Formal Reasoning on Imperative ML Programs

Arthur Charguéraud

Joint work with François Pottier

INRIA-Rocquencourt

Types, Logics and Semantics for State   Dagstuhl, 2008-02-04

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.

$$\frac{\Gamma \vdash v : \tau}{\Gamma; \{\} \vdash v : \tau}$$

HOL

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

Imperative language ↓ compilation, execution

Functional language ↓ specification, certification

– Running example: factorial

– Additional examples: quicksort, mutable queues

1) Typing Imperative Programs

A type system that extends System F with two ingredients.

Regions: are sets of values,

$[\rho]$ is the type of a value that belongs to region ρ.

Capabilities:
– describe ownership of regions,
(the exclusive right to read or write in a region)
– give the type of the corresponding piece of state.

$\{\rho:\theta\}$ is a singleton region (thus $[\rho]$ singleton type)

$\{\rho^*:\theta\}$ is a group region (used for aliasing).

Mechanisms for merging a singleton region into a group region and reverse are provided, extending the "Adoption and Focus" techniques (Fahndrich and DeLine 2002).

1) Typing Imperative Programs

Values types (non-linear):

$$\tau \quad := \quad \bot \mid \mathrm{unit} \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \mid \sigma_1 \rightarrow \sigma_2 \mid [\rho]$$

Memory types (linear):

$$\theta \quad := \quad \bot \mid \mathrm{unit} \mid \theta_1 + \theta_2 \mid \theta_1 \times \theta_2 \mid \sigma_1 \rightarrow \sigma_2 \mid [\rho] \mid \mathrm{ref}\ \theta \mid \mathrm{array}\ \theta$$

Computation types (linear):

$$\sigma \quad := \quad \exists \bar{\rho}.\, \tau.\, \bar{C}$$

Typing judgments:

– for values $\quad \overbrace{\Gamma}^{x\,:\,\tau} \vdash v : \tau$

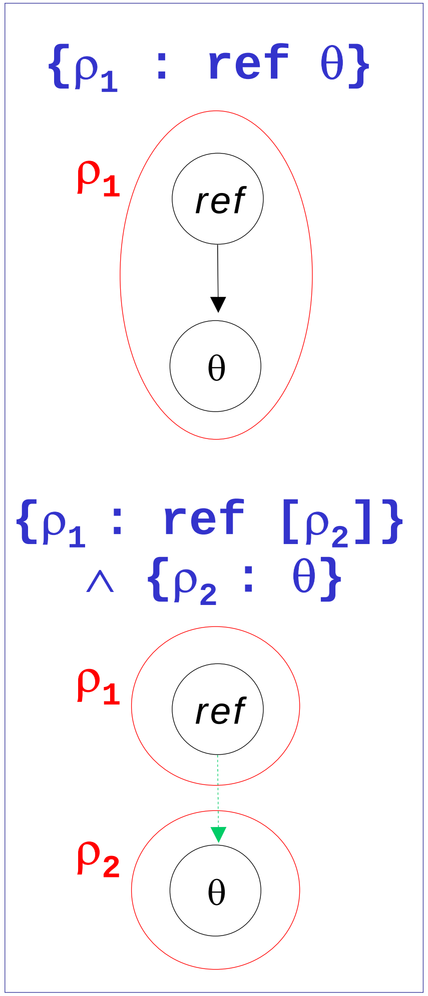
– for terms $\quad \Gamma ; \bar{C} \vdash t : \sigma$

1) Typing References

Our typing rules for reference primitives:

$$
\begin{array}{lll}
\text{ref} & : & \tau \to \exists\rho.[\rho]\,\{\rho : \text{ref } \tau\} \\
\text{get} & : & [\rho]\,\{\rho : \text{ref } \tau\} \;\to\; \tau\,\{\rho : \text{ref } \tau\} \\
\text{set} & : & \tau_2 \to [\rho]\,\{\rho : \text{ref } \tau_1\} \to \text{unit } \;\{\rho : \text{ref } \tau_2\}
\end{array}
$$

$$\{\rho_1 : \text{ref } \theta\} \equiv \exists\rho_2.\{\rho_1 : \text{ref } [\rho_2]\}\{\rho_2 : \theta\}$$



Effect-style notation:

