FabULous Interoperability for ML and a Linear Language

Gabriel Scherer[1,2(✉)], Max New[1], Nick Rioux[1], and Amal Ahmed[1,3]

[1] Northeastern University, Boston, USA
maxnew@ccs.neu.edu, rioux.n@husky.neu.edu, A.Ahmed@northeastern.edu
[2] Inria Saclay, Palaiseau, France
gabriel.scherer@inria.fr
[3] Inria Paris, Paris, France

Abstract. Instead of a monolithic programming language trying to cover all features of interest, some programming systems are designed by combining together simpler languages that cooperate to cover the same feature space. This can improve usability by making each part simpler than the whole, but there is a risk of *abstraction leaks* from one language to another that would break expectations of the users familiar with only one or some of the involved languages.

We propose a formal specification for what it means for a given language in a multi-language system to be usable without leaks: it should embed into the multi-language in a *fully abstract* way, that is, its contextual equivalence should be unchanged in the larger system.

To demonstrate our proposed design principle and formal specification criterion, we design a multi-language programming system that combines an ML-like statically typed functional language and another language with linear types and linear state. Our goal is to cover a good part of the expressiveness of languages that mix functional programming and linear state (ownership), at only a fraction of the complexity. We prove that the embedding of ML into the multi-language system is fully abstract: functional programmers should not fear abstraction leaks. We show examples of combined programs demonstrating in-place memory updates and safe resource handling, and an implementation extending OCaml with our linear language.

1 Introduction

Feature accretion is a common trend among mature but actively evolving programming languages, including C++, Haskell, Java, OCaml, Python, and Scala. Each new feature strives for generality and expressiveness, and may provide a large usability improvement to users of the particular problem domain or programming

Note: Due to severe space restrictions, many details have been omitted from this presentation of our work. We strongly encourage the reader to consult the complete version at https://arxiv.org/pdf/1707.04984.

C. Baier and U. Dal Lago (Eds.): FOSSACS 2018, LNCS 10803, pp. 146–162, 2018.
https://doi.org/10.1007/978-3-319-89366-2_8

style it was designed to empower (e.g., XML documents, asynchronous communication, staged evaluation). But feature creep in general-purpose languages may also make it harder for programmers to master the language as a whole, degrade the user experience (e.g., leading to more cryptic error messages), require additional work on the part of tooling providers, and lead to fragility in language implementations.

A natural response to increased language complexity is to define subsets of the language designed for a better programming experience. For instance, a subset can be easier to teach (e.g., "Core" ML[1], Haskell 98 as opposed to GHC Haskell, Scala mastery levels[2]); it can facilitate static analysis or decrease the risk of programming errors, while remaining sufficiently expressive for the target users' needs (e.g., MISRA C, Spark/Ada); it can enforce a common style within a company; or it can be designed to encourage a transition to deprecate some ill-behaved language features (e.g., strict Javascript).

Once a subset has been selected, it may be the case that users write whole programs purely in the subset (possibly using tooling to enforce that property), but programs will commonly rely on other libraries that are not themselves implemented in the same subset of the language. If users stay in the subset while using these libraries, they will only interact with the part of the library whose interface is expressible in the subset. But does the behavior of the library respect the expectations of users who only know the subset? When calling a function from within the subset breaks subset expectations, it is a sign of *leaky abstraction*.

How should we design languages with useful subsets that manage complexity and avoid abstraction leaks?

We propose to look at this question from a different, but equivalent, angle: instead of designing a single big monolithic language with some nicer subsets, we propose to consider *multi-language* programming systems where several smaller programming languages interact together to cover the same feature space. Each language or sub-combination of languages is a subset, in the above sense, of the multi-language, and there is a clear definition of *abstraction leaks* in terms of user experience: a user who only knows some of the languages of the system should be able to use the multi-language system, interacting with code written in the other languages, without have their expectations violated. If we write a program in Java and call a function that, internally, is implemented in Scala, there should be no surprises—our experience should be the same as when calling a pure Java function. Similarly, consider the subset of Haskell that does not contain `IO` (input-output as a type-tracked effect): the expectations of a user of this language, for instance in terms of valid equational reasoning, should not be violated by adding `IO` back to the language—in the absence of the abstraction-leaking `unsafePerformIO`.

We propose a *formal specification* for a "no abstraction leaks" guarantee that can be used as a design criterion to design new multi-language systems, with graceful interoperation properties. It is based on the formal notion of *full abstraction* which has previously been used to study the denotational semantics

[1] https://caml.inria.fr/pub/docs/u3-ocaml/ocaml-ml.html.
[2] http://www.scala-lang.org/old/node/8610.

of programming languages (Meyer and Sieber 1988; Milner 1977; Cartwright and Felleisen 1992; Jeffrey and Rathke 2005; Abramsky, Jagadeesan, and Malacaria 2000), and the formal property of compilers (Ahmed and Blume 2008, 2011; Devriese et al. 2016; New et al. 2016; Patrignani et al. 2015), but not for user-facing languages. A compiler C from a source language S to a target language T is *fully abstract* if, whenever two source terms s_1 and s_2 are indistinguishable in S, their translations $C(s_1)$ and $C(s_2)$ are indistinguishable in T. In a multi-language $G + E$ formed of a general-purpose, user-friendly language G and a more advanced language E—one that provides an *e*scape hatch for *e*xperts to write code that can't be implemented in G—we say that E does not *leak* into G if the embedding of G into the multi-language $G + E$ is fully abstract.

To demonstrate that our formal specification is reasonable, we design a novel multi-language programming system that satisfies it. Our multi-language λ^{UL} combines a general-purpose functional programming language λ^{U} (unrestricted) of the ML family with an advanced language λ^{L} (linear) with *linear types* and linear state. It is less convient to program in λ^{L}'s restrictive type system, but users can write programs in λ^{L} that could not be written in λ^{U}: they can use linear types, locally, to enforce resource usage protocols (typestate), and they can use linear state and the linear ownership discipline to write programs that do in-place update to allocate less memory, yet remain observationally pure.

Consider for example the following mixed-language program. The blue fragments are written in the general-purpose, user-friendly functional language, while the red fragments are written in the linear language. The boundaries UL and LU allow switching between languages. The program reads all lines from a file, accumulating them in a list, and concatenating it into a single string when the end-of-file (EOF) is reached.

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

The linear type system ensures that the file handle is properly closed: removing the close handle call would give a type error. On the other hand, only the parts concerned with the resource-handling logic need to be written in the red linear language; the user can keep all general-purpose logic (here, how to accumulate lines and what to do with them at the end) in the more convenient general-purpose blue language—and call this function from a blue-language program. Fine-grained boundaries allow users to rely on each language's strength and to use the advanced features only when necessary.

In this example, the file-handle API specifies that the call to line, which reads a line, returns the data at type ![String]. The latter represents how U values of type String can be put into a *lump* type to be passed to the linear world where they are treated as opaque blackboxes that must be passed back to the ML world for consumption. For other examples, such as in-place list manipulation or transient operations on an persistent data structure, we will need a deeper

form of interoperability where the linear world creates, dissects or manipulates U values. To enable this, our multi-language supports translation of types from one language to the other, using a *type compatibility* relation $\sigma \simeq \sigma$ between λ^{U} types σ and λ^{L} types σ.

