Multi-Language Programming Systems: a Linear Experiment

Gabriel Scherer Max New Amal Ahmed

Northeastern University

{gasche,maxnew,amal}@ccs.neu.edu

Abstract

Instead of a monolithic programming language trying to cover all features of interest, some programming systems are design 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 formally study this problem by reusing ideas from previous work on multi-language semantics, used to study modular compilation, that suggest a formal definition of what it means for a given language to be usable without leaks: it should embed into the multi-language system in a *fully abstract* way, that is, its contextual equivalence should be unchanged in the larger system. This strong formal requirement does not hold in most existing systems; is it a attainable goal for language design?

As a first experiment, we design a multi-language programming system that combines a simple ML-like language and a very simple linear language with linear state. The 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 typestate-like usage protocols.

1. Introduction

1.1 Motivation: Multi-Languages Against Complexity

Feature accretion is a common trend among mature but actively-evolving programming languages: C++, Haskell, Java, OCaml, Python, Scala, etc. Each new feature strives for generality and expressiveness, and may provide a large usability improvement to users of the particular problem domain or programming style it was designed to empower (XML documents, asynchronous communication, staged evaluation...), but it also makes it harder to master the language as a whole, requires additional work on the part of tooling providers, and may lead to fragility in tools or language implementations.

A natural response to growing language complexity is to define subsets of the language designed for better programming experience: a subset could be easier to teach while sufficiently educative ("Core" ML[1], Haskell 98 as opposed to "GHC Haskell", Scala mastery levels[2]), it could facilitate static analysis or decrease the risk of programming errors, while remaining sufficiently expressive for the target userbase needs (MISRA C, Spark/Ada), or be designed to encourage a transition to deprecate some ill-behaved language features (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 we stay in the subset while using these libraries, we will only interact with the part of the library whose interface is expressible in our subset; but does the behavior respect the expectations of a user that would 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 while avoiding abstraction leaks?

This question can be generalized to multi-language programming systems, where a piece of software is written by mixing together fragments written in different languages – such multi-language programs are ubiquitous. Multi-language systems are also used to manage complexity, by letting users write each part of the program using appropriate linguistic abstractions, without the need for a giant monolithic programming language that would cover all needs. In fact, the subset use-case is an instance of a multi-language system, if we consider the subset and the full language as two separate – yet related – programming languages.

The question of respecting user expectations also occurs there. Can a user of only one of the involved language correctly reason about the program fragment they are working on, or do they need to master all the languages used to reason about their code? When you write C primitives for a Python program, you would not expect this C code to break memory-

[1] https://caml.inria.fr/pub/docs/u3-ocaml/ocaml-ml.html

[2] http://www.scala-lang.org/old/node/8610

safety of Python user code – you expect some basic Python reasoning principles to be respected. If users cannot safely ignore the other, more complex languages when working on their code, usability is not really improved. Note that this issue also occurs when some of the languages or linguistic abstractions involved are not separate languages, but DSLs included or embedded into a general-purpose programming language.

How should we design multi-language programming systems, so that the languages interact gracefully together?

Of course, there is more to programming system design than piling features or languages on top of each other. Language designers occasionally strike gold and develop a powerful, unifying feature that subsumes several more specific-purpose features while extending the expressiveness of the language (delimited control, monads, continuation marks, dependent types...). But the availability of these features does not make the simpler subsumed features disappear overnight: their simplicity may make them more comfortable to use in the common case, more optimizable, and users will often reintroduce them as libraries. The fact that their are all implemented in terms of a powerful general feature will make it easier to make those sub-abstractions work well together, but we should still understand what exactly this "working well" requirement means and how to check it.

1.2 Formal Design of Multi-Language Systems

We propose to formally study those usability questions in programming language design. To do that, we rely on a existing body of work on the formal study of *multi-language semantics*, initially developed to formally study modular compilers let us link program fragments written in the source language, target language, and possibly some intermediate compilation languages. A formal notion used in these work, *full abstraction*, is of particular interest here. Consider a source language S and a target language T, both equipped with a given equivalence relation – the usual definition is that two fragments are equivalent if, whenever placed in a well-typed context, replacing one by the other does not change the program behavior. An embedding or translation from S to T is *fully abstract* if, whenever two program fragments in S are equivalent, then the translation of those fragments are equivalent: the translation preserves equational reasoning. Full abstraction gives a very strong notion of "graceful interaction": if the embedding of a single language into the multi-language system is fully abstract, we know that equational reasoning in this single language remains valid in the multi-language system.

We thus propose to evaluate the following design principle: in subset/superset or multi-language settings, the language that we think as "simple" should embed into its superset, or the multi-language, in a fully-abstract way. Is this a reasonable guiding principle, or is it unrealistic?

Of course, sometimes adding new behaviors that the previous system could not express is the very point of extending a language – adding non-termination, or non-determinism, or input-output to a previously pure language can let us distinguish programs that were previously indistinguishable. Full abstraction may not hold in every case, but we designers should know, and let our users know, when it does or does not hold – by intent or by mistake? Below are two reasons to believe that we could design realistic multi-languages into which the component languages embed fully-abstractly.

First, static type systems can make more translations fully-abstract. To define a multi-language formed of statically typed language, one specifies the relation between the types of the two languages. Consider our example of a pure language to which you add input-output. If both pure and impure functions have the same type in the extended system, then the embedding is not fully-abstract; but if an effect system tracks the difference in types, you may interpret functions in the subset as pure functions, and distinguishing contexts in the extended language would be become ill-typed. For an example of the real world, consider the subset of Haskell that does not contain `IO` (input-output as a type-tracked effect): it should embed into full Haskell in a fully-abstract way in absence of the abstraction-leaking `unsafePerformIO`.

Second, the notion full abstraction depends on the specified equivalence relation: you can obtain full abstraction by weakening the source equivalence relation. If you have a language with pure functions, but you wish to add impure functions later, you could specify your language with a weaker equivalence relation, by forbidding equational reasoning that reorder or duplicate function calls. The benefit of forcing you to formulate this weaker equivalence is that it precisely, formally expresses the limit of the equational reasoning principles that the subset-language designer can guarantee to its users.

It would be very optimistic to take an existing, commonly used multi-language system (Python+C, C+Assembly, OCaml+Coq, etc.) and hope for a reasonably concise, clarifying full-abstraction statement. Preservation of user reasoning is currently not enforced by language design, but by social contracts, careful library design and iterative bug fixing. In order to experiment with full abstraction as a guiding principle for multi-language system design, we therefore decided to build our own multi-language, as simple as possible.

We took inspiration from the ML community, where plenty of extension of ML languages were proposed to cover additional problem domains: dependent types for verification, linear types for resource or memory safety, type-and-effect systems, concurrency and parallelism, etc. In the present work, we propose a multi-language semantics for a simple ML language paired with a very simple linearly-typed language, such that ML embeds in the multi-language in a fully-abstract way. Our linear language $\lambda^{\mathbf{L}}$ is sensibly simpler, and in several ways less expressive, than advanced programming languages based on linear logic [11], separation logic [1], fine-grained permissions [6]: it is not designed to stand on its own, but to

$$\begin{array}{llll}
\textit{Types} & \sigma & ::= & \alpha \mid \sigma_1 \times \sigma_2 \mid 1 \mid \sigma_1 \to \sigma_2 \mid \\
 & & & \sigma_1 + \sigma_2 \mid \mu\alpha.\sigma \mid \forall\alpha.\sigma \\
\textit{Expressions} & e & ::= & x \mid \\
 & & & \langle e_1, e_2 \rangle \mid \pi_1\, e \mid \pi_2\, e \mid \\
 & & & \langle\rangle \mid e_1 ; e_2 \mid \\
 & & & \lambda(x:\sigma).\, e \mid e_1\, e_2 \mid \\
 & & & \mathsf{inj}_1\, e \mid \mathsf{inj}_2\, e \mid \mathsf{case}\, e' \,\mathsf{of}\, x_1.\, e_1 \mid x_2.\, e_2 \mid \\
 & & & \mathsf{fold}_{\mu\alpha.\sigma}\, e \mid \mathsf{unfold}\, e \mid \\
 & & & \Lambda\alpha.e \mid e\, [\sigma] \\
\textit{Values} & v & ::= & x \mid \langle v_1, v_2 \rangle \mid \langle\rangle \mid \lambda(x:\sigma).\, e \mid \\
 & & & \mathsf{inj}_1\, v \mid \mathsf{inj}_2\, v \mid \mathsf{fold}_{\mu\alpha.\sigma}\, v \mid \Lambda\alpha.v \\
\textit{Typing contexts} & \Gamma, \Delta & ::= & \cdot \mid \Gamma, x:\sigma \mid \Gamma, \alpha
\end{array}$$

Figure 1. Unrestricted language syntax

serve as a useful side-kick to a functional language, allowing safer resource handling, rather than a language of its own.

We found this experiment encouraging: the programming system that we obtain by mixing two different languages in a careful way has a good power-to-weight ratio. We will present some examples of the useful hybrid programs that can be written in this system. Finally, we hope that this experiment can help us better understand how to gracefully enable interoperation between linearity-agnostic languages and existing languages with linearity, such as Mezzo or Rust.

We claim the following contributions:

1. We design an extremely simple linear language, $\lambda^{\mathbf{L}}$, that supports linear state. This very simple design for linear state is a contribution of its own; in particular, erasing the terms from the linear typing rules gives exactly the standard presentation of linear logic. (Section 2)
2. We present a multi-language programming system $\lambda^{\mathsf{U}\mathbf{L}}$ combining a core ML language, λ^{U} (U for Unrestricted, as opposed to Linear) with $\lambda^{\mathbf{L}}$, such that the embedding of the ML language λ^{U} is fully abstract. (Section 3)
3. We evaluate the resulting language design by providing examples of hybrid $\lambda^{\mathsf{U}\mathbf{L}}$ programs exhibiting various programming patterns inaccessible to ML alone, such that safe in-place updates and typestate-like static protocol enforcement. (Section 4)

2. The λ^{U} and $\lambda^{\mathbf{L}}$ Languages

The unrestricted language λ^{U} is a run-off-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 Figure 1, and the typing rule in Figure 2 – the dynamic semantics are completely standard. Having binary sums, binary products and iso-recursive types lets us express algebraic datatypes in the usual way.

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

$$\frac{x:\sigma \in \Gamma}{\Gamma \vdash_{\mathsf{U}} x : \sigma}$$

$$\frac{\Gamma \vdash_{\mathsf{U}} e_1 : \sigma_1 \qquad \Gamma \vdash_{\mathsf{U}} e_2 : \sigma_2}{\Gamma \vdash_{\mathsf{U}} \langle e_1, e_2 \rangle : \sigma_1 \times \sigma_2} \qquad \frac{\Gamma \vdash_{\mathsf{U}} e : \sigma_1 \times \sigma_2}{\Gamma \vdash_{\mathsf{U}} \pi_i\, e : \sigma_i}$$

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

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

$$\frac{\Gamma \vdash_{\mathsf{U}} e : \sigma_i}{\Gamma \vdash_{\mathsf{U}} \mathsf{inj}_i\, e : \sigma_1 + \sigma_2}$$

$$\frac{\Gamma \vdash_{\mathsf{U}} e : \sigma_1 + \sigma_2 \qquad \begin{array}{c} \Gamma, x_1 : \sigma_1 \vdash_{\mathsf{U}} e_1 : \sigma \\ \Gamma, x_2 : \sigma_2 \vdash_{\mathsf{U}} e_2 : \sigma \end{array}}{\Gamma \vdash_{\mathsf{U}} \mathsf{case}\, e \,\mathsf{of}\, x_1.\, e_1 \mid x_2.\, e_2 : \sigma}$$

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

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

Figure 2. Unrestricted Language: Static Semantics

The novelty lies in the linear language $\lambda^{\mathbf{L}}$, 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 source language in which programs are written, because memory allocation returns *locations* ℓ that are not in the grammar of source terms. Reductions are defined on *configurations*, a local store paired with a term in in a slightly larger *extended* language. We give two type systems, a type system on source terms, that does not mention locations and stores – it is the one the 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.

We present the source language and type system in Section 2.1, except for the language fragment manipulating the linear store which is presented in Section 2.2. Finally, the extended terms, their typing and reduction semantics are presented in Section 2.3.

2.1 The Core of $\lambda^{\mathbf{L}}$

We present in Figure 3 the source syntax of our linear language $\lambda^{\mathbf{L}}$. For the syntactic categories of types $\boldsymbol{\sigma}$, and

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

Figure 3. Linear language: Source syntax

expressions $\mathbf{e}$, the last line contains the constructions related to the linear store that we only discuss in Section 2.2.