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

Rules derivable in our system:

$$
\begin{array}{lll}
\text{get} & : & [\rho] \to_{\{\rho:\text{ref } \tau\}} \tau \\
\text{set} & : & [\rho] \times \tau \to_{\{\rho:\text{ref } \tau\}} \text{unit} \\
\text{ref} & : & \tau \to_{\{\rho^\star:\text{ref } \tau\}} [\rho] \\
\text{map} & : & (\alpha \to_\epsilon \beta) \to \text{list } \alpha \to_\epsilon \text{list } \beta
\end{array}
$$

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 -> exists R, [R] $\wedge$ ε

get : [R] $->_{\varepsilon}$ int

set : int -> [R] $->_{\varepsilon}$ unit

for-loop : int -> int -> (int $->_{\varepsilon}$ unit) $->_{\varepsilon}$ unit

2) 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
```

Interesting cases:

A singleton capability $\{\rho:\theta\}$ is translated as a single value.

A group capability $\{\rho^*:\theta\}$ is translated as a map indexed by keys.

A value of type $[\rho]$ is translated as a key,

and it is usually erased if it is a singleton type.

2) Factorial: 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
```

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

– Value `r : [R]` is erased during translation.

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

3) Factorial: Description

Functional program:

```
let rec facto n =
 let R1 = 1 in
 let body i R =
   let R' = i * R in
   R' in
 let R2 = fold 2 n body R1 in
 R2
```

Higher-order logic formula:

```
Axiom facto : int -> int.
Axiom facto_descr : ∀ n,
 ∃ R1, R1 = 1 ∧
 ∃ body, (∀ i, ∀ R,
   ∃ R', R' = i * R ∧
   result2 body i R (= R')) ∧
 Let R2 =app4 fold 2 n body R1 in
 result1 facto n (= R2).
```

`result f x (= n)` defined as `safe f x /\ f x = n`

the application of function `f` to `x` is safe and returns `n`.

`Let y =app f x in P` defined as `safe f x -> ∃ y, y = f x /\ P`

if the application of `f` to `x` is safe then `y` is bound to `f x` in P

3) Strongest Post-Condiction

Idea: given a functional program, generate a higher-order logic formula that fully characterizes its behaviour.

$\mathrm{safe}\; f\; x$ is an abstract predicate (in Coq) that holds iff function f terminates without error on the input x.

$[\![\, v \,]\!]_r$ is the strongest post-condition of value v, in which r is the logical name associated to value v.

$[\![\, t \,]\!]_r^s$ is the strongest post-condition for term t, in which
– s is a proposition provable iff t terminates without error (s describes the "safety" of t),
– if s holds, then r is the logical name associated to the result of the evaluation of t.

3) Strongest Post-Condiction

A value, reflected by r:

$$
\begin{array}{ll}
[\![x]\!]_r & \equiv (r = x) \\
[\![(v_1, v_2)]\!]_r & \equiv \exists r_1, \exists r_2, (r = (r_1, r_2)) \wedge [\![v_1]\!]_{r_1} \wedge [\![v_2]\!]_{r_2} \\
[\![\mathrm{inj}^i\ v]\!]_r & \equiv \exists r_i, (r = \mathrm{inj}^i\ r_i) \wedge [\![v]\!]_{r_i} \\
[\![\lambda x.\, t]\!]_r & \equiv \forall x, [\![t]\!]_{(r\ x)}^{(\mathrm{safe}\, r\ x)} \\
[\![\mu r. \lambda x. t]\!]_r & \equiv [\![\lambda x.\, t]\!]_r
\end{array}
$$

A term, with safety reflected by s, and result by r:

$$
\begin{array}{ll}
[\![v]\!]_r^s & \equiv [\![v]\!]_r \wedge s \\
[\![\mathrm{let}\, x = t_1 \,\mathrm{in}\, t_2]\!]_r^s & \equiv \exists x, \exists s_1, [\![t_1]\!]_x^{s_1} \wedge (s_1 \Rightarrow [\![t_2]\!]_r^s) \\
[\![f\ x]\!]_r^s & \equiv \mathrm{safe}\, f\ x \Rightarrow s \wedge (r = f\ x) \\
[\![\mathrm{match}\ x\ \mathrm{with}\ \varnothing]\!]_r^s & \equiv \mathrm{True} \\
[\![\mathrm{match}\ x\ \mathrm{with}\ (p \rightarrow t) \mid c]\!]_r^s & \equiv \Big(\exists \mathrm{FV}(p), (x = p) \Rightarrow [\![t]\!]_r^s \Big) \\
& \quad \vee \Big([\forall \mathrm{FV}(p), (x \neq p)] \Rightarrow [\![\mathrm{match}\ x\ \mathrm{with}\ c]\!]_r^s \Big) \\
[\![\mathrm{assert}\ P\ \mathrm{in}\ t]\!]_r^s & \equiv P \Rightarrow [\![t]\!]_r^s
\end{array}
$$

