# Full abstraction for multi-language systems
# ML plus linear types

**Gabriel Scherer**, Amal Ahmed, Max New

Northeastern University, Boston

May 5, 2017

Section 1

# Full Abstraction for Multi-Language Systems: Introduction

# Multi-language systems

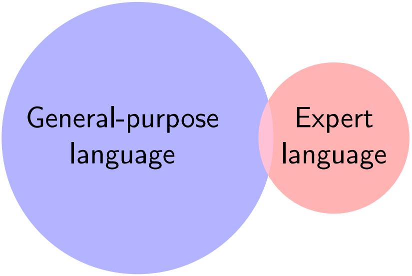
Languages of today tend to evolve into behemoths by piling features up: C++, Scala, GHC Haskell, OCaml...

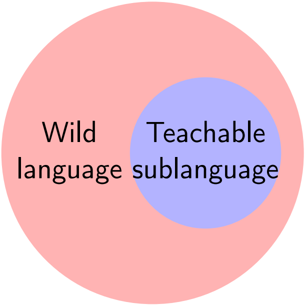
Multi-language systems: several languages working together to cover the feature space. (simpler?)

Multi-language system **design** may include designing new languages for interoperation.

Full abstraction to understand graceful language interoperability.

# Multi-language stories




Graceful interoperation?

Abstraction leaks?

(Several expert languages: not (yet?) in this work)

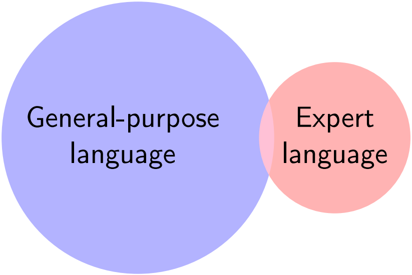
# Full abstraction

$[\![\_]\!] : S \longrightarrow T$ fully abstract:

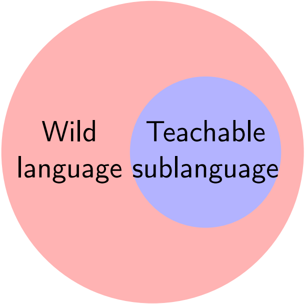
$$a \approx^{ctx} b \implies [\![a]\!] \approx^{ctx} [\![b]\!]$$

Full abstraction preserves (equational) reasoning.

# Full abstraction for multi-language systems




Graceful interoperation: $G \xrightarrow{f.a.} (G + E)$




No abstraction leaks: $T \xrightarrow{f.a.} W$

# Which languages?

ML sweet spot hard to beat,
but ML programmers yearn for language extensions.

ML plus:

- low-level memory, resource tracking, ownership
- effect system
- theorem proving
- ...

In this talk: a first ongoing experiment on **ML** plus **linear types**.

# Our case study

U (Unrestricted): general-purpose ML language
L (Linear): expert linear language.

$$U \xrightarrow{f.a.} (U + L)$$

Proof: by translating $L$ back into $U$ in an inefficient but correct way.

# Our case study

U (Unrestricted): general-purpose ML language
L (Linear): expert linear language.

$$U \xrightarrow{f.a.} (U + L)$$

Proof: by translating $L$ back into $U$ in an inefficient but correct way.

Note: extending $U$ preserves this result.

# Our case study

U (Unrestricted): general-purpose ML language
L (Linear): expert linear language.

$$U \xrightarrow{f.a.} (U + L)$$

Proof: by translating $L$ back into $U$ in an inefficient but correct way.

Note: extending $U$ preserves this result.

