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
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:

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

Typing of values:

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

Typing of primitives:

```latex
def
ref : \tau \rightarrow \exists \rho.\{\rho : \text{ref } \tau\} [\rho]
def
get : \{\rho : \text{ref } \tau\} [\rho] \rightarrow \{\rho : \text{ref } \tau\} \tau
def
set : \{\rho : \text{ref } \tau_1\} ([\rho] \times \tau_2) \rightarrow \{\rho : \text{ref } \tau_2\} \text{unit}
```

Typing with capabilities:

```ocaml
let \{R: \text{ref } \text{int}\} x = ref 7
let \{R: \text{ref } \text{int}\} y = get \{R: \text{ref } \text{int}\} x
let \{R: \text{ref } \text{char}\} _ = set \{R: \text{ref } \text{int}\} (x,'c')
```
Simple Reference – Translation

Typing with capabilities:

```plaintext
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:

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

```
set : \{\rho : ref \tau_1\} ([\rho] \times \tau_2) \rightarrow \{\rho : ref \tau_2\} unit
```

After some reductions:

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

Recall the imperative code:

```plaintext
let x = ref 7
let y = get x
let _ = set (x,'c')
```
Reference with Linear Contents

\( \tau \) ranges over non-linear "value types". Thus, "get" is restricted.

\[
\text{get} : \{\rho : \text{ref } \tau\} [\rho] \rightarrow \{\rho : \text{ref } \tau\} \tau
\]

This restriction is relieved through the "open" operation:

\[
\{\rho_1 : \text{ref ref int}\} \quad \text{open} \quad \{\rho_1 : \text{ref } [\rho_2]\}
\]

\[
\{\rho_2 : \text{ref int}\}
\]

\[
\text{OPEN-REF} : \{\rho_1 : \text{ref } \theta\} \equiv \exists \rho_2. \{\rho_1 : \text{ref } [\rho_2]\} \land \{\rho_2 : \theta\}
\]

\( \theta \) ranges over linear "memory types"
Matrices as 2D-arrays

ML type:

\[ x : \text{array (array int)} \]

In our system:

\[ x : [R] \]
\[ \{R : \text{array (array int)}\} \]

Type in translation:

\[ R : \text{array}^F (\text{array}^F \text{int}) \]
Matrices with Aliasable Rows

ML type:

\[ x : \text{array \ (array \ int)} \]

In our system:

\[ x : [R] \]
\[ \{ R : \text{array \ [\rho]} \} \]
\[ \{ \rho^* : \text{array \ int} \} \]

Type in translation:

\[ R : \text{array}^F \text{ key} \]
\[ \rho : \text{map} \text{ key \ (array}^F \text{ int)} \]
Adoption

\[
\begin{align*}
\lambda(h, x, \mathbb{1}). & \text{ let } k = \text{map\_fresh } h \text{ in } (\text{map\_add } (h, k, x), k) \\
\text{Translation: } & \text{ the coercion function corresponding to the subtyping rule}
\end{align*}
\]
Focus and Unfocus

\[ x: [\rho_1] \rightarrow \{\rho_1^*: \theta\} \]

Focus is implemented with \texttt{map_get} and unfocus with \texttt{map_set}.

\[
\begin{align*}
\text{FOCUS-RGN} & : \{\rho_1^*: \theta\} [\rho_1] \leq \exists \rho_2. \{\rho_1^*: \theta \setminus \rho_2\}\{\rho_2 : \theta\} [\rho_2] \\
\text{UNFOCUS-RGN} & : \{\rho_1^*: \theta \setminus \rho_2\}\{\rho_2 : \theta\} \leq \{\rho_1^*: \theta\}
\end{align*}
\]
Summary of the Key Ideas

The memory graph is partitioned into regions.

- Singleton region (1 item) \( \{\rho:\theta\} \rightarrow \text{a value} \)
- Group region (n ≥ 0 items) \( \{\rho^*:\theta\} \rightarrow \text{a map} \)

An item from region \( \rho \) admits type \([\rho]\) \rightarrow \text{a key} \)

A logical operation to transform regions \rightarrow \text{a coercion} \)