4) Factorial: Certification

Certification of function facto for the following specification:

```
Lemma facto_prop : forall n, n >= 0 ->
  result1 facto n (= (factorial n)).
```

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

– again, **result1 f n (= r)** means that the application of function **f** to argument **n** is safe and returns the value **r**

Proof is short and simple, and involves the two properties:

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

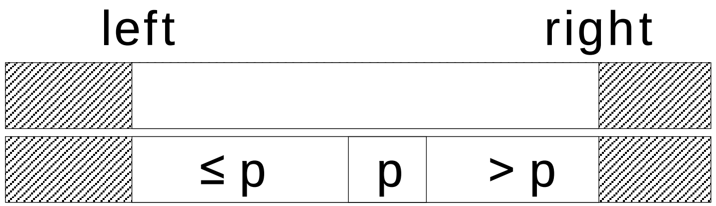
Demo 2: Quicksort

1) Imperative Source

Imperative program:

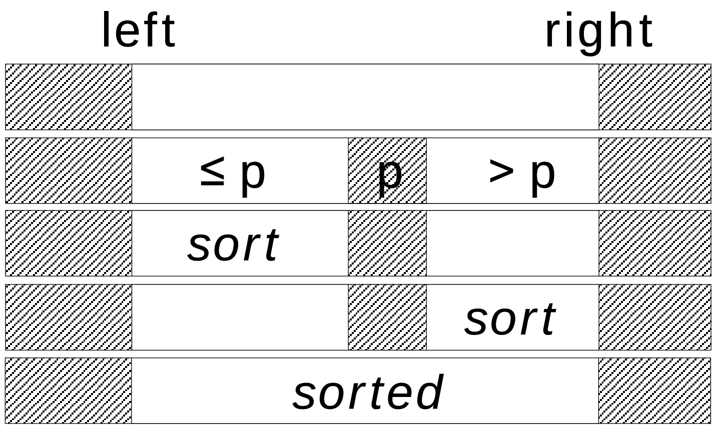
```
let quicksort smaller tab =

  let split left right =
     ...
```




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

  sort 0 (size tab)
```



2) Typing, Translation, Description

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: $\forall\alpha$, **(**$\forall\rho_1$ **p**$_2$**, [p**$_1$**] -> [p**$_2$**] !{**ρ_1**:**α**}!{**ρ_2**:**α**} -> bool) ->**
$\forall\rho$, **[**ρ**] {**ρ **: array** α**} -> unit {**ρ **: array** α**}**

Type of translation in System F:

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

Type of reflected function in Coq:

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

– **array**^{F} is the type of functional arrays (i.e. purely applicative),
– **obj** is a subtype **Set** restricted to inhabited comparable types.

3) Description, Specification

Description axioms:

```
Axiom quicksort:
  forall (A:obj), (A -> A -> bool) -> array^F A -> array^F A.
Axiom quicksort_descr: forall (A:obj) smaller tab,
  exists split, ... ∧ exists sort, ... ∧
  Let tab' =app3 sort 0 (size tab) tab in
  result2 (quicksort A) smaller tab (= tab')
```

Property proved about quicksort:

```
Lemma quicksort_spec : forall A smaller tab,
  total2 smaller ->
  total_order.rel smaller ->
  result2 (quicksort A) smaller tab (fun tab' =>
    permut tab tab' /\ sorted smaller tab').
```

Last 2 lines imply:

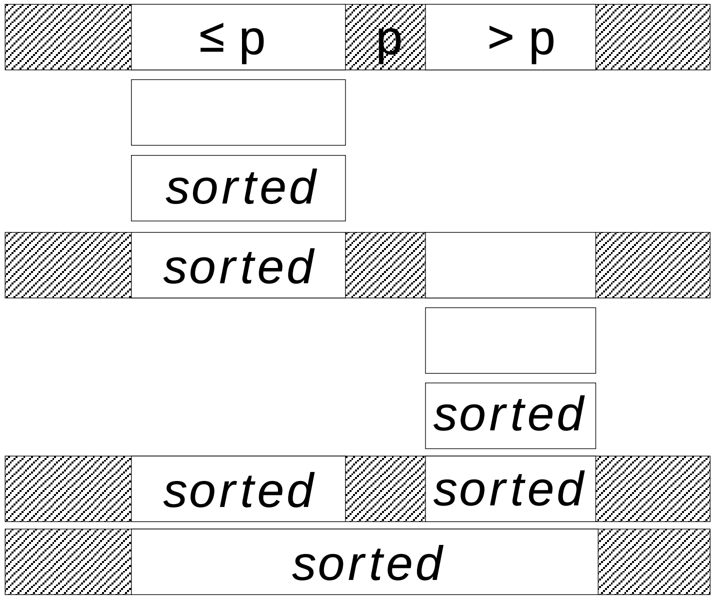
```
   tab' = quicksort A smaller tab ->
   permut tab tab' /\ sorted smaller tab'
```

4) Comments About the Proof

– 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.