We claim the following contributions:

1. We propose a formal specification of what it means for advanced language features to be introduced in a (multi-)language system without introducing a class of abstraction leaks that break equational reasoning. This specification captures a useful *usability* property, and we hope it will help us and others design more usable programming languages, much like the formal notion of *principal types* served to better understand and design type inference systems.
2. We design a simple linear language, λ^{L}, that supports linear state (Sect. 2). This simple design for linear state is a contribution of its own. A nice property of the language (shared by some other linear languages) is that the code has both an imperative interpretation—with in-place memory update, which provides resource guarantees—and a functional interpretation—which aids program reasoning. The imperative and functional interpretations have different resource usage, but the same input/output behavior.
3. We present a multi-language programming system λ^{UL} combining a core ML language, λ^{U} (U for Unrestricted, as opposed to Linear) with λ^{L} and prove that the embedding of the ML language λ^{U} in λ^{UL} is fully abstract (Sect. 3). Moreover, the multi-language is designed to ensure that our full abstraction result is stable under extension of the embedded ML language λ^{U}.

2 The λ^{U} and λ^{L} Languages

The unrestricted language λ^{U} is a run-of-the-mill idealized ML language with functions, pairs, sums, iso-recursive types and polymorphism. It is presented in its explicitly typed form—we will not discuss type inference in this work. The full syntax is described in Fig. 1, and the typing rules in Fig. 2. The dynamic semantics is completely standard. Having binary sums, binary products and iso-recursive types lets us express algebraic datatypes in the usual way.

The novelty lies in the linear language λ^{L}, which we present in several steps. As is common in λ-calculi with references, the small-step operational semantics is given for a language that is not exactly the surface language in which programs

$$
\begin{array}{lll}
\textit{Types} & \sigma & ::= \alpha \mid \sigma_1 \times \sigma_2 \mid 1 \mid \sigma_1 \to \sigma_2 \mid \sigma_1 + \sigma_2 \mid \mu\alpha.\,\sigma \mid \forall\alpha.\,\sigma \\
\textit{Expr.} & \mathsf{e} & ::= \mathsf{x} \mid \langle \mathsf{e}_1, \mathsf{e}_2 \rangle \mid \pi_1\,\mathsf{e} \mid \pi_2\,\mathsf{e} \mid \langle\rangle \mid \mathsf{e}_1; \mathsf{e}_2 \mid \lambda(\mathsf{x}:\sigma).\,\mathsf{e} \mid \mathsf{e}_1\ \mathsf{e}_2 \mid \\
 & & \quad \mathsf{inj}_{\mathsf{i}}\,\mathsf{e} \mid \mathsf{case}\,\mathsf{e}'\,\mathsf{of}\,\mathsf{x}_1.\,\mathsf{e}_1 \mid \mathsf{x}_2.\,\mathsf{e}_2 \mid \mathsf{fold}_{\mu\alpha.\sigma}\,\mathsf{e} \mid \mathsf{unfold}\,\mathsf{e} \mid \Lambda\alpha.\,\mathsf{e} \mid \mathsf{e}\,[\sigma] \\
\textit{Values} & \mathsf{v} & ::= \mathsf{x} \mid \langle \mathsf{v}_1, \mathsf{v}_2 \rangle \mid \langle\rangle \mid \lambda(\mathsf{x}:\sigma).\,\mathsf{e} \mid \mathsf{inj}_1\,\mathsf{v} \mid \mathsf{inj}_2\,\mathsf{v} \mid \mathsf{fold}_{\mu\alpha.\sigma}\,\mathsf{v} \mid \Lambda\alpha.\,\mathsf{v} \\
\textit{Contexts} & \Gamma & ::= \cdot \mid \Gamma, \mathsf{x}:\sigma \mid \Gamma, \alpha
\end{array}
$$

Fig. 1. Unrestricted language: syntax

$\boxed{\Gamma \vdash_{\mathrm{U}} \mathsf{e} : \sigma}$

$$\frac{\mathsf{x}{:}\sigma \in \Gamma}{\Gamma \vdash_{\mathrm{U}} \mathsf{x} : \sigma} \qquad \frac{}{\Gamma \vdash_{\mathrm{U}} \langle\rangle : 1} \qquad \frac{\Gamma \vdash_{\mathrm{U}} \mathsf{e} : 1 \qquad \Gamma \vdash_{\mathrm{U}} \mathsf{e}' : \sigma}{\Gamma \vdash_{\mathrm{U}} \mathsf{e}; \mathsf{e}' : \sigma}$$

$$\frac{\Gamma \vdash_{\mathrm{U}} \mathsf{e}_1 : \sigma_1 \qquad \Gamma \vdash_{\mathrm{U}} \mathsf{e}_2 : \sigma_2}{\Gamma \vdash_{\mathrm{U}} \langle \mathsf{e}_1, \mathsf{e}_2 \rangle : \sigma_1 \times \sigma_2} \qquad \frac{\Gamma \vdash_{\mathrm{U}} \mathsf{e} : \sigma_1 \times \sigma_2}{\Gamma \vdash_{\mathrm{U}} \pi_i\, \mathsf{e} : \sigma_i}$$

$$\frac{\Gamma, \mathsf{x}{:}\sigma \vdash_{\mathrm{U}} \mathsf{e} : \sigma'}{\Gamma \vdash_{\mathrm{U}} \lambda(\mathsf{x}{:}\sigma).\, \mathsf{e} : \sigma \to \sigma'} \qquad \frac{\Gamma \vdash_{\mathrm{U}} \mathsf{e} : \sigma' \to \sigma \qquad \Gamma \vdash_{\mathrm{U}} \mathsf{e}' : \sigma'}{\Gamma \vdash_{\mathrm{U}} \mathsf{e}\ \mathsf{e}' : \sigma}$$

$$\frac{\Gamma \vdash_{\mathrm{U}} \mathsf{e} : \sigma_i}{\Gamma \vdash_{\mathrm{U}} \mathsf{inj}_i\, \mathsf{e} : \sigma_1 + \sigma_2} \qquad \frac{\Gamma \vdash_{\mathrm{U}} \mathsf{e} : \sigma_1 + \sigma_2 \qquad \begin{array}{c}\Gamma, \mathsf{x}_1 : \sigma_1 \vdash_{\mathrm{U}} \mathsf{e}_1 : \sigma \\ \Gamma, \mathsf{x}_2 : \sigma_2 \vdash_{\mathrm{U}} \mathsf{e}_2 : \sigma\end{array}}{\Gamma \vdash_{\mathrm{U}} \mathsf{case}\, \mathsf{e}\, \mathsf{of}\, \mathsf{x}_1.\, \mathsf{e}_1 \mid \mathsf{x}_2.\, \mathsf{e}_2 : \sigma}$$

$$\frac{\Gamma \vdash_{\mathrm{U}} \mathsf{e} : \sigma[\mu\alpha.\, \sigma/\alpha]}{\Gamma \vdash_{\mathrm{U}} \mathsf{fold}_{\mu\alpha.\sigma}\, \mathsf{e} : \mu\alpha.\, \sigma} \qquad \frac{\Gamma \vdash_{\mathrm{U}} \mathsf{e} : \mu\alpha.\, \sigma}{\Gamma \vdash_{\mathrm{U}} \mathsf{unfold}\, \mathsf{e} : \sigma[\mu\alpha.\, \sigma/\alpha]}$$

$$\frac{\Gamma, \alpha \vdash_{\mathrm{U}} \mathsf{v} : \sigma}{\Gamma \vdash_{\mathrm{U}} \Lambda\alpha.\, \mathsf{v} : \forall\alpha.\, \sigma} \qquad \frac{\Gamma \vdash_{\mathrm{U}} \mathsf{e} : \forall\alpha.\, \sigma \qquad \Gamma \vdash \sigma'}{\Gamma \vdash_{\mathrm{U}} \mathsf{e}\, [\sigma'] : \sigma[\sigma'/\alpha]}$$

Fig. 2. Unrestricted language: static semantics

are written, because memory allocation returns *locations* ℓ that are not in the grammar of surface terms. Reductions are defined on *configurations*, a local store paired with a term in a slightly larger *internal* language. We have two type systems, a type system on surface terms, that does not mention locations and stores—which is the one a programmer needs to know—and a type system on configurations, which contains enough static information to reason about the dynamics of our language and prove subject reduction. Again, this follows the standard structure of syntactic soundness proofs for languages with a mutable store.