In technical terms, our linear type system is exactly propositional intuitionistic linear logic, extended with iso-recursive types. Intuitionistic linear logic is a subset of linear logic, without the negative/additive sums $\sigma_1 \parr \sigma_2$, whose syntax and type system can be easily expressed as a λ-calculus. For simplicity and because we did not need them, our current system also does not have polymorphism or negative/additive/lazy pairs $\sigma_1 \,\&\, \sigma_2$. Additive pairs would be a trivial addition, but polymorphism would require more work to define the multi-language semantics in Section 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 exactly once. Only some types have this linearity restriction, others allow to duplicate and share 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.

The types of linear values are the type of linear pairs $\sigma_1 \otimes \sigma_2$, of linear disjoint unions $\sigma_1 \oplus \sigma_2$ of linear functions $\sigma_1 \multimap \sigma_2$, and of the linear unit type $\mathbf{1}$. For example, a linear function must be called exactly once, and its result must in turn be consumed – such linear functions can safely capture linear resources. The expression-formers at these types use the same syntax as the unrestricted language λ^{U}, with the exception of linear pair deconstruction $\mathbf{let}\ \langle \mathbf{v}_1,\mathbf{v}_2 \rangle = \mathbf{e}_1\ \mathbf{in}\ \mathbf{e}_2$, which names both members of the deconstructed pair at once. A linear pair type with projection would only ever allow to observe one of the two members; this would correspond to the negative/lazy pairs $\sigma_1 \,\&\, \sigma_2$, where only one of the two members is ever computed.

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

$$
\begin{array}{llll}
(\Gamma_1, \mathbf{x}:!\sigma) \curlyvee (\Gamma_2, \mathbf{x}:!\sigma) & \overset{\text{def}}{=} & (\Gamma_1 \curlyvee \Gamma_2), \mathbf{x}:!\sigma & \\
(\Gamma_1, \mathbf{x}:\sigma) \curlyvee \Gamma_2 & \overset{\text{def}}{=} & (\Gamma_1 \curlyvee \Gamma_2), \mathbf{x}:\sigma & (\mathbf{x} \notin \Gamma_2) \\
\Gamma_1 \curlyvee (\Gamma_2, \mathbf{x}:\sigma) & \overset{\text{def}}{=} & (\Gamma_1 \curlyvee \Gamma_2), \mathbf{x}:\sigma & (\mathbf{x} \notin \Gamma_1)
\end{array}
$$

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

$$
\frac{}{!\Gamma, \mathbf{x}:\sigma \vdash_{\mathrm{L}} \mathbf{x} : \sigma}
$$

$$
\frac{\Gamma_1 \vdash_{\mathrm{L}} \mathbf{e}_1 : \sigma_1 \qquad \Gamma_2 \vdash_{\mathrm{L}} \mathbf{e}_2 : \sigma_2}{\Gamma_1 \curlyvee \Gamma_2 \vdash_{\mathrm{L}} \langle \mathbf{e}_1,\mathbf{e}_2 \rangle : \sigma_1 \otimes \sigma_2}
\qquad
\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{L}} \mathbf{e} : \sigma_1 \otimes \sigma_2 \\ \Delta, \mathbf{x}_1:\sigma_1, \mathbf{x}_2:\sigma_2 \vdash_{\mathrm{L}} \mathbf{e}' : \sigma\end{array}}{\Gamma \curlyvee \Delta \vdash_{\mathrm{L}} \mathbf{let}\ \langle \mathbf{x}_1,\mathbf{x}_2 \rangle = \mathbf{e}\ \mathbf{in}\ \mathbf{e}' : \sigma}
$$

$$
\frac{}{!\Gamma \vdash_{\mathrm{L}} \langle\rangle : \mathbf{1}}
\qquad
\frac{\Gamma \vdash_{\mathrm{L}} \mathbf{e} : \mathbf{1} \qquad \Delta \vdash_{\mathrm{L}} \mathbf{e}' : \sigma}{\Gamma \curlyvee \Delta \vdash_{\mathrm{L}} \mathbf{e};\mathbf{e}' : \sigma}
$$

$$
\frac{\Gamma, \mathbf{x}:\sigma \vdash_{\mathrm{L}} \mathbf{e} : \sigma'}{\Gamma \vdash_{\mathrm{L}} \lambda(\mathbf{x}:\sigma).\,\mathbf{e} : \sigma \multimap \sigma'}
$$

$$
\frac{\Gamma \vdash_{\mathrm{L}} \mathbf{e} : \sigma' \multimap \sigma \qquad \Delta \vdash_{\mathrm{L}} \mathbf{e}' : \sigma'}{\Gamma \curlyvee \Delta \vdash_{\mathrm{L}} \mathbf{e}\ \mathbf{e}' : \sigma}
$$

$$
\frac{\Gamma \vdash_{\mathrm{L}} \mathbf{e} : \sigma_i}{\Gamma \vdash_{\mathrm{L}} \mathbf{inj_i}\,\mathbf{e} : \sigma_1 \oplus \sigma_2}
\qquad
\frac{\Gamma \vdash_{\mathrm{L}} \mathbf{e} : \sigma_1 \oplus \sigma_2 \qquad \begin{array}{c}\Delta, \mathbf{x}_1 : \sigma_1 \vdash_{\mathrm{L}} \mathbf{e}_1 : \sigma \\ \Delta, \mathbf{x}_2 : \sigma_2 \vdash_{\mathrm{L}} \mathbf{e}_2 : \sigma\end{array}}{\Gamma \curlyvee \Delta \vdash_{\mathrm{L}} \mathbf{case}\,\mathbf{e}\ \mathbf{of}\ \mathbf{x}_1.\,\mathbf{e}_1 \mid \mathbf{x}_2.\,\mathbf{e}_2 : \sigma}
$$

$$
\frac{!\Gamma \vdash_{\mathrm{L}} \mathbf{e} : \sigma}{!\Gamma \vdash_{\mathrm{L}} \mathbf{share}(\varsigma\!:\!\Phi).\,\mathbf{e} : !\sigma}
\qquad
\frac{\Gamma \vdash_{\mathrm{L}} \mathbf{e} : !\sigma}{\Gamma \vdash_{\mathrm{L}} \mathbf{copy}^{\sigma}\ \mathbf{e} : \sigma}
$$

$$
\mu\alpha.\sigma \quad \underset{\mathbf{fold}_{\mu\alpha.\sigma}}{\overset{\mathbf{unfold}}{\rightleftarrows}} \quad \sigma[\mu\alpha.\sigma/\alpha]
$$

Figure 4. Linear Language: Source Static Semantics

The types of non-linear, duplicable values are the types of the form $!\sigma$ – the *exponential* modality of linear logic. If $\mathbf{e}$ has type σ, the term $\mathbf{share}\,\mathbf{e}$ has type $!\sigma$. Values of this type are not uniquely owned, they can be shared at will. If the term $\mathbf{e}$ has duplicable type $!\sigma$, then the type $\mathbf{copy}\ \mathbf{e}$ 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 source typing rules of $\lambda^{\mathbf{L}}$, presented in Figure 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 $\mathbf{e}_1$ is type-checked in the context Γ_1 and $\mathbf{e}_2$ in Γ_2, then the linear pair $\langle \mathbf{e}_1,\mathbf{e}_2 \rangle$ 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

$$\sigma ::= \ldots \mid \mathbf{Box}\ b\ \sigma \qquad\qquad b ::= 0 \mid 1$$

$$1 \ \underset{\mathbf{free}}{\overset{\mathbf{new}}{\multimap \atop \multimapinv}}\ \mathbf{Box}\ 0\ \sigma \qquad\qquad \mathbf{Box}\ 1\ \sigma \ \underset{\mathbf{box}}{\overset{\mathbf{unbox}}{\multimap \atop \multimapinv}}\ \mathbf{Box}\ 0\ \sigma \otimes \sigma$$

Figure 5. Linear language: Store Source Static Semantics

in $\Gamma_1 \curlyvee \Gamma_2$ cannot possibly appear in both Γ_1 and Γ_2: it must appear exactly once[3]. A good way to think of the linear judgment $\Gamma \vdash_L e : \sigma$ is that the evaluation of e *consumes* the linear variables of Γ; it is thus natural that the strict pair $\langle e_1, e_2 \rangle$ would need separate set of resources Γ_1 and Γ_2, as it evaluates both members to return a value. On the other hand, case elimination $\mathbf{case}\, e\, \mathbf{of}\ x_1 . e_1 \mid x_2 . e_2$ reuses the same context Δ in both branches e_1 and e_2: only one will be evaluated, so they do not compete for resources.

The variable rule does not expect a context of the form $\Gamma, x : \sigma$ but of the form $!\Gamma, x : \sigma$. $!\Gamma$ is a notation for the pair-wise application of the $(!)$ connective to all types of Γ – all types in $!\Gamma$ are of the form $!\sigma$. This means that the variable rule can only be used when all variables in the context are duplicable, except maybe the variable that is being used. A context of the form $\Gamma, x : \sigma$ would allow to forget some variable present in the context; in our judgment $\Gamma \vdash_L e : \sigma$, all non-duplicable variables in Γ must appear (once) in e.

The form $!\Gamma$ is also used in the typing rule for $\mathbf{share}\, e$: 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 Figure 4 defines them as primitive functions, at the given linear function type, in the empty context – using they does not consume resources. This notation also means that, operationally, these two operations shall be inverse of each other.

Lemma 2.1 (Context joining properties)
Context joining $(\curlyvee)$ is partial but associative and commutative. In particular, if $(\Gamma_1 \curlyvee \Gamma_2) \curlyvee \Delta$ is defined, then both $\Gamma_i \curlyvee \Delta$ are defined.

2.2 Linear Memory in λ^L

The source typing rules for the linear store are given in Figure 5. The linear type $\mathbf{Box}\ b\ \sigma$ represents a memory location that may hold a value of type σ. The parameter b is boolean: if it is 0, then the location is empty, it does not contain a value, and if it is 1 the location currently contains a value. The primitive operations to act on this type are given as linear isomorphisms: **new** turns a unit value into an empty location, it allocates; 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 $\mathbf{Box}\ 0\ \sigma \otimes \sigma$, and a full location, of type $\mathbf{Box}\ 1\ \sigma$.

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

$$\lambda(p : \mathbf{Box}\ 1\ \sigma \otimes \sigma).\ \begin{array}{l} \mathbf{let}\ \langle r, x \rangle = p\ \mathbf{in} \\ \mathbf{let}\ \langle l, xl \rangle = \mathbf{unbox}\ r\ \mathbf{in} \\ \langle \mathbf{box}\ \langle l, x \rangle, xl \rangle \end{array}$$

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 $\mathbf{Box}\ b\ \sigma$ 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 explicitating allocations and dereferences happening in higher-level purely functional language – in this view, empty cells allow memory reuse.

This view that $\mathbf{Box}\ b\ \sigma$ represents indirection through the memory suggests to encode lists of values of type σ by the type $\mathbf{LinList}\ \sigma \overset{\text{def}}{=} \mu\alpha. 1 \oplus \mathbf{Box}\ 1\ (\sigma \otimes \alpha)$ – 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 a in-place reverse function on lists of σ as follows:

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

This definition uses a fixpoint operator **fix** that can be defined, in the standard way, using the iso-recursive type $\mu\alpha.\alpha \multimap \sigma \multimap \sigma'$ of the strict fixpoint combinator on functions $\sigma \multimap \sigma'$.

Our linear λ-calculus 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:

$$\begin{array}{llll}
rev\_into & \mathbf{Nil} & acc & = acc \\
rev\_into & \mathbf{Cons}\ \langle x, xs \rangle @ l & acc & = rev\_into\ xs\ \langle x, acc \rangle @ l
\end{array}$$

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.

[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

This double view of linear state occurs in other programming systems with linear state. It was recently emphasized in O'Connor, Chen, Rizkallah, Amani, Lim, Murray, Nagashima, Sewell, and Klein [9], where the functional point 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 Extended Terms Syntax Typing

To give a dynamic semantics and prove it sound, we need to extend the language with explicit stores and store locations. Indeed, the allocating term $\mathbf{new}\ \langle\rangle$ should reduce to a "fresh location" ℓ allocated in some store ς, and neither are part of the source term syntax. The corresponding extended typing judgment is more complex, but remember that users do not need to know about it to reason about correctness of source program. It could be hidden in a soundness proof, but is also very useful to define a multi-language semantics in Section 3. Besides, the additional structure is designed to help reasoning about soundness during program execution; the same concepts may also help the programmer reason about program execution.

The syntax of extended terms and the extended type system are presented in Figure 6. Reduction will be defined on *configurations* $(\varsigma \mid \mathbf{e})$, which are pairs of a store ς and a term $\mathbf{e}$. Stores ς map *locations* ℓ to either nothing (the location is empty), written $[\ell \mapsto \cdot]$, or a value paired its own local store, noted $[\ell \mapsto (\varsigma \mid \mathbf{v})]$. Having local stores in this way, instead of a single global store as is typical in formalizations of ML, directly expresses the idea of "memory ownership" in the syntax: a term $\mathbf{e}$ "owns" the locations that appear in it, and a configuration $(\varsigma \mid \mathbf{e})$ is only well-typed if the domain of ς is exactly those locations. Each store slot, in turn, may contain value and the local store owned by the value; in particular, passing a full location of type $\mathbf{Box\ 1}\ \boldsymbol{\sigma}$ transfers ownership of the location, but also of the store fragment captured by the value.

