Formal Reasoning on Imperative ML Programs

Arthur Charguéraud

Joint work with François Pottier

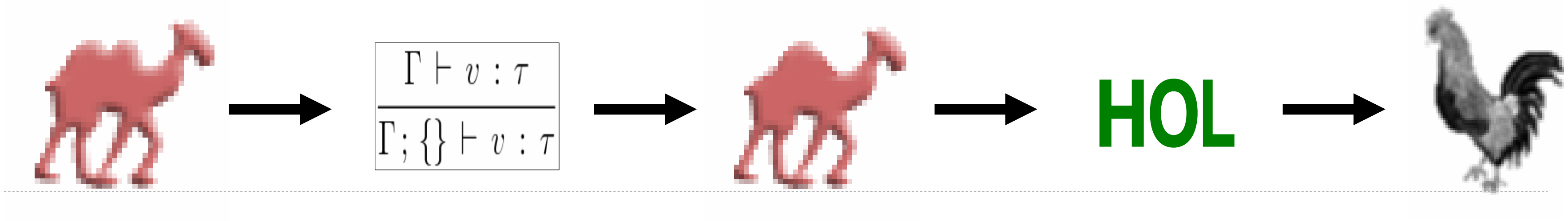
INRIA-Rocquencourt

Meeting CeProMi

Orsay, 2008-03-20

Overview

Our goal: design a type system that deals with non-aliasing and ownership transfers, supporting local reasoning, in order to ease the reasoning on higher-order imperative programs.



Imperative language → Typing derivations → Functional language → Higher-order formula → Coq proof script

Imperative language ↓ compilation, execution

Functional language ↓ specification, certification

– Presentation: type system and type-directed translation

– Illustration: factorial, mutable lists, union-find, quicksort, eratosthene, mutable queues

Ingredients

The type system extends System F with two ingredients.

Regions: are sets of one or several values (resp. σ or ρ)

– $[\alpha]$ is the type of inhabitant of region α

– for a singleton region σ, $[\sigma]$ is a singleton type

Capabilities: are written $\{\sigma:\theta\}$ or $\{\rho:\theta\}$

– describe ownership of regions,
i.e. the exclusive right to read or write in a region

– give the type θ of the corresponding piece of state

– $C_1 * C_2$ is the separating conjunction of C_1 and C_2

Grammar of Types

Values types (non-linear):

$$\tau \quad := \quad \bot \mid \top \mid \mathrm{unit} \mid [\rho] \mid [\sigma] \mid \chi_1 \rightarrow \chi_2 \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2$$

Memory types (linear):

$$\theta \quad := \quad \bot \mid \top \mid \mathrm{unit} \mid [\rho] \mid [\sigma] \mid \chi_1 \rightarrow \chi_2 \mid \theta_1 + \theta_2 \mid \theta_1 \times \theta_2 \mid \mathrm{ref}\,\theta$$

Computation types (linear):

$$\chi \quad := \quad \exists \bar{\rho}.\tau * \bar{C}$$

Typing judgments:

$x : \tau$

– for values: $\Delta \vdash v : \tau$

$x : \tau \quad y : C$

– for terms: $\Gamma \Vdash t : \chi$

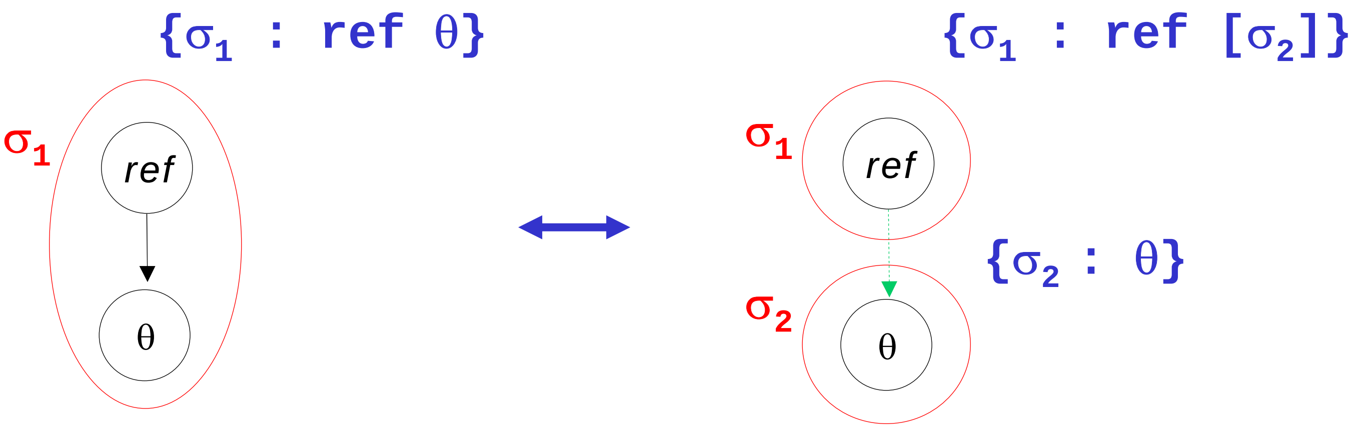
Typing References, Effects

Typing rules for reference primitives:

$$\begin{array}{lcl} \text{ref} & : & \tau \rightarrow \exists \sigma. [\sigma] * \{\sigma : \text{ref}\, \tau\} \\ \text{get} & : & [\sigma] * \{\sigma : \text{ref}\, \tau\} \rightarrow \tau * \{\sigma : \text{ref}\, \tau\} \\ \text{set} & : & ([\sigma] \times \tau_2) * \{\sigma : \text{ref}\, \tau_1\} \rightarrow \text{unit} * \{\sigma : \text{ref}\, \tau_2\} \end{array}$$