2.1 The Core of λ^{L}

Figure 3 presents the surface syntax of our linear language λ^{L}. For the syntactic categories of types σ, and expressions e, the last line contains the constructions related to the linear store that we only discuss in Sect. 2.2.

In technical terms, our linear type system is exactly propositional intuitionistic linear logic, extended with iso-recursive types. For simplicity and because we did not need them, our current system also does not have polymorphism or additive/lazy pairs $\sigma_1 \,\&\, \sigma_2$. Additive pairs would be a trivial addition, but polymorphism would require more work when we define the multi-language semantics in Sect. 3.

In less technical terms, our type system can enforce that values be used *linearly*, meaning that they cannot be duplicated or erased, they have to be deconstructed

$$
\begin{array}{lll}
\textit{Types} & \sigma & ::= \sigma_1 \otimes \sigma_2 \mid 1 \mid \sigma_1 \multimap \sigma_2 \mid \sigma_1 \oplus \sigma_2 \mid \mu\alpha.\,\sigma \mid \alpha \mid !\sigma \mid \mathsf{Box}\ 1\ \sigma \mid \mathsf{Box}\ 0 \\
\textit{Expr.} & \mathsf{e} & ::= \mathsf{x} \mid \langle \mathsf{e}_1, \mathsf{e}_2 \rangle \mid \mathsf{let}\ \langle \mathsf{v}_1, \mathsf{v}_2 \rangle = \mathsf{e}_1\ \mathsf{in}\ \mathsf{e}_2 \mid \langle\rangle \mid \mathsf{e}_1; \mathsf{e}_2 \mid \lambda(\mathsf{x}:\sigma).\,\mathsf{e} \mid \mathsf{e}_1\ \mathsf{e}_2 \mid \\
 & & \quad \mathsf{inj}_1\,\mathsf{e} \mid \mathsf{inj}_2\,\mathsf{e} \mid \mathsf{case}\,\mathsf{e}'\,\mathsf{of}\,\mathsf{x}_1.\,\mathsf{e}_1 \mid \mathsf{x}_2.\,\mathsf{e}_2 \mid \mathsf{fold}_{\mu\alpha.\sigma}\,\mathsf{e} \mid \mathsf{unfold}\,\mathsf{e} \mid \\
 & & \quad \mathsf{share}\,\mathsf{e} \mid \mathsf{copy}\,\mathsf{e} \mid \mathsf{new}\,\mathsf{e} \mid \mathsf{free}\,\mathsf{e} \mid \mathsf{box}\,\mathsf{e} \mid \mathsf{unbox}\,\mathsf{e} \\
\textit{Values} & \mathsf{v} & ::= \mathsf{x} \mid \langle \mathsf{v}_1, \mathsf{v}_2 \rangle \mid \langle\rangle \mid \lambda(\mathsf{x}:\sigma).\,\mathsf{e} \mid \mathsf{inj}_1\,\mathsf{v} \mid \mathsf{inj}_2\,\mathsf{v} \mid \mathsf{fold}_{\mu\alpha.\sigma}\,\mathsf{v} \mid \mathsf{share}\,\mathsf{v} \\
\textit{Contexts} & \Gamma & ::= \cdot \mid \Gamma, \mathsf{x}:\sigma
\end{array}
$$

Fig. 3. Linear language: surface syntax

exactly once. Only some types have this linearity restriction; others allow duplication and sharing of values at will. We can think of linear values as *resources* to be spent wisely; for any linear value somewhere in a term, there can be only one way to access this value, so we can interpret the language as enforcing an *ownership* discipline where whoever points to a linear value owns it.

In particular, linear functions of type $\sigma_1 \multimap \sigma_2$ must be called exactly once, and their results must in turn be consumed – they can safely capture linear resources. On the other hand, the non-linear, duplicable values are those at types of the form $!\sigma$ — the *exponential* modality of linear logic. If the term e has duplicable type $!\sigma$, then the term $\mathsf{copy}\ \mathsf{e}$ has type σ: this creates a local copy of the value that is uniquely-owned by its receiver and must be consumed linearly.

This resource-usage discipline is enforced by the surface typing rules of λ^{L}, presented in Fig. 4. They are exactly the standard (two-sided) logical rules of intuitionistic linear logic, annotated with program terms. The non-duplicability of linear values is enforced by the way contexts are merged by the inference rules: if e_1 is type-checked in the context Γ_1 and e_2 in Γ_2, then the linear pair $\langle \mathsf{e}_1, \mathsf{e}_2 \rangle$ is only valid in the combined context $\Gamma_1 \curlyvee \Gamma_2$. The $(\curlyvee)$ operation is partial; this combined context is defined only if the variables shared by Γ_1 and Γ_2 are duplicable—their type is of the form $!\sigma$. In other words, a variable at a non-duplicable type in $\Gamma_1 \curlyvee \Gamma_2$ cannot possibly appear in both Γ_1 and Γ_2: it must appear exactly once[3].

The expression $\mathsf{share}\,\mathsf{e}$ takes a term at some type σ and creates a "shared" term, whose value will be duplicable. Its typing rule uses a context of the form $!\Gamma$, which is defined as the pointwise application of the $(!)$ connectives to all the types in Γ. In other words, the context of this rule must only have duplicable types: a term can only be made duplicable if it does not depend on linear resources from the context. Otherwise, duplicating the shared value could break the unique-ownership discipline on these linear resources.

Finally, the linear isomorphism notation for fold and unfold in Fig. 4 defines them as primitive functions, at the given linear function type, in the empty context – using them does not consume resources. This notation also means that, operationally, these two operations shall be inverses of each other. The rules for the linear store type $\mathsf{Box}\ 1\ \sigma$ and $\mathsf{Box}\ 0$ are described in Sect. 2.2.

[3] Standard presentations of linear logic force contexts to be completely distinct, but have a separate rule to duplicate linear variables, which is less natural for programming.

$\boxed{\Gamma_1 \curlyvee \Gamma_2}$

$$(\Gamma_1, \mathsf{x}{:}!\sigma) \curlyvee (\Gamma_2, \mathsf{x}{:}!\sigma) \stackrel{\text{def}}{=} (\Gamma_1 \curlyvee \Gamma_2), \mathsf{x}{:}!\sigma$$
$$(\Gamma_1, \mathsf{x}{:}\sigma) \curlyvee \Gamma_2 \stackrel{\text{def}}{=} (\Gamma_1 \curlyvee \Gamma_2), \mathsf{x}{:}\sigma \qquad (\mathsf{x} \notin \Gamma_2)$$
$$\Gamma_1 \curlyvee (\Gamma_2, \mathsf{x}{:}\sigma) \stackrel{\text{def}}{=} (\Gamma_1 \curlyvee \Gamma_2), \mathsf{x}{:}\sigma \qquad (\mathsf{x} \notin \Gamma_1)$$

