Full abstraction for multi-language systems
ML plus linear types

Gabriel Scherer, Max New, Nicholas Rioux, Amal Ahmed

INRIA Saclay, France
Northeastern University, Boston

November 22, 2017
1 Full Abstraction for Multi-Language Systems: Introduction

2 Case Study: Unrestricted and Linear

3 How Fully Abstract Can We Go?
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?

(Several expert languages: not (yet?) in this work)
A question worth asking

What does it mean for two languages to “interact well together”? 
A question worth asking

What does it **mean** for two languages to “interact well together”?

- no segfaults?
A question worth asking

What does it **mean** for two languages to “interact well together”?

- no segfaults?
- the type systems are not broken?
  (correspondence between types on both sides)
A question worth asking

What does it mean for two languages to “interact well together”?

- no segfaults?
- the type systems are not broken? (correspondence between types on both sides)
- more?
A question worth asking

What does it \textbf{mean} for two languages to “interact well together”? 

- no segfaults?
- the type systems are not broken? (correspondence between types on both sides)
- more?

\[
\begin{align*}
\mathbb{[\_]} &: \text{Types}(L1) \to \text{Types}(L2) \\
\Gamma \vdash_{L1} e : \tau &\quad \Gamma \vdash_{L2} L1(e) : \mathbb{[}\tau]\end{align*}
\]

+ type soundness of the combined system
Full abstraction

\[\boxed{} : S \rightarrow T \text{ fully abstract:} \]

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

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.

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

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

Note: extending $U$ preserves this result.

Note: $L \rightarrow (U + L)$ not meant to be fully abstract.

(Not robust to extensions of $U$)
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: \( L \rightarrow (U + L) \) not meant to be fully abstract.
(Not robust to extensions of U)
Section 2

Case Study: Unrestricted and Linear
Unrestricted language: syntax

Types
\[ \sigma ::= \alpha \mid \sigma_1 \times \sigma_2 \mid 1 \mid \sigma_1 \rightarrow \sigma_2 \mid \sigma_1 + \sigma_2 \mid \mu \alpha. \sigma \mid \forall \alpha. \sigma \]

Expressions
\[ e ::= x \mid \langle e_1, e_2 \rangle \mid \pi_1 e \mid \pi_2 e \mid \langle \rangle \mid e_1 ; e_2 \mid \lambda (x : \sigma). e \mid e_1 e_2 \mid \text{inj}_1 e \mid \text{inj}_2 e \mid \text{case } e' \text{ of } x_1. e_1 \mid x_2. e_2 \mid \text{fold}_{\mu \alpha. \sigma} e \mid \text{unfold } e \mid \Lambda \alpha. e \mid e [\sigma] \]

Typing contexts
\[ \Gamma, \Gamma' ::= \cdot \mid \Gamma, x : \sigma \mid \Gamma, \alpha \]
Linear types: introduction

Resource tracking, unique ownership.

\[ \sigma \quad !\sigma \quad \Gamma \quad \quad \quad \quad !\Gamma \]

\[ \Gamma \vdash_1 e : \sigma \]

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

\[ \sigma ::= \sigma_1 \otimes \sigma_2 \mid 1 \mid \sigma_1 \rightarrow \sigma_2 \mid \]
\[ \sigma_1 \oplus \sigma_2 \mid \mu \alpha. \sigma \mid \alpha \mid \]
\[ !\sigma \mid \]
\[ \text{Box } b \ \sigma \]
Linear types: base

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

\[
\begin{align*}
\Gamma, x : \sigma & \vdash_1 x : \sigma \\
\Gamma & \vdash_1 \langle \rangle : 1 \\
\Gamma & \vdash_1 e : 1 & \Gamma' & \vdash_1 e' : \sigma \\
\Gamma & \vdash_1 e : \sigma_1 \otimes \sigma_2 & \Gamma', x_1 : \sigma_1, x_2 : \sigma_2 & \vdash_1 e' : \sigma \\
\Gamma & \vdash_1 \langle e_1, e_2 \rangle : \sigma_1 \otimes \sigma_2 \\
\Gamma & \vdash_1 e : \sigma' & \Gamma' & \vdash_1 e' : \sigma' \\
\Gamma & \vdash_1 \lambda (x : \sigma). e : \sigma \rightarrow \sigma' \\
\Gamma & \vdash_1 \text{inj}_i e : \sigma_1 \oplus \sigma_2 \\
\Gamma & \vdash_1 e : \sigma & \Gamma' & \vdash_1 e : \sigma' \\
\Gamma & \vdash_1 \text{copy}^\sigma e : \sigma \\
\Gamma & \vdash_1 e : !\sigma & \Gamma & \vdash_1 \text{share} e : !\sigma \\
\Gamma & \vdash_1 e : \sigma \\
\Gamma & \vdash_1 \text{case} e \text{ of } x_1. e_1 | x_2. e_2 : \sigma \\
\Gamma & \vdash_1 \text{let} \langle x_1, x_2 \rangle = e \text{ in } e' : \sigma \\
\end{align*}
\]
Applications

Protocol with resource handling requirements.

