Kindly Bent To Free Us

Gabriel Radanne Hannes Saffrich Peter Thiemann

February 3, 2020

Simplified API from the library `ocaml-tls`:

```
val channels : Tls.fd → in_channel * out_channel
(* Turn a file descr into input/output channels *)
```

```
let fd : Tls.fd = .....
let input, output = Tls.channels fd
let x = read_stuff input in
let () = close input in
...
let c = write output "thing" in (*Oups*)
...
```

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

This bug was found in the wild, and then fixed in `ocaml-tls`.

Simplified API from the library `ocaml-tls`:

```
val channels : Tls.fd → in_channel * out_channel
(* Turn a file descr into input/output channels *)
```

```
let fd : Tls.fd = .....
let input, output = Tls.channels fd
let x = read_stuff input in
let () = close input in
...
let c = write output "thing" in (*Oups*)
...
```

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

This bug was found in the wild, and then fixed in `ocaml-tls`.

Simplified API from the library `ocaml-tls`:

```
val channels : Tls.fd -> in_channel * out_channel
(* Turn a file descr into input/output channels *)
```

```
let fd : Tls.fd = .....
let input, output = Tls.channels fd
let x = read_stuff input in
let () = close input in
...
let c = write output "thing" in (*Oups*)
...
```

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

This bug was found in the wild, and then fixed in `ocaml-tls`.

Simplified API from the library `ocaml-tls`:

```
val channels : Tls.fd -> in_channel * out_channel
(* Turn a file descr into input/output channels *)
```

```
let fd : Tls.fd = .....
let input, output = Tls.channels fd
let x = read_stuff input in
let () = close input in
...
let c = write output "thing" in (*Oups*)
...
```

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

This bug was found in the wild, and then fixed in `ocaml-tls`.

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”
- . . .

Goals:

- Complete and principal type inference
- Impure and strict context
- Support both functional and imperative styles
- Works well with type abstraction

Non Goals:

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

Goals:

- Complete and principal type inference
- Impure and strict context
- Support both functional and imperative styles
- Works well with type abstraction

Non Goals:

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

The Affe language

Let's create an `LinArray` API together!

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

```
module LinArray : sig
  type (α : un) t : lin (* LinArrays are linear! *)
  val create : int → α → α t
  val free : α t → unit
end
```

Let's create an LinArray API together!

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

```
module LinArray : sig
  type (α : un) t : lin (* LinArrays are linear! *)
  val create : int → α → α t
  val free : α t → unit
end
```

```
let main () =
  let a = LinArray.create 3 "foo" (* : string t *)
  .... (* a is linear *)
  LinArray.free a ;
```

No type annotation!

Let's create an `LinArray` API together!

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

```
module LinArray : sig
  type (α : un) t : lin (* LinArrays are linear! *)
  val create : int → α → α t
  val free : α t → unit
end
```

```
let main () =
  let a = LinArray.create 3 "foo" (* : string t *)
  .... (* a is linear *)
  LinArray.free a ;
  f a (* ✘ No! *)
```

How to read the array ?

```
module LinArray : sig
  type (α : un) t : lin
  val create : int → α → α t
  val free : α t → unit
  val get : α t * int → α (* ? *)
end
```

```
let main () =
  let a = LinArray.create 3 "foo"
  let x = LinArray.get (a, 2) in
  LinArray.free a  (* ✘ No! *)
  print x
```

This doesn't work!

How to read the array ?

```
module LinArray : sig
  type (α : un) t : lin
  val create : int → α → α t
  val free : α t → unit
  val get : α t * int → α (* ? *)
end
```

```
let main () =
  let a = LinArray.create 3 "foo"
  let x = LinArray.get (a, 2) in
  LinArray.free a  (* ✘ No! *)
  print x
```

This doesn't work!

How to read the array ?

```
module LinArray : sig
  type (α : un) t : lin
  val create : int → α → α t
  val free : α t → unit
  val get : α t * int → α * α t (* ?? *)
end
```

```
let main () =
  let a = LinArray.create 3 "foo"
  let x, a = LinArray.get (a, 2) in
  LinArray.free a ;
  print x
```

This works, but is inconvenient!

How to read the array ?

```
module LinArray : sig
  type (α : un) t : lin
  val create : int → α → α t
  val free : α t → unit
  val get : &(α t) ∗ int → α
end
```

```
let main () =
  let a = LinArray.create 3 "foo"
  let x = LinArray.get (&a, 2) in (* Borrow *)
  LinArray.free a
```

We use borrows!

We temporarily give `&a` to `LinArray.get`.

A recap on borrows

Borrows allow to lend usage of something to someone else.

