Full abstraction for multi-language systems
ML plus linear types

Gabriel Scherer, Amal Ahmed, Max New

Northeastern University, Boston

May 5, 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)
Full abstraction

\[
\llbracket \_ \rrbracket : S \rightarrow T \text{ fully abstract: }
\]

\[
a \approx_{ctx} b \implies \llbracket a \rrbracket \approx_{ctx} \llbracket b \rrbracket
\]

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.

Note: extending \(U\) preserves this result.

\(L \rightarrow (U + L)\) 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.

\[ U \stackrel{f.a.}{\longrightarrow} (U + L) \]

Proof: by translating \( L \) back into \( U \) in an inefficient but correct way.

Note: extending \( U \) preserves this result.
Our case study

\textbf{U} (Unrestricted): general-purpose ML language
\textbf{L} (Linear): expert linear language.

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

Proof: by translating \textbf{L} back into \textbf{U} in an inefficient but correct way.

Note: extending \textbf{U} preserves this result.

Note: \textbf{L} \xrightarrow{} (\textbf{U} + \textbf{L}) not meant to be fully abstract.
(Not robust to extensions of \textbf{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.

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

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

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.
Applications

Protocol with resource handling requirements.

“This file descriptor must be closed”

open : !(Path → Handle)
line : !(Handle → (Handle ⊕ (String ⊗ Handle)))
close : !(Handle → 1)

(details about the boundaries come later)

Typestate.
(details about the boundaries come later)

open : !(Path → Handle)
line : !(Handle → (Handle ⊕ (String ⊗ Handle)))
close : !(Handle → 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)

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

\[
\begin{align*}
\sigma & | \sigma \\
\sigma & |
\sigma + ::= \cdots | [\sigma]
\end{align*}
\]

**Expressions**

\[
\begin{align*}
e & | e \\
e & + ::= \cdots | U\mathcal{L}(e) \\
e & + ::= \cdots | L\mathcal{U}(e)
\end{align*}
\]

**Contexts**

\[
\Gamma ::= \cdot | \Gamma, x: \sigma | \Gamma, \alpha | \Gamma, x: \sigma
\]

\[
\begin{align*}
!\Gamma \vdash_{ul} e : \sigma \\
!\Gamma \vdash_{ul} \mathcal{U}(e) : ![\sigma]
\end{align*}
\]

\[
\begin{align*}
!\Gamma \vdash_{ul} e : ![\sigma] \\
!\Gamma \vdash_{ul} \mathcal{L}(e) : \sigma
\end{align*}
\]
Interaction: compatibility

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

- \( \vdash_{ul} 1 \simeq !1 \)
- \( \vdash_{ul} \sigma_1 \simeq !\sigma_1 \)  \( \vdash_{ul} \sigma_2 \simeq !\sigma_2 \)
- \( \vdash_{ul} \sigma_1 \times \sigma_2 \simeq !(\sigma_1 \otimes \sigma_2) \)
- \( \vdash_{ul} \sigma_1 + \sigma_2 \simeq !(\sigma_1 \oplus \sigma_2) \)
- \( \vdash_{ul} \sigma \simeq ![\sigma] \)
- \( \vdash_{ul} \sigma \simeq !!\sigma \)
- \( \vdash_{ul} \sigma \rightarrow \sigma' \simeq !(\sigma \rightarrow !\sigma') \)
- \( \vdash_{ul} \sigma \simeq ![\sigma] \)
- \( \vdash_{ul} \sigma \simeq ![\sigma] \)
- \( \vdash_{ul} \sigma \simeq !([\sigma] \sigma) \)
- \( \vdash_{ul} \sigma \simeq ![Box 1 \sigma] \)

Interaction primitives and derived constructs:

- \( \sigma \text{ unlump} \)
- \( ![\sigma] \)
- \( \sigma \text{ lump}^\sigma \)
- \( \sigma LU(e) \text{ def } \sigma \text{ unlump } LU(e) \)
- \( UL^\sigma(e) \text{ def } UL(lump^\sigma e) \)
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.

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

(Cogent)
Remark on parametricity

(from Max New)

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

Not well-typed!

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

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!