“This file descriptor must be closed”

open : !(Path \rightarrow Handle)
line : !(Handle \rightarrow (Handle \oplus (String \otimes Handle)))
close : !(Handle \rightarrow 1)

(details about the boundaries come later)

Typestate.
open : !(Path \rightarrow \circ Handle)
line : !(Handle \rightarrow (Handle \oplus (!String \otimes Handle)))
close : !(Handle \rightarrow \circ 1)

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

Applications: in-place reuse of memory cells.
List reversal

type LList a = \( \mu t. 1 \oplus \text{Box} 1 (a \otimes t) \)

val reverse : LList a \( \rightarrow \) LList a

let reverse list = loop (inl ()) list

where rec loop tail = function
| inl () \rightarrow tail
| inr cell \rightarrow
  let (l, (x, xs)) = unbox cell in
  let cell = box (l, (x, tail)) in
  loop (inr cell) xs
List reversal (sweet)

```ml
let reverse : LList a ⇒ LList a = loop Nil list

where rec loop tail = function
  | Nil → tail
  | Cons l x xs → loop (Cons l x tail) xs
```

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

pattern Nil = inl ()

pattern Cons l x xs = inr (box (l, (x, xs)))
```

```ml
let reverse list = UL(share (reverse (copy (LU(list))))
```

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

```ocaml
type LList a = \mu t. 1 \oplus \text{Box} 1 (a \otimes t)
pattern Nil = \text{inl} ()
pattern Cons l x xs = \text{inr} (\text{box} (l, (x, xs)))

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

type List a = \mu t. 1 + (a \times 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**

\[
\begin{align*}
\text{σ} & \mid \text{σ} \\
\text{σ} & \mid \text{σ} \\
\text{σ} & \mid \text{σ} \\
\text{σ} & \mid \text{σ} \\
\text{σ} & \mid \text{σ} \\
\text{σ} & \mid \text{σ} \\
\text{σ} & \mid \text{σ} \\
\text{σ} & \mid \text{σ} \\
\end{align*}
\]

**Expressions**

\[
\begin{align*}
e & \mid e \\
e & \mid e \\
e & \mid e \\
e & \mid e \\
e & \mid e \\
e & \mid e \\
e & \mid e \\
e & \mid e \\
\end{align*}
\]

**Contexts**

\[
\begin{align*}
\text{Γ} & \mid \cdot \\
\text{Γ} & \mid \text{x:σ} \\
\text{Γ} & \mid \text{α} \\
\text{Γ} & \mid \text{x:σ} \\
\end{align*}
\]

\[
\begin{align*}
!\Gamma \vdash \text{lul e : σ} & \quad \quad !\Gamma \vdash \text{ul e : !σ} \\
!\Gamma \vdash \text{UL(e) : !σ} & \quad \quad !\Gamma \vdash \text{UL(e) : σ}
\end{align*}
\]
Interaction: compatibility

Compatibility relation \( \vdash_{ul} \sigma \simeq \sigma \)

\[
\begin{align*}
\vdash_{ul} 1 & \simeq !1 \\
\vdash_{ul} \sigma_1 & \simeq !\sigma_1 & \vdash_{ul} \sigma_2 & \simeq !\sigma_2 \\
\vdash_{ul} \sigma_1 \times \sigma_2 & \simeq !(\sigma_1 \otimes \sigma_2) \\
\vdash_{ul} \sigma_1 + \sigma_2 & \simeq !(\sigma_1 \oplus \sigma_2) \\
\vdash_{ul} \sigma & \simeq ![\sigma] \\
\vdash_{ul} \sigma & \simeq !!\sigma \\
\vdash_{ul} \sigma & \simeq !(\text{Box } 1 \ \sigma)
\end{align*}
\]

Interaction primitives and derived constructs:

\[
\begin{align*}
\sigma & \overset{\text{unlump}}{\longrightarrow} ![\sigma] \\
!\sigma & \overset{\text{when}}{\longrightarrow} \sigma \\
\sigma & \overset{\text{Lump}}{\longrightarrow} \text{lump}^\sigma \\
\sigma & \overset{\text{LU}(e)}{\longrightarrow} \text{LU}(\text{lump}^\sigma \ e)
\end{align*}
\]
Full abstraction

Theorem

The embedding of $\mathcal{U}$ into $\mathcal{UL}$ 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.

$$
\begin{align*}
[!\sigma] & \overset{\text{def}}{=} [\sigma] \\
[\text{Box} \ 0 \ \sigma] & \overset{\text{def}}{=} 1 \\
[\text{Box} \ 1 \ \sigma] & \overset{\text{def}}{=} 1 \times [\sigma] \\
[\sigma_1 \otimes \sigma_2] & \overset{\text{def}}{=} [\sigma_1] \times [\sigma_2]
\end{align*}
$$

(Cogent)
Remark on parametricity

(from Max New)

$$(\Lambda \alpha. \lambda(x: \alpha). \mathcal{UL}^\alpha(\alpha \mathcal{LU}(x))) [\sigma] \xrightarrow{U} ? \lambda(x: \sigma). \mathcal{UL}^\sigma(\sigma \mathcal{LU}(x))$$

Not well-typed!

$$(\Lambda \alpha. \lambda(x: \alpha). \mathcal{UL}^{!\alpha}(\!\alpha \mathcal{LU}(x))) [\sigma] \xrightarrow{U} \lambda(x: \sigma). \mathcal{UL}^{!\sigma}(\!\sigma \mathcal{LU}(x))$$

Logical relation (Max New, Nicholas Rioux)
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!