There are different types of borrows:

- Shared borrows `&a` which are *Unrestricted* (**`un`**)
- Exclusive borrows `&!a` which are *Affine* (**`aff`**)

We cannot use a borrow of `a` and `a` itself at the same time.
A borrow must not escape.

```
module LinArray : sig
  type (α : un) t : lin
  val create : int → α → α t
  val free : α t → unit
  val get : &(α t) * int → α
  val set : &!(α t) * int * α → unit
end
```

```
let main () =
  let a = create 3 "foo"
  let x = get (&a, 0) ^ get (&a, 1) in
    (* ✔ Multiple Shared borrows *)
  set (&!a, 2, x);
    (* ✔ One Exclusive borrow *)
  free a
```

```
module LinArray : sig
  type (α : un) t : lin
  val create : int → α → α t
  val free : α t → unit
  val get : &(α t) * int → α
  val set : &!(α t) * int * α → unit
end
```

```
let main () =
  let a = create 3 "foo"
  f (a, &a, 42)
  (* ✘ Using a and a borrow simultaneously! *)
```

```
module LinArray : sig
  type (α : un) t : lin
  val create : int → α → α t
  val free : α t → unit
  val get : &(α t) * int → α
  val set : &!(α t) * int * α → unit
end
```

```
let main () =
  let a = create 3 "foo"
  f (&!a, &a, 42)
  (* ✘ Conflicting borrows *)
```

```
module LinArray : sig
  type (α : un) t : lin
  val create : int → α → α t
  val free : α t → unit
  val get : &(α t) ∗ int → α
  val set : &!(α t) ∗ int ∗ α → unit
end
```

A slightly bigger piece of code:

```
let mk_fib_array n =
  let a = create n 1 in
  for i = 2 to n - 1 do
    let x = get (&a, i-1) + get (&a, i-2) in
    set (&!a, i, x)
  done;
  a
# mk_fib_array : int → int Array.t
```

Still no type annotations: everything is inferred.

Borrows must not escape $\Longrightarrow$ What is their scope ?

```
module LinArray : sig
  type (α : un) t : lin
  val create : int → α → α t
  val free : α t → unit
  val get : &(α t) * int → α
  val set : &!(α t) * int * α → unit
end
```

A slightly bigger piece of code:

```
let mk_fib_array n =
  let a = create n 1 in
  for i = 2 to n - 1 do
    let x = get (&a, i-1) + get (&a, i-2) in
    set (&!a, i, x)
  done;
  a
# mk_fib_array : int → int Array.t
```

Still no type annotations: everything is inferred.

Borrows must not escape $\implies$ What is their scope ?

```
module LinArray : sig
  type (α : un) t : lin
  val create : int → α → α t
  val free : α t → unit
  val get : &(α t) * int → α
  val set : &!(α t) * int * α → unit
end
```

A slightly bigger piece of code:

```
let mk_fib_array n =
  let a = create n 1 in
  for i = 2 to n - 1 do {|
    let x = {| get (&a, i-1) + get (&a, i-2) |} in
    set (&!,a, i) x
  |} done;
  a
# mk_fib_array : int → int Array.t
```

A borrow cannot escape a region `{| .... |}`.

Regions are inferred automatically, but can be manually provided.

Closures

Closures can capture linear and affine values:

```
let a = LinArray.create 10 "foo"
let f i = LinArray.set(&!a,i,"bar")
```

If `f` can be used multiple times, we violate the usage of `&!a`.

We infer:

```
val f : int -aff-> unit
```

Arrows are annotated with a kind (here, *Affine*) denoting their use.

$\rightarrow$ is equivalent to $\xrightarrow{\text{un}}$.

Closures

Closures can capture linear and affine values:

```
let a = LinArray.create 10 "foo"
let f i = LinArray.set(&!a,i,"bar")
```

If `f` can be used multiple times, we violate the usage of `&!a`.

We infer:

$$\texttt{val f : int} \xrightarrow{\texttt{aff}} \texttt{unit}$$

Arrows are annotated with a kind (here, *Affine*) denoting their use.

$\rightarrow$ is equivalent to $\xrightarrow{\texttt{un}}$.

Inference and polymorphism

So far, we have seen limited polymorphism.

What is the type of `compose` ?

```
let compose f g x = f (g x)
```

The type of `compose f g` depends on the linearity of `f` and `g`.

Inference and polymorphism

So far, we have seen limited polymorphism.

What is the type of `compose` ?

```
let compose f g x = f (g x)
```

The type of `compose f g` depends on the linearity of `f` and `g`.

```
val compose :
  (β -κ1-> α) → (γ -κ2-> β) -?-> γ -?-> α
```

$$(\beta \xrightarrow{\kappa_1} \alpha) \rightarrow (\gamma \xrightarrow{\kappa_2} \beta) \xrightarrow{?} \gamma \xrightarrow{?} \alpha$$

We would expect something of the form $\kappa_1 \sqcup \kappa_2$

Inference and polymorphism

So far, we have seen limited polymorphism.

What is the type of `compose` ?

```
let compose f g x = f (g x)
```

The type of `compose f g` depends on the linearity of `f` and `g`.

```
val compose :
```

$$(\kappa_1 \leq \kappa_2) \Rightarrow$$
$$(\beta \xrightarrow{\kappa_1} \alpha) \rightarrow (\gamma \xrightarrow{\kappa_2} \beta) \xrightarrow{\kappa_1} \gamma \xrightarrow{\kappa_2} \alpha$$

We use kind inequalities and subkinding to express such constraints.

