Kinds are calling conventions

PAUL DOWNEN, ZACHARY SULLIVAN, and ZENA M. ARIOLA, University of Oregon, USA
SIMON PEYTON JONES, Microsoft Research, UK

Curried functions apparently take one argument at a time, which is slow. So optimising compilers for higher order languages invariably have some mechanism for working around currying and passing several arguments at once, as many as the function can handle known as its *arity*. But such mechanisms are often ad-hoc, and do not work at all in higher order functions. We show how extensional, call-by-name functions have the correct behavior for directly expressing the arity of curried functions. And these extensional functions can stand side-by-side with functions native to practical programming languages, which do not use call-by-name evaluation. We use a kind system to distinguish the two in the same intermediate language, which allows us to express the arity of a function in its type, thereby giving a principled and compositional account of multi-argument curried functions. An unexpected, but significant, bonus is that our intermediate language is equally suitable for a call-by-value language and a call-by-need language, or any combination of the two.

CCS Concepts: • **Software and its engineering** → *Semantics*; *Compilers*;

ACM Reference Format:
Paul Downen, Zachary Sullivan, Zena M. Ariola, and Simon Peyton Jones. 2019. Kinds are calling conventions. *Proc. ACM Program. Lang.* 1, In submission, Article 1 (March 2019), 38 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Consider these two function definitions:

$$f_1 = \lambda x.\, \mathbf{let}\, z = h\ x\ x\ \mathbf{in}\ \lambda y.e\ y\ z$$
$$f_2 = \lambda x.\lambda y.\, \mathbf{let}\, z = h\ x\ x\ \mathbf{in}\ e\ y\ z$$

It is highly desirable for an optimising compiler to eta expand f_1 into f_2. The function f_1 takes only a single argument before returning a heap-allocated function closure; then that closure must subsequently be called by passing the second argument. In contrast, f_2 can take both arguments at once, without constructing an intermediate closure, and this can make a huge difference in practice [Marlow and Peyton Jones 2004]. But the eta expansion would be bad if $(h\ x\ x)$ was expensive, because in a call like $(map\ (f_2\ 3)\ xs)$, the expensive computation of $(h\ 3\ 3)$ would be performed once for each element of xs, whereas in $(map\ (f_1\ 3)\ xs)$ the value of $(h\ 3\ 3)$ would be computed only once. An optimising compiler should not cause an asymptotic slow-down!

So the question becomes "does $(h\ x\ x)$ do serious work?" We should transform f_1 into f_2 only if it does not. But what exactly do we mean by "serious work?" For example, suppose we knew that h was defined as $h = \lambda p.\lambda r.\lambda q.blah$; that is, h cannot begin to do any work until it is given three arguments. In this case, it is almost certainly a good idea to eta expand f_1. For this reason GHC, an optimising compiler for Haskell, keeps track of the *arity* of every in-scope variable, such as h, and uses that information to guide transformations. This notion of arity is not unique to Haskell or GHC: the same impact of eta expanding f_1 to f_2, with its potential performance improvement and risk of disastrous slow-down, is just as applicable in eager functional languages like OCaml [Dargaye and Leroy 2009]. In fact, the risks are even more dire in OCaml, where inappropriate eta

Authors' addresses: Paul Downen, pdownen@cs.uoregon.edu; Zachary Sullivan, zsulliva@cs.uoregon.edu; Zena M. Ariola, ariola@cs.uoregon.edu, Computer and Information Science, University of Oregon, 1477 E. 13th Ave. Eugene, Oregon, 97403, USA; Simon Peyton Jones, Microsoft Research, 21 Station Rd. Cambridge, CB1 2FB, UK, simonpj@microsoft.com.

2019. 2475-1421/2019/3-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

expansion can change the result of a program due to side effects like exceptions and mutable state. So arity information is crucial for correctly applying useful optimizations.

The problem is that the very notion of "arity" is a squishy, informal one, rooted in operational intuitions rather than solid invariants or principled theory. In this paper we resolve that problem:

- We describe $\mathcal{L}$, a statically-typed intermediate language suitable for program transformation and optimisation, in which the arity of a variable is directly expressed in its type instead of being an ad-hoc bolt-on (Section 3.1).
- We describe $\mathcal{L}X$, a restricted subset of $\mathcal{L}$ suitable for direct execution, along with a simple translation from $\mathcal{L}$ into $\mathcal{L}X$ (Section 3.3).
- We give a high-level operational semantics for $\mathcal{L}$, which is helpful in understanding what arity means (Section 3.5). But since arity is all about runtime costs, we give a low-level operational semantics for $\mathcal{L}X$ that is sufficiently close to the machine so that we can reason with confidence about operational matters (Section 3.6). And, of course, we show that the two line up (Section 3.7).
- It turns that our approach has a huge bonus: with a natural generalisation, the *same* intermediate language can be an equally good fit for a *call-by-value* language like OCaml or a *call-by-need* language like Haskell, a goal that has been remarkably elusive [Bolingbroke and Peyton Jones 2009]. It also has the side benefit of integrating with levity polymorphism [Eisenberg and Peyton Jones 2017] (Section 4) and giving a more systematic solution to the tricky issue of type erasure for call-by-value polymorphism (Section 4.3).

Unlike previous work on curried function optimization which we discuss in Section 5, our approach grows directly from deep roots in type theory; in particular *polarity* [Zeilberger 2009], which tells us the optimal evaluation strategy for a type based on its logical structure. Polarity brings two optimizations for compiling lazy functional languages—arity and call-by-value data representations [Peyton Jones and Launchbury 1991]—under the same umbrella. We discuss these deeper connections in Section 5.3.

An alternative approach is simply to uncurry every function but, as we discuss in Section 5.1, doing so is not straightforward in the presence of first-class polymorphism; nor does it help with compiling both call-by-value and call-by-need through a single intermediate language.

We have built a prototype implementation in GHC[1].

2 THE KEY IDEA

Informally, we define the *arity* of a function as the number of arguments it must receive before "doing serious work". So if `f` has arity 3, then `f`, `(f 3)`, and `(f 3 9)` are all partial applications of `f`; only when `f` is given three arguments will it compute anything. We begin by explaining why arity is important, before giving an intuitive introduction to our new approach.

2.1 Motivation

There are several reasons why an optimising compiler might care about the arity of a function:

- A function of arity n can be compiled into machine code that takes n arguments simultaneously, passed in machine registers. This is much, much faster than taking arguments one at a time, and returning an intermediate function closure after each application.
- In Haskell, the expression `(seq e1 e2)` evaluates `e1` to weak head-normal form, and then returns `e2`. Now suppose we define

```
loop1, loop2 :: Int -> Int
```

[1]URL suppressed for double-blind reviewing, but available on request.

```
loop1 = \x -> loop1 x
loop2 = loop2
```

`(seq loop1 True)` evaluates `loop1` to a function closure, and returns `True`. In contrast, `(seq loop2 True)` simply diverges because `loop2` diverges. So Haskell terms do not always enjoy eta equivalence; in general, $\lambda x.ex \neq e$. But eta equivalence *does* hold if e has arity greater than 0—in other words, if e is sure to evaluate to a lambda abstraction—so knowing the arity of an expression can unlock optimisations such as discarding unnecessary calls. For example, `(seq loop1 True)` becomes `True` since `loop1` has arity 1. But this arity information is not seen in the type: `loop1` and `loop2` have the same type, but different arities (1 versus 0).

- The trouble with limit eta equivalence is not unique to `seq` in Haskell: the same issue arises in eager functional languages, too. For example, consider similar definitions in OCaml:

  ```
  let rec loop1 x = loop1 x;;
  let rec loop2 x y = loop2 x y;;
  ```

  As before, `loop1` and `loop2` appear to be eta equivalent, but they are not the same function. For example, `let f = loop1 5 in true` diverges whereas `let f = loop2 5 in true` returns `true`. This is because with eager evaluation, as done by ML-family languages, all closures are computed in advance when bound, and `loop2 5` evaluates to a closure but `loop1 5` loops forever. So both eager and lazy functional languages have the same essential problem with restricted eta equivalence, which can block optimisations.
- Consider this (standard) definition:

  ```
  zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
  zipWith f (a:as) (b:bs) = f a b : zipWith f as bs
  zipWith f _      _      = []
  ```

  In GHC and the OCaml native-code compiler today, the call to `f` in the body of `zipWith` is an "unknown call:" the compiler knows nothing about the arity of the function being passed in. It might expect one argument, or two, or even three (so that `zipWith` would return a list of functions). Such unknown calls impose runtime overhead, which is frustrating because the vastly-common case is that `f` is an arity-2 function. If we could encode arity into `f`'s type we may write

  ```
  zipWith :: (a ⇝ b ⇝ c) -> [a] -> [b] -> [c]
  zipWith f (a:as) (b:bs) = f a b : zipWith f as bs
  zipWith f _      _      = []
  ```

  The new function arrows (⇝) in `f`'s type express the fact that `zipWith` must be passed a function that takes precisely two arguments. Now the call in `zipWith`'s body can be fast, passing two arguments in registers with no runtime arity checks. And this can be done locally, when compiling `zipWith`, without *directly* looking at its call sites (see Section 2.3).

