Functional Translation of a Calculus of Capabilities

Arthur Charguéraud

Joint work with François Pottier
INRIA
Separation in Data Structures

L1: odd values  L2: sorted

L1: odd values  L2: sorted

→ A **type system** able to capture disjointness of data structures
# Extending ML with Separation

<table>
<thead>
<tr>
<th>Technical starting point</th>
<th>⇒</th>
<th>System F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Materialization of ownership</td>
<td>⇒</td>
<td>Capability calculi</td>
</tr>
<tr>
<td>Description of disjointness</td>
<td>⇒</td>
<td>Separation Logic</td>
</tr>
<tr>
<td>Exclusivity of ownership</td>
<td>⇒</td>
<td>Linear Logic</td>
</tr>
<tr>
<td>Delimiting the scope of effects</td>
<td>⇒</td>
<td>Effects type systems</td>
</tr>
<tr>
<td>Fine-grained control of aliasing</td>
<td>⇒</td>
<td>Alias Types</td>
</tr>
<tr>
<td>Describing maybe-aliased data</td>
<td>⇒</td>
<td>Region calculi</td>
</tr>
</tbody>
</table>

→ A combination of many ideas into a single type system that targets a high-level programming language
Contributions

1) A type system controlling side-effects more accurately than ML

2) A fine-grained translation of typed imperative programs into a purely functional language
Capabilities

**Capability**: a static entity used to materialize ownership. Reading or writing a reference requires the capability on this ref.

Type of the function "get" that reads a reference:

- **in ML**:
  \[
  \forall \tau. \ (\text{ref } \tau) \rightarrow \tau
  \]

- **here**:
  \[
  \forall \tau. \ (\text{ref } \tau) \ {\cdot} \rightarrow \tau \ {\cdot}
  \]

  \[
  \forall \tau \sigma. \ (\text{ref } \tau)[\sigma] \ {\sigma} \rightarrow \tau \ {\sigma}
  \]

  \[
  \forall \tau \sigma. \ [\sigma] \ {\sigma : \text{ref } \tau} \rightarrow \tau \ {\sigma : \text{ref } \tau}
  \]

"at-sigma" singleton type for the location

the capability for the corresponding location

Ref: *Alias Types*, Smith, Walker, Morrisset, *ESOP'00*

Ref: *Linear Language with Locations*, Morrisett, Ahmed, Fluet, *TLCA'05*
Flow of Capabilities

A set of capabilities is available at each point in the program.

Skeleton of example:

```f
let f x y =
  ...
  let z = g x in
  ...
  z + y

input capabilities C1 and C2

call to g consumes C1

and produces C3

finally C2 and C3 are returned
```

Capabilities are treated **linearly**: they cannot be duplicated.

A **frame rule** is used to work locally on a subset of capabilities.

Ref: *Calculus of Capabilities*, Crary, Walker, Morrisset, *POPL'99*
Life-cycle of Capabilities

Type of the function "ref" that allocates a reference:

in ML: \( \tau \rightarrow (\text{ref } \tau) \)

here: \( \tau \rightarrow \exists \sigma. [\sigma] \{\sigma:\text{ref } \tau\} \)

Type of the function "set" that updates a reference:

in ML: \( \tau \rightarrow (\text{ref } \tau) \rightarrow \text{unit} \)

here: \( \tau \rightarrow [\sigma] \{\sigma:\text{ref } \tau_1\} \rightarrow \text{unit} \{\sigma:\text{ref } \tau_2\} \)

strong: \( \tau_2 \rightarrow [\sigma] \{\sigma:\text{ref } \tau_1\} \rightarrow \text{unit} \{\sigma:\text{ref } \tau_2\} \)

Type of the function "free" that de-allocates a reference:

in ML: \( (\text{ref } \tau) \rightarrow \text{unit} \) (unsafe)

here: \( [\sigma] \{\sigma:\text{ref } \tau\} \rightarrow \text{unit} \) (safe)
Invariants on Capabilities

If $l$ is a location, then

in ML: \[ l : \text{ref } \tau \]

here: \[ l : [\sigma] \text{ with capability } \{\sigma : \text{ref } \tau\} \]

Invariants

1) Whenever \{\sigma : \text{ref } \tau\} is available, the store maps a location of type \([\sigma]\) towards a value of type \(\tau\)

2) There can be at most one capability on a given location

3) If \{\sigma : \text{ref } \tau\} is not available, the location of type \([\sigma]\) cannot be accessed
Example with Aliasing

```
let r1 = ref 5           r1 : [σ₁] {σ₁:ref int}
let r2 = ref 7           r2 : [σ₂] {σ₂:ref int}
let r3 = r2             r3 : [σ₂]
let x = get r3          x : int
```

