Functional Translation of a Calculus of Capabilities

Arthur Charguéraud

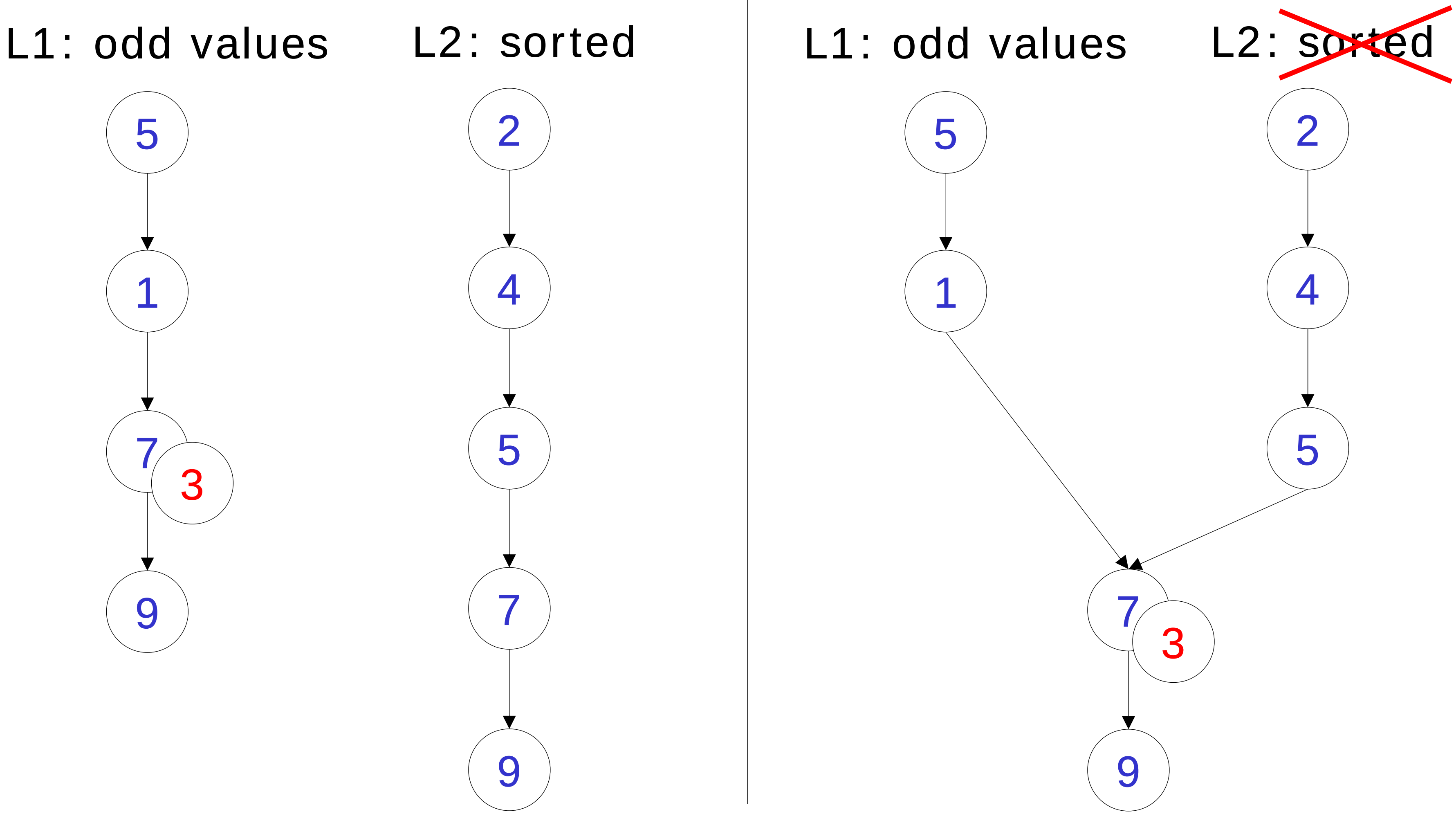
Joint work with François Pottier

INRIA

ICFP'08

Victoria, 2008-09-23

Separation in Data Structures



→ A **type system** able to capture disjointness of data structures

Extending ML with Separation

Technical starting point	$\Rightarrow$	System F
Materialization of ownership	$\Rightarrow$	Capability calculi
Description of disjointness	$\Rightarrow$	Separation Logic
Exclusivity of ownership	$\Rightarrow$	Linear Logic
Delimiting the scope of effects	$\Rightarrow$	Effects type systems
Fine-grained control of aliasing	$\Rightarrow$	Alias Types
Describing maybe-aliased data	$\Rightarrow$	Region calculi