2.2 The key idea: a new function arrow

Our key idea is to make *types encode arity*. To do this, we introduce a new function type $(\sigma \rightsquigarrow \tau)$ constructed with $\widetilde{\lambda} x.e$, alongside the existing one $(\sigma \to \tau)$, with the following intuitions:

- Terms of type $(\sigma \rightsquigarrow \tau)$ enjoy unconditional eta equivalence: if $e : \sigma \rightsquigarrow \tau$ then $(\widetilde{\lambda} x.\, e\, x) = e$.
- The type $(\sigma \rightsquigarrow \tau)$ is *unlifted*; that is, it is not inhabited by $\bot$ (a divergent value). This is why eta equivalence holds unconditionally; it is nonsensical to have a program of type $(\sigma \rightsquigarrow \tau)$.

- This eta equivalence also preserves *work equivalence*; that is, eta expansion does not change the number of reduction steps in a program run. For example, consider these two programs

$$\begin{array}{ll} \textbf{let } f_1 : \mathit{Int} \rightsquigarrow \mathit{Int} & \textbf{let } f_2 : \mathit{Int} \rightsquigarrow \mathit{Int} \\ \quad f_1 = \ \textbf{let } x = \mathit{ack}\ 2\ 3 & \quad f_2 = \widetilde{\lambda} y.\ \textbf{let } x = \mathit{ack}\ 2\ 3 \\ \qquad\quad \textbf{in } \widetilde{\lambda} y.x + y & \qquad\quad \textbf{in } x + y \\ \textbf{in } \mathit{map}\ f_1\ [1..1000] & \textbf{in } \mathit{map}\ f_2\ [1..1000] \end{array}$$

  With an ordinary λ, one would expect these two program to behave differently: in f_1, the (expensive) function *ack* would be called once, with x's value being shared by the 1000 calls of f_1. But in f_2, *ack* would be called once for each of the 1000 invocations of f_2. With our new lambda, however, the two are precisely equivalent, and in both cases *ack* is called 1000 times. In effect, the binding for f_1 is not memoised as a thunk, but is *treated in a call-by-name fashion*, a point we will return to.
- Although it has a special kind, a value of type $(\sigma \rightsquigarrow \tau)$, such as f_1 or f_2, is a first-class value: it can be passed to a function, returned as a result, or stored in a data structure.
- At run-time , if $f : \tau_1 \rightsquigarrow \ldots\ \tau_n \rightsquigarrow \rho$, where ρ is not of form $\rho_1 \rightsquigarrow \rho_2$, then f is bound to a heap-allocated function closure of arity exactly n. Like an ordinary function in a call-by-value language, but *unlike* an ordinary function value of type $\sigma \to \tau$ in a call-by-need language, it is *not* represented by a pointer to a thunk that will (if evaluated) return a function closure.
- But what if the result type ρ was a type variable a? Is it possible that a could be instantiated by $(\rho_1 \rightsquigarrow \rho_2)$ thereby changing f's arity, and making a nonsense of our claim that types encode arity? No: in our system you cannot instantiate a type variable with such a type, which corresponds to similar existing restrictions in GHC. For example, GHC does not allow you to instantiate a type variable with an unboxed, or unlifted type [Eisenberg and Peyton Jones 2017]. This restriction is enforced by the kind system and our new function arrow fits neatly into this framework.

Everything else flows from these intuitions.

2.3 Workers and wrappers

While we anticipate that expert programmers may want to write code that uses $(\rightsquigarrow)$ directly, our main focus is on using it internally, in the *intermediate language* of a compiler. How, then, might a compiler make use of the additional expressiveness? Consider

$$\begin{array}{l} \textbf{let } f : \mathit{Int} \to \mathit{Int} \to \mathit{Int} \\ \quad f = \lambda x.\lambda y.\mathit{blah} \\ \textbf{in } \ldots (f\ t) \ldots (f\ p\ q) \ldots \end{array}$$

Seeing that f begins with two lambdas, we would like to capture the fact that it has arity 2 with the type $\mathit{Int} \rightsquigarrow \mathit{Int} \rightsquigarrow \mathit{Int}$, but just changing the type of f can break type checking. Instead, the compiler can express this arity directly like so:

$$\begin{array}{l} \textbf{let } f_w : \mathit{Int} \rightsquigarrow \mathit{Int} \rightsquigarrow \mathit{Int} \\ \quad f_w = \widetilde{\lambda} x.\widetilde{\lambda} y.\mathit{blah} \\ \quad\ f : \mathit{Int} \to \mathit{Int} \to \mathit{Int} \\ \quad\ f = \lambda x.\lambda y.f_w\ x\ y \\ \textbf{in } \ldots (f\ t) \ldots (f\ p\ q) \ldots \end{array}$$

We have split f into a *wrapper* that performs impedance matching between the one-at-a-time f, and the all-at-once *worker* f_w. Now we can inline the wrapper (*i.e.,* f) at its call sites:

$$\begin{array}{l}\textbf{let } f_w : \mathit{Int} \rightsquigarrow \mathit{Int} \rightsquigarrow \mathit{Int} \\ \quad\quad f_w = \widetilde{\lambda} x . \widetilde{\lambda} y . \mathit{blah} \\ \textbf{in } \ldots (\lambda y . f_w\ t\ y) \ldots (f_w\ p\ q) \ldots\end{array}$$

Saturated calls simply call f_w directly, while under-saturated ones still need an ordinary λ, which visibly reflects the necessary runtime tests [Marlow and Peyton Jones 2004]. In general, these runtime tests check whether the number of arguments supplied at the call site is greater than, exactly equal to, or less than the arity of the function, and behave accordingly.

This so-called *worker/wrapper transform* is the standard way that GHC uses to move information from the *definition* of a function to its *call sites* [Gill and Hutton 2009]. We can use the same technique to transform `zipWith`, in Haskell (writing the new $\widetilde{\lambda} x . e$ as `\x ~> e`), like this:

```
$wzipWith :: (a ~> b ~> c) ~> [a] ~> [b] ~> [c]
$wzipWith f (a:as) (b:bs) = f a b : $wzipWith f as bs
$wzipWith f _ _ = []

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f as bs = $wzipWith (\a ~> \b ~> f a b) as bs
```

Again the wrapper does impedance-matching, but this time on the argument. Now at a typical call site such as `zipWith (||)`,[2] we can take the following steps

```
    zipWith (||)
==> $wzipWith (\a ~> \b ~> (||) a b)     {- inline wrapper for zipWith -}
==> $wzipWith (\a ~> \b ~> $w(||) a b)   {- inline wrapper for (||) -}
==> $wzipWith $w(||)                     {- eta reduce -}
```

It may not look as if much has happened, but the new code is much better: every call in the `$wzipWith` loop is now a fast call to a known-arity function.

2.4 Arity and Evaluation

Although they are separate concerns, data representation (the way in which objects are laid out in memory) is connected with evaluation strategy (the order in which computation steps are performed). For example, consider what happens if we have a type `Int#` for real, 64-bit machine integers suitable for storing directly in a register. In an expression like `f (g 1)`, where `f` and `g` both have type `Int# -> Int#`, is `(g 1)` evaluated before or after `f` is called? Or in other words: does the call `f (g 1)` use call-by-value or not? Since a value of type `Int#` is represented by a 64-bit machine integer, there is no way to delay the call to g. So the `Int#` type of machine integers forces a call-by-value evaluation order, even in a lazy-by-default language like Haskell. This insight lets GHC handle unboxed values [Peyton Jones and Launchbury 1991].

