happens if a functor takes a module containing affine values?

  $\Longrightarrow$ We need kind annotation on the functor arrow... ☹
- Separate compilation

Modules

Several distinct problematic:

- Type abstraction
- Linear/affine values in modules
- Functors
- Separate compilation

  What about linear/affine constants?

  $\Longrightarrow$ Should probably be forbidden...

Modules

Several distinct problematic:

- Type abstraction
- Linear/affine values in modules
- Functors
- Separate compilation

  What about linear/affine constants?

  $\Longrightarrow$ Should probably be forbidden...

  But what about `stdout` ?

Borrowing

Borrowing seem essential to express many patterns found in OCaml.

Read-only borrows, in `CCHashTrie`:

```
val add_mut : id -> key -> 'a -> 'a t -> 'a t
(* add_mut ~id k v m behaves like add k v m, except
   it will mutate in place whenever possible. *)
```

Mutable borrows, in `lacaml`:

```
val Lacaml.D.sycon :
  ... -> ?iwork:Common.int32_vec -> mat -> float
(* iwork is an optional preallocated work buffer *)
```

Borrowing

Borrowing seem essential to express many patterns found in OCaml.

Read-only borrows, in `CCHashTrie`:

```
val add_mut : id -> key -> 'a -> 'a t -> 'a t
(* add_mut ~id k v m behaves like add k v m, except
   it will mutate in place whenever possible. *)
```

Mutable borrows, in `lacaml`:

```
val Lacaml.D.sycon :
  ... -> ?iwork:Common.int32_vec -> mat -> float
(* iwork is an optional preallocated work buffer *)
```

Borrowing

Borrowing seem essential to express many patterns found in OCaml.

Read-only borrows, in `CCHashTrie`:

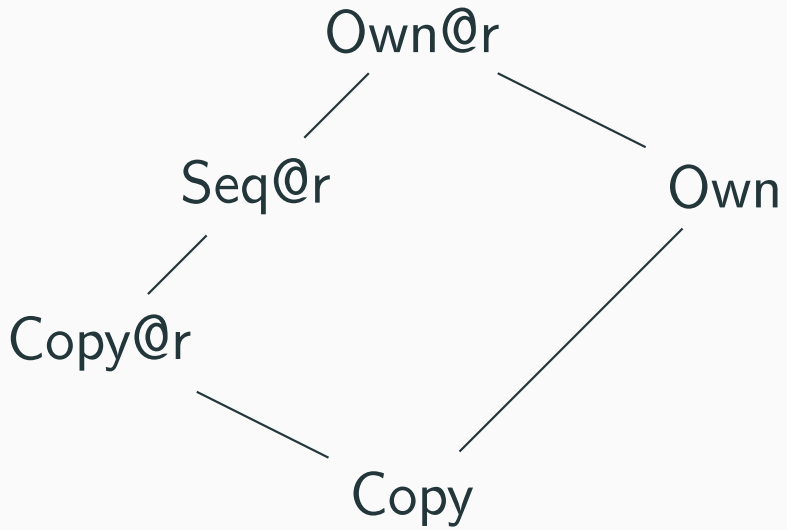
```
val add_mut : id -> key -> 'a -> 'a t -> 'a t
(* add_mut ~id k v m behaves like add k v m, except
   it will mutate in place whenever possible. *)
```

Mutable borrows, in `lacaml`:

```
val Lacaml.D.sycon :
  ... -> ?iwork:Common.int32_vec -> mat -> float
(* iwork is an optional preallocated work buffer *)
```

Borrowing

"Resource Polymorphism" has the following lattice:



It would requires:

- More syntactic annotations
- Regions

Conclusion

I presented a somewhat minimalistic approach to add linear types to an existing ML language (like OCaml).

- Based on kinds and constraints
- Works with type abstraction and modules
- Support type inference
- Doesn't break the whole ecosystem

The system is still small. We must look at concrete code pattern used in OCaml and decide how to support them.

Really??

Do you really think adding kinds, subkinding and qualified types to OCaml *is a good idea?*

Yes, I do!

- Qualified types are coming for modular implicits anyway.
- Having proper kinds would fix many weirdness (rows, . . .) and enable nice extensions (units of measures).
- I could make Eliom even better with them! ☺

Really??

Do you really think adding kinds, subkinding and qualified types to OCaml *is a good idea?*

Yes, I do!

- Qualified types are coming for modular implicits anyway.
- Having proper kinds would fix many weirdness (rows, . . .) and enable nice extensions (units of measures).
- I could make Eliom even better with them! ☺

Really??

Do you really think adding kinds, subkinding and qualified types to OCaml *is a good idea?*

Yes, I do!

- Qualified types are coming for modular implicits anyway.
- Having proper kinds would fix many weirdness (rows, . . .) and enable nice extensions (units of measures).
- I could make Eliom even better with them! ☺

Really??

> *Do you really think adding kinds, subkinding and qualified types to* OCaml *is a good idea?*

Yes, I do!

- Qualified types are coming for modular implicits anyway.
- Having proper kinds would fix many weirdness (rows, . . .) and enable nice extensions (units of measures).
- I could make Eliom even better with them! ☺

Close(Talk)

References

Martin Odersky, Martin Sulzmann, and Martin Wehr. 1999. Type Inference with Constrained Types. *TAPOS* 5, 1 (1999), 35–55.