Our extended type judgment $\boldsymbol{\Phi} \mid \boldsymbol{\Gamma} \vdash_{\mathrm{L}} \varsigma \mid \mathbf{e} : \boldsymbol{\sigma}$ checks configurations, not just terms, and relies not only on a typing context for variables $\boldsymbol{\Gamma}$ but also on a *store typing* $\boldsymbol{\Phi}$, which maps the locations of the configuration to typing assumptions of two forms: $(\cdot \mid \cdot \vdash \ell : \mathbf{Box\ 0}\ \boldsymbol{\sigma})$ indicates that ℓ must be empty in the configuration, and $(\boldsymbol{\Gamma} \mid \boldsymbol{\Phi} \vdash \ell : \mathbf{Box\ 1}\ \boldsymbol{\sigma})$ indicates that ℓ is full, and that the value it contains owns a local store of type ς and the resources in $\boldsymbol{\Gamma}$.

Just as linear variables must occur exactly once in a term, locations have linear types and thus occur exactly once in a term. Our typing judgment uses disjoint of store typings $\boldsymbol{\Phi}_1 \uplus \boldsymbol{\Phi}_2$ to enforce this linearity. Similarly, leaf rules such as the variable, unit and location rules enforce that both the store typing and the store be empty, which enforces that all locations are used in the term.

Locations ℓ are always linear, never duplicable. To allow sharing terms that contain locations, the extended language uses the extended construction $\mathbf{share}(\varsigma{:}\boldsymbol{\Phi}).\,\mathbf{e}$, that *captures*

Types	$\boldsymbol{\sigma}$	(unchanged from Figure 4)
Expressions	$\mathbf{e}$	$\ldots \mid \ell \mid \mathbf{share}(\varsigma{:}\boldsymbol{\Phi}).\,\mathbf{e}$
		with $\mathbf{share}\,\mathbf{e} \overset{\text{def}}{=} \mathbf{share}(\emptyset{:}\cdot).\,\mathbf{e}$
Values	$\mathbf{v}$	$\ldots \mid \ell \mid \mathbf{share}(\varsigma{:}\boldsymbol{\Phi}).\,\mathbf{v}$
Store	ς	$::= \emptyset \mid \varsigma[\ell \mapsto (\varsigma \mid \mathbf{v})] \mid \varsigma[\ell \mapsto \cdot]$
Configurations		$::= (\varsigma \mid \mathbf{e})$
Store typing	$\boldsymbol{\Phi}, \boldsymbol{\Psi}$	$::= \cdot \mid \boldsymbol{\Phi}, (\cdot \mid \cdot \vdash \ell : \mathbf{Box\ 0}\ \boldsymbol{\sigma})$
		$\mid \boldsymbol{\Phi}, (\boldsymbol{\Phi} \mid \boldsymbol{\Gamma} \vdash \ell : \mathbf{Box\ 1}\ \boldsymbol{\sigma})$

$\boxed{\boldsymbol{\Phi}_1 \uplus \boldsymbol{\Phi}_2}$ Disjoint union of (location $\mapsto$ judgments) mappings

$\boxed{\boldsymbol{\Phi} \mid \boldsymbol{\Gamma} \vdash_{\mathrm{L}} \varsigma \mid \mathbf{e} : \boldsymbol{\sigma}}$ $\boxed{\boldsymbol{\Gamma} \vdash_{\mathrm{L}} \mathbf{e} : \boldsymbol{\sigma} \overset{\text{def}}{=} \cdot \mid \boldsymbol{\Gamma} \vdash_{\mathrm{L}} \emptyset \mid \mathbf{e} : \boldsymbol{\sigma}}$

$$\frac{}{\cdot \mid !\boldsymbol{\Gamma}, \mathbf{x}:\boldsymbol{\sigma} \vdash_{\mathrm{L}} \emptyset \mid \mathbf{x} : \boldsymbol{\sigma}}$$

$$\frac{\boldsymbol{\Phi}_1 \mid \boldsymbol{\Gamma}_1 \vdash_{\mathrm{L}} \varsigma_1 \mid \mathbf{e}_1 : \boldsymbol{\sigma}_1 \qquad \boldsymbol{\Phi}_2 \mid \boldsymbol{\Gamma}_2 \vdash_{\mathrm{L}} \varsigma_2 \mid \mathbf{e}_2 : \boldsymbol{\sigma}_2}{\boldsymbol{\Phi}_1 \uplus \boldsymbol{\Phi}_2 \mid \boldsymbol{\Gamma}_1 \curlyvee \boldsymbol{\Gamma}_2 \vdash_{\mathrm{L}} \varsigma_1 \mathbin{+\!\!\!+} \varsigma_2 \mid \langle \mathbf{e}_1, \mathbf{e}_2 \rangle : \boldsymbol{\sigma}_1 \otimes \boldsymbol{\sigma}_2}$$

$$\frac{\begin{array}{c}\boldsymbol{\Phi} \mid \boldsymbol{\Gamma} \vdash_{\mathrm{L}} \varsigma \mid \mathbf{e} : \boldsymbol{\sigma}_1 \otimes \boldsymbol{\sigma}_2 \\ \boldsymbol{\Psi} \mid \boldsymbol{\Delta}, \mathbf{x}_1{:}\boldsymbol{\sigma}_1, \mathbf{x}_2{:}\boldsymbol{\sigma}_2 \vdash_{\mathrm{L}} \varsigma' \mid \mathbf{e}' : \boldsymbol{\sigma}\end{array}}{\boldsymbol{\Phi} \uplus \boldsymbol{\Psi} \mid \boldsymbol{\Gamma} \curlyvee \boldsymbol{\Delta} \vdash_{\mathrm{L}} \varsigma \mathbin{+\!\!\!+} \varsigma' \mid \mathbf{let}\ \langle \mathbf{x}_1, \mathbf{x}_2 \rangle = \mathbf{e}\ \mathbf{in}\ \mathbf{e}' : \boldsymbol{\sigma}}$$

$$\frac{}{\cdot \mid !\boldsymbol{\Gamma} \vdash_{\mathrm{L}} \langle\rangle \mid \emptyset : \mathbf{1}}$$

$$\frac{\boldsymbol{\Phi} \mid \boldsymbol{\Gamma} \vdash_{\mathrm{L}} \varsigma \mid \mathbf{e} : \mathbf{1} \qquad \boldsymbol{\Psi} \mid \boldsymbol{\Delta} \vdash_{\mathrm{L}} \varsigma' \mid \mathbf{e}' : \boldsymbol{\sigma}}{\boldsymbol{\Phi} \uplus \boldsymbol{\Psi} \mid \boldsymbol{\Gamma} \curlyvee \boldsymbol{\Delta} \vdash_{\mathrm{L}} \varsigma \mathbin{+\!\!\!+} \varsigma' \mid \mathbf{e}; \mathbf{e}' : \boldsymbol{\sigma}}$$

$$\frac{\boldsymbol{\Phi} \mid \boldsymbol{\Gamma}, \mathbf{x}:\boldsymbol{\sigma} \vdash_{\mathrm{L}} \varsigma \mid \mathbf{e} : \boldsymbol{\sigma}'}{\boldsymbol{\Phi} \mid \boldsymbol{\Gamma} \vdash_{\mathrm{L}} \varsigma \mid \lambda(\mathbf{x}:\boldsymbol{\sigma}).\,\mathbf{e} : \boldsymbol{\sigma} \multimap \boldsymbol{\sigma}'}$$

$$\frac{\boldsymbol{\Phi} \mid \boldsymbol{\Gamma} \vdash_{\mathrm{L}} \varsigma \mid \mathbf{e} : \boldsymbol{\sigma}' \multimap \boldsymbol{\sigma} \qquad \boldsymbol{\Psi} \mid \boldsymbol{\Delta} \vdash_{\mathrm{L}} \varsigma' \mid \mathbf{e}' : \boldsymbol{\sigma}'}{\boldsymbol{\Phi} \uplus \boldsymbol{\Psi} \mid \boldsymbol{\Gamma} \curlyvee \boldsymbol{\Delta} \vdash_{\mathrm{L}} \varsigma \mathbin{+\!\!\!+} \varsigma' \mid \mathbf{e}\ \mathbf{e}' : \boldsymbol{\sigma}}$$

$$\frac{\boldsymbol{\Phi} \mid \boldsymbol{\Gamma} \vdash_{\mathrm{L}} \varsigma \mid \mathbf{e} : \boldsymbol{\sigma}_i}{\boldsymbol{\Phi} \mid \boldsymbol{\Gamma} \vdash_{\mathrm{L}} \varsigma \mid \mathbf{inj}_{\mathbf{i}}\,\mathbf{e} : \boldsymbol{\sigma}_1 \oplus \boldsymbol{\sigma}_2}$$

$$\frac{\begin{array}{c}\boldsymbol{\Phi} \mid \boldsymbol{\Gamma} \vdash_{\mathrm{L}} \varsigma \mid \mathbf{e} : \boldsymbol{\sigma}_1 \oplus \boldsymbol{\sigma}_2 \qquad \boldsymbol{\Psi} \mid \boldsymbol{\Delta}, \mathbf{x}_1 : \boldsymbol{\sigma}_1 \vdash_{\mathrm{L}} \varsigma' \mid \mathbf{e}_1 : \boldsymbol{\sigma} \\ \boldsymbol{\Psi} \mid \boldsymbol{\Delta}, \mathbf{x}_2 : \boldsymbol{\sigma}_2 \vdash_{\mathrm{L}} \varsigma' \mid \mathbf{e}_2 : \boldsymbol{\sigma}\end{array}}{\boldsymbol{\Phi} \uplus \boldsymbol{\Psi} \mid \boldsymbol{\Gamma} \curlyvee \boldsymbol{\Delta} \vdash_{\mathrm{L}} \varsigma \mathbin{+\!\!\!+} \varsigma' \mid \mathbf{case}\,\mathbf{e}\ \mathbf{of}\ \mathbf{x}_1.\,\mathbf{e}_1 \mid \mathbf{x}_2.\,\mathbf{e}_2 : \boldsymbol{\sigma}}$$

$$\boldsymbol{\mu\alpha.\sigma} \quad \underset{\mathbf{fold}_{\mu\alpha.\sigma}}{\overset{\mathbf{unfold}}{\rightleftarrows}} \quad \boldsymbol{\sigma}[\boldsymbol{\mu\alpha.\sigma}/\boldsymbol{\alpha}]$$

$$\frac{\boldsymbol{\Phi} \mid !\boldsymbol{\Gamma} \vdash_{\mathrm{L}} \varsigma \mid \mathbf{e} : \boldsymbol{\sigma}}{\cdot \mid !\boldsymbol{\Gamma} \vdash_{\mathrm{L}} \emptyset \mid \mathbf{share}(\varsigma{:}\boldsymbol{\Phi}).\,\mathbf{e} : !\boldsymbol{\sigma}}$$

$$\frac{\boldsymbol{\Phi} \mid \boldsymbol{\Gamma} \vdash_{\mathrm{L}} \varsigma \mid \mathbf{e} : !\boldsymbol{\sigma}}{\boldsymbol{\Phi} \mid \boldsymbol{\Gamma} \vdash_{\mathrm{L}} \varsigma \mid \mathbf{copy}^{\boldsymbol{\sigma}}\ \mathbf{e} : \boldsymbol{\sigma}}$$

$$\frac{}{(\cdot \mid \cdot \vdash \ell : \mathbf{Box\ 0}\ \boldsymbol{\sigma}) \mid !\boldsymbol{\Gamma} \vdash_{\mathrm{L}} [\ell \mapsto \cdot] \mid \ell : \mathbf{Box\ 0}\ \boldsymbol{\sigma}}$$

$$\frac{\boldsymbol{\Phi} \mid \boldsymbol{\Gamma} \vdash_{\mathrm{L}} \varsigma \mid \mathbf{v} : \boldsymbol{\sigma}}{(\boldsymbol{\Phi} \mid \boldsymbol{\Gamma} \vdash \ell : \mathbf{Box\ 1}\ \boldsymbol{\sigma}) \mid \boldsymbol{\Gamma} \curlyvee !\boldsymbol{\Gamma}' \vdash_{\mathrm{L}} [\ell \mapsto (\varsigma \mid \mathbf{v})] \mid \ell : \mathbf{Box\ 1}\ \boldsymbol{\sigma}}$$

$$\mathbf{1} \quad \underset{\mathbf{free}}{\overset{\mathbf{new}}{\rightleftarrows}} \quad \mathbf{Box\ 0}\ \boldsymbol{\sigma} \qquad\qquad \mathbf{Box\ 1}\ \boldsymbol{\sigma} \quad \underset{\mathbf{box}}{\overset{\mathbf{unbox}}{\rightleftarrows}} \quad \mathbf{Box\ 0}\ \boldsymbol{\sigma} \otimes \boldsymbol{\sigma}$$

a local store $\varsigma : \Phi$. This notation is a binding construct: the locations in ς are bound by this shared term, and not visible outside this term. In particular, the typing rule for $\mathbf{share}(\varsigma{:}\Phi).\,e$ checks the term e in the store ς, but it is itself only valid paired with an empty store, under the empty store typing. When new copies of a shared term are made, the local store is copied as well: this is necessary to guarantee that locations remain linear – and for correctness of linear state update.