This type is the most general and is inferred.

A more general API

We can now generalize `LinArray` to arbitrary content:

```
module LinArray : sig
  type (α : κ) t : lin
  val create : (α : un) ⇒ int → α → α t
  val init : (int → α) → int → α t

  val free : (α : aff) ⇒ α t → unit

  val length : &(α t) → int

  val get : (α : un) ⇒ &(α t) * int → α
  val set : (α : aff) ⇒ &!(α t) * int * α → unit
end
```

Each operation quantifies the type of element it accepts.

What about iterations ?

A more general API

We can now generalize `LinArray` to arbitrary content:

```
module LinArray : sig
  type (α : κ) t : lin
  val create : (α : un) ⇒ int → α → α t
  val init : (int → α) → int → α t

  val free : (α : aff) ⇒ α t → unit

  val length : &(α t) → int

  val get : (α : un) ⇒ &(α t) * int → α
  val set : (α : aff) ⇒ &!(α t) * int * α → unit
end
```

Each operation quantifies the type of element it accepts.

What about iterations ?

Iterators and linearity

A naive fold function only works on unrestricted elements

```
val fold :
  (α : un) ⇒ (α → β → β) → α LinArray.t → β → β
```

Ideally, we would like to borrow the element while folding …

But the borrow shouldn't be captured!

```
val fold :
  (β:κ),(κ ≤ aff_r) ⇒
  (&(aff_{r+1},α) → β -aff_{r+1}→ β) → &(κ_1,α LinArray.t) → β -κ_1→ β
```

We can express such types using region variables.

Iterators and linearity

A naive fold function only works on unrestricted elements

```
val fold :
  (α : un) ⇒ (α → β → β) → α LinArray.t → β → β
```

Ideally, we would like to borrow the element while folding . . .
But the borrow shouldn't be captured!

```
val fold :
  (β:κ),(κ ≤ aff_r) ⇒
  (&(aff_{r+1},α) → β -aff_{r+1}→ β) → &(κ_1,α LinArray.t) → β -κ_1→ β
```

We can express such types using region variables.

A naive fold function only works on unrestricted elements

```
val fold :
  (α : un) ⇒ (α → β → β) → α LinArray.t → β → β
```

Ideally, we would like to borrow the element while folding . . .
But the borrow shouldn't be captured!

```
val fold :
  (β:κ),(κ ≤ aff_r) ⇒
  (&(aff_{r+1},α) → β -aff_{r+1}→ β) → &(κ_1,α LinArray.t) → β -κ_1→ β
```

We can express such types using region variables.

A glimpse at the theory

A glimpse at the theory

In the rest of this talk, we will take a closer look at:

- Kinds and constraints
- Inference

More precise syntax

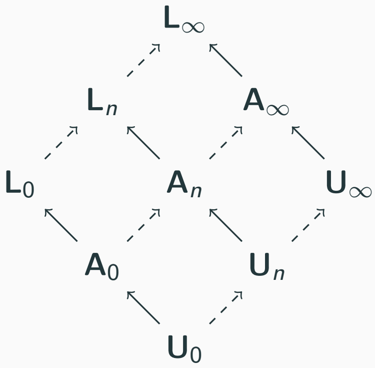
Let's clarify some syntax:

- Kind constants are composed of a "quality" (unrestricted **U**, Affine **A**, Linear **L**) and a "level" $n \in \mathbb{N}$.
- Borrows are noted $\&^{\mathbf{A}} a$ (Exclusive) and $\&^{\mathbf{U}} a$ (Shared).
- Borrow types are annotated with their kind: $\&^{b}(k, \tau)$.
- Regions annotated with their "nesting" and inner borrows.

Example of code:

$$
\begin{aligned}
\lambda a.\{\!|\ & \texttt{let}\ x = (f\ \&^{\mathbf{A}} a)\ \texttt{in} \\
& \{\!| g\ (\&^{\mathbf{A}} x) |\!\}^{2}_{\{x \mapsto \mathbf{A}\}}; \\
& \{\!| f\ (\&^{\mathbf{U}} x)\ (\&^{\mathbf{U}} x) |\!\}^{2}_{\{x \mapsto \mathbf{U}\}} \\
|\!\}^{1}_{\{a \mapsto \mathbf{A}\}} &
\end{aligned}
$$

Affe has subkinding. Kind constants respects the following lattice:



Resource management

To model resource management in the theory, we consider we consider the type $\mathrm{R}\ \tau$ of content $\tau : \mathbf{U}_0$

- `create`: $\forall \kappa_\alpha (\alpha : \kappa_\alpha).\ (\kappa_\alpha \leq \mathbf{U}_0) \Rightarrow \alpha \rightarrow \mathrm{R}\ \alpha$
- `observe`: $\forall \kappa \kappa_\alpha (\alpha : \kappa_\alpha).\ (\kappa_\alpha \leq \mathbf{U}_0) \Rightarrow \&^{\mathbf{U}}(\kappa, \mathrm{R}\ \alpha) \rightarrow \alpha$
- `update`:
  $\forall \kappa \kappa_\alpha (\alpha : \kappa_\alpha).\ (\kappa_\alpha \leq \mathbf{U}_0) \Rightarrow \&^{\mathbf{A}}(\kappa, \mathrm{R}\ \alpha) \rightarrow \alpha \xrightarrow{\mathbf{A}} \texttt{Unit}$
- `destroy`: $\forall \kappa_\alpha (\alpha : \kappa_\alpha).\ (\kappa_\alpha \leq \mathbf{U}_0) \Rightarrow \mathrm{R}\ \alpha \rightarrow \texttt{Unit}$

