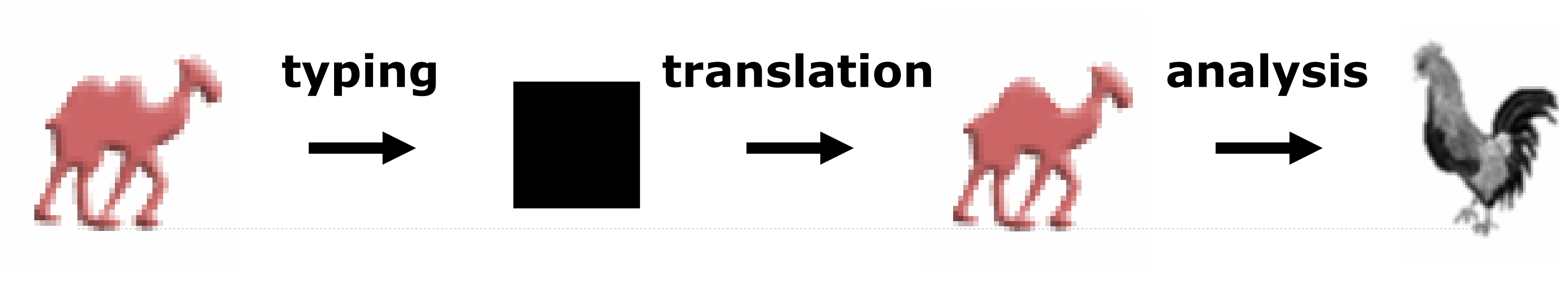
Functional Translation of a Calculus of Capabilities

Arthur Charguéraud

Joint work with François Pottier
INRIA-Rocquencourt

Workshop on Effects and Type Theory 2007-12-13

Overview



typing → **translation** → **analysis** →

Imperative language

Typing with capabilities

Functional language

HO logical formula

A type system

- System-F plus regions and linearly-treated capabilities (static)
- A capability is an exclusive read-and-write permission

A type-directed translation

- From imperative code towards an equivalent functional code
- The state is represented by the translation of capabilities

Simple Reference – Typing

Imperative source:

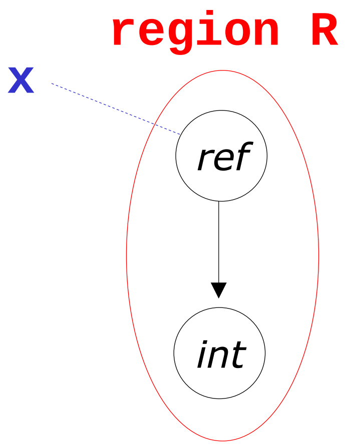
```
let x = ref 7
let y = get x
let _ = set (x,'c')
```

Typing of values:

```
7 : int       x  : [R]
y : int      'c' : char
_ : unit
```

Typing of primitives :

$$\begin{array}{lcl} \mathrm{ref} & : & \tau \longrightarrow \exists\rho.\{\rho : \mathrm{ref}\ \tau\}\,[\rho] \\ \mathrm{get} & : & \{\rho : \mathrm{ref}\ \tau\}\,[\rho] \longrightarrow \{\rho : \mathrm{ref}\ \tau\}\,\tau \\ \mathrm{set} & : & \{\rho : \mathrm{ref}\ \tau_1\}\,([\rho] \times \tau_2) \longrightarrow \{\rho : \mathrm{ref}\ \tau_2\}\,\mathrm{unit} \end{array}$$



Typing with capabilities:

```
let {R:ref int} x = ref 7
let {R:ref int} y = get {R:ref int} x
let {R:ref char} _ = set {R:ref int} (x,'c')
```

Simple Reference – Translation

Typing with capabilities:

```
let {R:ref int} x = ref 7
let {R:ref int} y = get {R:ref int} x
let {R:ref char} _ = set {R:ref int} (x,'c')
```

Functional translation:

```
let R1,x = (λa.(a,1)) 7
let R2,y = (λ(a,1).(a,a)) (R1,x)
let R3,_ = (λ(a1,(1,a2)).(a2,())) (R2,(x,'c'))
```

$$\text{set} \quad : \quad \{\rho : \text{ref } \tau_1\}\,([\rho] \times \tau_2) \longrightarrow \{\rho : \text{ref } \tau_2\}\,\text{unit}$$