It turns out that a similar story plays out for function types: the calling convention of higher-arity curried functions dictates their evaluation strategy. The main challenge involves arities higher than 1, where we want to perform an *unchecked* multiple-argument call to an unknown function

[2] `(||) :: Bool -> Bool -> Bool` is Haskell's Boolean-or function.

definition. Consider two different ways of passing an unknown function h to *zipWith*

$$\begin{array}{ll} \textbf{let } f : \mathit{Int} \to \mathit{Bool} \to \mathit{Int} & \textbf{let } g : \mathit{Int} \to \mathit{Bool} \to \mathit{Int} \\ \quad f = h & \quad g = \lambda x.\lambda y.h\ x\ y \\ \textbf{in } \mathit{zipWith}\ f\ \mathit{xs}\ \mathit{ys} & \textbf{in } \mathit{zipWith}\ g\ \mathit{xs}\ \mathit{ys} \end{array}$$

The definitions of f and g are eta equivalent, but g has the advantage of being clearly arity 2 whereas the arity of f is unclear. What difference does this make? We want *zipWith* to assume that f has arity 2 so it can perform a fast call by passing it two arguments. If it turns out that h actually has arity 1 (or 0), *zipWith* calling f will crash at runtime (since *zipWith*'s unchecked assumption is wrong), but *zipWith* calling g is fine (since the arities of the definition and call sites match up).

But can we sensibly perform this expansion? Not in a call-by-value language, where eta expansion is restricted to manifest values. Even though we can do the first expansion of h to $(\lambda x.h\ x)$ (because h is a call-by-value value), the second eta expansion is impossible because $(h\ x)$ is *not* known to be a value. The same issue arises in a call-by-need language, due to the desire to share work; perhaps $(h\ x)$ does a lot of work before returning a function. Moreover $(\mathit{seq}\ (g\ 3)\ \mathit{True})$ always returns *True*, whereas $(\mathit{seq}\ (f\ 3)\ \mathit{True})$ might diverge.

So eta expanding unknown functions is, in general, unsound in both Haskell (call by need) and ML (call-by-value). In contrast, a call-by-*name* language would have no such restrictions on eta expansion, making the two calls to *zipWith* above indeed equivalent. And this use of unconditional eta expansion is sound because with call-by-name evaluation, the only way a function expression gets evaluated at runtime is with a fully saturated calling context. With no other possibilities (like memoization and work sharing), no one can notice the difference between the presence or absence of a lambda. So the strong eta law of call-by-name evaluation accurately captures the semantics of higher-arity functions. The parallel between theory and practice is interestingly symmetric. Call-by-value is dual to call-by-name [Curien and Herbelin 2000; Filinski 1989; Wadler 2003] and functions are (approximately) dual to tuples [Downen and Ariola 2014b; Munch-Maccagnoni 2009; Zeilberger 2008]. So is it any wonder that the connection between function arity and call-by-name is dual to unboxed tuples and call-by-value?

But there is a good reason that no practical language uses call by name: it leads to asymptotically worse performance, because work is repeated over and over. Are we doomed to choosing between either slow curried functions or bad algorithmic performance? No! Fortunately, we know how to combine the three different evaluation strategies within the same program [Downen and Ariola 2018]. And by splitting the single vague function arrow type, $a \to b$, into different types, we can express their arity in the type:

- $(\to)$ represents an ordinary function type corresponding to the source language.
- $(\rightsquigarrow)$ represents a higher-arity function type.

The key is that $\rightsquigarrow$ arrows chains together according to currying. $\mathit{Int} \rightsquigarrow \mathit{Int} \rightsquigarrow \mathit{Int} \rightsquigarrow \mathit{Int}$ represents an arity 3 function, and *all* expressions of this type are equivalent to $\widetilde{\lambda} x.\widetilde{\lambda} y.\widetilde{\lambda} z.e$ for some body e. In contrast, $\mathit{Int} \to \mathit{Int} \to \mathit{Int} \to \mathit{Int}$ represents any other function, with no promise that all three λs will be available at once; each one might need to be computed with a separate step of evaluation.

3 AN ARITY-AWARE CALCULUS FOR CURRIED FUNCTIONS

Our aim here is to formalize the call arity optimization of [Marlow and Peyton Jones 2004] in terms of a statically checked feature of a core intermediate language. And similar to [Bolingbroke and Peyton Jones 2009], we seek an intermediate language that is equally appropriate for *eager* and *lazy* functional languages. We achieve both goals simultaneously with the same key idea: use the kind system to classify types by their evaluation strategy: call-by-value, call-by-name, or call-by-need.

Types and kinds

$k, l, r, s, t \in Kind$	$::= \mathsf{L}$		Call-by-need thunks
	$\mid \mathsf{V}$		Call-by-value values
	$\mid \mathsf{F}$		Call-by-name higher-arity functions
$a, b, c \in TyVar$			
$\sigma, \tau, \rho \in Type$	$::= a$		Polymorphic type variable
	$\mid \sigma \xrightarrow{r} \tau$		r-represented function type
	$\mid \forall^r a{:}r.\tau$		r-represented polymorphic type
	$\mid \mathit{Int}$		Call-by-value integers
	$\mid \mathit{List}^{\,r}\,\tau$		r-represented lists
	$\mid \mathit{Box}^{\,r}_{\,t}\,\tau$		r-box containing t-components

Expressions

$x, y, z \in Var$			
$e, u, v \in Expr$	$::= x$		Variable
	$\mid \mathbf{let}\ d\ \mathbf{in}\ e$		Let binding
	$\mid \lambda^r \mathbf{x}.e$		Abstraction
	$\mid e\ g$		Application
	$\mid n$		Integer literal
	$\mid K$		Data constructor
	$\mid \mathbf{case}\ e\ \mathbf{of}\ \{\overline{p_i \rightarrow u_i}\}$		Case analysis
$g \in Arg$	$::= e \mid \tau$		
$K \in Constructor$	$::= \mathit{Box}^{\,r}_{\,t}$		r-box containing a t-value
	$\mid \mathit{Nil}^{\,r}$		Empty r-list
	$\mid \mathit{Cons}^{\,r}$		Nonempty r-list constructor
$d \in Bind$	$::= x{:}\tau = e$		Non-recursive binding
	$\mid \mathbf{rec}\ x{:}\tau = e$		Recursive binding
$p \in Pattern$	$::= n$		Number pattern
	$\mid K\ \overline{(x{:}\tau)}$		Constructor pattern
	$\mid x{:}\tau$		Default pattern
$\mathbf{x}, \mathbf{a} \in BoundVar$	$::= x{:}\tau$		Type-annotated value variable
	$\mid a{:}k$		Kind-annotated type variable

Fig. 1. Syntax of the core calculus $\mathcal{L}$

In fact, we present *two* intermediate languages.[3] The first, $\mathcal{L}$, is suitable for transformation and optimisation (Sections 3.1 and 3.2). It has a high-level semantics which we give in Section 3.5.

The second, $\mathcal{L}X$, is a restricted subset of $\mathcal{L}$, designed for direct execution (Section 3.3), and equipped with a low-level abstract machine that makes explicit the operational impact of multiple-argument function calls (Section 3.6). We give a translation of $\mathcal{L}$ into $\mathcal{L}X$ in Section 3.4.

3.1 The core intermediate language $\mathcal{L}$

The syntax of our core calculus is given in Figure 1. For the most part, it is a standard explicitly-typed intermediate functional language, based on System F with data types.[4] The primary novelty

[3] GHC experts will recognise these two languages as analogues of GHC Core language and STG language.

[4] Note that we treat the branches of a **case** expression as an unordered list, and that the pattern of each branch must be different from all the others. This requires that each **case** expression has at most one default branch. We will also at times

can be seen in the syntax of kinds: instead of a single base kind (often written as $\star$), there are three different base kinds (V, L, and F), each specifying one of three possible evaluation strategies for the types that inhabit them. We can make this precise by considering a let-binding $\mathbf{let}\, x{:}\tau = u\, \mathbf{in}\, e$:

- V for types τ that follow a *call-by-value evaluation strategy*. To evaluate the let-expression, first evaluate the right hand side u, bind the result to x, and then evaluate the body e. Types of kind V include Int and $List^{\mathsf{V}}\tau$ (Figure 2). If x is bound to a V-type, then x points to the value itself; it cannot point to a thunk.[5]
- L for types τ that follow a *call-by-need evaluation strategy*. To evaluate the let-expression, bind x to a heap-allocated thunk for u, and evaluate e; if and when the value of x is needed, evaluate the thunk, and overwrite the thunk with its value. Types of L include $List^{\mathsf{L}}\tau$.
- F for types that follow a *call-by-name evaluation strategy*. To evaluate the let-expression, bind x to a heap-allocated closure for u, and evaluate e. If and when the value of x is needed (which will happen only if x is applied to some arguments), pass those arguments directly to the closure. Types of kind F include $\sigma \xrightarrow{\mathsf{F}} \tau$ and $\forall^{\mathsf{F}} a{:}k.\tau$.