Note: $L \longrightarrow (U + L)$ not meant to be fully abstract.
(Not robust to extensions of $U$)

# Section 2

## Case Study: Unrestricted and Linear

# Unrestricted language: syntax

| | | | |
|---|---|---|---|
| *Types* | $\sigma$ | $::=$ | $\alpha \mid \sigma_1 \times \sigma_2 \mid 1 \mid \sigma_1 \to \sigma_2 \mid$ <br> $\sigma_1 + \sigma_2 \mid \mu\alpha.\,\sigma \mid \forall\alpha.\,\sigma$ |
| *Expressions* | $\mathsf{e}$ | $::=$ | $\mathsf{x} \mid$ <br> $\langle \mathsf{e}_1, \mathsf{e}_2 \rangle \mid \pi_1\,\mathsf{e} \mid \pi_2\,\mathsf{e} \mid$ <br> $\langle\rangle \mid \mathsf{e}_1; \mathsf{e}_2 \mid$ <br> $\lambda(\mathsf{x}:\sigma).\,\mathsf{e} \mid \mathsf{e}_1\ \mathsf{e}_2 \mid$ <br> $\mathsf{inj}_1\,\mathsf{e} \mid \mathsf{inj}_2\,\mathsf{e} \mid \mathsf{case}\,\mathsf{e}'\,\mathsf{of}\,\mathsf{x}_1.\,\mathsf{e}_1 \mid \mathsf{x}_2.\,\mathsf{e}_2 \mid$ <br> $\mathsf{fold}_{\mu\alpha.\sigma}\,\mathsf{e} \mid \mathsf{unfold}\,\mathsf{e} \mid$ <br> $\Lambda\alpha.\,\mathsf{e} \mid \mathsf{e}\,[\sigma]$ |
| *Typing contexts* | $\Gamma, \Gamma'$ | $::=$ | $\cdot \mid \Gamma, \mathsf{x}:\sigma \mid \Gamma, \alpha$ |

# Linear types: introduction

Resource tracking, unique ownership.

$$\sigma \qquad\qquad !\sigma \qquad\qquad \Gamma \qquad\qquad !\Gamma$$

$$\Gamma \vdash_{\mathsf{l}} \mathsf{e} : \sigma$$

We own $\mathsf{e}$ at type $\sigma$ (duplicable or not), $\mathsf{e}$ owns the resources in $\Gamma$.

$$\begin{aligned}\sigma ::= {} & \sigma_1 \otimes \sigma_2 \mid 1 \mid \sigma_1 \multimap \sigma_2 \mid \\ & \sigma_1 \oplus \sigma_2 \mid \mu\alpha.\,\sigma \mid \alpha \mid \\ & !\sigma \mid \\ & \mathsf{Box}\ \mathsf{b}\ \sigma\end{aligned}$$

# Linear types: base

A **simple** but **useful** language with linear types.

$$\frac{}{!\Gamma, x{:}\sigma \vdash_l x : \sigma} \qquad \frac{}{!\Gamma \vdash_l \langle\rangle : 1} \qquad \frac{\Gamma \vdash_l e : 1 \qquad \Gamma' \vdash_l e' : \sigma}{\Gamma \boxplus \Gamma' \vdash_l e; e' : \sigma}$$

$$\frac{\Gamma_1 \vdash_l e_1 : \sigma_1 \qquad \Gamma_2 \vdash_l e_2 : \sigma_2}{\Gamma_1 \boxplus \Gamma_2 \vdash_l \langle e_1, e_2 \rangle : \sigma_1 \otimes \sigma_2} \qquad \frac{\Gamma \vdash_l e : \sigma_1 \otimes \sigma_2 \qquad \Gamma', x_1{:}\sigma_1, x_2{:}\sigma_2 \vdash_l e' : \sigma}{\Gamma \boxplus \Gamma' \vdash_l \mathsf{let}\ \langle x_1, x_2 \rangle = e\ \mathsf{in}\ e' : \sigma}$$

$$\frac{\Gamma, x{:}\sigma \vdash_l e : \sigma'}{\Gamma \vdash_l \lambda(x{:}\sigma).\, e : \sigma \multimap \sigma'} \qquad \frac{\Gamma \vdash_l e : \sigma' \multimap \sigma \qquad \Gamma' \vdash_l e' : \sigma'}{\Gamma \boxplus \Gamma' \vdash_l e\ e' : \sigma}$$