The typing rule for functions $\lambda(x : \sigma).\,e$ lets function bodies use an arbitrary store typing Φ. This would be unsound if our functions were duplicable, but it is a natural and expressive choice for linear, one-shot functions. To make a function duplicable, one can share it at type $!(\sigma \multimap \sigma')$, whose values are of the canonical form $\mathbf{share}(\varsigma{:}\Phi).\,\lambda(x : \sigma).\,e$. It is the sharing construct, not the function itself, that closes over the local store.

With the macro-expansion $\mathbf{share}\,e \overset{\text{def}}{=} \mathbf{share}(\emptyset{:}\cdot).\,e$, any term e of the source language (Figure 3 can be seen as a term of the extended language (Figure 6). In particular, we can prove that the source and extended typing judgments coincide on source terms.

Lemma 2.2
If e is a source term of λ^{L}, then the source judgment $\Gamma \vdash_{L} e : \sigma$ holds if and only if the extended judgment $\cdot \mid \Gamma \vdash_{L} \emptyset \mid e : \sigma$ holds.

The following technical results are used in the soundness proof for the language, Theorem 2.8 (Subject reduction for λ^{L}).

Lemma 2.3 (Inversion principle for λ^{L} values)
In any complete derivation of $\Phi \mid \Gamma \vdash_{L} \varsigma \mid v : \sigma$, either v is a variable x, or the derivation starts with the introduction rule for σ.

For example, if we have $\Phi \mid \Gamma \vdash_{L} \varsigma \mid v : !\sigma$, then we know that v is either a variable or of the form $\mathbf{share}(\varsigma'{:}\Psi).\,v'$ for some v', but also that $\varsigma = \emptyset$, $\Phi = \cdot$ and that Γ is of the form $!\Gamma'$ for some Γ'. The latter is immediate if v is $\mathbf{share}(\varsigma'{:}\Psi).\,v'$, and also holds if v is a variable.

Lemma 2.4 (Weakening of duplicable contexts)
$\Phi \mid \Delta \vdash_{L} \varsigma \mid e : \sigma$ *implies* $\Phi \mid !\Gamma, \Delta \vdash_{L} \varsigma \mid e : \sigma$.

2.4 Reduction of Extended Terms

Figure 7 gives a small-step operational semantics for the extended terms of λ^{L}. We separate the head reductions ($\overset{L}{\rightsquigarrow}$) from reductions in depth ($\overset{L}{\hookrightarrow}$). The head reduction of the linear types of the core language do not involve the store and are standard. For the store primitives of Figure 5 acting on $\mathbf{Box}\ b\ \sigma$, we reuse the isomorphism notation to emphasize that the related primitives are inverse of each other.

There are several reduction rules for $\mathbf{copy}\ (\mathbf{share}\,e)$, one for each type connective. These reductions perform a

head reduction $\boxed{e \overset{L}{\rightsquigarrow} e'}$ $\boxed{(\varsigma \mid e) \overset{L}{\rightsquigarrow} (\varsigma' \mid e')}$

$$\begin{array}{rcl}
\mathbf{let}\ \langle x_1, x_2\rangle = \langle v_1, v_2\rangle\ \mathbf{in}\ e & \overset{L}{\rightsquigarrow} & e[v_1/x_1][v_2/x_2] \\
\langle\rangle ; e & \overset{L}{\rightsquigarrow} & e \\
(\lambda(x : \sigma).\,e)\ v & \overset{L}{\rightsquigarrow} & e[v/x] \\
\mathbf{case}\ (\mathbf{inj_i}\ v)\ \mathbf{of}\ x_1.\,e_1 \mid x_2.\,e_2 & \overset{L}{\rightsquigarrow} & e_i[v/x_i] \\
\mathbf{unfold}\ (\mathbf{fold}_{\mu\alpha.\sigma}\ v) & \overset{L}{\rightsquigarrow} & v
\end{array}$$

$$\frac{e \overset{L}{\rightsquigarrow} e'}{(\varsigma \mid e) \overset{L}{\rightsquigarrow} (\varsigma \mid e')} \qquad (\emptyset \mid \langle\rangle)\ \underset{\underset{\mathbf{free}}{L}}{\overset{\overset{\mathbf{new}}{L}}{\leftrightarrows}}\ ([\ell \mapsto \cdot] \mid \ell)$$

$$(\varsigma[\ell \mapsto \cdot] \mid \langle \ell, v\rangle)\ \underset{\underset{\mathbf{unbox}}{L}}{\overset{\overset{\mathbf{box}}{L}}{\leftrightarrows}}\ ([\ell \mapsto (\varsigma \mid v)] \mid \ell)$$

$$\begin{array}{l}
\mathbf{copy}\ (\mathbf{share}(\varsigma_1 \mathbin{+\!\!+} \varsigma_2{:}\Phi_1 \uplus \Phi_2).\,\langle v_1, v_2\rangle) \\
\overset{L}{\rightsquigarrow} \qquad \text{if } \mathsf{locs}(\varsigma_i) = \mathsf{locs}(\Phi_i) = \mathsf{locs}(v_i) \\
\langle \mathbf{copy}\ \mathbf{share}(\varsigma_1{:}\Phi_1).\,v_1, \mathbf{copy}\ \mathbf{share}(\varsigma_2{:}\Phi_2).\,v_2\rangle
\end{array}$$

$$\begin{array}{ll}
\mathbf{copy}\ (\mathbf{share}(\emptyset{:}\cdot).\,\langle\rangle) & \overset{L}{\rightsquigarrow} \langle\rangle \\
\mathbf{copy}\ (\mathbf{share}(\varsigma{:}\Phi).\,\mathbf{inj_i}\ v) & \overset{L}{\rightsquigarrow} \mathbf{inj_i}\ (\mathbf{share}(\varsigma{:}\Phi).\,v) \\
\mathbf{copy}\ (\mathbf{share}(\varsigma{:}\Phi).\,\mathbf{fold}\ v) & \overset{L}{\rightsquigarrow} \mathbf{fold}\ (\mathbf{copy}\ (\mathbf{share}(\varsigma{:}\Phi).\,v))
\end{array}$$

$$(\emptyset \mid \mathbf{copy}\ (\mathbf{share}(\varsigma{:}\Phi).\,\lambda(x : \sigma).\,e)) \overset{L}{\rightsquigarrow} (\varsigma \mid \lambda(x : \sigma).\,e)$$

$$\mathbf{copy}\ (\mathbf{share}(\emptyset{:}\cdot).\,(\mathbf{share}(\varsigma{:}\Phi).\,v)) \overset{L}{\rightsquigarrow} \mathbf{share}(\varsigma{:}\Phi).\,v$$

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

$$\begin{array}{l}
\mathbf{copy}\ (\mathbf{share}([\ell \mapsto (\varsigma \mid v)]{:}(\Phi \mid !\Gamma \vdash \ell : \mathbf{Box}\ 1\ \sigma)).\,\ell) \\
\qquad \overset{L}{\rightsquigarrow} \mathbf{box}\ \langle \mathbf{new}\ \langle\rangle, \mathbf{copy}\ (\mathbf{share}(\varsigma{:}\Phi).\,v)\rangle
\end{array}$$

linear reduction contexts $\boxed{\Phi \mid \Gamma \vdash_{L} \varsigma \mid K[\square{:}\sigma] : \sigma'}$

$$\begin{array}{rl}
K ::= & \square{:}\sigma \mid \langle K, e_2\rangle \mid \langle v, K\rangle \mid \mathbf{let}\ \langle v_1, v_2\rangle = K\ \mathbf{in}\ e_2 \mid \\
& K ; e \mid K\ e \mid v\ K \mid \mathbf{copy}\ K \mid \\
& \mathbf{inj_1}\ K \mid \mathbf{inj_2}\ K \mid \mathbf{case}\ K\ \mathbf{of}\ x_1.\,e_1 \mid x_2.\,e_2 \mid \\
& \mathbf{fold}_{\mu\alpha.\sigma}\ K \mid \mathbf{unfold}\ K \mid \\
& \mathbf{new}\ K \mid \mathbf{free}\ K \mid \mathbf{box}\ K \mid \mathbf{unbox}\ K
\end{array}$$

typing rules of terms, plus: $\dfrac{}{\cdot \mid \cdot \vdash_{L} \emptyset \mid (\square{:}\sigma) : \sigma}$

reduction $\boxed{(\varsigma \mid e) \overset{L}{\hookrightarrow} (\varsigma' \mid e')}$

$$\frac{(\varsigma \mid e) \overset{L}{\rightsquigarrow} (\varsigma' \mid e')}{(\varsigma \mid e) \overset{L}{\hookrightarrow} (\varsigma' \mid e')} \qquad \frac{\Psi \mid \Gamma \vdash_{L} \varsigma'' \mid K[\square{:}\sigma] : \sigma' \qquad (\varsigma \mid e) \overset{L}{\hookrightarrow} (\varsigma' \mid e')}{(\varsigma'' \mathbin{+\!\!+} \varsigma \mid K[e]) \overset{L}{\hookrightarrow} (\varsigma'' \mathbin{+\!\!+} \varsigma' \mid K[e'])}$$

$$\frac{\begin{array}{c}\Phi \mid \Gamma \vdash_{L} \varsigma \mid e : \sigma \\ (\varsigma \mid e) \overset{L}{\hookrightarrow} (\varsigma' \mid e') \qquad \Phi' \mid \Gamma \vdash_{L} \varsigma' \mid e' : \sigma\end{array}}{(\emptyset \mid \mathbf{share}(\varsigma{:}\Phi).\,e) \overset{L}{\hookrightarrow} (\emptyset \mid \mathbf{share}(\varsigma'{:}\Phi').\,e')}$$

Figure 7. Linear language: Operational Semantics

deep copy of the value, stopping only on ground data ($\langle\rangle$), function values, and shared sub-terms: when copying a $\mathbf{!!\sigma}$ into a $\mathbf{!\sigma}$, there is no need for a deep copy. When it encounters a location, $\mathbf{copy}\ (\mathbf{share}\,\ell)$ reduces to a new allocation; if the location contains a value, the new location is filled with a copy of this value.

The copying rule for functions performs a copy of the local store ς of the shared function: the locations in ς are bound on the left-hand-side of the reduction, and free on the right-hand-side: this reduction step allocates fresh locations, and the store typing of the term changes from $\cdot$ on the left to $\mathbf{\Phi}$ on the right. The fact that reduction changes the store typing is not unique to this rule, it is also the case when directly copying locations. In ML languages with references, the store only grows during reduction, it is not the case for our linear store: our reduction may either allocate new locations or free existing ones.

We define a grammar of (deterministic) reduction contexts, which contain exactly one hole $\square$ in evaluation position. However, we only define *linear* contexts $\mathbf{K}$ that do not share their hole: we need a specific treatment of the $\mathbf{share(\varsigma{:}\Phi).e}$ reduction. Its subterm $\mathbf{e}$ is reduced in the local store ς, but may create or free locations in the store; so we need to update the local store and its store typing during the reduction.

Theorem 2.5 (Progress)
If $\mathbf{\Phi} \mid \mathbf{\Gamma} \vdash_{\mathrm{L}} \varsigma \mid \mathbf{e} : \mathbf{\sigma}$*, then either* $\mathbf{e}$ *is a value* $\mathbf{v}$ *or there exists* $(\varsigma' \mid \mathbf{e}')$ *such that* $(\varsigma \mid \mathbf{e}) \overset{\mathbf{L}}{\hookrightarrow} (\varsigma' \mid \mathbf{e}')$.