These kinds appear in source programs to annotate the binder in a type $\forall a{:}k.\tau$ or term $\lambda(a{:}k).e$. Kinds are also used to classify types (via the kinding judgement of Figure 2), but they also appear in three other less-standard places.

- The function type $\sigma \xrightarrow{r} \tau$ is parameterized by the kind r of the function abstraction itself. In effect, this gives us three different function types: one for each choice of r. The function abstraction form $\lambda^r x{:}\tau.e$ is correspondingly parameterized with its representation. Without the annotation, λ-abstractions would be ambiguous: is the function $\lambda x{:}\tau.x$ intended to have the type $\tau \xrightarrow{\mathsf{L}} \tau$, $\tau \xrightarrow{\mathsf{V}} \tau$, or $\tau \xrightarrow{\mathsf{F}} \tau$? By annotating the λ with the desired representation, the correct type can be inferred.
- Analogously, the polymorphic type $\forall^r a{:}k.\tau$ and corresponding abstraction $\lambda^r a{:}k.e$ are also annotated with their representation.
- Data types are also kinded (see Figure 1). To avoid clutter, in this paper we assume a handful of built-in types: Int, $List^{\mathsf{V}}\tau$, $List^{\mathsf{L}}\tau$, and a $Box^r_s\tau$ type we describe shortly. The two different list types allow for both eagerly evaluated lists ($List^{\mathsf{V}}$) which are always finite and lazily evaluated lists ($List^{\mathsf{L}}$) which may be infinite. The purpose of the $Box^r_s\tau$ type is to put an s value of type τ into a box that follows the evaluation strategy of r. This allows for composing the built-in types in more ways: for example, we can place a call-by-value Int into a lazy box of type $Box^{\mathsf{L}}_{\mathsf{V}}\, Int$, which can then be stored in a lazy list $List^{\mathsf{L}}\ (Box^{\mathsf{L}}_{\mathsf{V}}\, Int)$.[6]
  Adding user-defined data types, including GADTs, adds extra complications, but they are all orthogonal to the innovations of this paper, and we do not discuss them further.

We do not support kind polymorphism, but adding it is not hard, as we discuss in Section 4.

Through these kinds, $\mathcal{L}$ gives programmatic control of the evaluation strategy of abstractions and data structures, and hence is appropriate for modeling both eager and lazy functional languages. An eager language, like OCaml, can translate to $\mathcal{L}$ by V-annotating all function arrows, polymorphic $\forall$s, and λs from the source program, as well as the results of all data type constructors. Whereas a lazy language, like Haskell, can perform the analogous translation by annotating everything with

treat recursive and non-recursive **let** expressions the same, and write $\mathbf{let(rec)}\, x{:}\tau = u\, \mathbf{in}\, e$ to mean either $\mathbf{let}\, x{:}\tau = u\, \mathbf{in}\, e$ or $\mathbf{letrec}\, x{:}\tau = u\, \mathbf{in}\, e$.

[5] In GHC this kind is further sub-divided into call-by-value kinds with different representations. For example, GHC has one kind for types represented by 32-bit values (e.g. `Int32#`), one for types represented by 64-bit values (e.g. `Int#`), and yet another for types represented by a pointer to a heap-allocated value (e.g `Array# Int`). These distinctions are orthogonal to the purpose of the paper, so here we have a single call-by-value kind, V.

[6] GHC experts will notice that Box is a generalized version of the type-specific boxing mechanism. For example, the conversion from a strict unboxed `Int#` to the lazy `Int` is expressed by the data type definition `data Int = I# Int#`.

L. As we shall see, however, the F representation will be useful to *both* languages for optimizing curried functions. In this way, a compiler for a purely call-by-value functional language only needs to use the V and F fragment of the calculus (although it might still want to selectively use L for memoization), and one for a purely call-by-need functional language only needs to use the L and F fragment (although it might still want to use V for optimizing parameter passing).

Notation 1 (Default Annotations). We will abbreviate the types $\sigma \xrightarrow{\mathsf{F}} \tau$ and $\forall^{\mathsf{F}} a{:}k.\tau$ as $\sigma \rightsquigarrow \tau$ and $\widetilde{\forall} a{:}k$, respectively, and the corresponding abstractions $\lambda^{\mathsf{F}}\mathbf{x}.e$ as $\widetilde{\lambda}\mathbf{x}.e$. As further shorthand, we will write the call-by-need types $\sigma \xrightarrow{\mathsf{L}} \tau$ and $\forall^{\mathsf{L}} a{:}k.\tau$ as just $\sigma \to \tau$ and $\forall a{:}k.\tau$, respectively, and the corresponding abstractions $\lambda^{\mathsf{L}}\mathbf{x}.e$ as just $\lambda\mathbf{x}.e$. This effectively makes call-by-need the default evaluation strategy. V could have just as easily be used instead of L for a call-by-value default.

3.2 The type system of $\mathcal{L}$

The type system for $\mathcal{L}$ is given in Figure 2. The rules are all standard for System F with data types, and we assume the usual side condition that the typing rule for **case** expressions must exhaustively cover the possible patterns (which is always ensured when there is a default branch). However, there are a couple of additional unusual side conditions.

The first is that we want to ensure that if $\tau : \mathsf{F}$, then τ must be a function type or a forall; that is F classifies only function types. Moreover, these types must be *monomorphic*, in the sense that we cannot have some unknown type a of kind F. Formally:

Property 1 (Monomorphic F closures). *If* $\Theta \vdash \tau : \mathsf{F}$ *then either*

- $\tau = \sigma \rightsquigarrow \rho$, *where* $\Theta \vdash \sigma : s$ *and* $\Theta \vdash \rho : r$, *or*
- $\tau = \widetilde{\forall} a{:}k.\sigma$, *where* $\Theta, a{:}k \vdash \sigma : s$.

We guarantee this property, in the kind-checking judgement for types, by ensuring that both polymorphic type variables (a) and data type constructors (T) must have so-called *observable types*, written $obs(\tau)$, of kind L or V, but not F (Figure 2). This property matters because we want the *type* of a function to encode the function's *arity*, and that would go awry if the function had type $\widetilde{\forall} a{:}\mathsf{F}.\ Int \rightsquigarrow a$, where $a : \mathsf{F}$ could be instantiated to a ($\rightsquigarrow$) type (see Remark 1).

The second restriction is that recursive bindings can't bind eager variables of kind V, but must be for some *delayable type* of kind L or F; hence the side condition $delay(\tau)$ in the **letrec** rule. This restriction is standard in call-by-value languages, where it is called the "value restriction". It prevents nonsensical bindings like $\mathbf{let}\, x : Int = x + 1\, \mathbf{in}\, e$. We exclude such bindings not with a conventional syntactic restriction, but rather through the kind system: for L and F types, a **letrec** is absolutely fine. For example, even for a call-by-value language, the real recursive work of a function definition can be given a F function type, as in

$$
\begin{aligned}
&fact : Int \xrightarrow{\mathsf{V}} Int \\
&fact = \ \mathbf{letrec}\, fact' : Int \rightsquigarrow Int = \widetilde{\lambda} x{:}Int. \\
&\qquad\qquad \mathbf{if}\ (x == 0)\ \mathbf{then}\ 1\ \mathbf{else}\ n * fact'(n-1) \\
&\qquad \mathbf{in}\ \lambda^{\mathsf{V}} x{:}Int.fact'x
\end{aligned}
$$

Of course infinite loops like $\mathbf{letrec}\, x{:}\tau \xrightarrow{\mathsf{L}} \tau = x\, \mathbf{in}\, x$ are still possible, but this is not a problem at runtime since x is bound immediately without trying to evaluate the right-hand side.