After some reductions:

```
let R1,x = 7,1
let R2,y = R1,R1
let R3 = 'c'
```

x : [R] ⟷

Recall the imperative code:

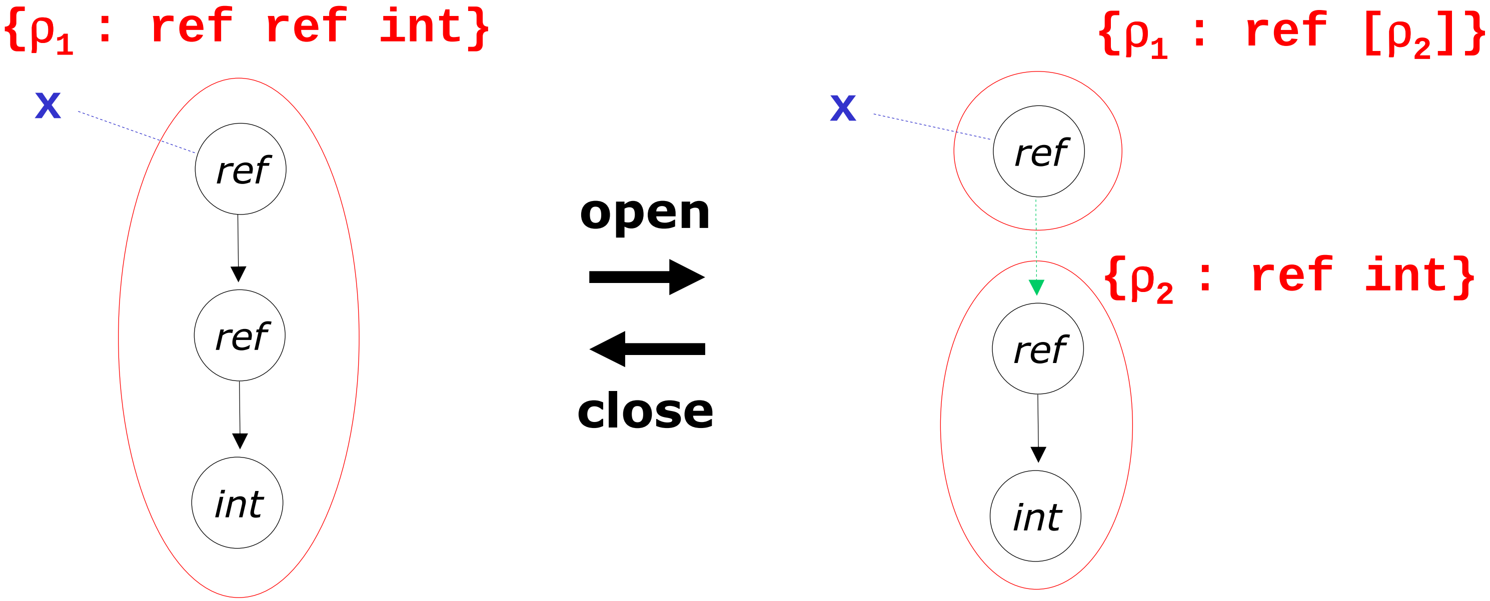
```
let x = ref 7
let y = get x
let _ = set (x,'c')
```

Reference with Linear Contents

τ ranges over non-linear "value types". Thus, "get" is restricted.

$$\boxed{\text{get} \quad : \quad \{\rho : \text{ref } \tau\}\,[\rho] \rightarrow \{\rho : \text{ref } \tau\}\,\tau}$$

This restriction is relieved through the "open" operation:



{ρ_1 : ref ref int}

{ρ_1 : ref [ρ_2]}

{ρ_2 : ref int}

open

close

$$\boxed{\text{OPEN-REF} \quad : \quad \{\rho_1 : \text{ref } \theta\} \;\equiv\; \exists \rho_2.\{\rho_1 : \text{ref } [\rho_2]\} \wedge \{\rho_2 : \theta\}}$$

θ ranges over linear "memory types"