Function "get" is here applied with type

```
[σ₂] {σ₂:ref int} → int {σ₂:ref int}
```
Example with Sharing

let r1 = ref 5
let r2 = ref r1
let r3 = ref r1
let r4 = get r3
let x = get r4

\[
\begin{align*}
\text{r1} & : [\sigma_1] & \{\sigma_1 : \text{ref int}\} \\
\text{r2} & : [\sigma_2] & \{\sigma_2 : \text{ref [\sigma_1]}\} \\
\text{r3} & : [\sigma_3] & \{\sigma_3 : \text{ref [\sigma_1]}\} \\
\text{r4} & : [\sigma_1] \\
\text{x} & : \text{int}
\end{align*}
\]


Building Data Structures

\[
\text{let } r1 = \text{ref } 5 \\
\text{let } r2 = \text{ref } r1
\]

\[
\begin{align*}
  r1 & : [\sigma_1] \\
  r2 & : [\sigma_2] \\
  \text{let } x & = \text{get } r2 \\
  x & : (\text{ref } \text{int})
\end{align*}
\]

\[
\{ \sigma_2 : \text{ref } [\sigma_1] \} \\
\{ \sigma_1 : \text{ref } \text{int} \}
\]

\[
\text{merge} \quad \leftrightarrow \quad \text{split}
\]

\[
\begin{align*}
  \text{get} & : [\sigma] \{ \sigma : \text{ref } \tau \} \rightarrow \tau \{ \sigma : \text{ref } \tau \}
\end{align*}
\]

\(\tau\) stands for a type free of the "ref" constructor

\text{BUG!}
Example: Mutable Binary Tree

tree \( \alpha = \text{ref} (\alpha \times \text{tree} \alpha \times \text{tree} \alpha) \)

L : [\sigma] with capability \{\sigma: \text{tree} \alpha\}

Note: the constructor for leaves has been hidden for simplicity.

\{\sigma: \text{ref} (\alpha \times \text{tree} \alpha \times \text{tree} \alpha)\}

can be traded against

\{\sigma: \text{ref} ([\sigma_1] \times [\sigma_2] \times [\sigma_3])\}

\{\sigma_1 : \alpha\}
\{\sigma_2 : \text{tree} \alpha\}
\{\sigma_3 : \text{tree} \alpha\}
Example: Graph with Pointers

in ML: \[ \text{node } \alpha = \text{ref } (\alpha \times \text{list } (\text{node } \alpha)) \]

here: \[ \text{node } \alpha \rho = \text{ref } (\alpha \times \text{list } [\rho]) \]

Capability on the "group region" \{\rho: \text{node } \alpha \rho\}
as opposed to "singleton regions" of the form \{\sigma: \text{node } \alpha\}

Ref: Adoption & Focus, Fahndrich, DeLine, PLDI'02
Ref: Connecting Effects & Uniqueness with Adoption, Boyland, Retert, POPL'05
Functional Translation

**Goal:** write a purely functional program equivalent to a given imperative program

**Standard monadic translation:** threads a map that represents the state of the store throughout the program

**But:**
- it threads more data than necessary
  - does not take advantage of separation properties
  - is not the identity over the pure fragment
  - does not match what a programmer would code
- the threaded map contains heterogeneous data
  - does not type-check in System F
Translation based on Capabilities

Fact: capabilities describe precisely which pieces of store need to be threaded at each point in the program

Idea: materialize capabilities as runtime values

Translated program:

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

- input the translation of capabilities \( C_1 \) and \( C_2 \)
- call to \( g \) consumes \( C_1 \)
- and produces \( C_3 \)
- finally \( C_2 \) and \( C_3 \) are returned
## Translating Capabilities and Types

<table>
<thead>
<tr>
<th>Source program</th>
<th>Translated program</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Static capability</strong></td>
<td><strong>Type of runtime value</strong></td>
</tr>
<tr>
<td>{σ: \text{ref} ; τ}</td>
<td>(τ)</td>
</tr>
<tr>
<td>{ρ: \text{ref} ; τ}</td>
<td>map key (τ)</td>
</tr>
<tr>
<td><strong>Type of runtime value</strong></td>
<td><strong>Type of runtime value</strong></td>
</tr>
<tr>
<td>[σ]</td>
<td>unit</td>
</tr>
<tr>
<td>[ρ]</td>
<td>key</td>
</tr>
</tbody>
</table>
A Few Examples

**Mutable trees:** represented as functional trees.

**Mutable lists:** the in-place list reversal function is translated to the reverse function for functional lists.

**Tarjan's union-find:** each instance of the union-find graph is represented using a map, each node is represented using a key.

**Landin's knot:** this fixpoint combinator implemented with a reference cell translates to the Y-combinator (which type-checks in System F with recursive types).
Conclusions

On-going work
- Extend the system to a full-blown language
- Augment the expressiveness of operations on group regions
- Set up a partial type-inference engine and implement it

Applications
- More precise types mean better documentation and fewer bugs
- Relaxing the value restriction (restriction now only on types)
- Support for safe deallocation (with runtime support for groups)
- Semi-automatic functional translation of imperative programs
- Should help for reasoning on imperative programs
- Should help for programming concurrent programs
Thanks!