Regions

Regions follow lexical scoping. For every borrow `&x` or `&!x`, We define a region such that:

1. The region contains at least `&x`/`&!x`.
2. The region is never larger than the scope of `x`.
3. An exclusive borrow `&!x` never share a region with any other borrow of `x`.
4. A use of `x` is never in the region of `&x`/`&!x`.

Region inference

The region inference algorithm in practice:

$$\begin{array}{l}\lambda a.\ \texttt{let}\ x = (f\ \&^{\mathbf{A}} a)\ \texttt{in} \\ \quad g\ (\&^{\mathbf{A}} x); \\ \quad f\ (\&^{\mathbf{U}} x)\ (\&^{\mathbf{U}} x)\end{array}$$

Region inference

The region inference algorithm in practice:

$$\lambda a.\{\!|\ \texttt{let}\ x = (f\ \&^{\mathbf{A}} a)\ \texttt{in}\\ g\ (\&^{\mathbf{A}} x);\\ f\ (\&^{\mathbf{U}} x)\ (\&^{\mathbf{U}} x)\\ |\!\}^{1}_{\{a \mapsto \mathbf{A}\}}$$

The region inference algorithm in practice:

$$
\begin{array}{l}
\lambda a.\{\!|\ \texttt{let}\ x = (f\ \&^{\mathbf{A}} a)\ \texttt{in} \\
\qquad \{\!|g\ (\&^{\mathbf{A}} x)|\!\}^{2}_{\{x \mapsto \mathbf{A}\}}; \\
\qquad f\ (\&^{\mathbf{U}} x)\ (\&^{\mathbf{U}} x) \\
\quad |\!\}^{1}_{\{a \mapsto \mathbf{A}\}}
\end{array}
$$

Region inference

The region inference algorithm in practice:

$$
\begin{aligned}
\lambda a.\{\!| \; & \texttt{let } x = (f \;\&^{\mathbf{A}} a) \texttt{ in} \\
& \{\!| g \; (\&^{\mathbf{A}} x) |\!\}^{2}_{\{x \mapsto \mathbf{A}\}}; \\
& \{\!| f \; (\&^{\mathbf{U}} x) \; (\&^{\mathbf{U}} x) |\!\}^{2}_{\{x \mapsto \mathbf{U}\}} \\
|\!\}^{1}_{\{a \mapsto \mathbf{A}\}} &
\end{aligned}
$$

Region inference

Another example with explicit region annotations:

$$\begin{array}{l}\texttt{let}\ r = \mathsf{ref}\ \ 0\ \texttt{in} \\ \ \lambda a.\ set\ r\ \{\!|g\ (\&^{\mathbf{U}} a)|\!\};\\ \qquad f\ (\&^{\mathbf{U}} a)\end{array} \rightsquigarrow \begin{array}{l}\texttt{let}\ r = \mathsf{ref}\ \ 0\ \texttt{in} \\ \ \lambda a.\ set\ r\ \{\!|g\ (\&^{\mathbf{U}} a)|\!\}^{1}_{\{a\mapsto \mathbf{U}\}};\\ \qquad \{\!|f\ (\&^{\mathbf{U}} a)|\!\}^{1}_{\{a\mapsto \mathbf{U}\}}\end{array}$$

Kinds during typing

A traditional linear rule for pairs:

$$\frac{\Gamma = \Gamma_1 \ltimes \Gamma_2 \qquad \Gamma_1 \vdash e_1 : \tau_1 \qquad \Gamma_2 \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

How to take kinds into account ?

Kinds during typing

We propagate constraints:

$$\frac{C \vdash_e \Gamma = \Gamma_1 \ltimes \Gamma_2 \qquad C \mid \Gamma_1 \vdash_s e_1 : \tau_1 \qquad C \mid \Gamma_2 \vdash_s e_2 : \tau_2}{C \mid \Gamma \vdash_s (e_1, e_2) : \tau_1 \times \tau_2}$$

And use a constraint-aware split:

$$\begin{array}{rcccccl}
(\sigma \leq \mathbf{U}_\infty) \vdash_e & (x : \sigma) & = & (x : \sigma) & \ltimes & (x : \sigma) & \text{Both} \\
\text{True} \vdash_e & B_x & = & B_x & \ltimes & \emptyset & \text{Left} \\
\text{True} \vdash_e & B_x & = & \emptyset & \ltimes & B_x & \text{Right} \\
& & & \vdots & & &
\end{array}$$