– Safety (termination without error) of the recursive function **`sort`** is proved by application of a strong induction principle (in Coq), since the size of the array strictly decreases on rec. 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.

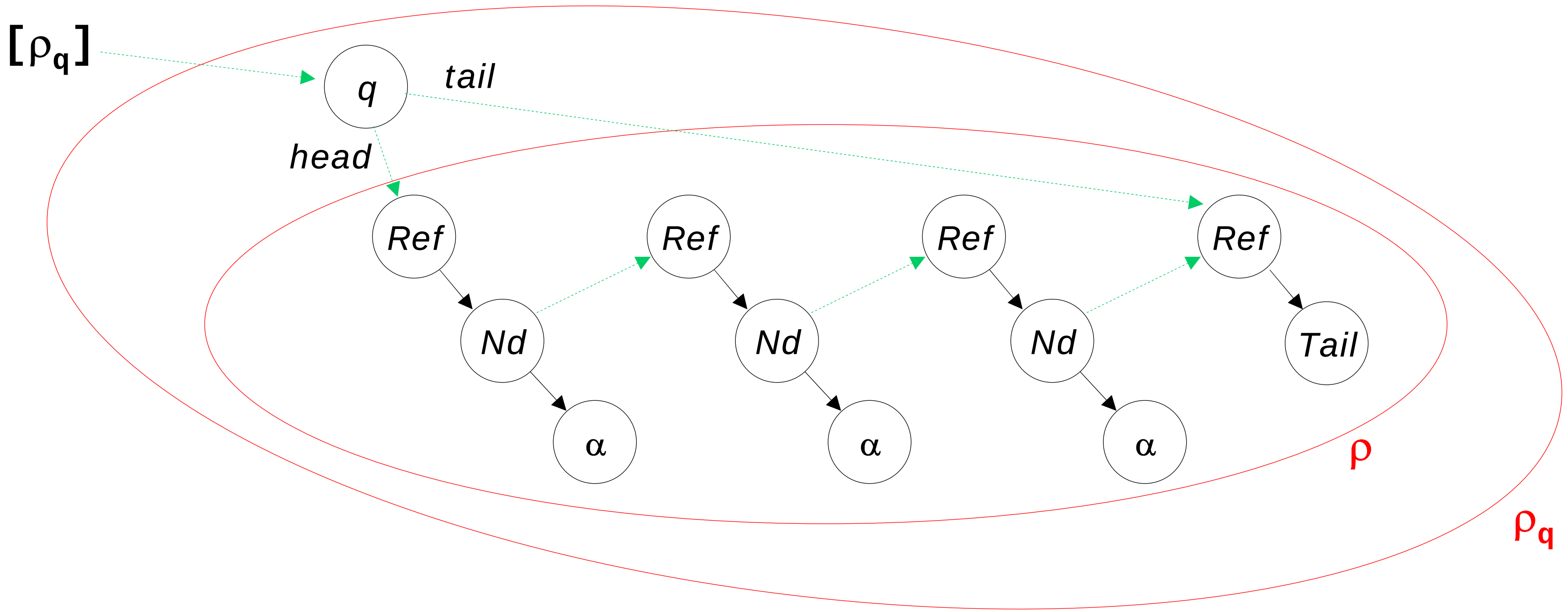


Demo 3: Mutable Queues

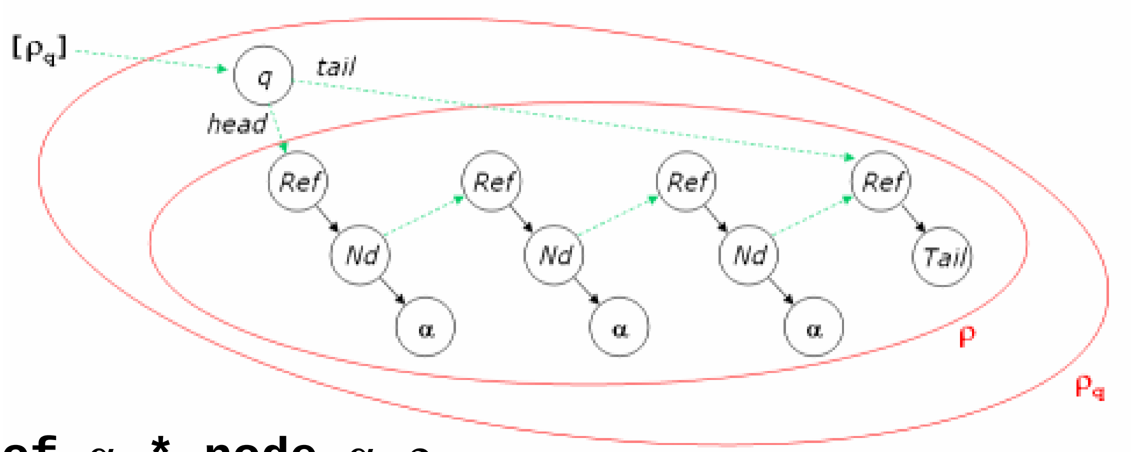
1) Typing

Types 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 }
```