Reference with linear contents:

$$\{\sigma_1 : \text{ref}\, \theta\} \equiv \exists \sigma_2. \{\sigma_1 : \text{ref}\, [\sigma_2]\} * \{\sigma_2 : \theta\}$$



Connection with Effects

Effect-style notation: (should be a star symbol)

$$\alpha \to_\epsilon \beta \quad \equiv \quad \alpha \wedge \epsilon \to \beta \wedge \epsilon$$

$$\text{map} \quad : \quad (\alpha \to_\epsilon \beta) \to \text{list}\ \alpha \to_\epsilon \text{list}\ \beta$$

Derivable rules:

$$\begin{array}{lll} \text{get} & : & [\rho] \to_{\{\rho:\text{ref}\ \tau\}} \tau \\ \text{set} & : & [\rho] \times \tau \to_{\{\rho:\text{ref}\ \tau\}} \text{unit} \\ \text{ref} & : & \tau \to_{\{\rho\,:\text{ref}\ \tau\}} [\rho] \end{array}$$

Original rules:

$$\begin{array}{lll} \text{ref} & : & \tau \to \exists \sigma.\, [\sigma] * \{\sigma : \text{ref}\ \tau\} \\ \text{get} & : & [\sigma] * \{\sigma : \text{ref}\ \tau\} \to \tau * \{\sigma : \text{ref}\ \tau\} \\ \text{set} & : & ([\sigma] \times \tau_2) * \{\sigma : \text{ref}\ \tau_1\} \to \text{unit} * \{\sigma : \text{ref}\ \tau_2\} \end{array}$$

Type-directed Translation

Idea: static capabilities from the source are given a runtime representation in the translated program.

Typed imperative program:

```
let f x {C1} {C2} =
   ...
   let y {C3} = g x {C2} in
   ...
   y {C1} {C3} in
```

Functional translation:

```
let f x c1 c2 =
   ...
   let y,c3 = g x c2 in
   ...
   y,c1,c3 in
```

– a singleton capability $\{\sigma:\theta\}$ is translated simply as a value,
– a value of type $[\sigma]$ is translated as unit.

– a group capability $\{\rho:\theta\}$ is translated as a map indexed by keys,
– a value of type $[\rho]$ is translated as a key.

Properties of the Translation

1) A program and a translation have similar behaviours

$\Rightarrow$ to reason about a well-typed imperative program, it suffices to reason on its functional translation

2) Translation is the identity on pure System F terms

$\Rightarrow$ the framework adds no overcost on pure components

3) Translated programs are well-typed in System F

$\Rightarrow$ allow to describe a structured store by a map without requiring dependent types

Example 1: Factorial

Typing

Imperative program:

```
let rec facto n =
  let r = ref 1 in
  for i = 2 to n do
    let p = i * (get r) in
    set p r;
  done;
  get r
```

Typing:

facto : int -> int
n : int
r : [R]
i : int
p : int
{R : ref int}
(R is a region name)

Typing of primitives: Let ε stand for **{R : ref int}** in

ref : int -> ∃R, [R] * ε
get : [R] ->_{ε} **int**
set : int -> [R] ->_{ε} **unit**
for-loop : int -> int -> (int ->_{ε} **unit)** ->_{ε} **unit**

Translation

Imperative program:

```
let rec facto n =
  let r = ref 1 in
  for i = 2 to n do
    let p = i * (get r) in
    let () = set p r in
    ()
  done;
  get r
```

Functional Program:

```
let rec facto n =
  let r,R1 = (),1 in
  let R2 = fold 2 n
    (fun i R ->
      let p = i * R in
      let R' = p in
      R') R1 in
  R2
```

Comments:

– The capability `{R : ref int}` is materialized in output code.

– For loop is translated as a `fold` on a sequence of integers.