$\boxed{\Gamma \vdash_{\mathrm{L}} \mathsf{e} : \sigma}$

$$\frac{}{!\Gamma, \mathsf{x}{:}\sigma \vdash_{\mathrm{L}} \mathsf{x} : \sigma} \qquad \frac{\Gamma_1 \vdash_{\mathrm{L}} \mathsf{e}_1 : \sigma_1 \qquad \Gamma_2 \vdash_{\mathrm{L}} \mathsf{e}_2 : \sigma_2}{\Gamma_1 \curlyvee \Gamma_2 \vdash_{\mathrm{L}} \langle \mathsf{e}_1, \mathsf{e}_2 \rangle : \sigma_1 \otimes \sigma_2}$$

$$\frac{\Gamma \vdash_{\mathrm{L}} \mathsf{e} : \sigma_1 \otimes \sigma_2 \qquad \Gamma', \mathsf{x}_1{:}\sigma_1, \mathsf{x}_2{:}\sigma_2 \vdash_{\mathrm{L}} \mathsf{e}' : \sigma}{\Gamma \curlyvee \Gamma' \vdash_{\mathrm{L}} \mathsf{let}\, \langle \mathsf{x}_1, \mathsf{x}_2 \rangle = \mathsf{e}\, \mathsf{in}\, \mathsf{e}' : \sigma}$$

$$\frac{}{!\Gamma \vdash_{\mathrm{L}} \langle\rangle : 1} \qquad \frac{\Gamma \vdash_{\mathrm{L}} \mathsf{e} : 1 \qquad \Gamma' \vdash_{\mathrm{L}} \mathsf{e}' : \sigma}{\Gamma \curlyvee \Gamma' \vdash_{\mathrm{L}} \mathsf{e}; \mathsf{e}' : \sigma} \qquad \frac{\Gamma, \mathsf{x}{:}\sigma \vdash_{\mathrm{L}} \mathsf{e} : \sigma'}{\Gamma \vdash_{\mathrm{L}} \lambda(\mathsf{x}{:}\sigma).\, \mathsf{e} : \sigma \multimap \sigma'}$$

$$\frac{\Gamma \vdash_{\mathrm{L}} \mathsf{e} : \sigma' \multimap \sigma \qquad \Gamma' \vdash_{\mathrm{L}} \mathsf{e}' : \sigma'}{\Gamma \curlyvee \Gamma' \vdash_{\mathrm{L}} \mathsf{e}\ \mathsf{e}' : \sigma} \qquad \frac{\Gamma \vdash_{\mathrm{L}} \mathsf{e} : \sigma_i}{\Gamma \vdash_{\mathrm{L}} \mathsf{inj}_{\mathsf{i}}\, \mathsf{e} : \sigma_1 \oplus \sigma_2}$$

$$\frac{\Gamma \vdash_{\mathrm{L}} \mathsf{e} : \sigma_1 \oplus \sigma_2 \qquad \begin{array}{c}\Gamma', \mathsf{x}_1 : \sigma_1 \vdash_{\mathrm{L}} \mathsf{e}_1 : \sigma \\ \Gamma', \mathsf{x}_2 : \sigma_2 \vdash_{\mathrm{L}} \mathsf{e}_2 : \sigma\end{array}}{\Gamma \curlyvee \Gamma' \vdash_{\mathrm{L}} \mathsf{case}\, \mathsf{e}\, \mathsf{of}\, \mathsf{x}_1.\, \mathsf{e}_1 \mid \mathsf{x}_2.\, \mathsf{e}_2 : \sigma} \qquad \frac{!\Gamma \vdash_{\mathrm{L}} \mathsf{e} : \sigma}{!\Gamma \vdash_{\mathrm{L}} \mathsf{share}\, \mathsf{e} : !\sigma} \qquad \frac{\Gamma \vdash_{\mathrm{L}} \mathsf{e} : !\sigma}{\Gamma \vdash_{\mathrm{L}} \mathsf{copy}\ \mathsf{e} : \sigma}$$

$$\mu\alpha.\,\sigma \underset{\mathsf{fold}_{\mu\alpha.\sigma}}{\overset{\mathsf{unfold}}{\multimap\!\circ}} \sigma[\mu\alpha.\,\sigma/\alpha] \qquad 1 \underset{\mathsf{free}}{\overset{\mathsf{new}}{\multimap\!\circ}} \mathsf{Box}\ 0 \qquad \mathsf{Box}\ 1\ \sigma \underset{\mathsf{box}}{\overset{\mathsf{unbox}}{\multimap\!\circ}} (\mathsf{Box}\ 0) \otimes \sigma$$

Fig. 4. Linear language: surface static semantics

head reduction $\boxed{\mathsf{e} \overset{\mathrm{L}}{\rightsquigarrow} \mathsf{e}'}$ $\boxed{(s \mid \mathsf{e}) \overset{\mathrm{L}}{\rightsquigarrow} (s' \mid \mathsf{e}')}$

$$(\emptyset \mid \langle\rangle) \underset{\mathrm{L}}{\overset{\mathrm{L}}{\rightsquigarrow}}{}_{\mathsf{free}}^{\mathsf{new}} ([\ell \mapsto \cdot] \mid \ell) \qquad (s[\ell \mapsto \cdot] \mid \langle \ell, \mathsf{v} \rangle) \underset{\mathrm{L}}{\overset{\mathrm{L}}{\rightsquigarrow}}{}_{\mathsf{unbox}}^{\mathsf{box}} ([\ell \mapsto (s \mid \mathsf{v})] \mid \ell)$$

$$(\lambda(\mathsf{x}{:}\sigma).\, \mathsf{e})\ \mathsf{v} \overset{\mathrm{L}}{\rightsquigarrow} \mathsf{e}[\mathsf{v}/\mathsf{x}] \qquad \mathsf{copy}\ (\mathsf{share}(s{:}\Psi).\, \mathsf{inj}_{\mathsf{i}}\, \mathsf{v}) \overset{\mathrm{L}}{\rightsquigarrow} \mathsf{inj}_{\mathsf{i}}\, \mathsf{copy}\ (\mathsf{share}(s{:}\Psi).\, \mathsf{v})$$

$$(\emptyset \mid \mathsf{copy}\ (\mathsf{share}(s{:}\Psi).\, \lambda(\mathsf{x}{:}\sigma).\, \mathsf{e})) \overset{\mathrm{L}}{\rightsquigarrow} (s \mid \lambda(\mathsf{x}{:}\sigma).\, \mathsf{e})$$

$$\mathsf{copy}\ (\mathsf{share}([\ell \mapsto \cdot] : (\cdot; \cdot \vdash \ell : \mathsf{Box}\ 0)).\, \ell) \overset{\mathrm{L}}{\rightsquigarrow} \mathsf{new}\ \langle\rangle$$

$$\mathsf{copy}\ (\mathsf{share}([\ell \mapsto (s \mid \mathsf{v})] : (\Psi; !\Gamma \vdash \ell : \mathsf{Box}\ 1\ \sigma)).\, \ell) \overset{\mathrm{L}}{\rightsquigarrow} \mathsf{box}\ \langle \mathsf{new}\ \langle\rangle, \mathsf{copy}\ (\mathsf{share}(s{:}\Psi).\, \mathsf{v}) \rangle$$

Fig. 5. Internal linear language: typing and reduction (excerpt)

2.2 Linear Memory in λ^{L}

The surface typing rules for the linear store are given at the end of Fig. 4. The linear type $\mathsf{Box}\ 1\ \sigma$ represents a memory location that holds a value of type σ. The type $\mathsf{Box}\ 0$ represents a location that has been allocated, but does not currently hold a value. The primitive operations to act on this type are given as linear isomorphisms: new allocates, turning a unit value into an empty location; conversely, free reclaims an empty location. Putting a value into the location and taking it out are expressed by box and unbox, which convert between a pair of an empty location and a value, of type $(\mathsf{Box}\ 0) \otimes \sigma$, and a full location, of type $\mathsf{Box}\ 1\ \sigma$.