Lemma 2.6 (Non-store-escaping substitution principle)
If $\quad \mathbf{\Phi} \mid \mathbf{\Gamma}, \mathbf{x} : \mathbf{\sigma} \vdash_{\mathrm{L}} \varsigma' \mid \mathbf{e} : \mathbf{\sigma}' \qquad \mathbf{\Psi} \mid \mathbf{\Delta} \vdash_{\mathrm{L}} \varsigma \mid \mathbf{v} : \mathbf{\sigma}$

$$\mathbf{\Gamma} \curlyvee \mathbf{\Delta} \qquad\qquad \mathbf{x} \notin \mathbf{\Phi}$$

then $\quad \mathbf{\Phi} \uplus \mathbf{\Psi} \mid \mathbf{\Gamma} \curlyvee \mathbf{\Delta} \vdash_{\mathrm{L}} \varsigma \mathbin{+\!\!+} \varsigma' \mid \mathbf{e}[\mathbf{v}/\mathbf{x}] : \mathbf{\sigma}'$

Lemma 2.7 (Context decomposition)
If $\mathbf{\Phi}' \mid \mathbf{\Gamma}' \vdash_{\mathrm{L}} \varsigma' \mid \mathbf{K}[\square{:}\mathbf{\sigma}] : \mathbf{\sigma}'$ *holds, then* $\mathbf{\Phi}'' \mid \mathbf{\Gamma}'' \vdash_{\mathrm{L}} \varsigma'' \mid \mathbf{K}[\mathbf{e}] : \mathbf{\sigma}'$ *holds if and only if there exists* $\mathbf{\Phi}, \mathbf{\Gamma}, \varsigma$ *such that* $\mathbf{\Phi}'' = \mathbf{\Phi} \uplus \mathbf{\Phi}'$, $\mathbf{\Gamma}'' = \mathbf{\Gamma} \curlyvee \mathbf{\Gamma}'$, $\varsigma'' = \varsigma \mathbin{+\!\!+} \varsigma'$ *and* $\mathbf{\Phi} \mid \mathbf{\Gamma} \vdash_{\mathrm{L}} \varsigma \mid \mathbf{e} : \mathbf{\sigma}$.

Theorem 2.8 (Subject reduction for $\lambda^{\mathbf{L}}$)
If $\mathbf{\Phi} \mid \mathbf{\Gamma} \vdash_{\mathrm{L}} \varsigma \mid \mathbf{e} : \mathbf{\sigma}$ *and* $(\varsigma \mid \mathbf{e}) \overset{\mathbf{L}}{\hookrightarrow} (\varsigma' \mid \mathbf{e}')$*, then there exists a (unique)* $\mathbf{\Phi}'$ *such that* $\mathbf{\Phi}' \mid \mathbf{\Gamma} \vdash_{\mathrm{L}} \varsigma' \mid \mathbf{e}' : \mathbf{\sigma}$.

3. Multi-Language Semantics

To formally define our multi-language semantics we create a combined language $\lambda^{\mathrm{U}\mathbf{L}}$ which let us compose term fragments from both λ^{U} and $\lambda^{\mathbf{L}}$ together, and we give an operational semantics to this combined language. Interaction 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 language 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 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 [2], is difficult because of semantic mismatches. In Section 4 (Hybrid program examples) 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 Figure 8 – the communication of values from one language to the other is only described in the next section. The multi-language $\lambda^{\mathrm{U}\mathbf{L}}$ has two distinct syntactic categories of types, values and expressions: those that come λ^{U} and those that come from $\lambda^{\mathbf{L}}$. Contexts, on the other hand, are mixed, they can have variables of both sorts – for a mixed context $\mathsf{\Gamma}$, the notation $\mathbf{!}\mathsf{\Gamma}$ only applies ($\mathbf{!}$) to its linear variables.

The typing rules of λ^{U} and $\lambda^{\mathbf{L}}$ are imported in our multi-language system, working on those two separate categories of program. They need to be extended to handle mixed contexts $\mathsf{\Gamma}$ instead of their original contexts Γ and $\mathbf{\Gamma}$. In the linear case, the rules look exactly the same. In the ML case, remark that the typing rules implicitly duplicate all the variables in the context; to remain sound in presence of linear variables, we state that those typing rules should use not an arbitrary context $\mathsf{\Gamma}$ instead of Γ, but a duplicable context $\mathbf{!}\mathsf{\Gamma}$.

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 allowed *language boundaries*, two new term formers $\mathcal{LU}(\mathrm{e})\,\mathcal{UL}(\varsigma : \mathbf{\Phi} \mid \mathbf{e})$ and that inject a ML term into the syntactic category of linear terms, and a linear configuration into the 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 $\lambda^{\mathbf{L}}$ and conversely. Allowing to send any type from one language into the other, for example by adding the counterpart of our language boundaries in the syntax of types, would be incorrect: 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 $\mathbf{!1}$, 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

Types $\sigma \mid \boldsymbol{\sigma}$

σ (unchanged from Figure 1)

$\boldsymbol{\sigma}$ $\ldots \mid [\sigma]$

Values $\mathsf{v} \mid \mathbf{v}$

v (unchanged from Figure 1)

$\mathbf{v}$ $\ldots \mid [\mathsf{v}]$

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

e $\ldots \mid \mathcal{UL}(\varsigma : \boldsymbol{\Phi} \mid \mathbf{e})$

with $\mathcal{UL}(\mathbf{e}) \overset{\text{def}}{=} \mathcal{UL}(\emptyset : \cdot \mid \mathbf{e})$

$\mathbf{e}$ $\ldots \mid \mathcal{LU}(\mathsf{e})$

Contexts $\Gamma ::= \cdot \mid \Gamma, x : \sigma \mid \Gamma, \alpha \mid \Gamma, \mathbf{x} : \boldsymbol{\sigma}$

Typing rules $\boxed{\Gamma \vdash_{\mathrm{LU}} \mathsf{e} : \sigma}$ $\boxed{\boldsymbol{\Phi} \mid \Gamma \vdash_{\mathrm{UL}} \varsigma \mid \mathbf{e} : \boldsymbol{\sigma}}$

with $\Gamma \vdash_{\mathrm{UL}} \mathbf{e} : \boldsymbol{\sigma} \overset{\text{def}}{=} \cdot \mid \Gamma \vdash_{\mathrm{UL}} \emptyset \mid \mathbf{e} : \boldsymbol{\sigma}$

(Typing rules of $\Gamma \vdash_{\mathrm{U}} \mathsf{e} : \sigma$ reused, with mixed context $!\Gamma$)

(Typing rules of $\boldsymbol{\Phi} \mid \boldsymbol{\Gamma} \vdash_{\mathrm{L}} \varsigma \mid \mathbf{e} : \boldsymbol{\sigma}$ reused, with mixed context Γ)

$$\frac{!\Gamma \vdash_{\mathrm{LU}} \mathsf{e} : \sigma}{\cdot \mid !\Gamma \vdash_{\mathrm{UL}} \emptyset \mid \mathcal{LU}(\mathsf{e}) : ![\sigma]} \qquad \frac{\boldsymbol{\Phi} \mid !\Gamma \vdash_{\mathrm{UL}} \varsigma \mid \mathbf{e} : ![\sigma]}{!\Gamma \vdash_{\mathrm{LU}} \mathcal{UL}(\varsigma : \boldsymbol{\Phi} \mid \mathbf{e}) : \sigma}$$

Reduction rules

(Reduction rules of λ^{U} and $\lambda^{\mathbf{L}}$ reused unchanged)

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

$$\frac{\begin{array}{c}\boldsymbol{\Phi} \mid \Gamma \vdash_{\mathrm{UL}} \varsigma \mid \mathbf{e} : \boldsymbol{\sigma} \\ (\varsigma \mid \mathbf{e}) \overset{\mathbf{L}}{\hookrightarrow} (\varsigma' \mid \mathbf{e}') \qquad \boldsymbol{\Phi}' \mid \Gamma \vdash_{\mathrm{UL}} \varsigma' \mid \mathbf{e}' : \boldsymbol{\sigma}\end{array}}{\mathcal{UL}(\varsigma : \boldsymbol{\Phi} \mid \mathbf{e}) \overset{\mathrm{U}}{\hookrightarrow} \mathcal{UL}(\varsigma' : \boldsymbol{\Phi}' \mid \mathbf{e}')}$$

Figure 8. Multi-language: lump and boundaries

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 $\mathsf{v} : \mathsf{y}$ is a 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 the language boundaries insert lumps when going from λ^{U} to $\lambda^{\mathbf{L}}$, and remove them when going back from $\lambda^{\mathbf{L}}$ to λ^{U}. In particular, arbitrary linear types cannot occur at the boundary, they must be of the form $![\sigma]$.

Finally, boundaries have reduction rules: a term or configuration inside a boundary in reduction position is reduced until it becomes a value – then a lump is added or removed depending on the boundary direction. Note that because the

Interaction context $\boldsymbol{\Sigma} ::= \cdot \mid \boldsymbol{\Sigma}, \alpha \simeq !\boldsymbol{\beta}$

Compatibility relation $\boxed{\boldsymbol{\Sigma} \vdash_{\mathrm{UL}} \sigma \simeq \boldsymbol{\sigma}}$

with $\sigma \simeq \boldsymbol{\sigma} \overset{\text{def}}{=} \cdot \vdash_{\mathrm{UL}} \sigma \simeq \boldsymbol{\sigma}$

$$\frac{}{\boldsymbol{\Sigma} \vdash_{\mathrm{UL}} 1 \simeq !\mathbf{1}} \qquad \frac{\boldsymbol{\Sigma} \vdash_{\mathrm{UL}} \sigma_1 \simeq !\boldsymbol{\sigma}_1 \qquad \boldsymbol{\Sigma} \vdash_{\mathrm{UL}} \sigma_2 \simeq !\boldsymbol{\sigma}_2}{\boldsymbol{\Sigma} \vdash_{\mathrm{UL}} \sigma_1 \times \sigma_2 \simeq !(\boldsymbol{\sigma}_1 \otimes \boldsymbol{\sigma}_2)}$$

$$\frac{\boldsymbol{\Sigma} \vdash_{\mathrm{UL}} \sigma_1 \simeq !\boldsymbol{\sigma}_1 \qquad \boldsymbol{\Sigma} \vdash_{\mathrm{UL}} \sigma_2 \simeq !\boldsymbol{\sigma}_2}{\boldsymbol{\Sigma} \vdash_{\mathrm{UL}} \sigma_1 + \sigma_2 \simeq !(\boldsymbol{\sigma}_1 \oplus \boldsymbol{\sigma}_2)}$$

$$\frac{\boldsymbol{\Sigma} \vdash_{\mathrm{UL}} \sigma \simeq !\boldsymbol{\sigma} \qquad \boldsymbol{\Sigma} \vdash_{\mathrm{UL}} \sigma' \simeq !\boldsymbol{\sigma}'}{\boldsymbol{\Sigma} \vdash_{\mathrm{UL}} \sigma \to \sigma' \simeq !(!\boldsymbol{\sigma} \multimap !\boldsymbol{\sigma}')} \qquad \frac{}{\boldsymbol{\Sigma} \vdash_{\mathrm{UL}} \sigma \simeq ![\sigma]}$$

$$\frac{\boldsymbol{\Sigma} \vdash_{\mathrm{UL}} \sigma \simeq !\boldsymbol{\sigma}}{\boldsymbol{\Sigma} \vdash_{\mathrm{UL}} \sigma \simeq !!\boldsymbol{\sigma}} \qquad \frac{\boldsymbol{\Sigma} \vdash_{\mathrm{UL}} \sigma \simeq !\boldsymbol{\sigma}}{\boldsymbol{\Sigma} \vdash_{\mathrm{UL}} \sigma \simeq !(\mathbf{Box}\ \mathbf{1}\ \boldsymbol{\sigma})}$$

$$\frac{\boldsymbol{\Sigma}, \alpha \simeq !\boldsymbol{\beta} \vdash_{\mathrm{UL}} \sigma \simeq !\boldsymbol{\sigma}}{\boldsymbol{\Sigma} \vdash_{\mathrm{UL}} \mu\alpha.\sigma \simeq !(\mu\boldsymbol{\beta}.\boldsymbol{\sigma})} \qquad \frac{(\alpha \simeq !\boldsymbol{\beta}) \in \boldsymbol{\Sigma}}{\boldsymbol{\Sigma} \vdash_{\mathrm{UL}} \alpha \simeq !\boldsymbol{\beta}}$$

Interaction primitives and derived constructs:

$$\begin{tikzcd} ![\sigma] \arrow[r, shift left, "{}^{\sigma}\mathsf{unlump}"] & \boldsymbol{\sigma} \arrow[l, shift left, "\mathsf{lump}^{\sigma}"] \end{tikzcd} \quad \text{whenever} \quad \cdot \vdash_{\mathrm{UL}} \sigma \simeq \boldsymbol{\sigma}$$

$${}^{\sigma}\mathcal{LU}(\mathsf{e}) \overset{\text{def}}{=} {}^{\sigma}\mathsf{unlump}\ \mathcal{LU}(\mathsf{e}) \qquad \mathcal{UL}^{\sigma}(\mathbf{e}) \overset{\text{def}}{=} \mathcal{UL}(\mathsf{lump}^{\sigma}\ \mathbf{e})$$

Figure 9. Multi-language: static interaction semantics

$\mathbf{v}$ in $\mathcal{UL}(\varsigma : \boldsymbol{\Phi} \mid \mathbf{v})$ is at a duplicable type $![\sigma]$, we know by inversion that the store is empty.