Specification

Specification of the function in Coq:

```
Lemma facto_prop : forall n, n >= 0 ->
   result facto n (eq (factorial n)).
```

– **factorial n** is a defined in the logic as the generalized product of naturals in the set [1,n]

– again, **result f n (eq r)** means that the application of function **f** to argument **n** is safe and returns the value **r**

Note: the proof would involve the two properties of factorial

```
Lemma factorial_0 : factorial 0 = 1.
Lemma factorial_n : forall n, n > 0 ->
  factorial n = n * factorial (n-1).
```

Example 2: Mutable lists

Definition and Constructors

$$
\begin{array}{lcl}
\mathrm{mlist}\,\theta & := & \mu L.\,\mathrm{ref}\,(\mathrm{unit} + \theta \times L) \\
 & \rhd & \mu L.\,(\mathrm{unit} + [\![\theta]\!] \times L)
\end{array}
$$

$$
\begin{array}{lcl}
\mathrm{empty} & : & \forall\theta.\,\mathrm{unit} \to \exists\sigma.\,[\sigma] * \{\sigma : \mathrm{mlist}\,\theta\} \\
 & := & \lambda().\,\mathrm{ref}\,(\mathrm{inj}^1\,()) \\
 & \rhd & \lambda().\,((), \mathrm{inj}^1\,())
\end{array}
$$

$$
\begin{array}{lcl}
\mathrm{cons} & : & \forall\theta\,\sigma_x\,\sigma_l.\,[\sigma_x] \times [\sigma_l] * \{\sigma_x : \theta\} * \{\sigma_l : \mathrm{mlist}\,\theta\} \\
 & & \quad \to \exists\sigma.\,[\sigma] * \{\sigma : \mathrm{mlist}\,\theta\} \\
 & := & \lambda(h,t).\,\mathrm{ref}\,(\mathrm{inj}^2\,(h,t)) \\
 & \rhd & \lambda(h,t).\,((), (\mathrm{inj}^2\,(h,t)))
\end{array}
$$

Iterator and Reverse

$$\text{iter} \quad : \quad \forall\theta\,\epsilon.\,(\forall\sigma_x.\,[\sigma_x] \rightarrow_{\epsilon*\{\sigma_x:\theta\}} \text{unit}) \rightarrow \forall\sigma.\,[\sigma] \rightarrow_{\epsilon*\{\sigma:\text{mlist}\,\theta\}} \text{unit}$$

$$\begin{array}{lll}
\text{reverse} & : & \forall\sigma\,\theta.\,[\sigma] * \{\sigma : \text{mlist}\,\theta\} \rightarrow \exists\sigma'.\,[\sigma'] * \{\sigma' : \text{mlist}\,\theta\} \\
& := & \text{let } f = \mu\, aux.\lambda(l,p).\,\text{match}\,(\text{get}\,l)\,\text{with} \\
& & \quad \mid \text{inj}^1\,() \Rightarrow p \\
& & \quad \mid \text{inj}^2\,(h,t) \Rightarrow \text{set}\,(l, \text{inj}^2\,(h,p))\,;\ aux\,(t,l) \\
& & \text{in } \lambda l.\,(f\,(l, \text{empty}\,())) \\
& \rhd & \text{let } f = \mu\, aux.\lambda((),(),l,p).\,\text{match}\,l\,\text{with} \\
& & \quad \mid \text{inj}^1\,() \Rightarrow ((),p) \\
& & \quad \mid \text{inj}^2\,(h,t) \Rightarrow \\
& & \qquad \text{let } l' = \text{inj}^2\,(h,p)\,\text{in} \\
& & \qquad aux\,((),(),t,l') \\
& & \text{in } \lambda((),l).\,(f\,((),(),l,\text{empty}\,()))
\end{array}$$

Example 3: Union-find

Creation of a new node

$$
\begin{array}{lll}
\text{node}\,\rho & := & \text{ref}\,(\text{unit} + [\rho]) \\
 & \rhd & \text{unit} + \text{key}
\end{array}
\qquad
\left[\begin{array}{l} [\rho] \\ \{\rho : \text{node}\,\rho\} \end{array}\right.
$$

$$
\begin{array}{lll}
\text{new\_node} & : & \forall \rho.\, \text{unit} \rightarrow_{\{\rho : \text{node}\,\rho\}} [\rho] \\
 & := & \lambda().\, \text{ref}\,(\text{inj}^1\,()) \\
 & \rhd & \lambda((), r). \\
 & & \quad \text{let}\, k = \text{map\_fresh}\, r\, \text{in} \\
 & & \quad \text{let}\, r' = \text{map\_add}\, r\, k\, (\text{inj}^1\,())\, \text{in} \\
 & & \quad (k, r')
\end{array}
$$

A new reference is allocated and is then adopted by region ρ.

Others operations

$$\text{find} \quad : \quad \forall\rho.\,[\rho] \rightarrow_{\{\rho:\text{node}\,\rho\}} [\rho]$$

Modifies the map translating the capability on region ρ by reading and writing into that map.

$$\text{unify} \quad : \quad \forall\rho.\,[\rho] \times [\rho] \rightarrow_{\{\rho:\text{node}\,\rho\}} \text{unit}$$

Calls "find" twice and then update the map.

$$\text{are\_unified} \quad : \quad \forall\rho.\,[\rho] \times [\rho] \rightarrow_{\{\rho:\text{node}\,\rho\}} \text{bool}$$

Calls "find" twice. The comparison of two pointers in the source translates as the comparison of two keys.