For example, the following program takes a full reference and a value, and swaps the value with the content of the reference:

$$\lambda(\mathsf{p} : (\mathsf{Box}\ 1\ \sigma) \otimes \sigma).\ \mathsf{let}\ \langle \mathsf{r}, \mathsf{x} \rangle = \mathsf{p}\ \mathsf{in}\ \mathsf{let}\ \langle \mathsf{l}, \mathsf{x}_l \rangle = \mathsf{unbox}\ \mathsf{r}\ \mathsf{in}\ \langle \mathsf{box}\ \langle \mathsf{l}, \mathsf{x} \rangle, \mathsf{x}_l \rangle$$

The programming style following from this presentation of linear memory is functional, or applicative, rather than imperative. Rather than insisting on the mutability of references—which is allowed by the linear discipline—we may think of the type $\mathsf{Box}\ 1\sigma$ as representing the indirection through the heap that is implicit in functional programs. In a sense, we are not writing imperative programs with a mutable store, but rather making explicit the allocations and dereferences happening in higher-level purely functional language. In this view, empty cells allow memory reuse.

This view that $\mathsf{Box}\ 1\ \sigma$ represents indirection through the memory suggests we can encode lists of values of type σ by the type $\mathsf{LinList}\,\sigma \stackrel{\text{def}}{=} \mu\alpha.1 \oplus \mathsf{Box}\ 1\ (\sigma \otimes \alpha)$. The placement of the box inside the sum mirrors the fact that empty list is represented as an immediate value in functional languages. From this type definition, one can write an in-place reverse function on lists of σ as follows:

$$\begin{array}{l}
\mathsf{fix}\ \lambda(\mathsf{rev_into} : \mathsf{LinList}\ \sigma \multimap \mathsf{LinList}\ \sigma \multimap \mathsf{LinList}\ \sigma).\\
\quad \lambda(\mathsf{xs} : \mathsf{LinList}\ \sigma).\ \lambda(\mathsf{acc} : \mathsf{LinList}\ \sigma).\\
\quad \mathsf{case}\ \mathsf{unfold}\ \mathsf{xs}\ \mathsf{of}\\
\quad \mid \mathsf{y}.\ (\mathsf{y}; \mathsf{acc})\\
\quad \mid \mathsf{y}.\ \mathsf{let}\ \langle \mathsf{l}, \mathsf{p} \rangle = \mathsf{unbox}\ \mathsf{y}\ \mathsf{in}\\
\qquad \mathsf{let}\ \langle \mathsf{xs}, \mathsf{x} \rangle = \mathsf{p}\ \mathsf{in}\\
\qquad \mathsf{rev_into}\ \mathsf{xs}\ (\mathsf{fold}\ (\mathsf{inj}_2\ (\mathsf{box}\ \langle \mathsf{l}, \langle \mathsf{x}, \mathsf{acc} \rangle \rangle))))
\end{array}$$

Our linear language λ^{L} is a formal language that is not terribly convenient to program directly. We will not present a full surface language in this work, but one could easily define syntactic sugar to write the exact same function as follows:

$$\begin{array}{ll}
\mathsf{rev_into}\ \mathsf{Nil} & \mathsf{acc} = \mathsf{acc}\\
\mathsf{rev_into}\ (\mathsf{Cons}\ \langle \mathsf{x}, \mathsf{xs} \rangle @\, \mathsf{l}) & \mathsf{acc} = \mathsf{rev_into}\ \mathsf{xs}\ (\mathsf{Cons}\ \langle \mathsf{x}, \mathsf{acc} \rangle @\, \mathsf{l})
\end{array}$$

One can read this function as the usual functional `rev_append` function on lists, annotated with memory reuse information: if we assume we are the unique owner of the input list and won't need it anymore, we can reuse the memory of its cons cells (given in this example the name l) to store the reversed list.

On the other hand, if you read the box and unbox as imperative operations, this code expresses the usual imperative pointer-reversal algorithm.

This double view of linear state occurs in other programming systems with linear state. It was recently emphasized in O'Connor et al. (2016), where the functional point of view is seen as easing formal verification, while the imperative view is used as a compilation technique to produce efficient C code from linear programs.

2.3 Internal λ^{L} Syntax and Typing

To give a dynamic semantics for λ^{L} and prove it sound, we need to extend the language with explicit stores and store locations. Indeed, the allocating term $\mathsf{new}\,\langle\rangle$ should reduce to a "fresh location" ℓ allocated in some store s, and neither are part of the surface-language syntax. The corresponding internal typing judgment is more complex, but note that users do not need to know about it to reason about correctness of surface programs. The internal typing is essential for the soundness proof, but also useful for defining the multi-language semantics in Sect. 3.

We work with *configurations* $(s \mid \mathsf{e})$, which are pairs of a store s and a term e. Our internal typing judgment $\Psi; \Gamma \vdash_{\mathrm{L}} s \mid \mathsf{e} : \sigma$ checks configurations, not just terms, and relies not only on a typing context for variables Γ but also on a *store typing* Ψ, which maps the locations of the configuration to typing assumptions.

Unfortunately, due to space limits, we will not present this part of the type system – which is not directly exposed to users of the language. See some examples of reduction rules in Fig. 5, and the long version of this work.

2.4 Reduction of Internal Terms

In the long version of this work we give a reduction relation between linear configurations $(s \mid \mathsf{e}) \overset{\mathsf{L}}{\hookrightarrow} (s' \mid \mathsf{e}')$ and prove a subject reduction result.

Theorem 1 (Subject reduction for λ^{L}). *If* $\Psi; \Gamma \vdash_{\mathrm{L}} s \mid \mathsf{e} : \sigma$ *and* $(s \mid \mathsf{e}) \overset{\mathsf{L}}{\hookrightarrow} (s' \mid \mathsf{e}')$*, then there exists a (unique)* Ψ' *such that* $\Psi'; \Gamma \vdash_{\mathrm{L}} s' \mid \mathsf{e}' : \sigma$.

3 Multi-language Semantics

To formally define our multi-language semantics we create a combined language $\lambda^{\mathsf{U}\mathsf{L}}$ which lets us compose term fragments from both λ^{U} and λ^{L} together, and we give an operational semantics to this combined language. Interoperability is enabled by specifying how to transport values across the language boundaries.

Multi-language systems in the wild are not defined in this way: both languages are given a semantics, by interpretation or compilation, in terms of a shared lower-level language (C, assembly, the JVM or CLR bytecode, or Racket's core forms), and the two languages are combined at that level. Our formal multi-language description can be seen as a model of such combinations, that gives a specification of the expected observable behavior of this language combination.

Another difference from multi-languages in the wild is our use of very fine-grained language boundaries: a term written in one language can have its subterms written in the other, provided the type-checking rules allow it. Most multi-language systems, typically using Foreign Function Interfaces, offer coarser-grained composition at the level of compilation units. Fine-grained composition of existing languages, as done in the Eco project (Barrett et al. 2016), is difficult because of semantic mismatches. In the full version of this work we demonstrate that fine-grained composition is a rewarding language design, enabling new programming patterns.

3.1 Lump Type and Language Boundaries

The core components the multi-language semantics are shown Fig. 6—the communication of values from one language to the other will be described in the next section. The multi-language $\lambda^{\mathsf{U}\mathsf{L}}$ has two distinct syntactic categories of types, values, and expressions: those that come from λ^{U} and those that come from λ^{L}. Contexts, on the other hand, are mixed, and can have variables of both sorts. For a mixed context Γ, the notation $!\Gamma$ only applies ($!$) to its linear variables.