Capabilities and Types

Capabilities:

\[ C ::= \emptyset | C_1 \land C_2 | \{ \rho : \theta \} | \{ \rho^* : \theta \} | \{ \rho_1^* : \theta \setminus \rho_2 \} \]

Value types:

\[ \tau ::= \text{unit} | \tau_1 + \tau_2 | \tau_1 \times \tau_2 | \sigma_1 \rightarrow \sigma_2 | [\rho] \]

Memory types:

\[ \theta ::= \text{unit} | \theta_1 + \theta_2 | \theta_1 \times \theta_2 | \sigma_1 \rightarrow \sigma_2 | [\rho] | \text{ref } \theta \]
Typing Judgements

Typing of values:

\[
\Gamma \vdash v : \tau
\]

\(x : \tau\)

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

Typing of terms:

input capability

\[
\Gamma ; C \vdash t : (\exists \rho.C'.\tau)
\]

output capability

If \(t\) evaluates to \(v\), then

\[
\Gamma \vdash v : \tau
\]

Call this pattern a "computation type" and write it \(\sigma\)

Functions have a type of the form \(\sigma_1 \rightarrow \sigma_2\)
Typing Rules

Typing of values: $\Gamma \vdash v : \tau$

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

Value, viewed as a term:

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

Abstraction:

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

Application:

$$\frac{\Gamma \vdash v : (\sigma_1 \rightarrow \sigma_2) \quad \Gamma ; C \vdash t : \sigma_1}{\Gamma ; C \vdash (v \ t) : \sigma_2}$$
The Frame Typing Rule

"Frame" rule:

\[ \frac{\Gamma; C_2 \vdash t : \sigma}{\Gamma; (C_1 \land C_2) \vdash t : (C_1 \land \sigma)} \]

"Let" rule, combining frame (derivable):

\[ \frac{\Gamma; C_1 \vdash t_1 : (\exists \rho. C_2, \tau) \quad (\Gamma; x : \tau); (C_2 \land C_3) \vdash t_2 : \sigma}{\Gamma; (C_1 \land C_3) \vdash (\text{let } x = t_1 \text{ in } t_2) : \sigma} \]
Translation Judgements

Translation of values: \[ \Gamma \vdash v : \tau \triangleright w \]

Translation of terms: \[ \Gamma ; C \triangleright c \vdash t : \sigma \triangleright u \]

c translates C \hspace{1cm} u translates t : \sigma

Abstraction:

\[
\frac{\left( \Gamma , \ x : \tau \right) ; \ C \triangleright y \vdash t : \sigma \triangleright u}{\Gamma \vdash (\lambda x. t) : (\exists \rho. C. \tau) \rightarrow \sigma \triangleright (\lambda(y, x). u)}
\]

y is the translation of capability C
Subtyping Rules

Weaken result type:

\[ \Gamma ; C \vdash t : \sigma_1 \quad \sigma_1 \leq \sigma_2 \]
\[ \Gamma ; C \vdash t : \sigma_2 \]

Strengthen input capability:

\[ \Gamma ; C_2 \vdash t : \sigma \quad C_1 \leq C_2 \]
\[ \Gamma ; C_1 \vdash t : \sigma \]

Associated translations:

\[ \Gamma ; C \triangleright c \vdash t : \sigma_1 \triangleright u \quad \sigma_1 \leq \sigma_2 \triangleright w \]
\[ \Gamma ; C \triangleright c \vdash t : \sigma_2 \triangleright (w u) \]

\[ \Gamma ; C_2 \triangleright (w c) \vdash t : \sigma \triangleright u \quad C_1 \leq C_2 \triangleright w \]
\[ \Gamma ; C_1 \triangleright c \vdash t : \sigma \triangleright u \]
Simulation Diagram

\[ t / m \rightarrow t' / m' \]

reduction step in the imperative language

typing

\[ \tau \rightarrow \tau' \]

reduction steps in the functional language

\[ u \rightarrow u' \]

translation

typing
Related Work

Line of work on regions and capabilities:

- Tofte & Talpin: allocation in a stack of regions
- Calculus of Capabilities: capability = right to deallocate regions
- Alias Types: types in capabilities on singleton regions
- Adoption & Focus: group regions, adoption and focus operations
- 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!