$\rightarrow$ 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: $\forall\tau.\quad (\mathrm{ref}\ \tau) \quad\to\quad \tau$

$\forall\tau.\quad (\mathrm{ref}\ \tau)\ \{\cdot\} \quad\to\quad \tau\ \{\cdot\}$

$\forall\tau\,\sigma.\quad (\mathrm{ref}\ \tau)_{[\sigma]}\ \{\sigma\} \quad\to\quad \tau\ \{\sigma\}$

here: $\forall\tau\,\sigma.\quad [\sigma]\ \{\sigma : \mathrm{ref}\ \tau\} \to \tau\ \{\sigma : \mathrm{ref}\ \tau\}$

"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:

```
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:	τ	$\to$	$(\mathrm{ref}\ \tau)$
here:	τ	$\to$	$\exists\sigma.\ [\sigma]\ \{\sigma{:}\mathrm{ref}\ \tau\}$

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

in ML:	τ	$\to$	$(\mathrm{ref}\ \tau)$	$\to$	unit
here:	τ	$\to$	$[\sigma]\ \{\sigma{:}\mathrm{ref}\ \tau\}$	$\to$	$\mathrm{unit}\ \{\sigma{:}\mathrm{ref}\ \tau\}$
strong:	τ_2	$\to$	$[\sigma]\ \{\sigma{:}\mathrm{ref}\ \tau_1\}$	$\to$	$\mathrm{unit}\ \{\sigma{:}\mathrm{ref}\ \tau_2\}$

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

in ML:	$(\mathrm{ref}\ \tau)$	$\to$	unit	(unsafe)
here:	$[\sigma]\ \{\sigma{:}\mathrm{ref}\ \tau\}$	$\to$	unit	(safe)

Invariants on Capabilities

If ℓ is a location, then

in ML: ℓ : ref τ

here: ℓ : [σ] with capability {σ:ref τ}

Invariants

1) Whenever {σ:ref τ} is available, the store maps a location of type [σ] towards a value of type τ
2) There can be at most one capability on a given location
3) If {σ:ref τ} is not available, the location of type [σ] cannot be accessed

Example with Aliasing

```
let r1 = ref 5

let r2 = ref 7

let r3 = r2

let x = get r3
```

$r1 : [\sigma_1]$ $\{\sigma_1 : \text{ref int}\}$

$r2 : [\sigma_2]$ $\{\sigma_2 : \text{ref int}\}$

$r3 : [\sigma_2]$

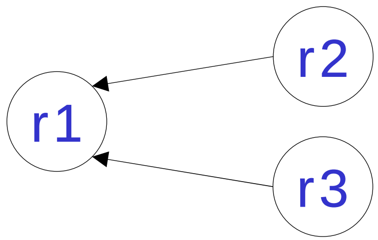
$x : \text{int}$

Function "get" is here applied with type

$$[\sigma_2]\ \{\sigma_2 : \text{ref int}\} \rightarrow \text{int}\ \{\sigma_2 : \text{ref int}\}$$

Example with Sharing

`let r1 = ref 5`	$r1 : [\sigma_1]$	$\{\sigma_1 : \text{ref int}\}$
`let r2 = ref r1`	$r2 : [\sigma_2]$	$\{\sigma_2 : \text{ref } [\sigma_1]\}$
`let r3 = ref r1`	$r3 : [\sigma_3]$	$\{\sigma_3 : \text{ref } [\sigma_1]\}$
`let r4 = get r3`	$r4 : [\sigma_1]$	
`let x = get r4`	$x : \text{int}$	



Building Data Structures