The typing rules of λ^{U} and λ^{L} are imported into our multi-language system, working on those two separate categories of program. They need to be extended to handle mixed contexts Γ instead of their original contexts Γ and Γ. In the linear case, the rules look exactly the same. In the ML case, the typing rules implicitly duplicate all the variables in the context. It would be unsound to extend them to arbitrary linear variables, so they use a duplicable context $!\Gamma$.

To build interesting multi-language programs, we need a way to insert a fragment coming from a language into a term written in another. This is done using *language boundaries*, two new term formers $\mathcal{LU}(\mathsf{e})$ and $\mathcal{UL}(s{:}\Psi \mid \mathsf{e})$ that inject an ML term into the syntactic category of linear terms, and a linear configuration into the syntactic category of ML terms.

Of course, we need new typing rules for these term-level constructions, clarifying when it is valid to send a value from λ^{U} into λ^{L} and vice versa. It would be incorrect to allow sending any type from one language into the other—for instance, by adding the counterpart of our language boundaries in the syntax of types—since values of linear types must be uniquely owned so they cannot possibly be sent to the ML side as the ML type system cannot enforce unique ownership.

On the other hand, any ML value could safely be sent to the linear world. For closed types, we could provide a corresponding linear type (1 maps to $!1$, etc.), but an ML value may also be typed by an abstract type variable α, in which case we can't know what the linear counterpart should be. Instead of trying to

provide translations, we will send any ML type σ to the *lump type* $[\sigma]$, which embeds ML types into linear types. A lump is a blackbox, not a type translation: the linear language does not assume anything about the behavior of its values—the values of $[\sigma]$ are of the form $[\mathsf{v}]$, where $\mathsf{v} : \sigma$ is an ML value that the linear world cannot use. More precisely, we only propagate the information that ML values are all duplicable by sending σ to $![\sigma]$.

The typing rules for language boundaries insert lumps when going from λ^{U} to λ^{L}, and remove them when going back from λ^{L} to λ^{U}. In particular, arbitrary linear types cannot occur at the boundary, they must be of the form $![\sigma]$.

$$
\begin{array}{rl}
\textit{Types } \sigma \mid \sigma & \\
\quad \sigma & \qquad \text{(unchanged from Figure 1)} \\
\quad \sigma & \quad + ::= \cdots \mid [\sigma] \\
\\
\textit{Values } \mathsf{v} \mid \mathsf{v} & \\
\quad \mathsf{v} & \qquad \text{(unchanged from Figure 1)} \\
\quad \mathsf{v} & \quad + ::= \cdots \mid [\mathsf{v}] \\
\\
\textit{Expressions } \mathsf{e} \mid \mathsf{e} & \\
\quad \mathsf{e} & \quad + ::= \cdots \mid \mathcal{UL}(s{:}\Psi \mid \mathsf{e}) \\
& \qquad \text{with} \quad \mathcal{UL}(\mathsf{e}) \stackrel{\text{def}}{=} \mathcal{UL}(\emptyset{:}\cdot \mid \mathsf{e}) \\
\quad \mathsf{e} & \quad + ::= \cdots \mid \mathcal{LU}(\mathsf{e}) \\
\\
\multicolumn{2}{c}{\textit{Contexts } \Gamma ::= \cdot \mid \Gamma, \mathsf{x}{:}\sigma \mid \Gamma, \alpha \mid \Gamma, \mathsf{x}{:}\sigma}
\end{array}
$$

Typing rules $\boxed{\Gamma \vdash_{\text{LU}} \mathsf{e} : \sigma}$ $\boxed{\Psi \mid \Gamma \vdash_{\text{UL}} s \mid \mathsf{e} : \sigma}$

with $\Gamma \vdash_{\text{UL}} \mathsf{e} : \sigma \stackrel{\text{def}}{=} \cdot \mid \Gamma \vdash_{\text{UL}} \emptyset \mid \mathsf{e} : \sigma$

(Typing rules of $\Gamma \vdash_{\text{U}} \mathsf{e} : \sigma$ reused, with mixed context $!\Gamma$)
(Typing rules of $\Psi; \Gamma \vdash_{\text{L}} s \mid \mathsf{e} : \sigma$ reused, with mixed context Γ)

$$\frac{!\Gamma \vdash_{\text{LU}} \mathsf{e} : \sigma}{\cdot \mid !\Gamma \vdash_{\text{UL}} \emptyset \mid \mathcal{LU}(\mathsf{e}) : ![\sigma]} \qquad \frac{\Psi \mid !\Gamma \vdash_{\text{UL}} s \mid \mathsf{e} : ![\sigma]}{!\Gamma \vdash_{\text{LU}} \mathcal{UL}(s{:}\Psi \mid \mathsf{e}) : \sigma}$$

Reduction rules

(Reduction rules of λ^{U} and λ^{L} reused unchanged) $\qquad \dfrac{\mathsf{e} \overset{\mathsf{U}}{\hookrightarrow} \mathsf{e}'}{\mathcal{LU}(\mathsf{e}) \overset{\mathsf{L}}{\hookrightarrow} \mathcal{LU}(\mathsf{e}')}$

$$\frac{}{\mathcal{LU}(\mathsf{v}) \overset{\mathsf{L}}{\rightsquigarrow} [\mathsf{v}]} \qquad \frac{}{\mathcal{UL}(\emptyset{:}\cdot \mid \mathsf{share}\,[\mathsf{v}]) \overset{\mathsf{U}}{\hookrightarrow} \mathsf{v}}$$

$$\frac{\Psi \mid \Gamma \vdash_{\text{UL}} s \mid \mathsf{e} : \sigma \qquad (s \mid \mathsf{e}) \overset{\mathsf{L}}{\hookrightarrow} (s' \mid \mathsf{e}') \qquad \Psi' \mid \Gamma \vdash_{\text{UL}} s' \mid \mathsf{e}' : \sigma}{\mathcal{UL}(s{:}\Psi \mid \mathsf{e}) \overset{\mathsf{U}}{\hookrightarrow} \mathcal{UL}(s'{:}\Psi' \mid \mathsf{e}')}$$

Fig. 6. Multi-language: lump and boundaries

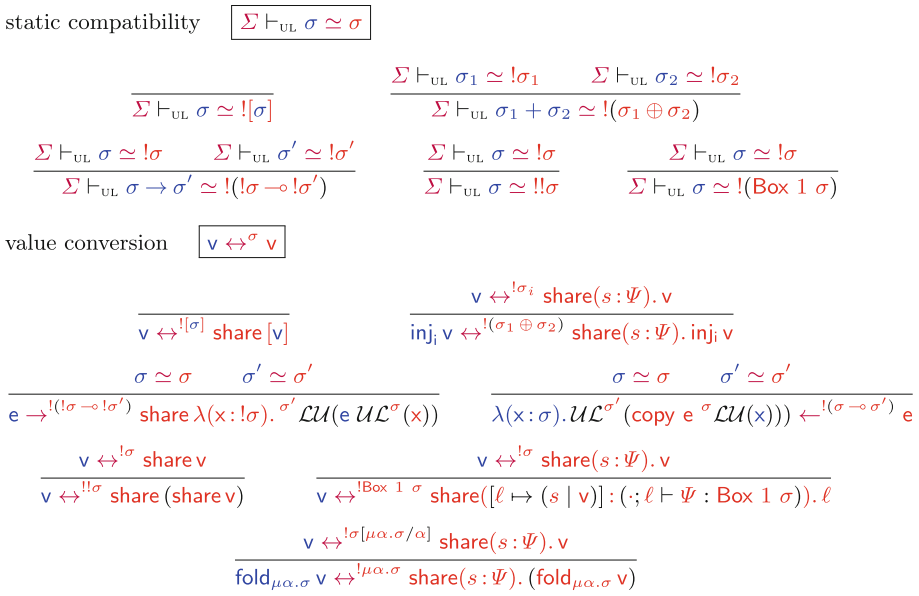

Fig. 7. Interoperability: static and dynamic semantics (excerpt)

Finally, boundaries have reduction rules: a term or configuration inside a boundary in reduction position is reduced until it becomes a value, and then a lump is added or removed depending on the boundary direction. Note that because the v in $\mathcal{UL}(s:\Psi \mid \mathsf{v})$ is at a duplicable type $![\sigma]$, we know by inversion that the store is empty.