Matrices as 2D-arrays

ML type:

```
x : array (array int)
```

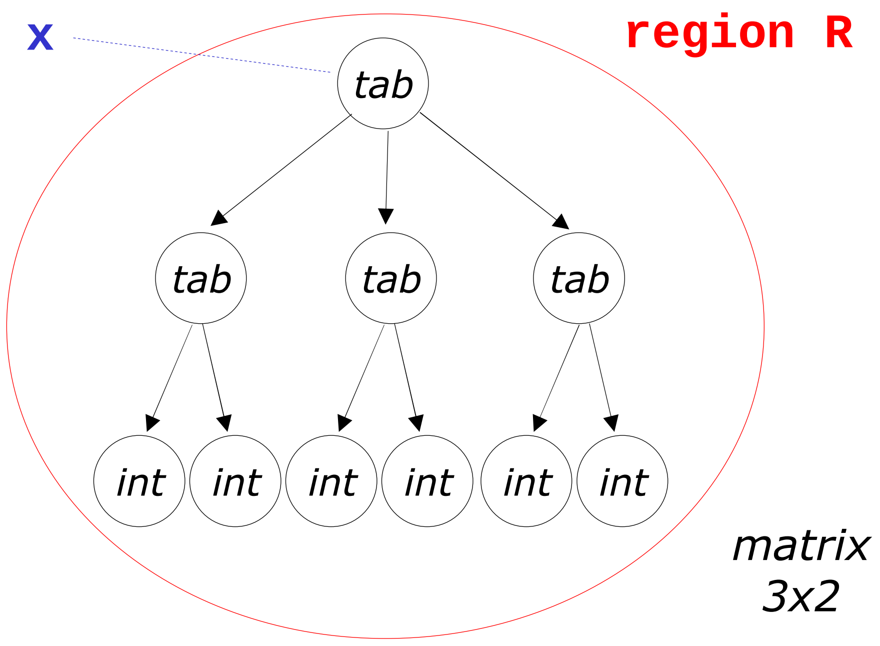
In our system:

```
x : [R]
{R : array (array int)}
```

Type in translation:

R : array^F (array^F int)

Corresponding memory graph:



Matrices with Aliasable Rows

ML type:

```
x : array (array int)
```

In our system:

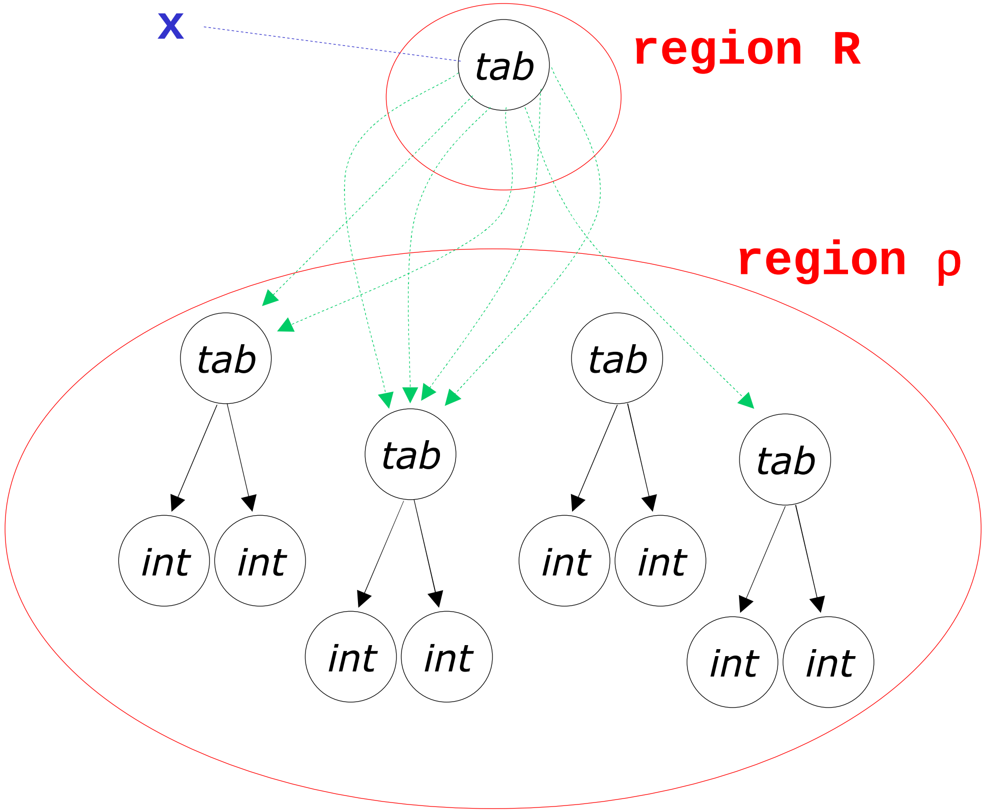
```
x : [R]
{R : array [ρ]}
{ρ*: array int}
```