3.2 Interaction Semantics: 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 unchanged. To provide a real interaction, we provide a way to extract values out of a lump $![\sigma]$, use it at some linear type $\boldsymbol{\sigma}$, and put it back in before sending the result to λ^{U}.

The correspondence between intuitionistic types σ and linear types $\boldsymbol{\sigma}$ is specified by a heterogeneous *compatibility relation* $\sigma \simeq \boldsymbol{\sigma}$ defined in Figure 9 (Multi-language: static interaction semantics). The specification of this relation is that if $\sigma \simeq \boldsymbol{\sigma}$ holds, then the space of values of $![\sigma]$ and $\boldsymbol{\sigma}$ 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 position of the index $\boldsymbol{\sigma}$ emphasizes that the *input* e of $\mathsf{lump}^{\sigma}\ \mathbf{e}$ has type $\boldsymbol{\sigma}$, while the *output* of ${}^{\sigma}\mathsf{unlump}\ \mathbf{e}$ has type $\boldsymbol{\sigma}$.)

For example, we have $\boldsymbol{![}(\sigma \to \sigma')\boldsymbol{]} \simeq \boldsymbol{!(![}\sigma\boldsymbol{]} \multimap \boldsymbol{![}\sigma'\boldsymbol{])}$. Given a lumped ML function, we can unlump it to see it as a linear function. We can call it from the linear side, but have to pass it a duplicable argument, as a ML function may duplicate its argument. Conversely, we can convert a linear function into a lumped function type to pass it to the ML side, but it has to have a duplicable return type, given that the ML side may freely share the return value.

Our $\mathsf{lump}^{\boldsymbol{\sigma}}$ and ${}^{\boldsymbol{\sigma}}\mathsf{unlump}$ primitives are only indexed by the linear type $\boldsymbol{\sigma}$, because compatible ML type σ can be uniquely recovered, as per the following result.

Lemma 3.1 (Determinism of the compatibility relation) *If $\sigma \simeq \boldsymbol{\sigma}$ and $\sigma' \simeq \boldsymbol{\sigma}$ then $\sigma = \sigma'$.*

Note that the converse property does not hold: for a given σ, there are many $\boldsymbol{\sigma}$ such that $\sigma \simeq \boldsymbol{\sigma}$. For example, we have $1 \simeq \boldsymbol{!1}$ but also $1 \simeq \boldsymbol{!!1}$. This corresponds to the fact that the linear types are more fine-grained, and make distinctions (inner duplicability, dereference of full locations) that are erased in the ML world. The $\sigma \simeq \boldsymbol{![}\sigma\boldsymbol{]}$ case also allows you to (un)lump as deeply or as shallowly as you need: $\sigma_1 \times (\sigma_2 \times)$ is compatible with both $\boldsymbol{!(![}\sigma_1\boldsymbol{]} \otimes \boldsymbol{![}\sigma_2 \times\boldsymbol{])}$ and $\boldsymbol{!(![}\sigma_1\boldsymbol{]} \otimes \boldsymbol{(![}\sigma_2\boldsymbol{]} \otimes \boldsymbol{![]))}$. We could not systematically translate the complete type σ, as type variables cannot be translate and need to remain lumped: allowing lumps to “stop” the translation at arbitrary depth is a natural generalization.

The term $\mathcal{LU}(\mathsf{e})$ turns a $\mathsf{e} : \sigma$ into a lumped type $\boldsymbol{![}\sigma\boldsymbol{]}$, and we need to unlump it with some ${}^{\boldsymbol{\sigma}}\mathsf{unlump}$ for a compatible $\sigma \simeq \boldsymbol{\sigma}$ to interact with it on the linear side. It is common to combine both operations and we provide a syntactic sugar for it, ${}^{\boldsymbol{\sigma}}\mathcal{LU}(\mathsf{e})$. Similarly $\mathcal{UL}^{\boldsymbol{\sigma}}(\mathbf{e})$ first lumps a linear term then sends the result to the ML world.

The following technical result provides some confidence in the definition of compatibility for linear types, and is useful when reasoning on the operational semantics in the next section.

Lemma 3.2 (Substitution of recursive hypotheses) *If $\Sigma, \alpha \simeq \boldsymbol{!\beta} \vdash_{\mathrm{UL}} \sigma \simeq \boldsymbol{\sigma}$, $\alpha \notin \boldsymbol{\sigma}$, and $\Sigma \vdash_{\mathrm{UL}} \sigma' \simeq \boldsymbol{!\sigma'}$ then $\Sigma \vdash_{\mathrm{UL}} \sigma[\sigma'/\alpha] \simeq \boldsymbol{\sigma}[\boldsymbol{\sigma'}/\boldsymbol{\beta}]$.*

3.3 Interaction Semantics: Dynamic Semantics

We were careful to define the compatibility relation such that $\sigma \simeq \boldsymbol{\sigma}$ only holds when $\boldsymbol{![}\sigma\boldsymbol{]}$ and $\boldsymbol{\sigma}$ are isomorphic, in the sense that any value of one can be converted into a value of another. Figure 10 defines the operational semantics of the lumping and unlumping operations precisely as realizing these isomorphisms. For concision, we specify the isomorphisms as relations, following the inductive structure of the compatibility judgment itself. We write $(\overset{\sim}{\leftrightarrow})$ when a rule can be read bidirectionally to convert in either directions (assuming the same direction holds of the premises), and $(\overset{\sim}{\leftarrow})$ or $(\overset{\sim}{\rightarrow})$ for rules that only describe how to convert values in one direction.

$$\langle\rangle \overset{\sim}{\leftrightarrow}^{\boldsymbol{!1}} \mathbf{share}\,\boldsymbol{\langle\rangle}$$

$$\frac{\mathsf{v}_1 \overset{\sim}{\leftrightarrow}^{\boldsymbol{!\sigma_1}} \mathbf{share}(\boldsymbol{\varsigma_1{:}\Phi_1}).\,\mathbf{v}_1 \qquad \mathsf{v}_2 \overset{\sim}{\leftrightarrow}^{\boldsymbol{!\sigma_2}} \mathbf{share}(\boldsymbol{\varsigma_2{:}\Phi_2}).\,\mathbf{v}_2 \qquad \begin{array}{l}\mathsf{locs}(\boldsymbol{\varsigma_1}) = \mathsf{locs}(\boldsymbol{\Phi_1}) = \mathsf{locs}(\mathbf{v}_1)\\ \mathsf{locs}(\boldsymbol{\varsigma_2}) = \mathsf{locs}(\boldsymbol{\Phi_2}) = \mathsf{locs}(\mathbf{v}_2)\end{array}}{\langle\mathsf{v}_1,\mathsf{v}_2\rangle \overset{\sim}{\leftrightarrow}^{\boldsymbol{!(\sigma_1 \otimes \sigma_2)}} \mathbf{share}(\boldsymbol{\varsigma_1} + \boldsymbol{\varsigma_2{:}\Phi_1} \uplus \boldsymbol{\Phi_2}).\,\boldsymbol{\langle}\mathbf{v}_1,\mathbf{v}_2\boldsymbol{\rangle}}$$

$$\frac{\mathsf{v} \overset{\sim}{\leftrightarrow}^{\boldsymbol{!\sigma_i}} \mathbf{share}(\boldsymbol{\varsigma{:}\Phi}).\,\mathbf{v}}{\mathsf{inj_i}\,\mathsf{v} \overset{\sim}{\leftrightarrow}^{\boldsymbol{!(\sigma_1 \oplus \sigma_2)}} \mathbf{share}(\boldsymbol{\varsigma{:}\Phi}).\,\mathbf{inj_i}\,\mathbf{v}}$$

$$\frac{\sigma \simeq \boldsymbol{\sigma} \qquad \sigma' \simeq \boldsymbol{\sigma'}}{\mathsf{e} \overset{\sim}{\rightarrow}^{\boldsymbol{!(!\sigma \multimap !\sigma')}} \mathbf{share}\,\boldsymbol{\lambda(\mathbf{x} : !\sigma)}.\,{}^{\boldsymbol{\sigma'}}\mathcal{LU}(\mathsf{e}\,\mathcal{UL}^{\boldsymbol{\sigma}}(\mathbf{x}))}$$

$$\frac{\sigma \simeq \boldsymbol{\sigma} \qquad \sigma' \simeq \boldsymbol{\sigma'}}{\lambda(\mathsf{x} : \sigma).\,\mathcal{UL}^{\boldsymbol{\sigma'}}(\mathbf{copy}\,\mathbf{e}\,{}^{\boldsymbol{\sigma}}\mathcal{LU}(\mathsf{x})) \overset{\sim}{\leftarrow}^{\boldsymbol{!(\sigma \multimap \sigma')}} \mathbf{e}}$$

$$\frac{}{\mathsf{v} \overset{\sim}{\leftrightarrow}^{\boldsymbol{![}\sigma\boldsymbol{]}} \mathbf{share}\,\boldsymbol{[}\mathsf{v}\boldsymbol{]}} \qquad \frac{\mathsf{v} \overset{\sim}{\leftrightarrow}^{\boldsymbol{!\sigma}} \mathbf{share}\,\mathbf{v}}{\mathsf{v} \overset{\sim}{\leftrightarrow}^{\boldsymbol{!!\sigma}} \mathbf{share}\,(\mathbf{share}\,\mathbf{v})}$$

$$\frac{\mathsf{v} \overset{\sim}{\leftrightarrow}^{\boldsymbol{!\sigma}} \mathbf{share}(\boldsymbol{\varsigma{:}\Phi}).\,\mathbf{v}}{\mathsf{v} \overset{\sim}{\leftrightarrow}^{\boldsymbol{!\mathrm{Box}\ 1\ \sigma}} \mathbf{share}(\boldsymbol{[\ell \mapsto (\varsigma \mid \mathbf{v})]{:}(\cdot \mid \ell \vdash \Phi : \mathrm{Box}\ 1\ \sigma)}).\,\boldsymbol{\ell}}$$

$$\frac{\mathsf{v} \overset{\sim}{\leftrightarrow}^{\boldsymbol{!\sigma[\mu\alpha.\sigma/\alpha]}} \mathbf{share}(\boldsymbol{\varsigma{:}\Phi}).\,\mathbf{v}}{\mathsf{fold}_{\mu\alpha.\sigma}\,\mathsf{v} \overset{\sim}{\leftrightarrow}^{\boldsymbol{!\mu\alpha.\sigma}} \mathbf{share}(\boldsymbol{\varsigma{:}\Phi}).\,(\mathbf{fold}_{\boldsymbol{\mu\alpha.\sigma}}\,\mathbf{v})}$$

$$(\boldsymbol{\emptyset} \mid \mathbf{share}\,\boldsymbol{[}\mathsf{v}\boldsymbol{]}) \;\underset{\mathsf{lump}^{\boldsymbol{\sigma}}}{\overset{{}^{\boldsymbol{\sigma}}\mathsf{unlump}}{\overset{\mathbf{L}}{\underset{\mathbf{L}}{\hookrightarrow}}}}\; (\boldsymbol{\emptyset} \mid \mathbf{v}) \qquad \begin{array}{l}\text{whenever} \quad \mathsf{v} \overset{\sim}{\rightarrow}^{\boldsymbol{\sigma}} \mathbf{v}\\[2ex] \text{whenever} \quad \mathsf{v} \overset{\sim}{\leftarrow}^{\boldsymbol{\sigma}} \mathbf{v}\end{array}$$

Figure 10. Multi-language: dynamic interaction semantics

Theorem 3.3 (Value translations are functional) *If $\sigma \simeq \boldsymbol{\sigma}$, then for any closed value $\mathsf{v} : \sigma$ there is a unique $\mathbf{v} : \boldsymbol{\sigma}$ such that $\mathsf{v} \overset{\sim}{\rightarrow}^{\boldsymbol{\sigma}} \mathbf{v}$, and conversely for any closed value $\mathbf{v} : \boldsymbol{\sigma}$ there is a unique $\mathsf{v} : \sigma$ such that $\mathsf{v} \overset{\sim}{\leftarrow}^{\boldsymbol{\sigma}} \mathbf{v}$.*

Lemma 3.4 (Lumping cancellation) *The lump conversions $\mathsf{lump}^{\boldsymbol{\sigma}}$ and ${}^{\boldsymbol{\sigma}}\mathsf{unlump}$ cancel each other modulo $\beta\eta$. In particular,*

$${}^{\boldsymbol{\sigma}}\mathcal{LU}((\mathcal{UL}^{\boldsymbol{\sigma}}((\boldsymbol{\varsigma} \mid \mathbf{v})))) =_{\beta\eta} \mathbf{v} \qquad \mathcal{UL}^{\boldsymbol{\sigma}}(({}^{\boldsymbol{\sigma}}\mathcal{LU}(\mathsf{v}))) =_{\beta\eta} \mathsf{v}$$

Implementation consideration In a realistic implementation of this multi-language system, we would expect the representation choices made for λ^{U} and $\lambda^{\mathbf{L}}$ to be such that, for some but not all compatible pairs $\sigma \simeq \boldsymbol{\sigma}$, the types σ and $\boldsymbol{\sigma}$ actually have the exact same representation, making the conversion an efficient no-op. An implementation could even restrict the compatibility relation to accept only the pairs that can be implemented in this way; it would reject some $\lambda^{\mathrm{U}\mathbf{L}}$ programs, but the “graceful interaction” result that is our essential contribution would still hold.