3.2 Interoperability: Static Semantics

If the linear language could not interact with lumped values at all, our multi-language programs would be rather boring, as the only way for the linear extension to provide a value back to ML would be to have received it from λ^{U} and pass it back unchanged (as in the lump embedding of Matthews and Findler (2009)). To provide a real interaction, we provide a way to extract values out of a lump $![\sigma]$, use it at some linear type σ, and put it back in before sending the result to λ^{U}.

The correspondence between intuitionistic types σ and linear types σ is specified by a heterogeneous *compatibility relation* $\sigma \simeq \sigma$ – defined in full in Fig. 7. The specification of this relation is that if $\sigma \simeq \sigma$ holds, then the space of values of $![\sigma]$ and σ are isomorphic: we can convert back and forth between them. When this relation holds, the term-formers lump^{σ} and ${}^{\sigma}\mathsf{unlump}$ perform the conversion.

The term $\mathcal{LU}(\mathsf{e})$ turns a $\mathsf{e} : \sigma$ into a lumped type $![\sigma]$, and we need to unlump it with some ${}^{\sigma}\mathsf{unlump}$ for a compatible $\sigma \simeq \sigma$ to interact with it on the linear

side. It is common to combine both operations and we provide syntactic sugar for it: ${}^{\sigma}\mathcal{LU}(\mathsf{e})$. Similarly $\mathcal{UL}^{\sigma}(\mathsf{e})$ first lumps a linear term then sends the result to the ML world.

3.3 Interoperability: Dynamic Semantics

When the relation $\sigma \simeq \sigma$ holds, we can define a relation $\mathsf{v} \leftrightarrow^{\sigma} \mathsf{v}$ between the values of σ and the values of σ – see the long version of this work. It is functional in both direction: with our definition v is uniquely determined from v and conversely. We then define the reduction rule for (un)lumping: if $\mathsf{v} \leftrightarrow^{\sigma} \mathsf{v}$, then

$$(\emptyset \mid {}^{\sigma}\mathsf{unlump}\ (\mathsf{share}\ [\mathsf{v}])) \overset{\mathsf{L}}{\hookrightarrow} (\emptyset \mid \mathsf{v}) \qquad (\emptyset \mid \mathsf{lump}^{\sigma}\ \mathsf{v}) \overset{\mathsf{L}}{\hookrightarrow} (\emptyset \mid \mathsf{share}\ [\mathsf{v}])$$

3.4 Full Abstraction from λ^{U} into λ^{UL}

We can now state the major meta-theoretical result of this work, which is the proposed multi-language design extends the simple language λ^{U} in a way that provably has, in a certain sense, "no abstraction leaks".

Definition 1 (Contextual equivalence in λ^{U}). *We say that* e, e' *such that* $\Gamma \vdash_{\mathrm{U}} \mathsf{e}, \mathsf{e}' : \sigma$ *are* contextually equivalent*, written* $\mathsf{e} \approx^{ctx}_{\mathrm{U}} \mathsf{e}'$*, if, for any expression context* $C[\square]$ *such that* $\cdot \vdash_{\mathrm{U}} C[\mathsf{e}] : 1$*, the closed terms* $C[\mathsf{e}]$ *and* $C[\mathsf{e}']$ *are equi-terminating.*

Definition 2 (Contextual equivalence in λ^{UL}). *We say that* e, e' *such that* $\Gamma \vdash_{\mathrm{LU}} \mathsf{e}, \mathsf{e}' : \sigma$ *are* contextually equivalent*, written* $\mathsf{e} \approx^{ctx}_{\mathrm{LU}} \mathsf{e}'$*, if, for any expression context* $C[\square]$ *such that* $\cdot \vdash_{\mathrm{LU}} C[\mathsf{e}] : 1$*, the closed terms* $C[\mathsf{e}]$ *and* $C[\mathsf{e}']$ *are equi-terminating.*

Theorem 2 (Full Abstraction). *The embedding of* λ^{U} *into* λ^{UL} *is fully-abstract:*

$$\Gamma \vdash_{\mathrm{U}} \mathsf{e} \approx^{ctx}_{\mathrm{U}} \mathsf{e}' : \sigma \qquad \Longrightarrow \qquad \Gamma \vdash_{\mathrm{LU}} \mathsf{e} \approx^{ctx}_{\mathrm{LU}} \mathsf{e}' : \sigma$$

4 Conclusion and Related Work

Having a stack of usable, interoperable languages, extensions or dialects is at the forefront of the Racket approach to programming environments, in particular for teaching (Felleisen et al. 2004).

Our multi-language semantics builds on the seminal work by Matthews and Findler (2009), who gave a formal semantics of interoperability between a dynamically and a statically typed language. Others have followed the Matthews-Findler approach of designing multi-language systems with fine-grained boundaries—for instance, formalizing interoperability between a simply and dependently typed language (Osera et al. 2012); between a functional and typed assembly language (Patterson et al. 2017); between an ML-like and an

affinely typed language, where linearity is enforced at runtime on the ML side using stateful contracts (Tov and Pucella 2010); and between the source and target languages of compilation to specify compiler correctness (Perconti and Ahmed 2014). However, all these papers address only the question of soundness of the multi-language; we propose a formal treatment of *usability* and absence of abstraction leaks.

The only work to establish that a language embeds into a multi-language in a fully abstract way is the work on fully abstract compilation by Ahmed and Blume (2011) and New et al. (2016) who show that their compiler's source language embeds into their source-target multi-language in a fully abstract way. But the focus of this work was on fully abstract compilation, not on usability of user-facing languages.

The Eco project (Barrett et al. 2016) is studying multi-language systems where user-exposed languages are combined in a very fine-grained way; it is closely related in that it studies the user experience in a multi-language system. The choice of an existing dynamic language creates delicate interoperability issues (conflicting variable scoping rules, etc.) as well as performance challenges. We propose a different approach, to design new multi-languages from scratch with interoperability in mind to avoid legacy obstacles.

We are not aware of existing systems exploiting the simple idea of using promotion to capture uniquely-owned state and dereliction to copy it—common formulations would rather perform copies on the contraction rule.

The general idea that linear types can permit reuse of unused allocated cells is not new. In Wadler (1990), a system is proposed with both linear and non-linear types to attack precisely this problem. It is however more distant from standard linear logic and somewhat ad-hoc; for example, there is no way to permanently turn a uniquely-owned value into a shared value, it provides instead a local *borrowing* construction that comes with ad-hoc restrictions necessary for safety. (The inability to *give up* unique ownership, which is essential in our list-programming examples, seems to also be missing from Rust, where one would need to perform a costly operation of traversing the graph of the value to turn all pointers into `Arc` nodes.)