Example 4: Eratosthene

Typing of Imperative Source

```
let rec iter_primes n f =                    {P:ref bool}
  if (n < 2) then () else begin
    let p = ref true in                                    ε
    let g m =
       if n mod m = 0 then p := false ; f m in
    iter_prime (n-1) g;
    if !p then f n
  end
```

Types involved:

iter_primes : $\forall\varepsilon$, $\texttt{int -> (int ->}_{\varepsilon}\texttt{ int) ->}_{\varepsilon}\texttt{ unit}$

p : [P]   with {P : ref bool}

Recursive call to iter_primes at type:

$\texttt{int -> (int ->}_{\varepsilon*\{P:ref\ bool\}}\texttt{ int) ->}_{\varepsilon*\{P:ref\ bool\}}\texttt{ unit}$

Translation

```
let rec iter_primes n f e1 =
  if (n < 2) then e1 else begin
    let p1 = true in
    let g m (e'1,p'1) =
      let p'2 = if n mod m = 0 then false else p'1 in
      let e'2 = f m e'1 in
      (e'2,p'2) in
    let e2,p2 = iter_primes (n-1) g (e1,p1) in
    let e3 = if p2 then (f n e2) else e2 in
  e3
  end
```

: $\forall\alpha$, int -> (int -> α -> int*α) -> α -> α

(was) : $\forall\varepsilon$, int -> (int ->$_\varepsilon$ int) ->$_\varepsilon$ unit

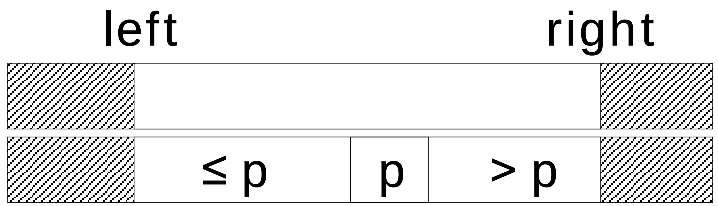
Example 5: Quicksort

Imperative Source

Imperative program:

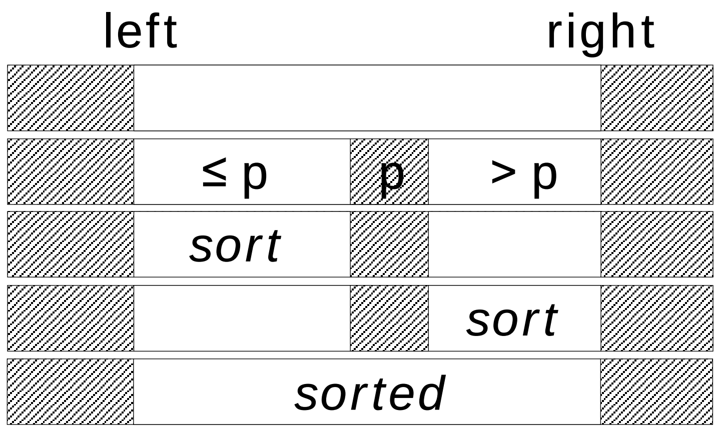
```
let quicksort smaller tab =

  let split left right =
     ...
```




```
  let sort left right =
     let piv = split left right
     sort left piv;
     sort (piv+1) right;

  sort 0 (size tab)
```



Typing and Translation

Type of source in System F + imperative features:

quicksort: $\forall\alpha$, **(**α **->** α **-> bool) -> array** α **-> unit**

Type of source in System F + regions & capabilities:

quicksort: $\forall\alpha$, **(**$\forall\sigma_1\ \sigma_2$, **[**$\sigma_1$**] -> [**$\sigma_2$**] ->**$_{!\{\sigma 1:\alpha\}!\{\sigma 2:\alpha\}}$ **bool)**
-> $\forall\sigma$, **[**σ**] ->**$_{\{\sigma\ :\ \text{array}\ \alpha\}}$ **unit**

Type of translation in System F:

quicksort: $\forall\alpha$, **(**α **->** α **-> bool) -> array**^{F} α **-> array**^{F} α

where **array**^{F} is the type of purely applicative arrays.

Specification of Quicksort

```
Lemma quicksort_spec : forall A tab smaller Smaller,
  total_order.rel Smaller ->
  correspond2 smaller Smaller ->
  result2 (quicksort A) smaller tab (fun tab' =>
    permut tab tab' /\ sorted Smaller tab').
```

In-bounds Checks

Accesses to a cell of the array is garrantied inbound by typing.

An integer can be cast into a valid array cell pointer,
generating a proof obligation at that point.

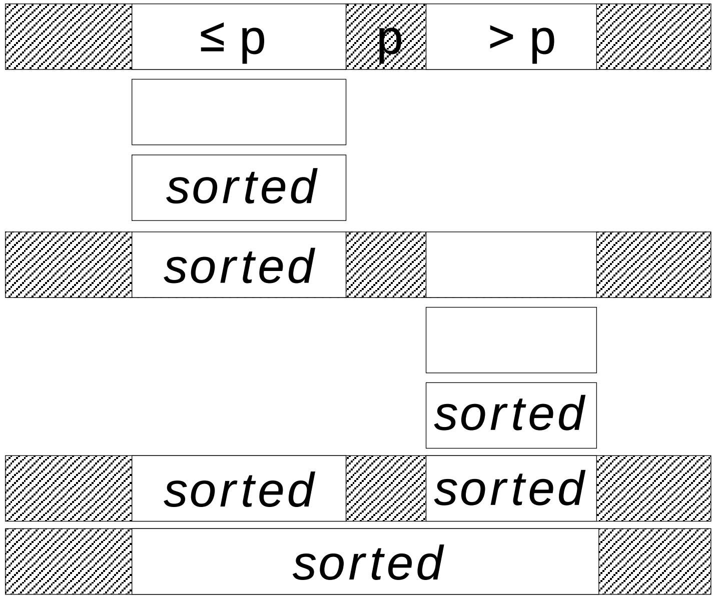
```
Given
<ρ:array>   garranties that the region is an array
{ρ:ref θ}   the capability is translated as map h

Subtyping operation is
shift : [ρ] -> int -> [?ρ]
cast  : [?ρ] <= [ρ]
        if translation is k then the cast generates
            assert k ∈ dom(h)
```