3.4 Multi-Languages and Parametricity

We discussed the design choice of manipulating lumps $[\sigma]$ of any ML type, not just the type variable that motivates them. In presence of polymorphism, this generalization is also an important design choice to preserve parametricity.

Let us define $\mathbf{id}^{\sigma}(\mathbf{e}) \overset{\mathrm{def}}{=} \mathsf{lump}^{\sigma}\ ({}^{\sigma}\,\mathsf{unlump}\ \mathbf{e})$, and consider a polymorphic term of the form $\Lambda\alpha.\mathcal{UL}(\dots \mathbf{id}^{![\alpha]} \dots)$. The (un)lumping operations on a lumped type such as $![\alpha]$ are just the identity: the lumped value is passed around unchanged, so $\mathbf{id}^{![\alpha]}(\mathbf{v})$ shall reduce to $\mathbf{v}$. Now, if we instantiate this polymorphic term with a ML type σ, it will reduce to a term $\mathcal{UL}(\dots \mathbf{id}^{![\sigma]} \dots)$ whose unlumping operation is still on a lumped type, so is still exactly the identity.

On the contrary, if we allowed lumps only on type variables, we would have to push the lump inside σ, and the (un)lumping operations would become more complex: if σ starts with a ML product type $\_ \times \_$, it would be turned into a shared linear pair $!(\_ \otimes \_)$ by unlumping, and back into a ML pair by lumping. In general, $\mathbf{id}^{\sigma}$ may perform deep η-expansions of lumped values. The fact that, after instantiation of the polymorphic term, we get a monomorphic term that has a different (but η-equivalent) computational behavior would cause meta-theoretic difficulties; this is the approach that was adopted in previous work on multi-languages with polymorphism, Perconti and Ahmed [10], and it made some proofs substantially more complex.

In contrast, our handling of lump types as turning arbitrary types into blackboxes makes type instantiation obviously parametric.

Lemma 3.5 (Substitution of polymorphic variables)
If $\Sigma \vdash_{\mathrm{UL}} \sigma \simeq \boldsymbol{\sigma}$ and $\alpha \notin \Sigma$, then $\Sigma \vdash_{\mathrm{UL}} \sigma[\sigma'/\alpha] \simeq \boldsymbol{\sigma}[\sigma'/\alpha]$ and their lumping operations coincide on all values.

3.5 Full Abstraction From λ^{U} Into $\lambda^{\mathsf{U}\mathbf{L}}$

We can now state and prove 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 3.6 (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 $\mathsf{C}[\square]$ such that $\cdot \vdash_{\mathrm{U}} \mathsf{C}[\mathsf{e}] : 1$, the closed terms $\mathsf{C}[\mathsf{e}]$ and $\mathsf{C}[\mathsf{e}']$ are equi-terminating.*

Definition 3.7 (Contextual equivalence in $\lambda^{\mathsf{U}\mathbf{L}}$)
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 $\mathsf{C}[\square]$ such that $\cdot \vdash_{\mathrm{LU}} \mathsf{C}[\mathsf{e}] : 1$, the closed terms $\mathsf{C}[\mathsf{e}]$ and $\mathsf{C}[\mathsf{e}']$ are equi-terminating.*

The proof of full abstraction is actually rather simple. It relies on the idea, that we already mentioned in Section 2.2 (Linear memory in $\lambda^{\mathbf{L}}$), that linear state can be seen as either being imperatively mutated, but also as a purely functional

$$
\begin{aligned}
\llbracket \mathbf{1} \rrbracket &\overset{\mathrm{def}}{=} 1 \\
\llbracket \boldsymbol{\sigma}_1 \otimes \boldsymbol{\sigma}_2 \rrbracket &\overset{\mathrm{def}}{=} \llbracket \boldsymbol{\sigma}_1 \rrbracket \times \llbracket \boldsymbol{\sigma}_2 \rrbracket \\
&\dots \\
\llbracket !\boldsymbol{\sigma} \rrbracket &\overset{\mathrm{def}}{=} \llbracket \boldsymbol{\sigma} \rrbracket
\end{aligned}
\qquad
\begin{aligned}
\llbracket \mathbf{Box}\ \mathbf{0}\ \boldsymbol{\sigma} \rrbracket &\overset{\mathrm{def}}{=} 1 \\
\llbracket \mathbf{Box}\ \mathbf{1}\ \boldsymbol{\sigma} \rrbracket &\overset{\mathrm{def}}{=} 1 \times \llbracket \boldsymbol{\sigma} \rrbracket \\
\llbracket [\sigma] \rrbracket &\overset{\mathrm{def}}{=} \sigma \\
\llbracket \mu\boldsymbol{\alpha}.\boldsymbol{\sigma} \rrbracket &\overset{\mathrm{def}}{=} \mu\alpha_{\boldsymbol{\alpha}}.\llbracket \boldsymbol{\sigma} \rrbracket \\
\llbracket \boldsymbol{\alpha} \rrbracket &\overset{\mathrm{def}}{=} \alpha_{\boldsymbol{\alpha}}
\end{aligned}
$$

$$
\begin{aligned}
\llbracket \langle\rangle \rrbracket &\overset{\mathrm{def}}{=} \langle\rangle \\
\llbracket \mathbf{e}; \mathbf{e}' \rrbracket &\overset{\mathrm{def}}{=} \llbracket \mathbf{e} \rrbracket; \llbracket \mathbf{e}' \rrbracket \\
\llbracket \boldsymbol{\lambda}(\mathbf{x} : \boldsymbol{\sigma}).\,\mathbf{e} \rrbracket &\overset{\mathrm{def}}{=} \lambda(\mathsf{x}_{\mathbf{x}} : \llbracket \boldsymbol{\sigma} \rrbracket).\,\llbracket \mathbf{e} \rrbracket \\
\llbracket \mathbf{e}\ \mathbf{e}' \rrbracket &\overset{\mathrm{def}}{=} \llbracket \mathbf{e} \rrbracket\ \llbracket \mathbf{e}' \rrbracket
\end{aligned}
\qquad
\begin{aligned}
\llbracket \mathcal{LU}(\mathsf{e}) \rrbracket &\overset{\mathrm{def}}{=} \llbracket \mathsf{e} \rrbracket \\
\llbracket \mathbf{new}\ \mathbf{e} \rrbracket &\overset{\mathrm{def}}{=} \llbracket \mathbf{e} \rrbracket; \langle\rangle \\
\llbracket \mathbf{free}\ \mathbf{e} \rrbracket &\overset{\mathrm{def}}{=} \llbracket \mathbf{e} \rrbracket; \langle\rangle \\
\llbracket \mathbf{box}\ \mathbf{e} \rrbracket &\overset{\mathrm{def}}{=} \langle \langle\rangle, \pi_2\,\llbracket \mathbf{e} \rrbracket \rangle \\
\llbracket \mathbf{unbox}\ \mathbf{e} \rrbracket &\overset{\mathrm{def}}{=} \langle \langle\rangle, \pi_2\,\llbracket \mathbf{e} \rrbracket \rangle
\end{aligned}
$$

$$
\begin{aligned}
\llbracket \mathbf{x} \rrbracket &\overset{\mathrm{def}}{=} \mathsf{x}_{\mathbf{x}} \\
\llbracket \boldsymbol{\ell} \rrbracket &\overset{\mathrm{def}}{=} \mathsf{x}_{\boldsymbol{\ell}} \\
\llbracket (\emptyset \mid \mathbf{e}) \rrbracket &\overset{\mathrm{def}}{=} \llbracket \mathbf{e} \rrbracket \\
\llbracket (\varsigma[\boldsymbol{\ell} \mapsto \cdot] \mid \mathbf{e}) \rrbracket &\overset{\mathrm{def}}{=} \llbracket (\varsigma \mid \mathbf{e}) \rrbracket[\langle\rangle/\mathbf{x}_{\boldsymbol{\ell}}] \\
\llbracket (\varsigma[\boldsymbol{\ell} \mapsto (\varsigma' \mid \mathbf{v})] \mid \mathbf{e}) \rrbracket &\overset{\mathrm{def}}{=} \llbracket (\varsigma \mid \mathbf{e}) \rrbracket[\langle \langle\rangle, \llbracket (\varsigma' \mid \mathbf{v}) \rrbracket \rangle/\mathsf{x}_{\boldsymbol{\ell}}]
\end{aligned}
$$

Figure 11. Pure semantics of linear state

feature that just explicits memory layout. In absence of aliasing, we can give a purely functional semantics to linear state operations – instead of the store-modifying semantics of Figure 7 (Linear language: Operational Semantics) – and in fact this semantics determines a translation from linear programs back into pure ML programs. Those ML programs will not have the same allocation behavior as the initial linear programs (in-place programs won't be in-place anymore), but they are observably equivalent in that they are equi-terminating and return the same outputs from the same inputs.

The definition of the functional translation of linear contexts, terms and types is given in Figure 11. To simplify the translation of terms and the statement of Theorem 3.8, we assume that a global injective mapping is chosen from linear variables $\mathbf{x}$ and locations $\boldsymbol{\ell}$ to ML variables $\mathsf{x}_{\mathbf{x}}$ and $\mathsf{x}_{\boldsymbol{\ell}}$, from linear type variables $\boldsymbol{\alpha}$ to ML type variables $\alpha_{\boldsymbol{\alpha}}$, and that the inputs terms and types have all bound variables distinct from each other and their free variables.

Lemma 3.8 (Compositionality)
$\llbracket \mathsf{C}[\mathsf{e}] \rrbracket = \llbracket \mathsf{C} \rrbracket[\llbracket \mathsf{e} \rrbracket]$.

Lemma 3.9 (Projection)
If $\mathsf{e} \in \lambda^{\mathsf{U}\mathbf{L}}$ is in the λ^{U} subset, then $\llbracket \mathsf{e} \rrbracket = \mathsf{e}$.

Theorem 3.10 (Termination equivalence)
The reduction of $\llbracket \mathsf{e} \rrbracket$ in λ^{U} terminates if and only if the reduction of e in $\lambda^{\mathsf{U}\mathbf{L}}$ terminates.

Theorem 3.11 (Full Abstraction)
The embedding of λ^{U} into $\lambda^{\mathsf{U}\mathbf{L}}$ 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. Hybrid Program Examples

4.1 In-Place Transformations

In Section 2.2 (Linear memory in $\lambda^{\mathbf{L}}$) we proposed a program for in-place reversal of linear lists defined by the type $\mathbf{LinList}\ \sigma \overset{\text{def}}{=} \mu\alpha.1 \oplus \mathbf{Box}\ 1\ (\sigma \otimes \alpha)$. We can also define a type of ML lists $\mathsf{List}\ \sigma \overset{\text{def}}{=} \mu\alpha.1 + \sigma \times \alpha$. Note that ML lists are compatible with shared linear lists, in the sense that $\mathsf{List}\ \sigma \simeq \mathbf{!}(\mathbf{LinList}\ \mathbf{!}[\sigma])$. This enables writing in-place list-manipulation functions in $\lambda^{\mathbf{L}}$, and exposing them to beginners at a λ^{U} type: $\mathsf{rev}\ \mathsf{xs} \overset{\text{def}}{=}$

$$\mathcal{UL}^{\mathbf{LinList}\ \mathbf{!}[\sigma]}(\mathbf{share}\,(\mathbf{rev\_into}\ \mathbf{copy}\ ({}^{\mathbf{LinList}\ \mathbf{!}[\sigma]}\mathcal{LU}(\mathsf{xs}))\ \mathbf{Nil}))$$

This example is arguably silly, as the allocations that are avoided by doing an in-place traversal are paid when copying the shared list to obtain a uniquely-owned version. A better example of list operations that can profitably be sent on the linear side is merge sort: a ML implementation allocates the size of the list on each merge layer, while the surprisingly readable λ^{U} implementation only allocates for the first copy.