The third restriction is dual to that for **letrec**: case analysis can only be applied to expressions of observable types (*i.e.,* types of observable kinds), enforced by the $obs\,\tau$ side condition in the **case** rule. The idea here is that all you can do to a function $f : \sigma \rightsquigarrow \tau$ is *apply it*. Unlike a normal function of type $g : \sigma \to \tau$, you cannot *evaluate it* with **case**. Why? Because doing

Typing environments

$$\Gamma ::= \bullet \mid \Gamma, x : \tau \qquad\qquad \Theta ::= \bullet \mid \Theta, a : k$$

Constraints on representations and types

$$\frac{}{obs(\mathsf{L})} \quad \frac{}{obs(\mathsf{V})} \quad \frac{\Theta \vdash \tau : r \quad obs(r)}{\Theta \vdash obs(\tau)} \qquad \frac{}{delay(\mathsf{L})} \quad \frac{}{delay(\mathsf{F})} \quad \frac{\Theta \vdash \tau : r \quad delay(r)}{\Theta \vdash delay(\tau)}$$

Checking kinds $\boxed{\Theta \vdash \tau : k}$

$$\frac{}{\Theta \vdash Int : \mathsf{V}}\ IntTy \qquad \frac{\Theta \vdash \tau : t \quad obs(r)}{\Theta \vdash Box^{\,r}_{\,t}\,\tau : r}\ BoxTy \qquad \frac{\Theta \vdash \tau : r \quad obs(r)}{\Theta \vdash List^{\,r}\,\tau : r}\ ListTy$$

$$\frac{obs(r)}{\Theta, a : r \vdash a : r}\ TyVar \qquad \frac{\Theta \vdash \sigma : s \quad \Theta \vdash \tau : t}{\Theta \vdash \sigma \xrightarrow{r} \tau : r} \rightarrow Ty \qquad \frac{\Theta, a : k \vdash \tau : t \quad obs(k)}{\Theta \vdash \forall^r a{:}k.\tau : r}\ \forall Ty$$

Constructor types $\boxed{K : \tau}$

$$Box^{\,r}_{\,t} : \widetilde{\forall} a{:}t.a \rightsquigarrow Box^{\,r}_{\,t}\,a \qquad Nil^{\,r} : \widetilde{\forall} a{:}r.\ List^{\,r}\,a \qquad Cons^{\,r} : \widetilde{\forall} a{:}r.a \rightsquigarrow List^{\,r}\,a \rightsquigarrow List^{\,r}\,a$$

Checking types $\boxed{\Theta; \Gamma \vdash e : \tau}$

$$\frac{}{\Theta; \Gamma, x : \tau \vdash x : \tau}\ Var \qquad \frac{}{\Theta; \Gamma \vdash n : Int}\ Num \qquad \frac{K : \tau}{\Theta; \Gamma \vdash K : \tau}\ Constr$$

$$\frac{\Theta; \Gamma, x : \sigma \vdash e : \tau}{\Theta; \Gamma \vdash \lambda^r x{:}\sigma.e : \sigma \xrightarrow{r} \tau} \rightarrow I \qquad \frac{\Theta; \Gamma \vdash e : \sigma \xrightarrow{r} \tau \quad \Theta; \Gamma \vdash u : \sigma}{\Theta; \Gamma \vdash e\ u : \tau} \rightarrow E$$

$$\frac{obs(k) \quad \Theta, a : k; \Gamma \vdash e : \tau}{\Theta; \Gamma \vdash \lambda^r a{:}k.e : \forall^r a{:}k.\tau} \forall I \qquad \frac{\Theta; \Gamma \vdash e : \forall^r a{:}k.\tau \quad \Theta \vdash \sigma : k}{\Theta; \Gamma \vdash e\ \sigma : \tau[\sigma/a]} \forall E$$

$$\frac{\Theta; \Gamma \vdash u : \tau \quad \Theta; \Gamma, x : \tau \vdash e : \rho}{\Theta; \Gamma \vdash \mathbf{let}\, x{:}\tau = u\, \mathbf{in}\, e : \rho}\ Let \qquad \frac{\Theta; \Gamma, x : \tau \vdash u : \tau \quad \Theta; \Gamma, x : \tau \vdash e : \rho \quad \Theta \vdash delay(\tau)}{\Theta; \Gamma \vdash \mathbf{letrec}\, x{:}\tau = u\, \mathbf{in}\, e : \rho}\ LetRec$$

$$\frac{\Theta; \Gamma \vdash e : \tau \quad \Theta \vdash obs(\tau) \quad \Theta; \Gamma_i \vdash p_i : \tau \quad \Theta; \Gamma, \Gamma_i \vdash u_i : \rho \quad \ldots}{\Theta; \Gamma \vdash \mathbf{case}\, e\, \mathbf{of}\, \{\overline{p_i \rightarrow u_i}\} : \rho}\ Case$$

Checking patterns $\boxed{\Theta; \Gamma \vdash p : \tau}$

$$\frac{\Theta \vdash obs(\tau)}{\Theta; x : \tau \vdash (x{:}\tau) : \tau}\ DefPat \qquad \frac{}{\Theta; \bullet \vdash n : Int}\ NumPat \qquad \frac{}{\Theta; x : \tau \vdash Box^{\,r}_{\,t}\ \tau\ (x{:}\tau) : Box^{\,r}_{\,t}\tau}\ BoxPat$$

$$\frac{}{\Theta; \bullet \vdash Nil^{\,r}\ \tau : List^{\,r}\,\tau}\ NilPat \qquad \frac{}{\Theta; x : \tau, y : List^{\,r}\,\tau \vdash Cons^{\,r}\ \tau\ (x{:}\tau)\ (y{:}\,List^{\,r}\,\tau) : List^{\,r}\,\tau}\ ConsPat$$

Fig. 2. Type system of $\mathcal{L}$

so would threaten η-equivalence, by distinguishing $\bot$ from $\lambda x.(\bot x)$, where $\bot$ stands for an infinite loop. In particular, notice how $\mathbf{case}\ \bot\ \mathbf{of}\ \{z{:}\tau \rightarrow 5\}$ will loop forever, but after applying η-expansion, $\mathbf{case}\ \lambda x.(\bot x)\ \mathbf{of}\ \{z{:}\tau \rightarrow 5\}$ evaluates to 5 instead. Forbidding such **case** analysis protects η-equivalence for the type $\sigma \rightsquigarrow \tau$ (and similarly $\widetilde{\forall} a{:}k.\tau$), meaning that these function types are *extensional*, in contrast to the non-extensional function type $\sigma \rightarrow \tau$.

Remark 1. As mentioned in Property 1, polymorphism is forbidden for types of kind F. For example, observe how the definition

$$applyToOne : \widetilde{\forall} a{:}\mathsf{F}.(Int \rightsquigarrow a) \rightsquigarrow a$$
$$applyToOne = \widetilde{\lambda} a{:}\mathsf{F}.\widetilde{\lambda} f{:}(Int \rightsquigarrow a).\ f\ 1$$

which applies the higher-order function parameter f is rejected; $\widetilde{\forall} a{:}\mathsf{F}.(Int \rightsquigarrow a) \rightsquigarrow a$ is an ill-kinded type due to the unsatisfied constraint $obs(\mathsf{F})$ when checking the quantifier

$$\frac{a : \mathsf{F} \vdash (Int \rightsquigarrow a) \rightsquigarrow a : \mathsf{F} \quad obs(\mathsf{F})}{\vdash \widetilde{\forall} a{:}\mathsf{F}.(Int \rightsquigarrow a) \rightsquigarrow a : \mathsf{F}}\ \forall Ty$$

Types of kind F must be monomorphic. The reason for this constraint is that, without knowing what a is, there is not enough information to properly η-expand *applyToOne*. For example, if we specialize a to the unary $b \rightsquigarrow c$, then we should η-expand the definition once as in

$$applyToOne\ (b \rightsquigarrow c) : (Int \rightsquigarrow b \rightsquigarrow c) \rightsquigarrow b \rightsquigarrow c$$
$$applyToOne\ (b \rightsquigarrow c) =_\eta \widetilde{\lambda} f{:}(Int \rightsquigarrow b \rightsquigarrow c).\widetilde{\lambda} x{:}b.\ f\ 1\ x$$

However, we could also specialize a to $b \rightsquigarrow c \rightsquigarrow d$, which would lead to an alternative definition:

$$applyToOne\ (b \rightsquigarrow c \rightsquigarrow d) : (Int \rightsquigarrow b \rightsquigarrow c \rightsquigarrow c) \rightsquigarrow b \rightsquigarrow c \rightsquigarrow d$$
$$applyToOne\ (b \rightsquigarrow c \rightsquigarrow d) =_\eta \widetilde{\lambda} f{:}(Int \rightsquigarrow b \rightsquigarrow c \rightsquigarrow d).\widetilde{\lambda} x{:}b.\widetilde{\lambda} y{:}c.\ f\ 1\ x\ y$$

Without specializing a in advance, we don't know which is properly η-expanded.