Kinds during typing

We propagate constraints:

$$\frac{C \vdash_e \Gamma = \Gamma_1 \ltimes \Gamma_2 \qquad C \mid \Gamma_1 \vdash_s e_1 : \tau_1 \qquad C \mid \Gamma_2 \vdash_s e_2 : \tau_2}{C \mid \Gamma \vdash_s (e_1, e_2) : \tau_1 \times \tau_2}$$

And use a constraint-aware split:

$$\begin{array}{rcccccl}
(\sigma \leq \mathbf{U}_\infty) \vdash_e & (x : \sigma) & = & (x : \sigma) & \ltimes & (x : \sigma) & \text{Both} \\
\text{True} \vdash_e & B_x & = & B_x & \ltimes & \emptyset & \text{Left} \\
\text{True} \vdash_e & B_x & = & \emptyset & \ltimes & B_x & \text{Right} \\
& & & \vdots & & &
\end{array}$$

Kinds during typing

We propagate constraints:

$$\frac{C \vdash_e \Gamma = \Gamma_1 \ltimes \Gamma_2 \qquad C \mid \Gamma_1 \vdash_s e_1 : \tau_1 \qquad C \mid \Gamma_2 \vdash_s e_2 : \tau_2}{C \mid \Gamma \vdash_s (e_1, e_2) : \tau_1 \times \tau_2}$$

And use a constraint-aware split:

$$\begin{array}{rcccccl}
(\sigma \leq \mathbf{U}_\infty) \vdash_e (x : \sigma) & = & (x : \sigma) & \ltimes & (x : \sigma) & & \text{Both} \\
\text{True} \vdash_e \quad B_x & = & B_x & \ltimes & \emptyset & & \text{Left} \\
\text{True} \vdash_e \quad B_x & = & \emptyset & \ltimes & B_x & & \text{Right} \\
& & \vdots & & & &
\end{array}$$

How to split with regions

To handle regions and borrows, we need special binders:

$$
\begin{array}{lcccr}
\cdot \vdash_e (\&^{\mathbf{U}} x : \sigma) & = & (\&^{\mathbf{U}} x : \sigma) & \ltimes & (\&^{\mathbf{U}} x : \sigma) & \text{Borrow} \\
\cdot \vdash_e (x : \sigma) & = & [x : \sigma]^n_b & \ltimes & (x : \sigma) & \text{Susp} \\
\cdot \vdash_e (\&^b x : \sigma) & = & [x : \sigma]^n_{\mathbf{U}} & \ltimes & (\&^b x : \sigma) & \text{SuspB} \\
\cdot \vdash_e [x : \sigma]_b & = & [x : \sigma]^n_{\mathbf{U}} & \ltimes & [x : \sigma]_b & \text{SuspS}
\end{array}
$$

$(\&^b x : \sigma)$ means a borrow is usable.

$[x : \sigma]^n_b$ means a borrow *will be usable* when we enter a region.

When we enter a region $\{\!|\ \dots\ |\!\}^n_{\{x \mapsto b\}}$, we transform the binders of x in the environment:

$$(b_n \leq k) \wedge (k \leq b_\infty) \vdash_e [x : \tau]^n_b \rightsquigarrow_n (\&^b x : \&^b(k, \tau))$$

Constraints

Constraints are a list of inequalities: $(k \leq k')^*$

We can only use constraints in schemes:

$$\sigma ::= \forall\kappa^*\forall(\alpha : k)^*.(C \Rightarrow \tau) \qquad \text{Type schemes}$$

$$\theta ::= \forall\kappa^*.(C \Rightarrow k_i^* \rightarrow k) \qquad \text{Kind schemes}$$

We use these constraints to verify everything!

Constraints

Constraints are a list of inequalities: $(k \leq k')^*$

We can only use constraints in schemes:

$\sigma ::= \forall \kappa^* \forall (\alpha : k)^*.(C \Rightarrow \tau)$	Type schemes
$\theta ::= \forall \kappa^*.(C \Rightarrow k_i^* \rightarrow k)$	Kind schemes

We use these constraints to verify everything!

Constraint and regions

Consider the following program :

$$\texttt{let}\ x = \texttt{create}()\ \texttt{in}$$
$$\{\!|g\ (\&^{\mathbf{A}} x)|\!\}^{n}_{\{x \mapsto \mathbf{A}\}}$$

We deduce the following:

$$(x : \tau) \wedge (\&^{\mathbf{A}} x : \&^{b}(k, \tau)) \wedge (\mathbf{A}_n \leq k) \wedge (k \leq \mathbf{A}_\infty)$$
$$(g : \&^{\mathbf{A}}(k, \tau) \xrightarrow{\kappa} \tau') \wedge (\tau' : k') \wedge (k' \leq \mathbf{L}_{n-1})$$

Finally, we must verify and normalize the constraints

Constraint and regions

Consider the following program :

$$\texttt{let}\ x = \texttt{create}()\ \texttt{in}$$
$$\{\!\![g\ (\&^{\mathbf{A}} x)]\!\!\}^{n}_{\{x \mapsto \mathbf{A}\}}$$