1) Typing, continued



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. {ρ* : ref (cell α ρ)}. queue 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
```

2) Push Operation

Type in source:

```
push : ∀α, [X] -> [Q] {X:α}{Q:Queue α} -> unit {Q:Queue α}
```

Type in translation:

```
push : ∀α, α -> Queue α -> Queue α
```

Specification:

```
push_spec : forall (A:obj) x q L,
  isQueue q L ->
  result2 push x q (fun q' => isQueue q' (L ++ x::nil))
```

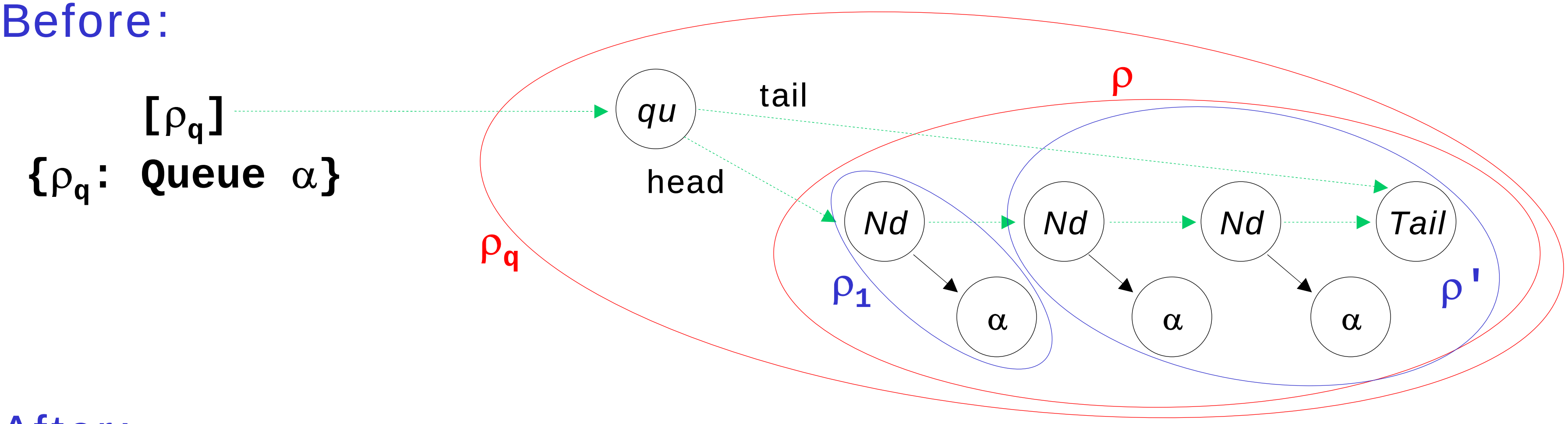
where:

"isQueue q L" is an inductive relation that holds if the object **q** (of type **(map key (cell α))*{head:key;tail:key}**) describes a queue whose elements are the value from list **L** (of type **list α**).

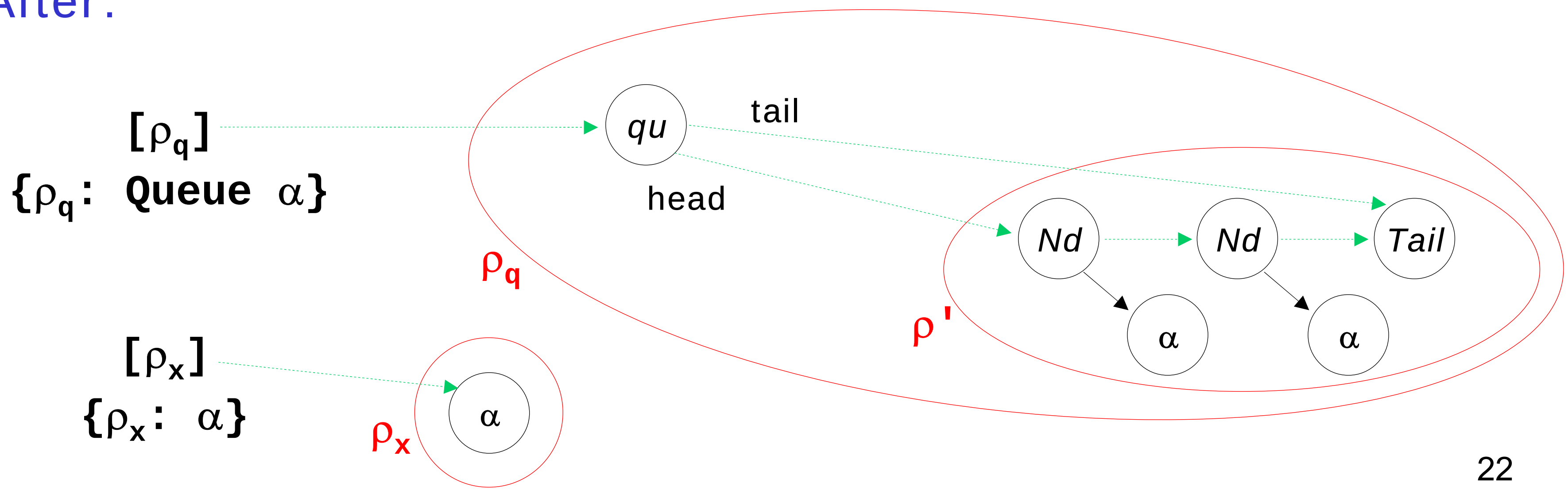
3) Pop Operation

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

Before:

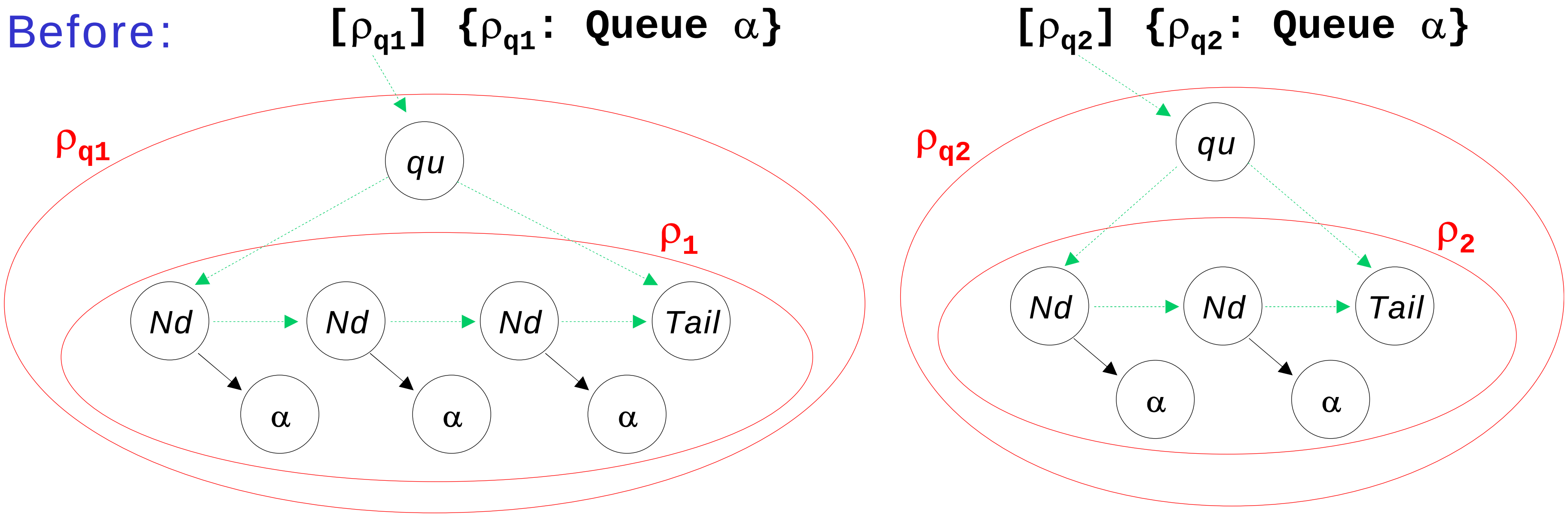


After:

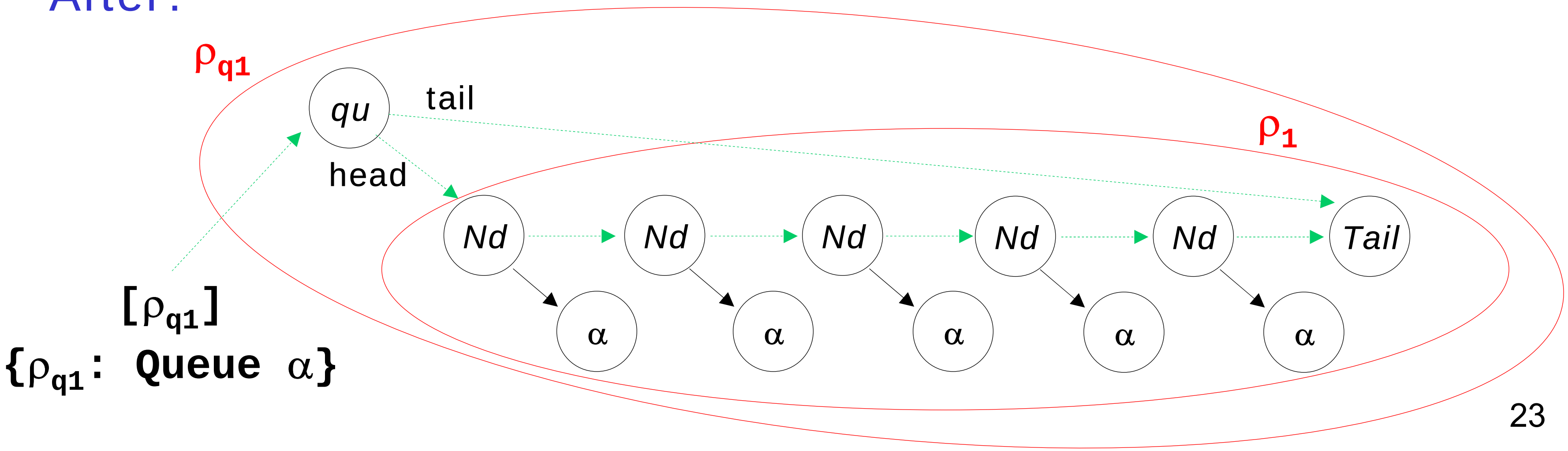


4) Append Operation

Before:



After:



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

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

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

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

– The "Why" tool, Hoare Logic for CBV Function Programs

Conclusion

Results:

– could be a practical way for reasoning on imperative programs,

– which may lead to nice-looking specifications and proofs.

– which adds no overcost when reasoning on pure components,

– which reuses Coq as a target logic and as a proof-assistant,

Main components of our type system and translation in the draft:

Functional Translation of a Calculus of Capabilities (Pottier & I).

Future work:

– formalization of advanced features of the type system,

– formalization of the strongest post-condition algorithm,

– proving coherence of adding such axioms to Coq,

– implementations of the several tools involved.

Thanks!