$$\frac{\Gamma \vdash_l e : \sigma_i}{\Gamma \vdash_l \mathsf{inj_i}\ e : \sigma_1 \oplus \sigma_2} \qquad \frac{\Gamma \vdash_l e : \sigma_1 \oplus \sigma_2 \qquad (\Gamma', x_i : \sigma_i \vdash_l e_i : \sigma)_{i \in \{1,2\}}}{\Gamma \boxplus \Gamma' \vdash_l \mathsf{case}\ e\ \mathsf{of}\ x_1.\, e_1 \mid x_2.\, e_2 : \sigma}$$

$$\frac{!\Gamma \vdash_l e : \sigma}{!\Gamma \vdash_l \mathsf{share}\ e : !\sigma} \qquad \frac{\Gamma \vdash_l e : !\sigma}{\Gamma \vdash_l \mathsf{copy}^{\sigma}\ e : \sigma}$$

$$\mu\alpha.\,\sigma \quad \overset{\mathsf{unfold}}{\multimap}{}^{13} \quad \sigma[\mu\alpha.\,\sigma/\alpha]$$

# Applications

Protocol with resource handling requirements.

"This file descriptor must be closed"

$$\begin{array}{lcl} \mathsf{open} & : & !(![\mathsf{Path}] \multimap \mathsf{Handle}) \\ \mathsf{line} & : & !(\mathsf{Handle} \multimap (\mathsf{Handle} \oplus (![\mathsf{String}] \otimes \mathsf{Handle}))) \\ \mathsf{close} & : & !(\mathsf{Handle} \multimap 1) \end{array}$$

(details about the boundaries come later)

Typestate.

(details about the boundaries come later)

$$\begin{array}{lcl} \mathsf{open} & : & !(![\mathsf{Path}] \multimap \mathsf{Handle}) \\ \mathsf{line} & : & !(\mathsf{Handle} \multimap (\mathsf{Handle} \oplus (![\mathsf{String}] \otimes \mathsf{Handle}))) \\ \mathsf{close} & : & !(\mathsf{Handle} \multimap 1) \end{array}$$

```
let concat_lines path : String = UL(
  loop (open LU(path)) LU(Nil)
  where rec loop handle LU(acc : List String) =
    match line handle with
    | EOF handle ->
      close handle; LU(rev_concat "\n" acc)
    | Next line handle ->
      loop handle LU(Cons UL(line) acc))
```

(U values are passed back and forth, never inspected)

# Linear types: linear locations

Box 1 $\sigma$: full cell

Box 0 : empty cell

$$1 \underset{\text{free}}{\overset{\text{new}}{\rightleftarrows}} \text{Box } 0 \qquad\qquad \text{Box } 1\ \sigma \underset{\text{box}}{\overset{\text{unbox}}{\rightleftarrows}} (\text{Box } 0) \otimes \sigma$$

Applications: in-place reuse of memory cells.

# List reversal

```
type LList a = μt. 1 ⊕ Box 1 (a ⊗ t)

val reverse : LList a ⊸ LList a
let reverse list = loop (inl ()) list
  where rec loop tail = function
  | inl () → tail
  | inr cell →
    let (l, (x, xs)) = unbox cell in
    let cell = box (l, (x, tail)) in
    loop (inr cell) xs
```

# List reversal (sweet)

```
type LList a = μt. 1 ⊕ Box 1 (a ⊗ t)
pattern Nil = inl ()
pattern Cons l x xs = inr (box (l, (x, xs)))

val reverse : LList a ⊸ LList a
let reverse list = loop Nil list
  where rec loop tail = function
  | Nil → tail
  | Cons l x xs → loop (Cons l x tail) xs

type List a = μt. 1 + (a × t)
let reverse list = UL(share (reverse (copy (LU(list)))))
```