We deduce the following:

$$(x : \tau) \wedge (\&^{\mathbf{A}} x : \&^{b}(k, \tau)) \wedge (\mathbf{A}_n \leq k) \wedge (k \leq \mathbf{A}_\infty)$$
$$(g : \&^{\mathbf{A}}(k, \tau) \xrightarrow{\kappa} \tau') \wedge (\tau' : k') \wedge (k' \leq \mathbf{L}_{n-1})$$

Finally, we must verify and normalize the constraints

Constraint and regions

Consider the following program :

$$\texttt{let}\ x = \texttt{create}()\ \texttt{in}$$
$$\{\!|g\ (\&^{\mathbf{A}} x)|\!\}^{n}_{\{x \mapsto \mathbf{A}\}}$$

We deduce the following:

$$(x : \tau) \wedge (\&^{\mathbf{A}} x : \&^{b}(k, \tau)) \wedge (\mathbf{A}_n \leq k) \wedge (k \leq \mathbf{A}_\infty)$$
$$(g : \&^{\mathbf{A}}(k, \tau) \xrightarrow{\kappa} \tau') \wedge (\tau' : k') \wedge (k' \leq \mathbf{L}_{n-1})$$

Finally, we must verify and normalize the constraints

Constraint and regions

Consider the following program :

$$\texttt{let}\ x = \texttt{create}()\ \texttt{in}$$
$$\{\!\!\{ g\ (\&^{\mathbf{A}} x) \}\!\!\}^{n}_{\{x \mapsto \mathbf{A}\}}$$

We deduce the following:

$$(x : \tau) \wedge (\&^{\mathbf{A}} x : \&^{b}(k, \tau)) \wedge (\mathbf{A}_n \leq k) \wedge (k \leq \mathbf{A}_\infty)$$
$$(g : \&^{\mathbf{A}}(k, \tau) \xrightarrow{\kappa} \tau') \wedge (\tau' : k') \wedge (k' \leq \mathbf{L}_{n-1})$$

Finally, we must verify and normalize the constraints

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

Raw constraints:

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

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

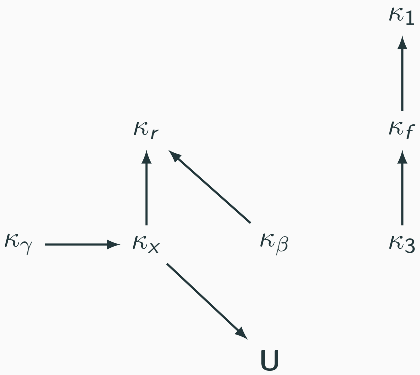
Raw constraints:

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

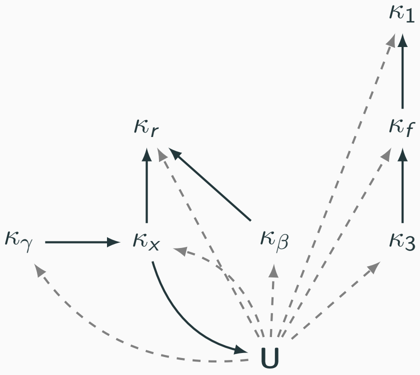
We unify the types and discover new constraints:

$$\alpha_r = (\gamma \xrightarrow{\kappa_3} \beta) \xrightarrow{\kappa_2} \gamma \xrightarrow{\kappa_1} \beta * \gamma$$
$$(\kappa_x \leq \mathbf{U}) \wedge (\kappa_\gamma \leq \kappa_x) \wedge (\kappa_x \leq \kappa_r) \wedge (\kappa_\beta \leq \kappa_r) \wedge (\kappa_3 \leq \kappa_f) \wedge (\kappa_f \leq \kappa_1)$$

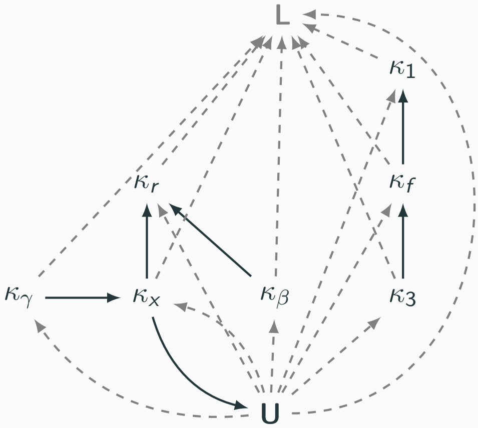
$$(\gamma : \kappa_\gamma)(\beta : \kappa_\beta).\ (\gamma \xrightarrow{\kappa_3} \beta) \xrightarrow{\kappa_2} \gamma \xrightarrow{\kappa_1} \beta * \gamma$$