3.3 The executable language $\mathcal{L}X$

The relative flexibility of the intermediate language $\mathcal{L}$ makes it easier to write, transform, and optimise programs. However, the downside of this flexibility is that it is more difficult to give precise operational intuitions about how a program is executed. For this reason, we also define $\mathcal{L}X$, a restricted subset of $\mathcal{L}$, which is suitable for direct execution by a standard abstract machine model. The syntax of $\mathcal{L}X$ is the same as that of $\mathcal{L}$ (Figure 1); the $\mathcal{L}X$ subset is carved out by the more stringent type system in Figure 3, which can be summarized as follows:

(1) All arguments must be an atomic parameter P: either a variable (x) for function application or a type (σ) for polymorphic instantiation. In effect, $\mathcal{L}X$ is in A-normal form [Sabry and Felleisen 1993].
(2) All constructors must be saturated (*i.e.,* fully applied), which is expressed by the side condition of the $\eta Constr$ rule: an expression of the form $K\ \overline{P}$ is only well-typed when K is applied to enough arguments to return a result of an observable type, which is one of the data types of $\mathcal{L}X$ (integers, lists, or boxes).
(3) The body of every λ-abstraction, and the right hand side of every **let** or **letrec**, must be fully η-expanded, having a $\widetilde{\lambda}$ present for every leading $\rightsquigarrow$ arrow and $\widetilde{\forall}$ quantifier in its type. In effect, the F-kinded fragment of $\mathcal{L}X$ is in η-long form. This property is established by the judgement $\Theta; \Gamma \vdash^{\mathsf{F}}_{\eta} e : \tau$.
(4) Just as in $\mathcal{L}$, the expression discriminated by a **case** must be of an observable type, and the right hand side of a **letrec** must be delayable.
(5) Unlike $\mathcal{L}$, the right hand side of a non-recursive **let** must be of a delayable type.
(6) Again unlike $\mathcal{L}$, the expression returned by a **let** and in every branch of a **case** must be of an observable type.

Parameters

$$P \in Param ::= x \mid \tau \qquad \overline{P} ::= P_0, P_1, \ldots$$

Restricted typing rules $\boxed{\Theta; \Gamma \vdash_\eta e : \tau}$

$$\frac{}{\Theta; \Gamma, x : \tau \vdash_\eta x : \tau}\ \eta Var \qquad \frac{}{\Theta; \Gamma \vdash_\eta n : Int}\ \eta Num \qquad \frac{\Theta; \Gamma \vdash K\ \overline{P} : \tau \quad \Theta \vdash obs(\tau)}{\Theta; \Gamma \vdash_\eta K\ \overline{P} : \tau}\ \eta Constr$$

$$\frac{\Theta; \Gamma, x : \tau \vdash^{\mathsf{F}}_\eta e : \sigma}{\Theta; \Gamma \vdash_\eta \lambda^r x{:}\tau.e : \tau \xrightarrow{r} \sigma}\ \eta{\to}I \qquad \frac{\Theta; \Gamma, x : \tau \vdash_\eta e : \tau \xrightarrow{r} \sigma}{\Theta; \Gamma, x : \tau \vdash_\eta e\ x : \sigma}\ \eta{\to}E$$

$$\frac{\Theta, a : k; \Gamma \vdash^{\mathsf{F}}_\eta e : \tau}{\Theta; \Gamma \vdash_\eta \lambda^r a{:}k.e : \forall^r a{:}k.\tau}\ \eta\forall I \qquad \frac{\Theta; \Gamma \vdash_\eta e : \forall^r a{:}k.\tau \quad \Theta \vdash \sigma : k}{\Theta; \Gamma \vdash_\eta e\ \sigma : \tau[\sigma/a]}\ \eta\forall E$$

$$\frac{\Theta; \Gamma \vdash^{\mathsf{F}}_\eta u : \tau \quad \Theta; \Gamma, x : \tau \vdash_\eta e : \rho \quad \Theta \vdash delay(\tau) \quad \Theta \vdash obs(\rho)}{\Theta; \Gamma \vdash_\eta \mathbf{let}\, x{:}\tau = u\, \mathbf{in}\, e : \rho}\ \eta Let$$

$$\frac{\Theta; \Gamma, x : \tau \vdash^{\mathsf{F}}_\eta u : \tau \quad \Theta; \Gamma, x : \tau \vdash_\eta e : \rho \quad \Theta \vdash delay(\tau) \quad \Theta \vdash obs(\rho)}{\Theta; \Gamma \vdash_\eta \mathbf{letrec}\, x{:}\tau = u\, \mathbf{in}\, e : \rho}\ \eta LetRec$$

$$\frac{\Theta; \Gamma \vdash_\eta e : \tau \quad \Theta \vdash obs(\tau) \quad \Theta; \Gamma_i \vdash p_i : \tau \quad \Theta; \Gamma, \Gamma_i \vdash u_i : \rho \quad \ldots \quad \Theta \vdash obs(\rho)}{\Theta; \Gamma \vdash_\eta \mathbf{case}\, e\, \mathbf{of}\, \{\overline{p_i \to u_i}\}}\ \eta Case$$

Checking η-expansion $\boxed{\Theta; \Gamma \vdash^{\mathsf{F}}_\eta e : \tau}$

$$\frac{\Theta; \Gamma, x : \tau \vdash^{\mathsf{F}}_\eta e : \sigma}{\Theta; \Gamma \vdash^{\mathsf{F}}_\eta \widetilde{\lambda} x{:}\tau.e : \tau \rightsquigarrow \sigma}\ \eta{\rightsquigarrow}I \qquad \frac{\Theta, a : k; \Gamma \vdash^{\mathsf{F}}_\eta e : \tau}{\Theta; \Gamma \vdash^{\mathsf{F}}_\eta \widetilde{\lambda} a{:}k.e : \widetilde{\forall} a{:}k.\tau}\ \eta\widetilde{\forall} \qquad \frac{\Theta \vdash obs(\tau) \quad \Theta; \Gamma \vdash_\eta e : \tau}{\Theta; \Gamma \vdash^{\mathsf{F}}_\eta e : \tau}\ \eta Obs$$

Plus the same kind and pattern rules as in Figure 2.

Fig. 3. Type system of $\mathcal{L}X$

These restrictions force $\widetilde{\lambda}$-abstractions to appear only in very constrained ways. First, a $\widetilde{\lambda}$-abstraction can appear only as the right hand side of a **let** or **letrec**, the body of another $\widetilde{\lambda}$-abstraction, or in the function position of a saturated call. In particular, it is not possible to write a $\widetilde{\lambda}$-abstraction as the returned result of a **let** binding or **case** branch. This limitation helps to understand the effect of call-by-name functions on sharing. For example, the expression *fact* 1000 will be repeated on every application of $\mathbf{let}\, n{:}\, Int\ =\ fact\ 1000\, \mathbf{in}\, \widetilde{\lambda}\mathbf{x}.n + x$. The lack of sharing is easier to see in the η-expanded $\widetilde{\lambda}\mathbf{x}.\, \mathbf{let}\, n{:}\, Int\ =\ fact\ 1000\, \mathbf{in}\, n + x$, which is the correct way to write this expression according to Figure 3.