Sub-arrays for Recursive Calls

Recursive calls to the sort function thread only a submap of the map describing the entire array.

This gives us for free the fact that other cells of the array have not been modified.



$$\{\rho\text{:ref } \theta\} \equiv \exists \rho',\ \{\rho\text{:ref } \theta \setminus K\} * \{\rho'\text{:ref } \theta\}$$
$$* <\rho' \subset \rho> * <\rho_{|K} \subset \rho'>$$

where K set of keys

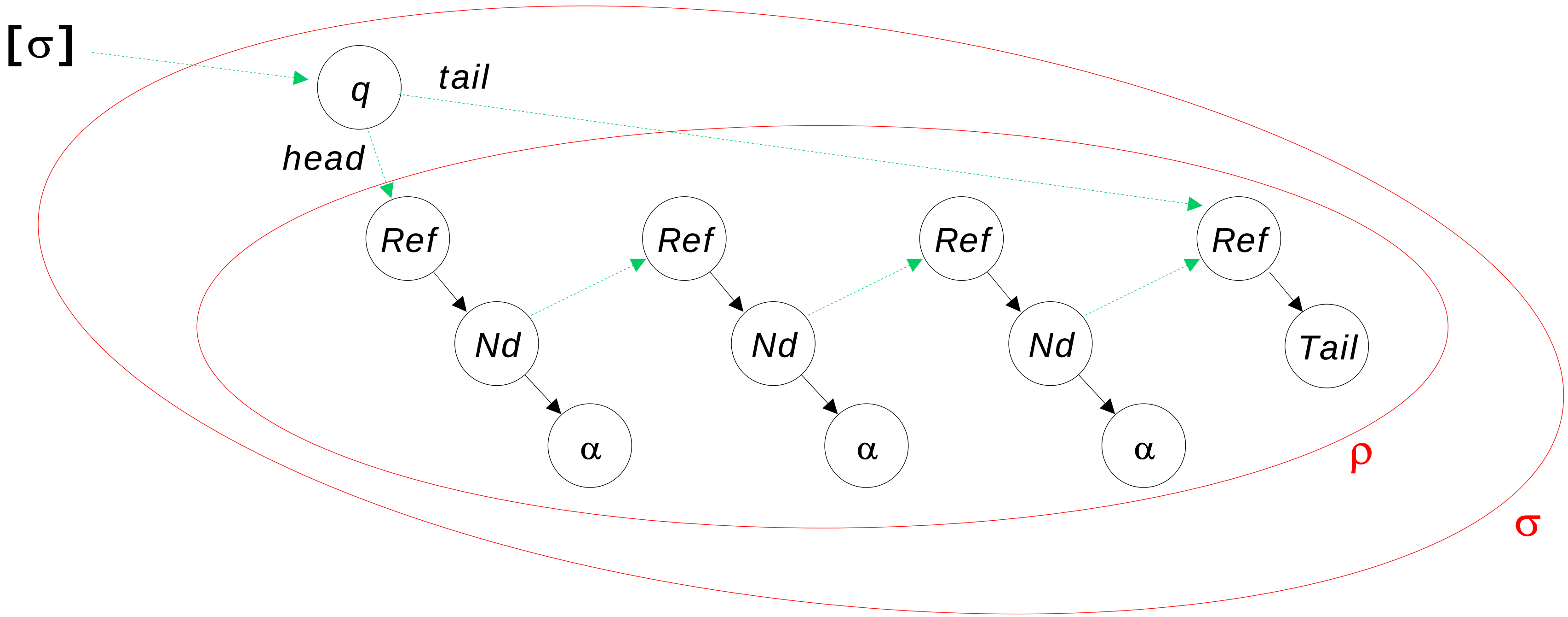
cast : $[\rho'] <= [\rho]$ **is for free**