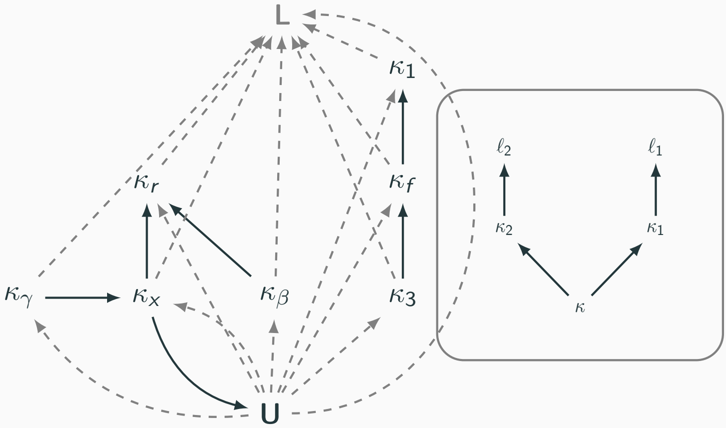
$$(\gamma : \kappa_\gamma)(\beta : \kappa_\beta).\ (\gamma \xrightarrow{\kappa_3} \beta) \xrightarrow{\kappa_2} \gamma \xrightarrow{\kappa_1} \beta * \gamma$$



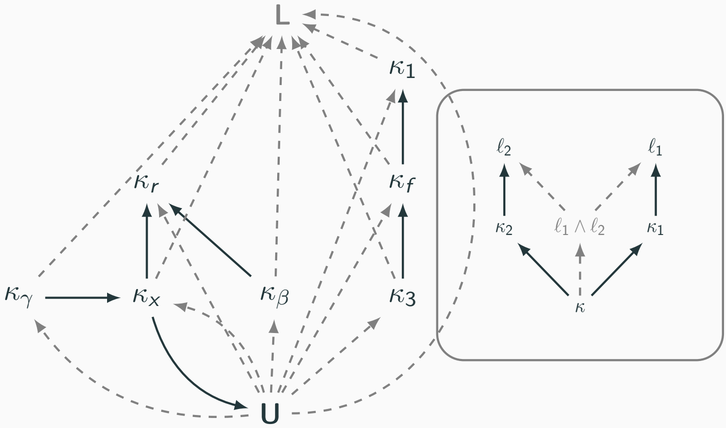
$$(\gamma : \kappa_\gamma)(\beta : \kappa_\beta).\ (\gamma \xrightarrow{\kappa_3} \beta) \xrightarrow{\kappa_2} \gamma \xrightarrow{\kappa_1} \beta * \gamma$$



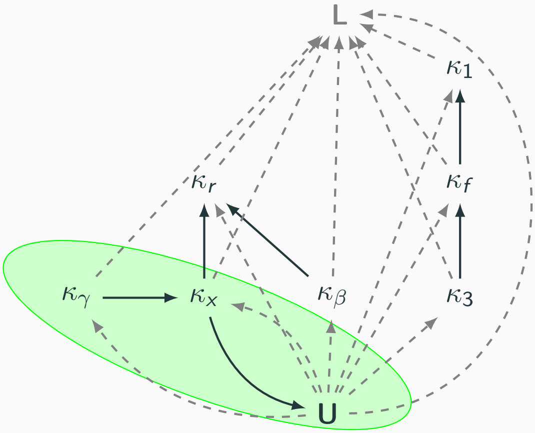
$$(\gamma : \kappa_\gamma)(\beta : \kappa_\beta).\ (\gamma \xrightarrow{\kappa_3} \beta) \xrightarrow{\kappa_2} \gamma \xrightarrow{\kappa_1} \beta * \gamma$$



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

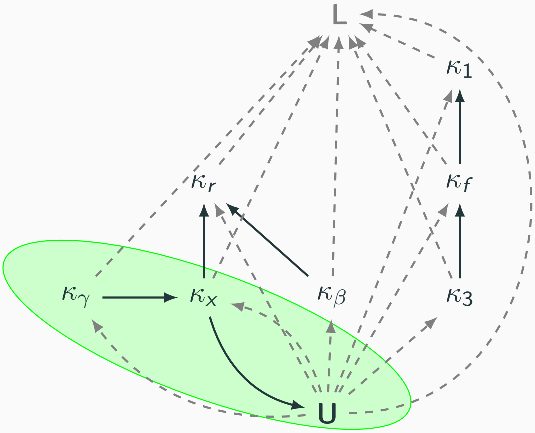


$$(\gamma : \kappa_\gamma)(\beta : \kappa_\beta).\ (\gamma \xrightarrow{\kappa_3} \beta) \xrightarrow{\kappa_2} \gamma \xrightarrow{\kappa_1} \beta * \gamma$$



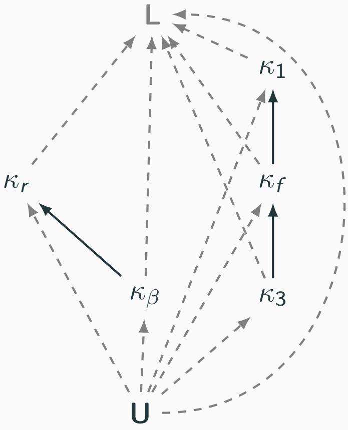
$$(\gamma : \kappa_\gamma)(\beta : \kappa_\beta).\ (\gamma \xrightarrow{\kappa_3} \beta) \xrightarrow{\kappa_2} \gamma \xrightarrow{\kappa_1} \beta * \gamma$$