Second, named applications of call-by-name functions must be saturated. For example, instead of

$$\begin{array}{l}\mathbf{let}\ f : Int \rightsquigarrow Int \rightsquigarrow Int\ = \ldots \\ \quad\ \ g : Int \rightsquigarrow Int\ \ \ = f\ 1 \\ \mathbf{in}\, g\ 2\end{array}$$

we would have to η-expand g like so:

$$\begin{array}{l}\mathbf{let}\ f : Int \rightsquigarrow Int \rightsquigarrow Int\ = \ldots \\ \quad\ \ g : Int \rightsquigarrow Int\ \ \ = \widetilde{\lambda} x{:}\, Int\, .f\ 1\ x \\ \mathbf{in}\, g\ 2\end{array}$$

Type-driven transformation $\boxed{\mathcal{E}[\![e]\!]\overline{P}{:}\tau}$

$$\mathcal{E}[\![\widetilde{\lambda}\mathbf{x}.e]\!]\bullet{:}\sigma = \widetilde{\lambda}\mathbf{x}.\mathcal{E}[\![e]\!]\bullet{:}\tau \qquad (\text{if } e : \tau)$$
$$\mathcal{E}[\![e]\!]\overline{P}{:}\sigma{\rightsquigarrow}\tau = \widetilde{\lambda}x{:}\sigma.\mathcal{E}[\![e]\!]\overline{P}x{:}\tau \qquad (\text{if } e \neq \widetilde{\lambda}y{:}\sigma.e')$$
$$\mathcal{E}[\![e]\!]\overline{P}{:}\widetilde{\forall}a{:}k.\tau = \widetilde{\lambda}a{:}k.\mathcal{E}[\![e]\!]\overline{P}a{:}\tau \qquad (\text{if } e \neq \widetilde{\lambda}a{:}k.e')$$
$$\mathcal{E}[\![e]\!]\overline{P}{:}\tau = C[\![e]\!]\overline{P} \qquad (\text{if } obs(\tau))$$

Type-generic transformation $\boxed{C[\![e]\!]\overline{P}}$

$$C[\![e]\!]\overline{P} = e\ \overline{P} \qquad (\text{if } e = x, n, \text{ or } K)$$
$$C[\![\lambda^r\mathbf{x}.e]\!]\overline{P} = (\lambda^r\mathbf{x}.\mathcal{E}[\![e]\!]\bullet{:}\tau)\ \overline{P} \qquad (\text{if } e : \tau)$$
$$C[\![e\ u]\!]\overline{P} = C[\![\mathbf{let}\ x{:}\tau = u\ \mathbf{in}\ e\ x]\!]\overline{P} \qquad (\text{if } u \notin Param \text{ and } u : \tau)$$
$$C[\![e\ P']\!]\overline{P} = C[\![e]\!]P'\overline{P}$$
$$C[\![\mathbf{let(rec)}\ x{:}\tau = u\ \mathbf{in}\ e]\!]\overline{P} = \mathbf{let(rec)}\ x{:}\tau = \mathcal{E}[\![u]\!]\bullet{:}\tau\ \mathbf{in}\ C[\![e]\!]\overline{P} \qquad (\text{if } delay(\tau))$$
$$C[\![\mathbf{let}\ x{:}\tau = u\ \mathbf{in}\ e]\!]\overline{P} = \mathbf{case}\ C[\![u]\!]\bullet\ \mathbf{of}\ \{x{:}\tau \to C[\![e]\!]\overline{P}\} \qquad (\text{if } \tau : \mathsf{V})$$
$$C[\![\mathbf{case}\ e\ \mathbf{of}\ \{\overline{p_i \to u_i}\}]\!]\overline{P} = \mathbf{case}\ C[\![e]\!]\bullet\ \mathbf{of}\ \{\overline{p_i \to C[\![u_i]\!]\overline{P}}\}$$

Fig. 4. Pre-processing transformation from $\mathcal{L}$ to $\mathcal{L}X$

This more restrictive, η-expansion enforcing, type system makes sure that multi-arity function calls are safe to execute, because the arities will match at runtime (as we will soon see in the abstract machine in Section 3.6).

3.4 Compiling $\mathcal{L}$ into $\mathcal{L}X$

It is straightforward to translate $\mathcal{L}$ into $\mathcal{L}X$ as shown in Figure 4. The compilation has two different modes ($\mathcal{E}[\![e]\!]{:}\tau$ and $C[\![e]\!]$) which depend on one another. The first mode ($\mathcal{E}[\![e]\!]{:}\tau$) is driven by the type of the $\mathcal{L}$ expression being processed, which it uses to η-expand that expression as appropriate and build up a chain of parameters until the longest possible chain of curried F abstractions have been created. This makes the implicit type information of F types explicit in the syntax of the expression, so that expressions of F do indeed *look like* manifest values.

The other mode ($C[\![e]\!]$) does not depend on the type of the expression, and instead handles the other conditions of the runtime type system. In particular, the parameters ($\overline{P}$) that were built up from η-expansion are pushed into the bodies of **let** and **case** expressions. Pushing the arguments inward into these block-forms of expression has the net effect of making sure that the return type of every **let** and **case** is no longer a F type, satisfying the runtime typing constraint that they return an observable result. This mode of pre-processing also handles other issues relevant to evaluation strategy by naming non-parameter arguments with a **let** binding, and converting **let** bindings of V types, which are meant to be strict, into **case** expressions with only a default branch, which are manifestly strict.

The overall result of the η-expanding pre-processing is that we can transform $\mathcal{L}$ expression into $\mathcal{L}X$ ones: $\mathcal{E}[\![e]\!]{:}\tau$ transforms any $\mathcal{L}$ expression of type τ into a $\mathcal{L}X$ expression of the same type is transformed into a well-typed $\mathcal{L}X$ expression of the same type, and $C[\![e]\!]$ transforms $\mathcal{L}$ programs

(expressions of observable types) into $\mathcal{L}X$ programs. Therefore, we can prepare any well-typed $\mathcal{L}$ expression for direct execution by first pre-processing it.

Proposition 1 (Compilation). *For any $\mathcal{L}$ expression $\Theta;\Gamma \vdash e : \tau$*

(1) $\Theta;\Gamma \vdash^{\mathsf{F}}_{\eta} (\mathcal{E}[\![e]\!]\bullet{:}\tau) : \tau$ *is an $\mathcal{L}X$ expression, and*
(2) $\Theta;\Gamma \vdash_{\eta} (C[\![e]\!]\bullet) : \tau$ *is an $\mathcal{L}X$ expression if $\Theta \vdash obs(\tau)$.*

3.5 Operational semantics and equational theory for $\mathcal{L}$

To understand how to evaluate expressions of the intermediate language $\mathcal{L}$, we give a high-level operational semantics in the form of a rewriting and equational theory in Figure 5. Primarily, the operational semantics of $\mathcal{L}$ corresponds to an extension of the call-by-need λ-calculus [Ariola et al. 1995] in order to support lazy data structures like lists. This is achieved by giving a name to all non-trivial expression arguments when they are first seen with the *name* rule. That way, the components of constructors are shared. For example, the result of *fact* 1000 will be shared in the first component of the list $Cons^{\mathsf{L}}(Box^{\mathsf{L}}_{\mathsf{V}}\, Int\,)\ (fact\ 1000)\ xs$. This way, the β-reduction of applications and the reduction of **case** expressions can be simplified by the assumption that sharing and evaluation order are already taken care of by the *name* rule. The *float* rule performs both the *lift* and *assoc* reductions of the call-by-need λ-calculus, and covers **case** commutations, too.

Naming arguments also has the pleasant benefit of giving a uniform presentation of the three evaluation strategies by condensing the difference between call-by-name, call-by-value, and call-by-need evaluation into **let** expressions. The memoization of call-by-need evaluation requires us to deal with delayed bindings, and so the difference between the three can be seen in the rules for evaluating a **let**. When encountering an expression $\mathbf{let}\, x{:}\tau = u\, \mathbf{in}\, e$, one of three things will happen depending on the representation of the type τ:

- If $\tau : \mathsf{F}$, then u will be delayed, e will be evaluated first due to the fact that $\mathbf{let}\, x{:}\tau = u\, \mathbf{in}\, \square$ is a binding context for the delayable type τ. If x is ever required during evaluation of e, then u will be copied in for x as it, duplicating the evaluation of u each time x is used.
- If $\tau : \mathsf{V}$, then u will be eagerly evaluated first, due to the fact that $\mathbf{let}\, x{:}\tau = \square\, \mathbf{in}\, e$ is a frame context. When (and if) u reduces to a weak head-normal form W, e will be evaluated in the context of the simplified binding of x to W. Note that evaluating u might allocate several bindings (represented by contexts B_1 to B_n), which will all be shared when attention is moved to e. This is achieved by the *float* rule like so:

  $$\mathbf{let}\, x{:}\tau = u\, \mathbf{in}\, e \mapsto^{*} \mathbf{let}\, x{:}\tau = B_1[\ldots B_n[W]]\, \mathbf{in}\, e \mapsto^{*}_{float} B_1[\ldots B_n[\mathbf{let}\, x{:}\tau = W\, \mathbf{in}\, e]]$$

  which continues with the evaluation of e.
- If $\tau : \mathsf{L}$, then u will be delayed and e will be evaluated first, like in the case for F. However, the first time that x is needed in e, the context $\mathbf{let}\, x{:}\tau = u\, \mathbf{in}\, E[x]$ is converted to a frame context of the form $\mathbf{let}\, x{:}\tau = \square\, \mathbf{in}\, E[x]$. In this case, the evaluation will switch to work on u, which will proceed in the same manner as in the case for V.