cast : $[\rho] <= [\rho']$ **generates** **assert k** $\in$ **dom(h)**

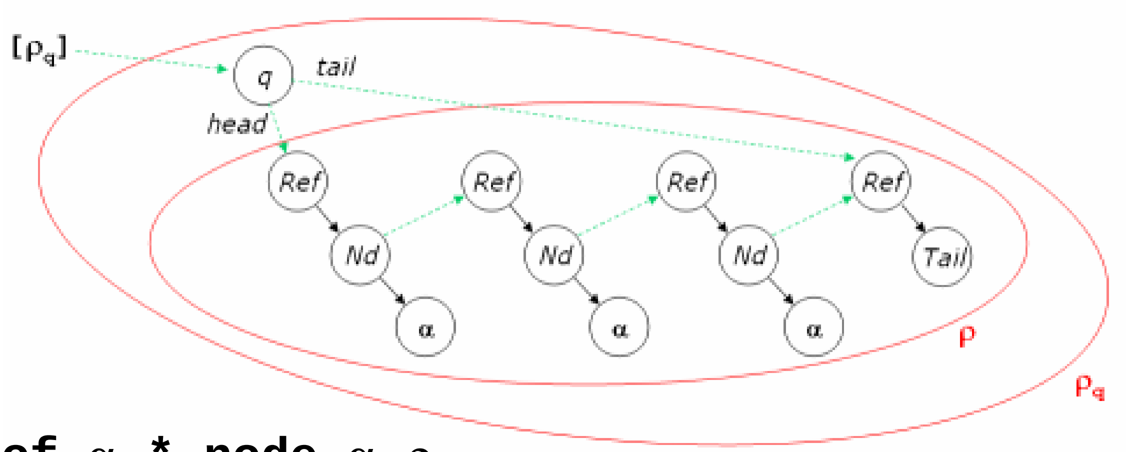
Example 6: Mutable Queues

Typing in ML

```
cell α = Tail | Node of α * node α
node α = ref (cell α)
queue α = { mutable head : node α;
            mutable tail : node α }
```



Typing and Translation of Types



Types in source:

```
cell α ρ = Tail | Node of α * node α ρ
node α ρ = [ρ]
queue α ρ = { mutable head : node α ρ;
              mutable tail : node α ρ }
Queue α = ∃ρ. (queue α ρ) ∗ {ρ* : ref (cell α ρ)}
```

Types in translation:

```
cell α = Tail | Node of α * key
node α = key
queue α = { head : node α; tail : node α }
Queue α = (map key (cell α)) * queue α
```

Operations

Type in source:

```
create : unit -> ∃Q.[Q]∗{Q:Queue α}
push   : [X] -> [Q]∗{X:α}∗{Q:Queue α} -> unit∗{Q:Queue α}
pop    : [Q]∗{Q:Queue α} -> ∃X.[X]∗{X:α}∗{Q:Queue α}
append : ([Q1]x[Q2])∗{Q1:Queue α}∗{Q2:Queue α}
          -> unit∗{Q1:Queue α}
```

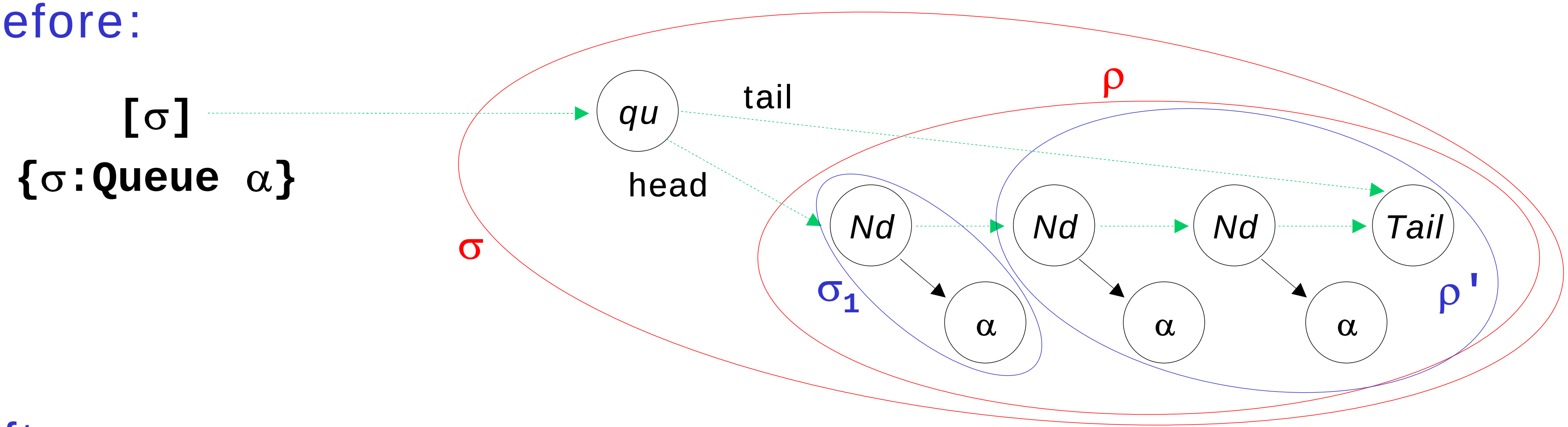
Type in translation:

```
create : unit -> Queue β
push   : β -> Queue β -> Queue β
pop    : Queue β -> β * Queue β
append : Queue β -> Queue β -> Queue β
```