```
let r1 = ref 5

let r2 = ref r1
```

$r1 : [\sigma_1]$

$r2 : [\sigma_2]$

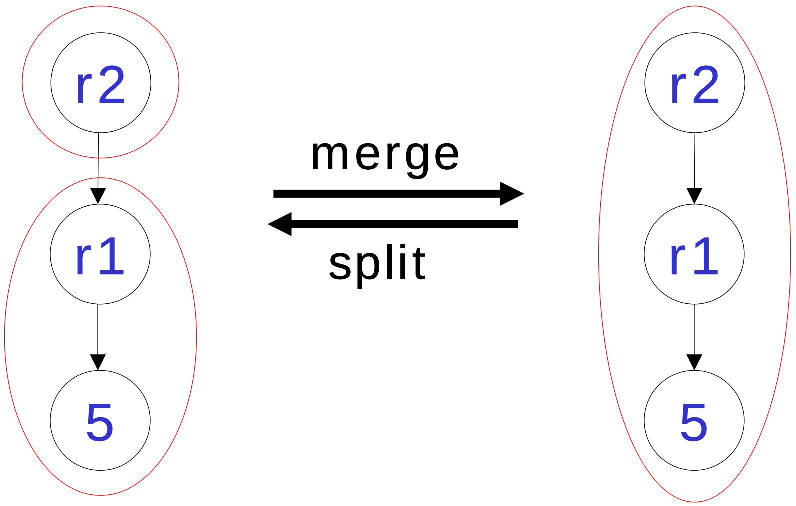
```
let x = get r2
```

$x : (\text{ref int})$

BUG!

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

$\{\sigma_1 : \text{ref int}\}$



$\{\sigma_2 : \text{ref (ref int)}\}$

get : $[\sigma]\ \{\sigma : \text{ref } \tau\} \rightarrow \tau\ \{\sigma : \text{ref } \tau\}$

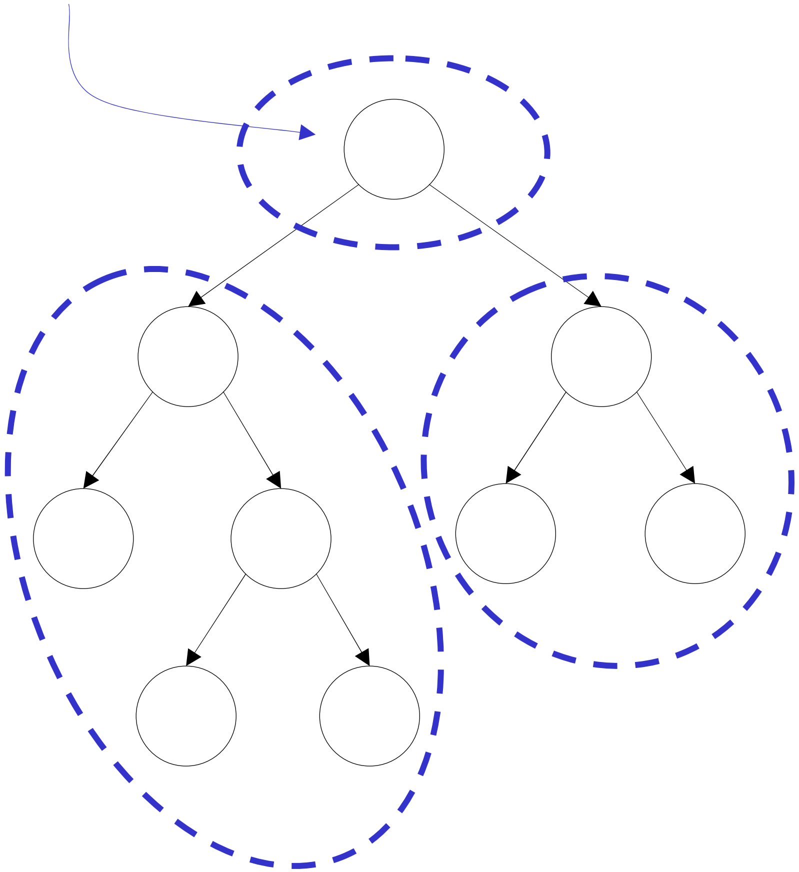
τ stands for a type free of the "ref" constructor

Example: Mutable Binary Tree

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

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

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



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

can be traded against

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

$\{\sigma_1 : \alpha\}$

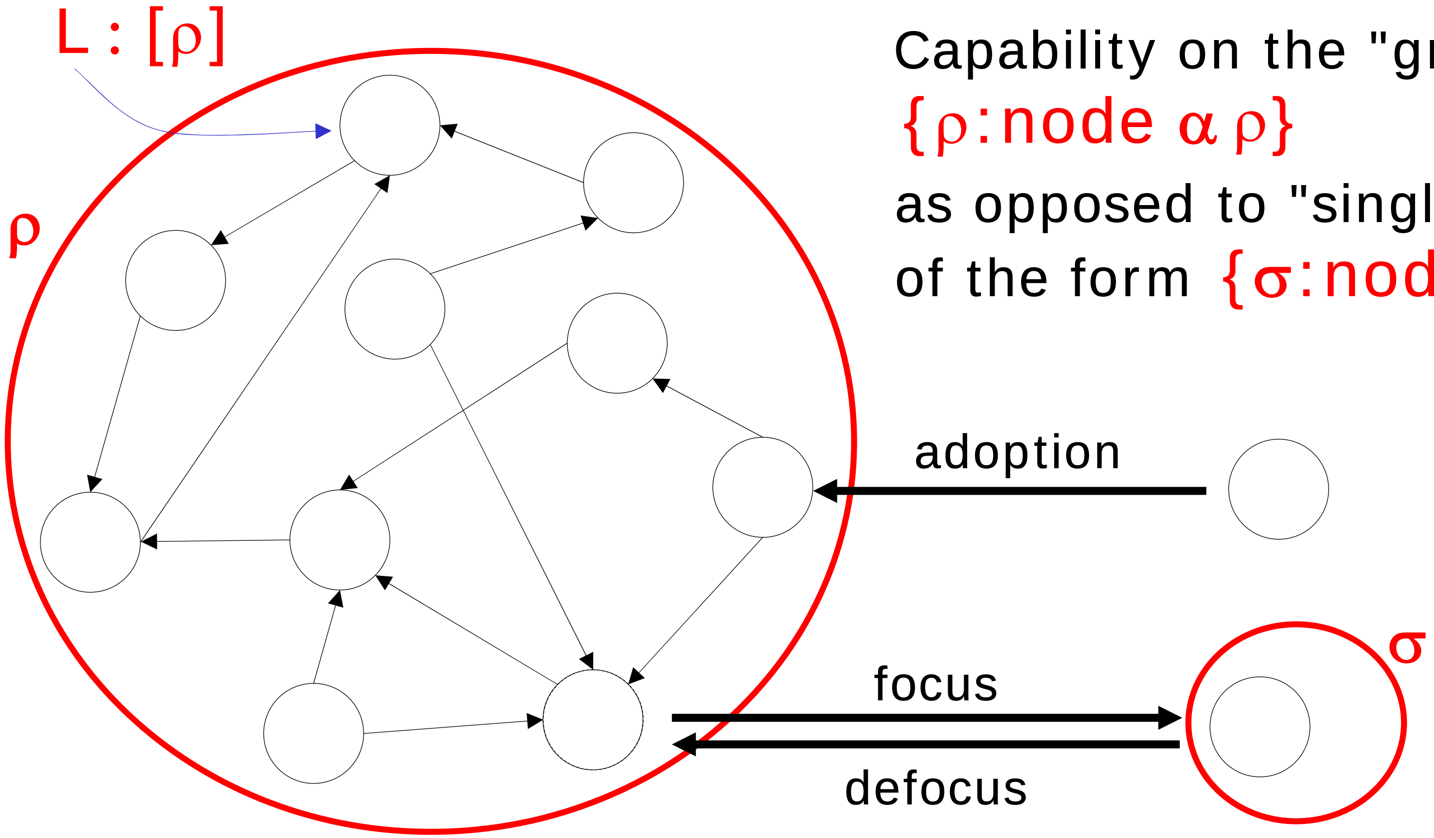
$\{\sigma_2 : \text{tree } \alpha\}$

$\{\sigma_3 : \text{tree } \alpha\}$

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

$\rightarrow$ does not take advantage of separation properties

$\rightarrow$ is not the identity over the pure fragment

$\rightarrow$ does not match what a programmer would code

– the threaded map contains heterogeneous data

$\rightarrow$ 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:

```
let f x y c1 c2 =

  ...

  let z,c3 = g x c1 in

  ...

  z+y,c2,c3
```

input the **translation of** capabilities **C1** and **C2**

call to **g** consumes **C1**

and produces **C3**

finally **C2** and **C3** are returned

Translating Capabilities and Types

Source program	*Translated program*
Static capability	Type of runtime value
$\{\sigma\text{:ref } \tau\}$	τ
$\{\rho\text{:ref } \tau\}$	map key τ
Type of runtime value	Type of runtime value
$[\sigma]$	unit
$[\rho]$	key

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!