Type in translation:

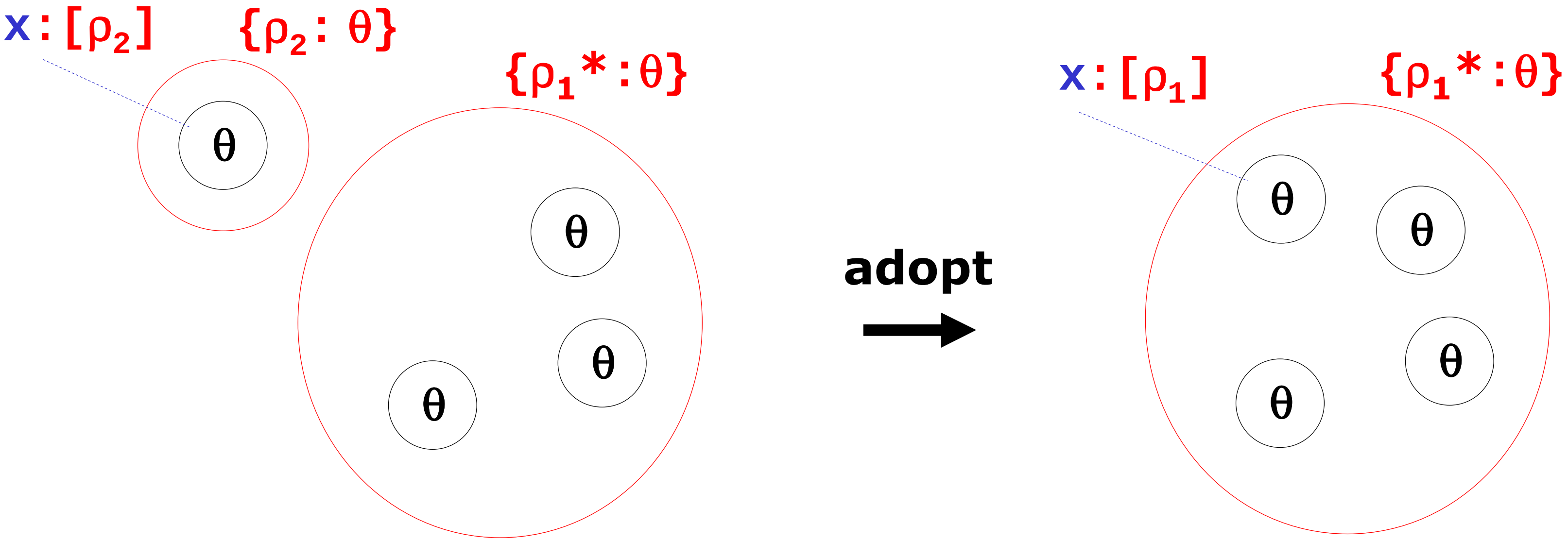
R : array^F key

ρ : map key (array^F int)

Corresponding memory graph:



Adoption



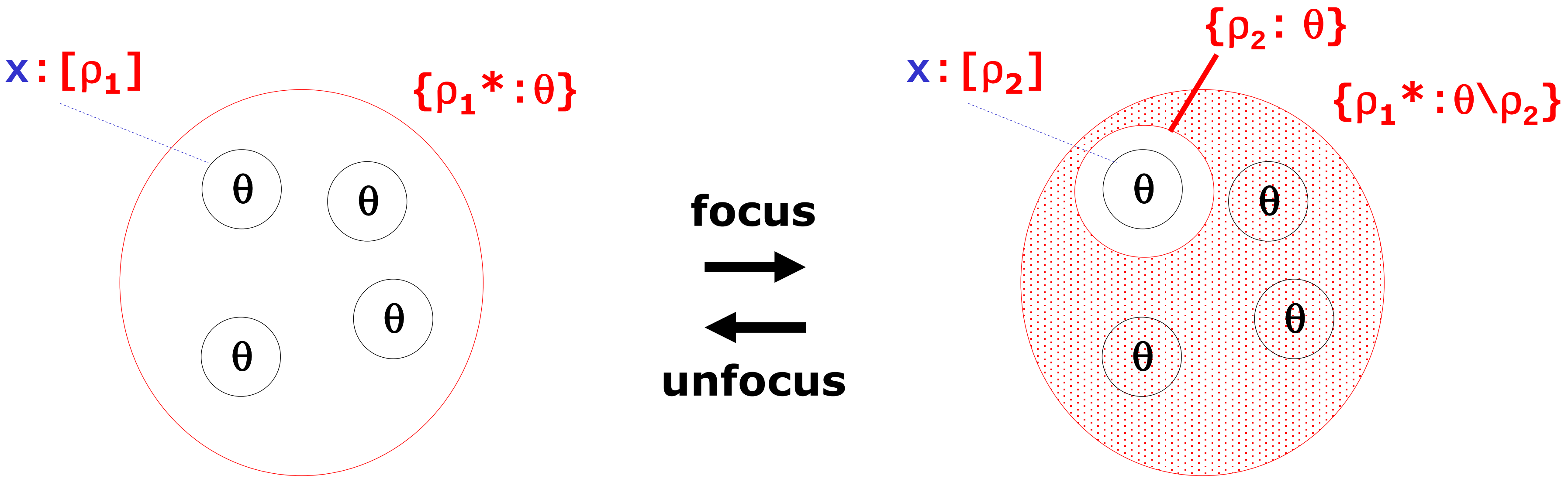
$$\boxed{\text{ADOPT} \quad : \quad \{\rho_1{}^{\star} : \theta\}\{\rho_2 : \theta\}\,[\rho_2] \leq \{\rho_1{}^{\star} : \theta\}\,[\rho_1]}$$

Translation:

$$\boxed{\lambda(h, x, \mathbb{1}).\,\mathrm{let}\, k = \mathrm{map\_fresh}\, h\, \mathrm{in}\, (\mathrm{map\_add}\,(h, k, x), k)}$$

the coercion function corresponding to the subtyping rule

Focus and Unfocus



$$\begin{array}{lcl} \textsc{Focus-Rgn} & : & \{\rho_1{}^\star : \theta\}\,[\rho_1] \leq \exists \rho_2.\{\rho_1{}^\star : \theta \setminus \rho_2\}\{\rho_2 : \theta\}\,[\rho_2] \\ \textsc{Unfocus-Rgn} & : & \{\rho_1{}^\star : \theta \setminus \rho_2\}\{\rho_2 : \theta\} \leq \{\rho_1{}^\star : \theta\} \end{array}$$

Focus is implemented with map_get and unfocus with map_set.

Summary of the Key Ideas

The memory graph is partitioned into regions.

– Singleton region (1 item) $\{\rho:\theta\}$ → a value

– Group region ($n \geq 0$ items) $\{\rho^*:\theta\}$ → a map

An item from region ρ admits type $[\rho]$ → a key

A logical operation to transform regions → a coercion

Capabilities and Types

Capabilities:

$$C \quad := \quad \overbrace{\emptyset}^{\text{empty}} \;|\; \overbrace{C_1 \wedge C_2}^{\text{pair}} \;|\; \overbrace{\{\rho : \theta\}}^{\text{singleton}} \;|\; \overbrace{\{\rho^\star : \theta\}}^{\text{group}} \;|\; \overbrace{\{{\rho_1}^\star : \theta \setminus \rho_2\}}^{\text{suspended}}$$

Value types:

$$\tau \quad := \quad \text{unit} \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \mid \sigma_1 \rightarrow \sigma_2 \mid \overbrace{[\rho]}^{\text{"at-rho"}}$$