4.2 Typestate Protocols

Linear types can enforce proper allocation and deallocation of resources, and in general any automata/typestate-like protocols on their usage by encoding the state transitions as linear transformations. In the simple example of file-descriptor handling, additional safety compared to ML programming can be obtained by exposing file-handling functions on the λ^{U} side, with linear types. Consider a linear library providing the following operations

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

then a user willing to read files from their ML program would have to do it inside a linear boundary, enforcing safe resource handling. But the granularity of our handles means that the user can easily open a nested boundary in the line-consuming code, and write it in simple ML again. Only the resource-consumption structure needs to be in the linear fragment:

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

5. Conclusion

Two languages that are each as simple as possible can be combined in a multi-language system, who can be sensibly simpler than monolithic languages covering the same feature space yet reasonably expressive. In our case study, a very simple system mirroring the standard rules of intuitionistic linear logic can be equipped with linear state and usefully complement a general-purpose functional ML language.

Fine-grained language boundaries allow interesting programming patterns to emerge, and we believe that full abstraction provides a stringent but rewarding notion of what it means for the isolated languages to "gracefully" embed in the system, avoiding abstraction leaks for the other parts.

5.1 Related Work

Having a stack of usable, interoperable languages, extensions or dialects is the forefront of the Racket approach to programming environments, in particular for teaching [5]. The Racket community did seminal work on specifying formal semantics for multi-language systems [8], but it only addresses the question of soundness of the multi-language; we propose a formal treatment not only of correctness, but also *usability*.

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 [12], 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 [7] 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 [9], in which linear state is also viewed as both function 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.

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. This restricts expressiveness in well-understood ways [4]: shared values cannot point to linear values.

Finally, on the side of the semantics, our system is related to LNL [3], 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 has 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**.

References

[1] Thibaut Balabonski, François Pottier, and Jonathan Protzenko. The design and formalization of Mezzo, a permission-based programming language. *ACM Transactions on Programming Languages and Systems*, 38(4):14:1–14:94, August 2016. doi: http://dx.doi.org/10.1145/2837022.

[2] Edd Barrett, Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. Fine-grained language composition: A case study. In *ECOOP*, July 2016.

[3] P Nick Benton. A mixed linear and non-linear logic: Proofs, terms and models. In *International Workshop on Computer Science Logic*, 1994.

[4] Manuel Fahndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *PLDI'02*, PLDI '02, 2002.

[5] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. The teachscheme! project: Computing and programming for every student. *Computer Science Education*, 14(1):55–77, 2004.

[6] Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. Foundations of typestate-oriented programming. *TOPLAS*, 2014.

[7] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource Aware ML. In *24rd International Conference on Computer Aided Verification (CAV'12)*, volume 7358 of *Lecture Notes in Computer Science*, pages 781–786. Springer, 2012.

[8] Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(3):12, 2009.

[9] Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. Refinement through restraint: Bringing down the cost of verification. In *ICFP*, 2016.

[10] James T Perconti and Amal Ahmed. Verifying an open compiler using multi-language semantics. In *ESOP*, 2014.

[11] Jesse A Tov and Riccardo Pucella. Practical affine types. In *POPL*, 2011.

[12] Philip Wadler. Linear Types Can Change the World! In *Programming Concepts and Methods*, 1990.

A. Proof Outlines

Proof (Theorem 2.5 (Progress))

By induction on the typing derivation of $\mathbf{e}$, using induction hypothesis in the evaluation order corresponding to the structure of contexts $\mathbf{K}$. If one induction hypothesis returns a reduction, we build a bigger reduction ($\overset{\mathbf{L}}{\hookrightarrow}$) for the whole term. If all induction hypotheses return a value, the proof depends on whether the head term-former is an introduction/construction form or an elimination/destruction form. An introduction form whose subterms are values is a value. For elimination forms, we use Theorem 2.3 (Inversion principle for $\lambda^{\mathbf{L}}$ values) on the eliminated subterm (a value), to learn that it starts with an introduction form, and thus forms a head redex with the head elimination form, so we build a head reduction ($\overset{\mathbf{L}}{\rightsquigarrow}$). □

Proof (Theorem 2.6 (Non-store-escaping substitution principle))

The proof, summarized below, proceeds by induction on the typing derivation of $\mathbf{e}$.

Most cases need an additional case analysis on whether the substituted type σ is a duplicable type of the form $!\sigma''$, as it influence whether it may appear in zero or several subterms of $\mathbf{e}$. (This is a price to pay for contraction and weakening happening in all rules for convenience, instead of being isolated in separate structural rules.)

For example, in the variable case, $\mathbf{e}$ may be the variable $\mathbf{x}$ itself, in which case we know that Γ is empty and conclude immediately. But $\mathbf{e}$ may also be another variable $\mathbf{y}$ if $\mathbf{x}$ is duplicable and has been dropped. In that case, we perform an inversion (Theorem 2.3) on the $\mathbf{v}$ premise to learn that Φ is empty and Δ is duplicable, and can thus use Theorem 2.4 (Weakening of duplicable contexts).

In the $\langle \mathbf{e}_1, \mathbf{e}_2 \rangle$ case, if $\mathbf{x}$ is a linear variable it only occurs in one subterm on which we apply our induction hypothesis. If $\mathbf{x}$ is duplicable, inversion on the $\mathbf{v}$ premises again tells us that Δ is duplicable. We know by assumption that $(\Gamma_1 \curlyvee \Gamma_2) \curlyvee \Delta$; because Δ is duplicable, we can deduce from Theorem 2.1 (Context joining properties) that the $\Gamma_i \curlyvee \Delta$ are also defined, which let us apply an induction hypothesis on both subterms $\mathbf{e}_i$. To conclude, we need the computation

$$\begin{aligned} &(\Gamma_1 \curlyvee \Delta) \curlyvee (\Gamma_2 \curlyvee \Delta) \\ = {}& \Gamma_1 \curlyvee \Gamma_2 \curlyvee (\Delta \curlyvee \Delta) \\ = {}& \Gamma_1 \curlyvee \Gamma_2 \curlyvee \Delta \end{aligned}$$

which again comes from duplicability of Δ.

The assumption $\mathbf{x} \notin \Phi$ enforces that the resource $\mathbf{x}$ is consumed in the term $\mathbf{e}$ itself, not in one of the values $[\ell \mapsto (\varsigma \mid \mathbf{v})]$ in the store: otherwise $\mathbf{x}$ would appear in the store typing $(\Gamma \mid \ell \vdash \Phi : \mathbf{Box\ 1})$ of this location in Φ. It is used in the case where $\mathbf{e} : \sigma$ is a full location $\ell : \mathbf{Box\ 1}\ \sigma'$. If $\mathbf{x}$ could appear in the value of ℓ in the store, we would have substitute it in the store as well – in our substitution statement, only the term is modified. Here we know that this value is unused, so it has a duplicable type and we can perform an inversion in the other cases. $\square$

Proof (Theorem 2.8 (Subject reduction for λ^{L}))

The proof is done by induction on the reduction derivation.

The head-reduction rules involving substitutions rely on Theorem 2.6 (Non-store-escaping substitution principle); note that in each of them, for example $(\lambda(x:\sigma').\,e')\ e''$, the substituted variable x is bound in the term e, and thus does not appear in the store ς: the non-store-escaping hypothesis holds.

For the copy rule and the store operators, we build a valid derivation for the reduced configuration by inverting the typing derivation of the reducible configuration.

In the non-head-reduction cases, the share case is by direction, and the context case $K[e]$ uses Theorem 2.7 (Context decomposition) to obtain a typing derivation for e, and the same lemma again rebuild a derivation of the reduced term $K[e']$. $\square$

Proof (Theorem 3.1 (Determinism of the compatibility relation))

By induction on the syntax of σ: the judgment $\Sigma \vdash_{UL} \sigma \simeq \sigma$ is syntax-directed in its σ component. $\square$

Proof (Theorem 3.2 (Substitution of recursive hypotheses))

By induction on the $\sigma \simeq \sigma$ derivation. There are two leaf cases: the case recursive hypotheses, which is immediate, and the case of lump $\sigma \simeq ![\sigma]$. In this latter case, notice that σ is a type of λ^{U}, so in particular it does not contain the variable β; and we assumed $\alpha \notin \sigma$ so we also have $\alpha \notin \sigma$, so $\sigma[\sigma'/\alpha] = \sigma \simeq ![\sigma] = \sigma[\sigma'/\beta]$. $\square$

Proof (Theorem 3.3 (Value translations are functional))

Remark that in the statement of the term, when we quantify over all closed values $v : \sigma$, we implicitly assume that in the general case values of v live in the empty global store – otherwise we would have a value of the form $\Phi \mid \cdot \vdash_{L} \varsigma \mid v : \sigma$. This is valid because all types σ in the image of the type-compatibility relation are duplicable types of the form $!\sigma'$, so by Theorem 2.3 (Inversion principle for λ^{L} values) we know that v is in fact of the form a $\mathrm{share}(\varsigma{:}\Phi).\,e'$, living in the empty store.)

The two sides of the result are proved simultaneously by induction on $\sigma \simeq \sigma$, using inversion to reason on the shapes of v and v. Note that the inductive cases remain on closed values: the only variable-binder constructions, λ-abstractions, do not use the recursion hypothesis.

In the recursive case $\mu\alpha.\sigma \simeq !(\mu\alpha.\sigma)$, to use the induction hypothesis on the folded values we need to know that the unfolded types $\sigma[\mu\alpha.\sigma/\alpha]$ and $\sigma[\mu\alpha.\sigma/\alpha]$ are compatible. This is exactly Theorem 3.2 (Substitution of recursive hypotheses), using the hypothesis $\mu\alpha.\sigma \simeq !(\mu\alpha.\sigma)$ itself. $\square$

Proof

By induction on σ, and then by parallel induction on the derivations of type compatibility and value compatibility. The parallel cases are symmetric by definition, only the function case $!(!\sigma \multimap !\sigma')$ needs to be checked. A simple computation, using the induction hypothesis on the smaller types $!\sigma$ and $!\sigma'$, shows that composing the two function translations gives an η-expansion – plus the $\beta\eta$-steps from the induction hypotheses. $\square$

Proof (Theorem 3.5 (Substitution of polymorphic variables))

By induction on $\sigma \simeq \sigma$. In the variable leaf case, we know $\alpha \notin \Sigma$. In the lump leaf case $\sigma \simeq ![\sigma]$, the goal $\sigma[\sigma'/\alpha] \simeq ![\sigma[\sigma'/\alpha]]$ is immediate. $\square$

Proof (Theorem 3.10 (Termination equivalence))

The translation respects the evaluation structure: a value is translated into a value, and a position in the original term is reducible if and only if the same position is reducible in the translation – both properties are checked by direct induction, on values and evaluation contexts.

Furthermore, the translation was carefully chosen (especially for the store operations) so that there is a redex in the translated term if and only if there is a redex in the original term, and the reduction of the translation is also the translation of the reduction. For example, we have

$$\begin{array}{rl} & (\varsigma[\ell \mapsto \cdot] \mid \mathbf{box}\ \langle \ell, v\rangle) \\ \overset{L}{\hookrightarrow} & ([\ell \mapsto (\varsigma \mid v)] \mid \ell) \end{array}$$

$$\begin{array}{rl} & [\![(\varsigma[\ell \mapsto \cdot] \mid \mathbf{box}\ \langle \ell, v\rangle)]\!] \\ = & [\![\mathbf{box}]\!]\ \langle \langle\rangle, v[\varsigma/\varsigma]\rangle \\ = & \langle \langle\rangle, \pi_2\ \langle \langle\rangle, v[\varsigma/\varsigma]\rangle\rangle \\ \overset{L}{\hookrightarrow} & \langle \langle\rangle, v[\varsigma/\varsigma]\rangle \\ = & \langle \langle\rangle, [\![(\varsigma \mid v)]\!]\rangle \\ = & [\![([\ell \mapsto (\varsigma \mid v)] \mid \ell)]\!] \end{array}$$

where $[\varsigma/\varsigma]$ denotes the composed substitution $[\langle\langle\rangle, [\![(\varsigma' \mid v)]\!]\rangle / x_{\ell}]$ for each $[\ell \mapsto (\varsigma' \mid v)]$ in ς. $\square$

Proof (Theorem 3.11 (Full Abstraction))

To show that two λ^{U} terms e, e' are contextually equivalent in λ^{UL}, we are given a context C in λ^{UL} and must prove that $C[e]$, $C[e]$ are equi-terminating.

From Theorem 3.10 (Termination equivalence) we know that $C[e]$ and $[\![C[e]]\!]$ are equi-terminating, and from Theorem 3.8 (Compositionality) that $[\![C[e]]\!]$ is equal to $[\![C]\!][\![e]\!]$, which is equal to $[\![C]\!][e]$ by Theorem 3.9 (Projection). Similarly, $C[e']$ and $[\![C]\!][e']$ are equi-terminating. Because $[\![C]\!]$ is a context in λ^{U}, we can use our assumption that $e \approx^{ctx}_{U} e'$ to conclude that $[\![C]\!][e]$ and $[\![C]\!][e']$ are equi-terminating. $\square$