The RAML project (Hoffmann et al. 2012) also combines linear logic and memory reuse: its *destructive match* operator will implicitly reuse consumed cells in new allocations occurring within the match body. Multi-languages give us the option to explore more explicit, flexible representations of those low-level concern, without imposing the complexity to all programmers.

A recent related work is the Cogent language (O'Connor et al. 2016), in which linear state is also viewed as both functional and imperative – the latter view enabling memory reuse. The language design is interestingly reversed: in Cogent, the linear layer is the simple language that everyone uses, and the non-linear language is a complex but powerful language that is used when one really has to, named C.

Our linear language λ^{L} is sensibly simpler, and in several ways less expressive, than advanced programming languages based on linear logic (Tov and Pucella 2011), separation logic (Balabonski et al. 2016), fine-grained permissions (Garcia et al. 2014): it is not designed to stand on its own, but to serve as a useful sidekick to a functional language, allowing safer resource handling.

One major simplification of our design compared to more advanced linear or separation-logic-based languages is that we do not separate physical locations from the logical capability/permission to access them (e.g., as in Ahmed et al. (2007)). This restricts expressiveness in well-understood ways (Fahndrich and DeLine 2002): shared values cannot point to linear values.

Alms (Tov and Pucella 2011), Quill (Morris 2016) and Linear Haskell (Bernardy et al. 2018) add linear types to a functional language, trying hard not to lose desirable usability property, such as type inference or the genericity of polymorphic higher-order functions. This is very challenging; for example, Linear Haskell gives up on principality of inference[4]. Our multi-language design side-steps this issue as the general-purpose language remains unchanged. Language boundaries are more rigid than an ideal no-compromise language, as they force users to preserve the distinction between the general-purpose and the advanced features; it is precisely this compromise that gives a design of reduced complexity.

Finally, on the side of the semantics, our system is related to LNL (Benton 1994), a calculus for linear logic that, in a sense, is itself built as a multi-language system where (non-duplicable) linear types and (duplicable) intuitionistic types interact through a boundary. It is not surprising that our design contains an instance of this adjunction: for any σ there is a unique σ such that $\sigma \simeq !\sigma$, and converting a σ value to this σ and back gives a $!\sigma$ and is provably equivalent, by boundary cancellation, to just using share.

Acknowledgments. We thank our anonymous reviewers for their feedback, as well as Neelakantan Krishnaswami, François Pottier, Jennifer Paykin, Sylvie Boldot and Simon Peyton-Jones for our discussions on this work.

This work was supported in part by the National Science Foundation under grants CCF-1422133 and CCF-1453796, and the European Research Council under ERC Starting Grant SECOMP (715753). Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of our funding agencies.

References

Abramsky, S., Jagadeesan, R., Malacaria, P.: Full abstraction for PCF. Inf. Comput. **163**(2), 409–470 (2000)

Ahmed, A., Blume, M.: Typed closure conversion preserves observational equivalence. In: International Conference on Functional Programming (ICFP), Victoria, British Columbia, Canada, pp. 157–168, September 2008

[4] Thanks to Stephen Dolan for pointing out that $\lambda f.\lambda x.\ f\ x$ has several incompatible Linear Haskell types.

Ahmed, A., Blume, M.: An equivalence-preserving CPS translation via multi-language semantics. In: International Conference on Functional Programming (ICFP), Tokyo, Japan, pp. 431–444, September 2011

Ahmed, A., Fluet, M., Morrisett, G.: L3: a linear language with locations. Fundamenta Informaticae **77**(4), 397–449 (2007)

Balabonski, T., Pottier, F., Protzenko, J.: The design and formalization of Mezzo, a permission-based programming language. ACM Trans. Program. Lang. Syst. **38**(4), 14:1–14:94 (2016)

Barrett, E., Bolz, C.F., Diekmann, L., Tratt, L.: Fine-grained language composition: a case study. In: ECOOP (2016)

Benton, P.N.: A mixed linear and non-linear logic: proofs, terms and models. In: Pacholski, L., Tiuryn, J. (eds.) CSL 1994. LNCS, vol. 933, pp. 121–135. Springer, Heidelberg (1995). https://doi.org/10.1007/BFb0022251

Bernardy, J.-P., Boespflug, M., Newton, R.R., Jones, S.P., Spiwack, A.: Linear haskell: practical linearity in a higher-order polymorphic language. PACMPL **2**(POPL), 5:1–5:29 (2018). https://doi.org/10.1145/3158093

Cartwright, R., Felleisen, M.: Observable sequentiality and full abstraction. In: ACM Symposium on Principles of Programming Languages (POPL), Albuquerque, New Mexico, pp. 328–342 (1992)

Devriese, D., Patrignani, M., Piessens, F.: Fully-abstract compilation by approximate back-translation. In: ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg, Florida (2016)

Fahndrich, M., DeLine, R.: Adoption and focus: practical linear types for imperative programming. In: PLDI 2002 (2002)

Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S.: The teachscheme! project: computing and programming for every student. Comput. Sci. Educ. **14**(1), 55–77 (2004)

Garcia, R., Tanter, É., Wolff, R., Aldrich, J.: Foundations of typestate-oriented programming. TOPLAS **36**, 12 (2014)

Hoffmann, J., Aehlig, K., Hofmann, M.: Resource aware ML. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 781–786. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_64

Jeffrey, A., Rathke, J.: Java JR: fully abstract trace semantics for a Core Java Language. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 423–438. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31987-0_29

Matthews, J., Findler, R.B.: Operational semantics for multi-language programs. ACM Trans. Program. Lang. Syst. (TOPLAS) **31**(3), 12 (2009)

Meyer, A.R., Sieber, K.: Towards fully abstract semantics for local variables. In: ACM Symposium on Principles of Programming Languages (POPL), San Diego, California, pp. 191–203 (1988)

Milner, R.: Fully abstract models of typed lambda calculi. Theor. Comput. Sci. **4**(1), 1–22 (1977)

Morris, J.G.: The best of both worlds: linear functional programming without compromise. In: ICFP (2016)

New, M.S., Bowman, W.J., Ahmed, A.: Fully abstract compilation via universal embedding. In: International Conference on Functional Programming (ICFP), Nara, Japan, September 2016

O'Connor, L., Chen, Z., Rizkallah, C., Amani, S., Lim, J., Murray, T., Nagashima, Y., Sewell, T., Klein, G.: Refinement through restraint: bringing down the cost of verification. In: ICFP (2016)

Osera, P.M., Sjöberg, V., Zdancewic, S.: Dependent interoperability. In: Programming Languages Meets Program Verification (PLPV), January 2012

Patrignani, M., Agten, P., Strackx, R., Jacobs, B., Clarke, D., Piessens, F.: Secure compilation to protected module architectures. ACM Trans. Program. Lang. Syst. **37**(2), 6:1–6:50 (2015)

Patterson, D., Perconti, J., Dimoulas, C., Ahmed, A.: FunTAL: reasonably mixing a functional language with assembly. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Barcelona, Spain, June 2017. http://www.ccs.neu.edu/home/amal/papers/funtal.pdf

Perconti, J.T., Ahmed, A.: Verifying an open compiler using multi-language semantics. In: Shao, Z. (ed.) ESOP 2014. LNCS, vol. 8410, pp. 128–148. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54833-8_8

Tov, J.A., Pucella, R.: Stateful contracts for affine types. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 550–569. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11957-6_29

Tov, J.A., Pucella, R.: Practical affine types. In: POPL (2011)

Wadler, P.: Linear types can change the world! In: Programming Concepts and Methods (1990)