σ is a pair (C, τ)

Memory types:

$$\theta \quad := \quad \text{unit} \mid \theta_1 + \theta_2 \mid \theta_1 \times \theta_2 \mid \sigma_1 \rightarrow \sigma_2 \mid [\rho] \mid \overbrace{\text{ref } \theta}^{\text{references}}$$

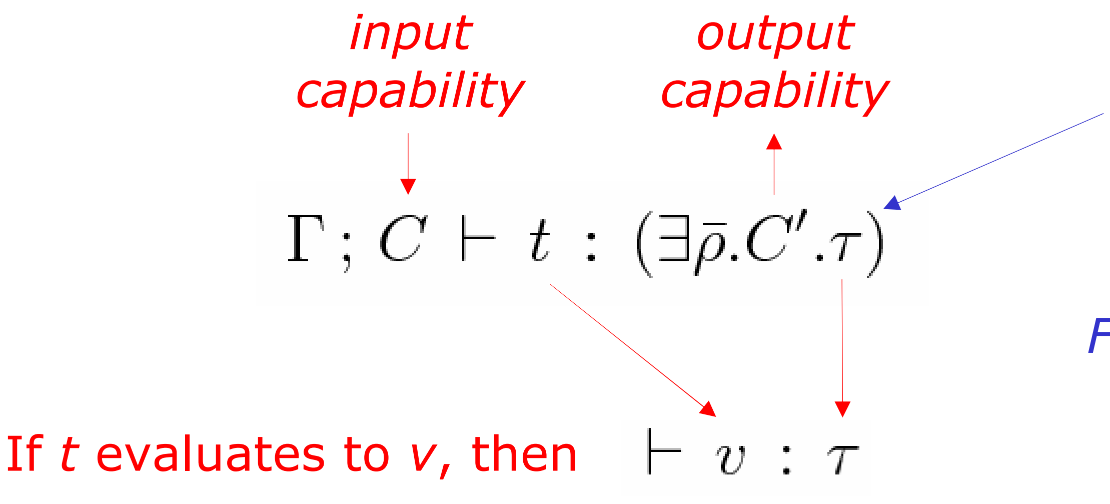
Typing Judgements

Typing of values:

$$\underbrace{\Gamma}_{x : \tau} \vdash v : \tau$$

A variable must have a value type: it is ultimately substituted by a value

Typing of terms:



input capability

output capability

$$\Gamma ; C \vdash t : (\exists \bar{\rho}.C'.\tau)$$

If *t* evaluates to *v*, then $\vdash v : \tau$

Call this pattern a "computation type" and write it $\boldsymbol{\sigma}$

Functions have a type of the form $\sigma_1 \to \sigma_2$

Typing Rules

Typing of values: $\Gamma \vdash v : \tau$ Typing of terms: $\Gamma ; C \vdash t : \sigma$

Value, viewed as a term:

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

Abstraction:

$$\frac{(\Gamma, x : \tau) ; C \vdash t : \sigma}{\Gamma \vdash (\lambda x.\, t) : (\exists \bar{\rho}. C. \tau) \rightarrow \sigma}$$

Application:

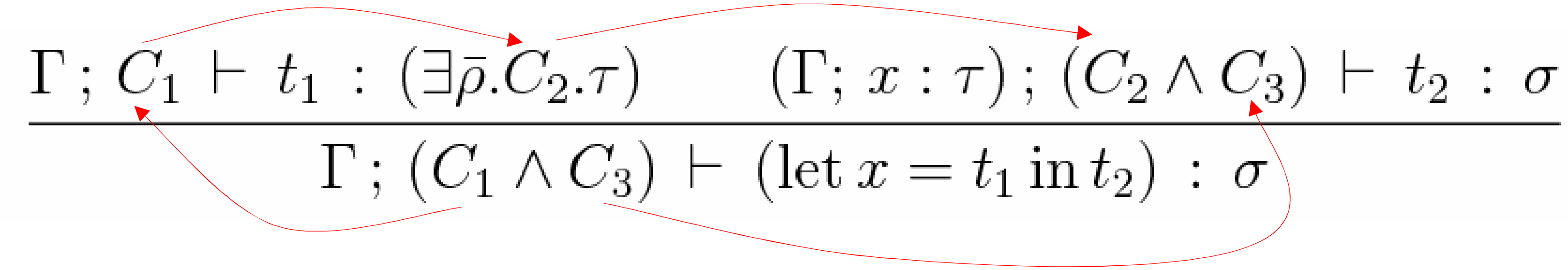
$$\frac{\Gamma \vdash v : (\sigma_1 \rightarrow \sigma_2) \qquad \Gamma ; C \vdash t : \sigma_1}{\Gamma ; C \vdash (v\ t) : \sigma_2}$$