(U values are created from the L side from a compatible type)

```
let partition p li = partition_aux p (Nil, Nil) li
partition_aux p (yes, no) = function
| Nil -> (yes, no)
| Cons l x xs ->
  let (yes, no) =
    if copy p x then (Cons l x yes, no) else (yes, Cons l x no)
  in partition_aux p (yes, no) xs

let lin_quicksort li = quicksort_aux li Nil
let quicksort_aux li acc = match li with
| Nil -> acc
| Cons l head li ->
  let p = share (fun x -> x < head) in
  let (below, above) = partition p li in
  quicksort_aux below (Cons l head (quicksort_aux above acc))

quicksort li UL(li) = UL(share (lin_quicksort (copy li)))
```

# Interaction: lump

*Types* $\sigma \mid \sigma$

$$\begin{aligned} &\sigma \\ &\sigma \quad +::= \cdots \mid [\sigma] \end{aligned}$$

*Expressions* $\mathsf{e} \mid \mathsf{e}$

$$\begin{aligned} &\mathsf{e} \quad +::= \cdots \mid \mathcal{UL}(\mathsf{e}) \\ &\mathsf{e} \quad +::= \cdots \mid \mathcal{LU}(\mathsf{e}) \end{aligned}$$

*Contexts* $\Gamma ::= \cdot \mid \Gamma, \mathsf{x}{:}\sigma \mid \Gamma, \alpha \mid \Gamma, \mathsf{x}{:}\sigma$

$$\frac{!\Gamma \vdash_{\mathsf{lu}} \mathsf{e} : \sigma}{!\Gamma \vdash_{\mathsf{ul}} \mathcal{LU}(\mathsf{e}) : ![\sigma]} \qquad \frac{!\Gamma \vdash_{\mathsf{ul}} \mathsf{e} : ![\sigma]}{!\Gamma \vdash_{\mathsf{lu}} \mathcal{UL}(\mathsf{e}) : \sigma}$$

# Interaction: compatibility

Compatibility relation $\boxed{\vdash_{\mathsf{ul}} \sigma \simeq \sigma}$

$$\frac{}{\vdash_{\mathsf{ul}} 1 \simeq\, !1} \qquad \frac{\vdash_{\mathsf{ul}} \sigma_1 \simeq\, !\sigma_1 \qquad \vdash_{\mathsf{ul}} \sigma_2 \simeq\, !\sigma_2}{\vdash_{\mathsf{ul}} \sigma_1 \times \sigma_2 \simeq\, !(\sigma_1 \otimes \sigma_2)}$$

$$\frac{\vdash_{\mathsf{ul}} \sigma_1 \simeq\, !\sigma_1 \qquad \vdash_{\mathsf{ul}} \sigma_2 \simeq\, !\sigma_2}{\vdash_{\mathsf{ul}} \sigma_1 + \sigma_2 \simeq\, !(\sigma_1 \oplus \sigma_2)} \qquad \frac{\vdash_{\mathsf{ul}} \sigma \simeq\, !\sigma \qquad \vdash_{\mathsf{ul}} \sigma' \simeq\, !\sigma'}{\vdash_{\mathsf{ul}} \sigma \to \sigma' \simeq\, !(!\sigma \multimap\, !\sigma')}$$

$$\frac{}{\vdash_{\mathsf{ul}} \sigma \simeq\, ![\sigma]} \qquad \frac{\vdash_{\mathsf{ul}} \sigma \simeq\, !\sigma}{\vdash_{\mathsf{ul}} \sigma \simeq\, !!\sigma} \qquad \frac{\vdash_{\mathsf{ul}} \sigma \simeq\, !\sigma}{\vdash_{\mathsf{ul}} \sigma \simeq\, !(\mathsf{Box}\ 1\ \sigma)}$$

Interaction primitives and derived constructs:

$$![\sigma] \;\underset{\mathsf{lump}^{\sigma}}{\overset{{}^{\sigma}\mathsf{unlump}}{\rightleftarrows}}\; \sigma \quad \text{when} \quad \vdash_{\mathsf{ul}} \sigma \simeq \sigma$$

$${}^{\sigma}\mathcal{LU}(\mathsf{e}) \stackrel{\text{def}}{=} {}^{\sigma}\mathsf{unlump}\ \mathcal{LU}(\mathsf{e})$$

$$\mathcal{UL}^{\sigma}(\mathsf{e}) \stackrel{\text{def}}{=} \mathcal{UL}(\mathsf{lump}^{\sigma}\ \mathsf{e})$$

# Full abstraction

**Theorem**

The embedding of U into UL is fully abstract.