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



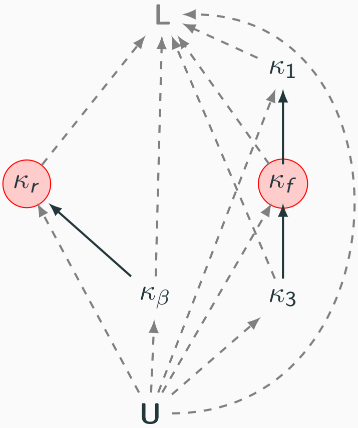
$$(\gamma : \kappa_\gamma)(\beta : \kappa_\beta).\ (\gamma \xrightarrow{\kappa_3} \beta) \xrightarrow{\kappa_2} \gamma \xrightarrow{\kappa_1} \beta * \gamma$$

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



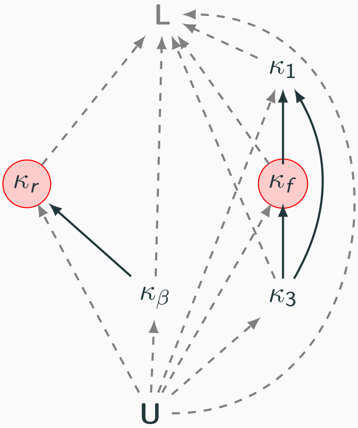
$$(\gamma : \kappa_\gamma)(\beta : \kappa_\beta).\ (\gamma \xrightarrow{\kappa_3} \beta) \xrightarrow{\kappa_2} \gamma \xrightarrow{\kappa_1} \beta * \gamma$$

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



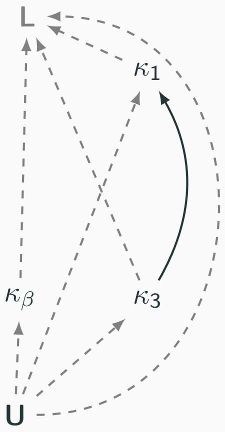
$$(\gamma : \kappa_\gamma)(\beta : \kappa_\beta).\ (\gamma \xrightarrow{\kappa_3} \beta) \xrightarrow{\kappa_2} \gamma \xrightarrow{\kappa_1} \beta * \gamma$$

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



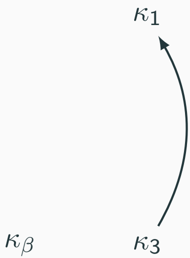
$$(\gamma : \kappa_\gamma)(\beta : \kappa_\beta).\ (\gamma \xrightarrow{\kappa_3} \beta) \xrightarrow{\kappa_2} \gamma \xrightarrow{\kappa_1} \beta * \gamma$$

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



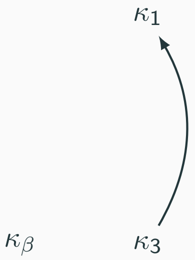
$$(\gamma : \kappa_\gamma)(\beta : \kappa_\beta).\ (\gamma \xrightarrow{\kappa_3} \beta) \xrightarrow{\kappa_2} \gamma \xrightarrow{\kappa_1} \beta * \gamma$$

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



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

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



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

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

Constraints – Tricky bits

Some tricky bits on constraints:

- Kinds might be polymorphic, and not all instances will have the same kinds
- Constraint solving is perf-sensitive! Adding too much power there (notably, disjunctions) is problematic.

Conclusion

I presented Affe:

- Support functional *and* imperative programming styles thanks to linear types, borrows and regions.
- Novel use of kinds and constraints to verify these properties
- Complete and principal type inference
- Design compatible with OCaml

In the paper "Kindly bent to free us" (on Arxiv), you can find:

- Several examples of functional, imperative or mixed programming
- Complete account of the theory:
  - A "logical" version of the type system
  - A resource-aware semantics and the proof of soundness
  - An inference algorithm based on HM(X) and the proofs of completeness/principality

Conclusion

I presented Affe:

- Support functional *and* imperative programming styles thanks to linear types, borrows and regions.
- Novel use of kinds and constraints to verify these properties
- Complete and principal type inference
- Design compatible with OCaml

In the paper "Kindly bent to free us" (on Arxiv), you can find:

- Several examples of functional, imperative or mixed programming
- Complete account of the theory:
  - A "logical" version of the type system
  - A resource-aware semantics and the proof of soundness
  - An inference algorithm based on HM(X) and the proofs of completeness/principality

Future work

Area of future work:

- Mechanizing the formalization
  $\Longrightarrow$ Ongoing work by Hannes Saffrich
- Design associated optimisations
  $\Longrightarrow$ Collaboration with Guillaume Munch-Maccagnoni
- Investigate pattern matching
- Extend the expressivity further (at the price of inference ?)

Finally, this kind system should be able to support other features (unboxing, for instance)

Close(Talk)

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! ☺

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?

  $\implies$ 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?

  $\implies$ Should probably be forbidden...

Modules

Several distinct problematic:

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

  What about linear/affine constants?

  $\implies$ Should probably be forbidden...

  But what about `stdout` ?

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
- ...

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

References

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