The Frame Typing Rule

"Frame" rule:

$$\frac{\Gamma\,;\,C_2 \vdash t : \sigma}{\Gamma\,;\,(C_1 \wedge C_2) \vdash t : (C_1 \wedge \sigma)}$$

"Let" rule, combining frame (derivable):



$$\frac{\Gamma\,;\,C_1 \vdash t_1 : (\exists\bar{\rho}.C_2.\tau) \qquad (\Gamma;\,x:\tau)\,;\,(C_2 \wedge C_3) \vdash t_2 : \sigma}{\Gamma\,;\,(C_1 \wedge C_3) \vdash (\mathrm{let}\;x = t_1\;\mathrm{in}\;t_2) : \sigma}$$

Translation Judgements

Translation of values: $\Gamma \vdash v : \tau \rhd w$

Translation of terms: $\Gamma; C \rhd c \vdash t : \sigma \rhd u$

c translates C

u translates t:σ

Abstraction:

y is the translation of capability *C*

$$\frac{(\Gamma, x : \tau); C\ \underline{\rhd y} \vdash t : \sigma\ \underline{\rhd u}}{\Gamma \vdash (\lambda x.t) : (\exists\bar{\rho}.C.\tau) \to \sigma\ \underline{\rhd (\lambda(y, x).u)}}$$

Subtyping Rules

Weaken result type:

$$\frac{\Gamma\,;C \vdash t : \sigma_1 \qquad \sigma_1 \le \sigma_2}{\Gamma\,;C \vdash t : \sigma_2}$$

Strengthen input capability:

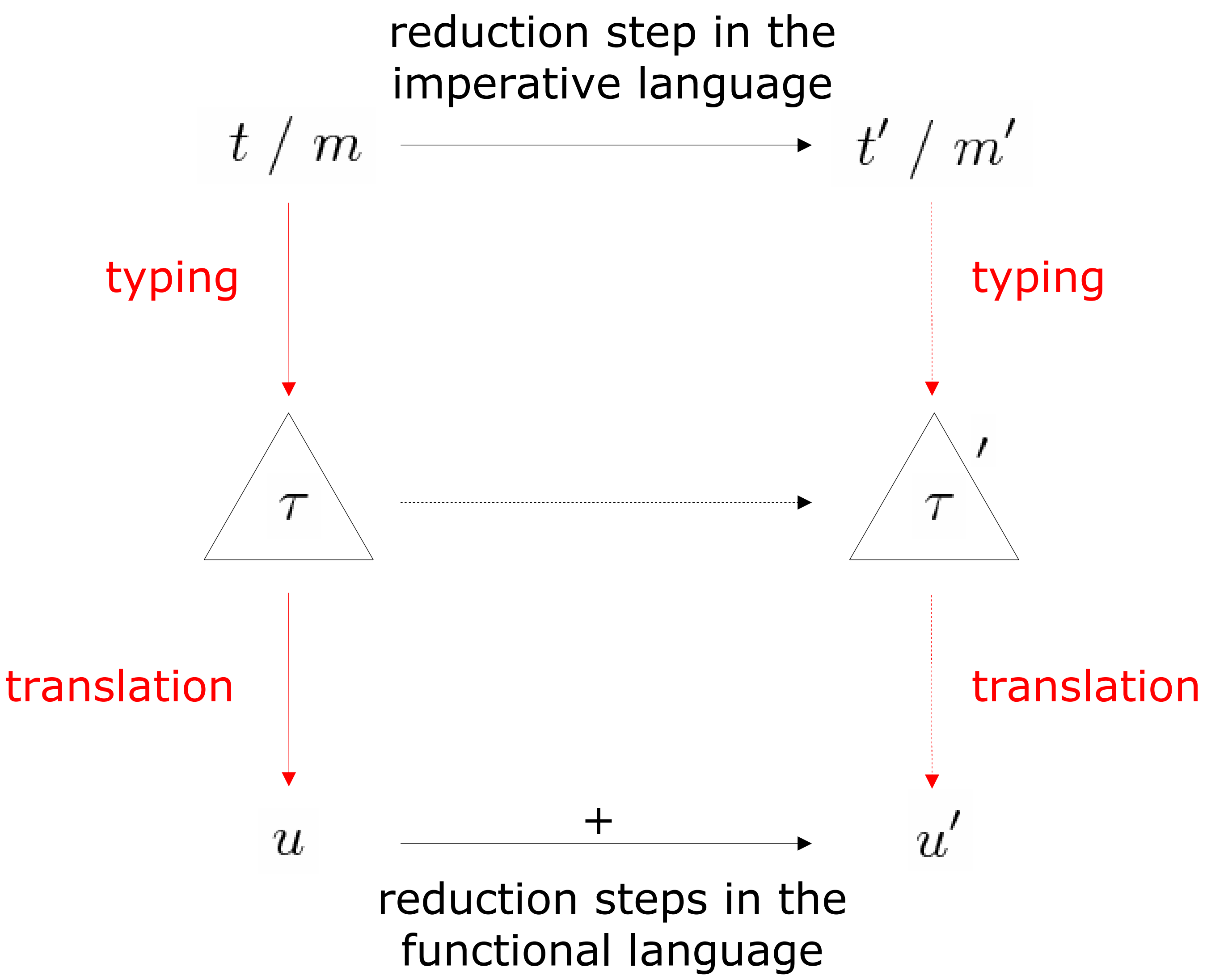
$$\frac{\Gamma\,;C_2 \vdash t : \sigma \qquad C_1 \le C_2}{\Gamma\,;C_1 \vdash t : \sigma}$$

Associated translations:

$$\frac{\Gamma; C \ \rhd c \vdash t : \sigma_1 \ \underline{\rhd\ u} \qquad \sigma_1 \le \sigma_2 \rhd w}{\Gamma; C \ \rhd c \vdash t : \sigma_2 \ \underline{\rhd\ (w\ u)}}$$

$$\frac{\Gamma; C_2 \ \underline{\rhd (w\ c)} \vdash t : \sigma \rhd u \qquad C_1 \le C_2 \rhd w}{\Gamma; C_1 \ \underline{\rhd c} \vdash t : \sigma \rhd u}$$

Simulation Diagram



$$\begin{tikzcd}
t \,/\, m \arrow[r, "\text{reduction step in the imperative language}"] \arrow[d, "\text{typing}"'] & t' \,/\, m' \arrow[d, dotted, "\text{typing}"] \\
\tau \arrow[r, dotted] \arrow[d, "\text{translation}"'] & \tau' \arrow[d, dotted, "\text{translation}"] \\
u \arrow[r, "+", "\text{reduction steps in the functional language}"'] & u'
\end{tikzcd}$$

Related Work

Line of work on regions and capabilities:

94 – Tofte & Talpin: allocation in a stack of regions

99 – Calculus of Capabilities: capability = right to deallocate regions

00 – Alias Types: types in capabilities on singleton regions

02 – Adoption & Focus: group regions, adoption and focus operations

05 – Boyland: per-field adoption

More related work:

- Separation Logic, Stateful Views: separating conjunction, frame
- Monads, Effects: static control of access to regions (less expr.)
- Monadic Translation, Why tool: does not support aliasing

Conclusions

Our contribution: validation of the concept of

"translation of capabilities"

- merge and extend earlier works on calculi of capabilities,
- introduce a functional translation, directed by typing derivations.

On-going work:

- support for arrays, including pointer arithmetics,
- support operations such as fusion and splitting of regions,
- add some type inference to diminish the need for annotations,
- lift a logic for reasoning on functional programs, and become able to state properties about imperative programs directly.

Thanks!