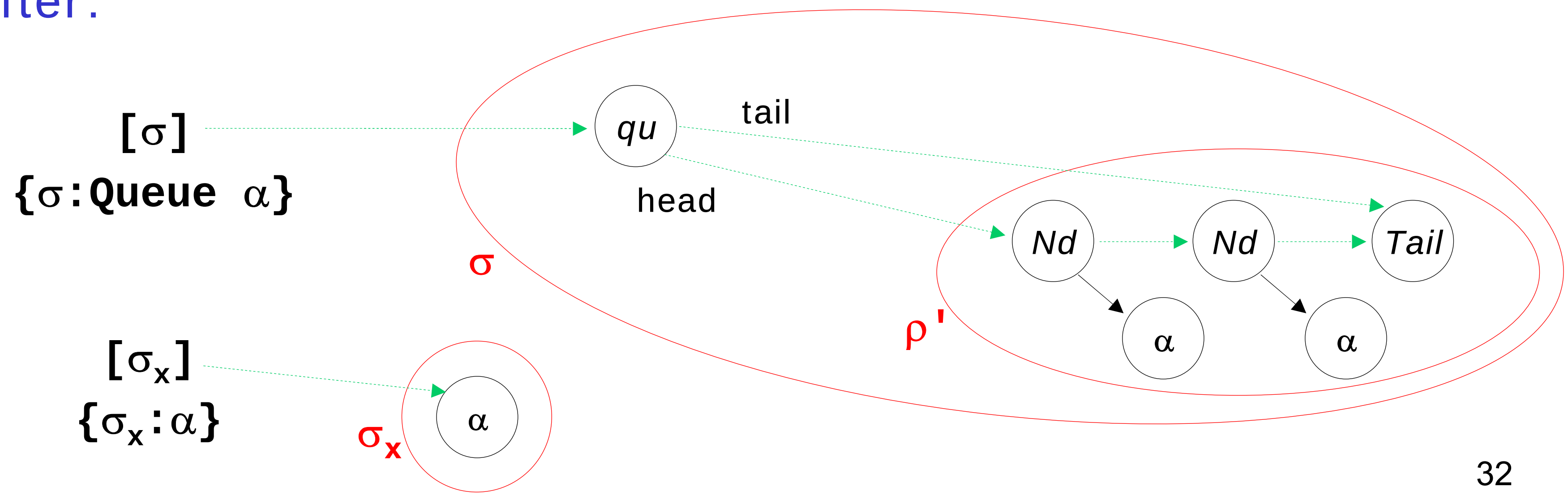
Pop: Cut Regions

Reminder: **Queue α = ∃ρ. (queue α ρ) * {ρ* : ref (cell α ρ)}**

Before:



After:

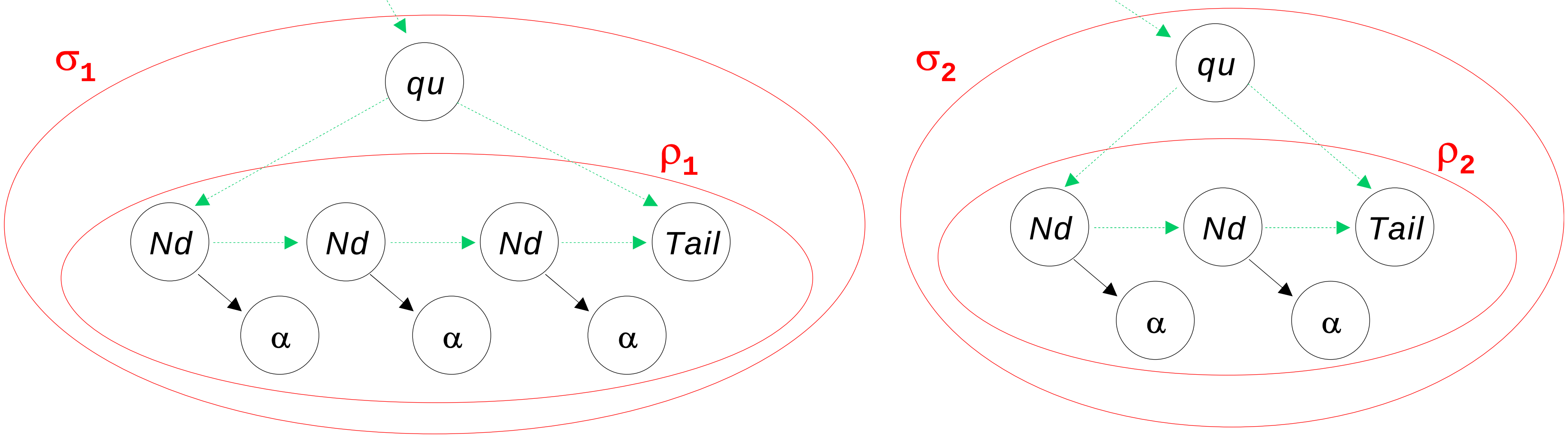


Append: Merge Regions

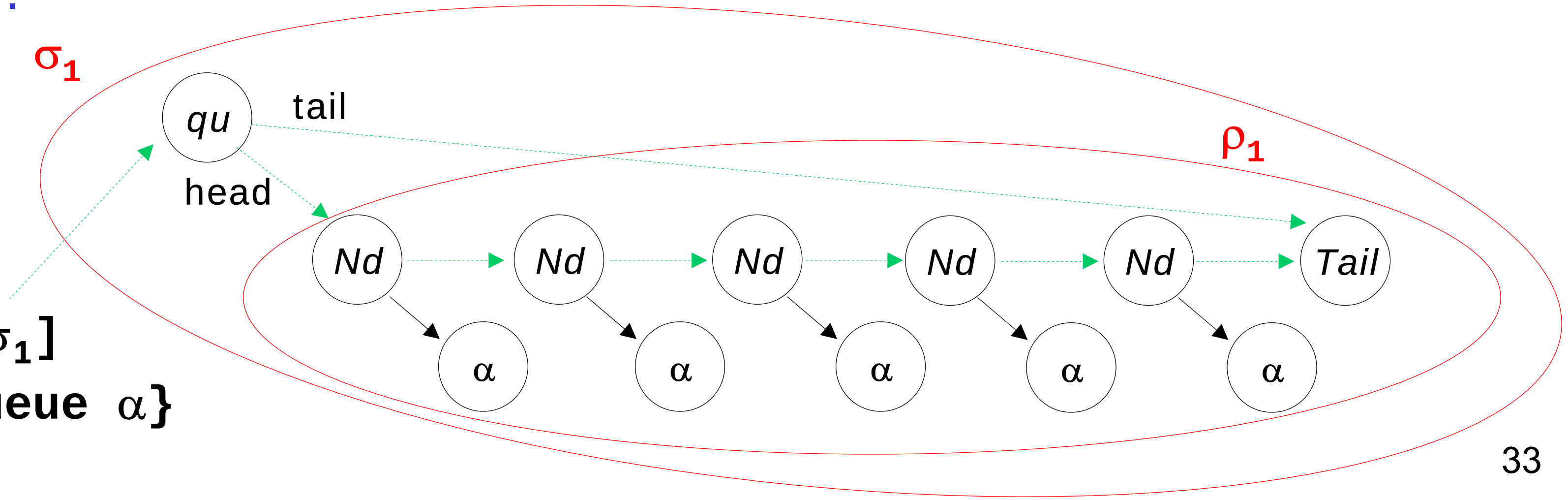
Before:

$[\sigma_1]$ $\{\sigma_1\text{:Queue } \alpha\}$

$[\sigma_2]$ $\{\sigma_2\text{:Queue } \alpha\}$



After:



$[\sigma_1]$
$\{\sigma_1\text{:Queue } \alpha\}$

A Few Typing Rules

Main Typing Rules

VAR

$$\frac{(x : \tau) \in \Delta}{\Delta \vdash x : \tau \;\; \triangleright\; x}$$

FUN

$$\frac{\Delta, x : \chi_1 \Vdash t : \chi_2 \;\; \triangleright\; u}{\Delta \vdash (\lambda x.\, t) : (\chi_1 \to \chi_2) \;\; \triangleright\; (\lambda x.\, u)}$$

VAL

$$\frac{\Delta \vdash v : \tau \;\; \triangleright\; w}{\Delta \Vdash v : \tau \;\; \triangleright\; w}$$

FRAME

$$\frac{\Gamma \Vdash t : \chi \;\; \triangleright\; u}{(\Gamma, x : C) \Vdash t : (\chi * C) \;\; \triangleright\; (u, x)}$$

APP

$$\frac{\Delta \vdash v : (\chi_1 \to \chi_2) \;\; \triangleright\; w \qquad \Delta, \Gamma \Vdash t : \chi_1 \;\; \triangleright\; u}{\Delta, \Gamma \Vdash (v\, t) : \chi_2 \;\; \triangleright\; (w\, u)}$$

LET

$$\frac{\Delta, \Gamma_1 \vdash t_1 : \chi_1 \;\; \triangleright\; u_1 \qquad \Delta, x : \chi_1, \Gamma_2 \vdash t_2 : \chi_2 \;\; \triangleright\; u_2}{\Delta, \Gamma_1, \Gamma_2 \vdash (\mathrm{let}\, x = t_1 \,\mathrm{in}\, t_2) : \chi_2 \;\; \triangleright\; (\mathrm{let}\, x = u_1 \,\mathrm{in}\, u_2)}$$

Conclusions

Related Work

Regions and Capabilities

– Stack of Region, Tofte, Talpin, and later effects type systems

– Calculus of Capabilities, Crary, Walker, Morrisset

– Alias Types, Smith, Walker, Morrisset

– Adoption & Focus, Fahndrich, DeLine

– Connecting Effects & Uniqueness with Adoption, Boyland, Retert

Other Related Works

– Separation Logic, Stateful Views – Monads, Monadic Translation

– The "Why" tool, Filliâtre

– Linear Language with Locations, Linear Regions are all You Need

– From Algol to Poly. Linear λ-calculus, O'Hearn, Reynolds

– Logics for higher-order functions, Honda and *al*

– Hoare Logic for CBV Function Programs, Pottier, Regis-Gianas

Future Work

Soon

– Formalization of advanced features of the type system.

– Setting up of a convenient way to reason about functional programs using Coq (using strongest post-condition?).

Then

– Partial type inference, user-level syntax.

– Implementation.

– Realistic demos.

Thanks!