Proof: by pure interpretation of the linear language into ML.

# Full abstraction

**Theorem**

The embedding of U into UL is fully abstract.

Proof: by pure interpretation of the linear language into ML.

$$
\begin{aligned}
\lceil !\sigma \rceil &\stackrel{\text{def}}{=} \lceil \sigma \rceil \\
\lceil \text{Box } 0\ \sigma \rceil &\stackrel{\text{def}}{=} 1 \\
\lceil \text{Box } 1\ \sigma \rceil &\stackrel{\text{def}}{=} 1 \times \lceil \sigma \rceil \\
\lceil \sigma_1 \otimes \sigma_2 \rceil &\stackrel{\text{def}}{=} \lceil \sigma_1 \rceil \times \lceil \sigma_2 \rceil
\end{aligned}
$$

(Cogent)

# Remark on parametricity

(from Max New)

$$(\Lambda\alpha.\,\lambda(\mathsf{x}:\alpha).\,\mathcal{U}\mathcal{L}^{\alpha}({}^{\alpha}\mathcal{L}\mathcal{U}(\mathsf{x})))\ [\sigma] \quad \overset{\mathsf{U}}{\hookrightarrow}? \quad \lambda(\mathsf{x}:\sigma).\,\mathcal{U}\mathcal{L}^{\sigma}({}^{\sigma}\mathcal{L}\mathcal{U}(\mathsf{x}))$$

Not well-typed!

$$(\Lambda\alpha.\,\lambda(\mathsf{x}:\alpha).\,\mathcal{U}\mathcal{L}^{![\alpha]}({}^{![\alpha]}\mathcal{L}\mathcal{U}(\mathsf{x})))\ [\sigma] \quad \overset{\mathsf{U}}{\hookrightarrow} \quad \lambda(\mathsf{x}:\sigma).\,\mathcal{U}\mathcal{L}^{![\sigma]}({}^{![\sigma]}\mathcal{L}\mathcal{U}(\mathsf{x}))$$

Logical relation (Max New, Nicholas Rioux)

# Questions

## But not the end!

Implementation ?

Implementability? (Cell size.)

Limitation: no separation of pointer and capability.

Does this approach scale to a language usable in practice?
(Polymorphism in L?)
(Without losing its simplicity?)

Your questions.

## Section 3

# How Fully Abstract Can We Go?

I used to think of Full Abstraction as an **ideal** property that would never be reached in practice.

I changed my mind. The statement can be **weakened** to fit many situations, and remains a useful specification.

I will now present some (abstract) examples of this approach.

# Weak Trick 1: restrict the interaction types

The no-interaction multi-language: always fully abstract!

Types restrict interaction: “only integers”, “only ground types”.

Extend the scope of safe interaction by adding more types.
**Design tool**.

Idea: the idealist will still have a useful system.

# Weak Trick 2: weaken the source equivalence

Full abstraction is relative to the source equivalence.

Contextual equivalence makes a closed-world assumption.
Good, sometimes too strong.

Safe impure language: forbid reordering of calls.

Safe impure language: add impure counters for user reasoning.

Or use types with weaker equivalence principles: **linking types**
(Daniel Patterson, Amal Ahmed)

Idea: full abstraction forces you to **specify** the right thing.

# Questions

Compare different ways to specify a weaker equivalence for full abstraction?

- through explicit term equations?
- through types?
- by adding phantom features?

Does our multi-language design scale to more than two languages?
(Yes, I think)

Are boundaries multi-language designs also convenience boundaries?
(good or bad?)

Your questions.

Thanks!