In addition to the operational reduction rules, needed to run programs to get an answer, we also consider some additional reductions that might be used for optimization in a compiler. The main ones of interest are the η rules for both functional and polymorphic types. Call-by-value functions only support a restricted η rule for values (because an infinite $loop : \sigma \xrightarrow{\mathsf{V}} \tau$ is observably different from $\lambda^{\mathsf{V}}\mathbf{x}.loop\ x$), and call-by-need functions don't support η at all (due to the fact that a **case** can force evaluation of an infinite $loop : \sigma \xrightarrow{\mathsf{L}} \tau$)! But η-equivalence holds unconditionally for expressions of type $\sigma \rightsquigarrow \tau$, which is one of our key insights.

In addition to the η laws, we also include the ability to *lift* applications into any tail-calling context, which includes both **let** and **case** expressions, and the conversion from a strict **let** to a

Answers and evaluation contexts

$$
\begin{array}{lll}
A \in \mathit{Answer} & ::= W \mid B[A] & \\
W \in \mathit{WHNF} & ::= \lambda^r \mathbf{x}.e \mid K\ \overline{P} \mid n & \\
P \in \mathit{Param} & ::= x \mid \tau & \\
E \in \mathit{EvalCxt} & ::= \Box & \text{Empty evaluation context} \\
 & \mid F[E] & \text{Pushing a stack frame} \\
 & \mid B[E] & \text{Allocating a binding} \\
B \in \mathit{BindCxt} & ::= \mathbf{let(rec)}\ x{:}\tau = e\ \mathbf{in}\ \Box \quad (\text{if } \mathit{delay}(\tau)) & \text{Delayed binding} \\
 & \mid \mathbf{let(rec)}\ x{:}\tau = W\ \mathbf{in}\ \Box & \text{Value binding} \\
F \in \mathit{FrameCxt} & ::= \Box\ P & \text{Applying a parameter} \\
 & \mid \mathbf{case}\ \Box\ \mathbf{of}\ \{\overline{p_i \to e_i}\} & \text{Performing case analysis} \\
 & \mid \mathbf{let}\ x{:}\tau = \Box\ \mathbf{in}\ e \quad (\text{if } \tau : \mathsf{V}) & \text{Evaluating a strict let} \\
 & \mid \mathbf{let(rec)}\ x{:}\tau = \Box\ \mathbf{in}\ E[x] \quad (\text{if } \tau{:}\mathsf{L}, x \notin BV(E)) & \text{Forcing a needed thunk} \\
L \in \mathit{TailCxt} & ::= \Box & \text{Empty tail context} \\
 & \mid \mathbf{let(rec)}\ x{:}\tau = e\ \mathbf{in}\ L & \text{Result of a let binding} \\
 & \mid \mathbf{case}\ e\ \mathbf{of}\ \{\overline{p_i \to L_i}\} & \text{Results of branching}
\end{array}
$$

Pattern matching $\boxed{W \hookrightarrow p[\overline{P/x}]}$

$$\overline{n \hookrightarrow n[\bullet]} \qquad \overline{\mathit{Box}^r_t\ \tau\ P \hookrightarrow \mathit{Box}^r_t\ (x{:}\tau)[P/x]}$$

$$\overline{\mathit{Nil}^r\ \tau \hookrightarrow \mathit{Nil}^r[\bullet]} \qquad \overline{\mathit{Cons}^r\ \tau\ P\ P' \hookrightarrow \mathit{Cons}^r\ (x{:}\tau)\ (y{:}\mathit{List}^r\ \tau)[P/x, P'/y]}$$

Operational rules $\boxed{e \mapsto e'}$

$$
\begin{array}{rll}
(\beta_P) & (\lambda^r \mathbf{x}.e)\ P \mapsto e[P/\mathbf{x}] & \\
(\mathit{match}) & \mathbf{case}\ W\ \mathbf{of}\ \{\overline{p_i \to u_i}; p \to e; \ldots\} \mapsto e[\overline{P/x}] & (\text{if } W \hookrightarrow p[\overline{P/x}]) \\
(\mathit{default}) & \mathbf{case}\ W\ \mathbf{of}\ \{\overline{p_i \to u_i}; x{:}\tau \to e\} \mapsto \mathbf{let}\ x{:}\tau = W\ \mathbf{in}\ e & (\text{if } W \not\hookrightarrow p_i) \\
(\mathit{lookup}) & \mathbf{let(rec)}\ x{:}\tau = e\ \mathbf{in}\ E[x] \mapsto \mathbf{let(rec)}\ x{:}\tau = e\ \mathbf{in}\ E[e] & \\
 & \qquad (\text{if } x \notin BV(E) \text{ and } e \in \mathit{WHNF} \text{ or } \tau{:}\mathsf{F}) & \\
(\mathit{float}) & F[B[A]] \mapsto B[F[A]] & \\
(\mathit{name}) & e\ u \mapsto \mathbf{let}\ x{:}\tau = u\ \mathbf{in}\ e\ x & (\text{if } u \notin \mathit{Param} \text{ and } u : \tau)
\end{array}
$$

Extensional rules $\boxed{e \to e'}$

$$
\begin{array}{rll}
(\eta_{\rightsquigarrow}) & e \to \widetilde{\lambda} x{:}\sigma.e\ x & (\text{if } e : \sigma \rightsquigarrow \tau) \\
(\eta_{\widetilde{\forall}}) & e \to \widetilde{\lambda} a{:}k.e\ a & (\text{if } e : \widetilde{\forall} a{:}k.\tau) \\
(\mathit{lift}) & L[\overline{e}]\ P \to L[\overline{e\ P}] & \\
(\mathit{case/let_V}) & \mathbf{let}\ x{:}\tau = u\ \mathbf{in}\ e \to \mathbf{case}\ u\ \mathbf{of}\ \{x{:}\tau \to e\} & (\text{if } u \notin \mathit{WHNF} \text{ and } \tau : \mathsf{V})
\end{array}
$$

Typed equality $\boxed{\Theta; \Gamma \vdash e = e' : \tau}$

$$\frac{e \mapsto e'}{e \to e'} \qquad \frac{\Theta; \Gamma \vdash e : \tau \quad e \to e' \quad \Theta; \Gamma \vdash e' : \tau}{\Theta; \Gamma \vdash e = e' : \tau}$$

Plus rules for reflexivity, symmetry, transitivity, and congruence.

Fig. 5. Operational semantics and equational theory for $\mathcal{L}$

case with only a default branch, which can help to clarify evaluation order without needing type information. The interest of these extra, extensional, reduction rules is that they capture the entire pre-processing transformation in terms of individual small steps.

Proposition 2 (η Elaboration)**.** *Given* $\Theta; \Gamma \vdash e : \tau$*,*

(1) $\Theta; \Gamma \vdash (\mathcal{E}[\![e]\!]\bullet{:}\tau) = e : \tau$.
(2) $\Theta; \Gamma \vdash (C[\![e]\!]\bullet) = e : \tau$*, and*

Each of these reduction rules preserve the types of expressions, keeping $\mathcal{L}$ type safe.

Definition 1 (Programs, Answers, and Cycles)**.** An $\mathcal{L}$ *program* is an expression $\Theta; \bullet \vdash e : \tau$ where $\Theta \vdash obs(\tau)$, an *answer* is an expression of the form A, and a *cycle* is an expression of the form $E_1[\mathbf{letrec}\, x{:}\tau = E_3[x]\, \mathbf{in}\, E_2[x]]$ where x is not bound by E_2.

Proposition 3 (Type Safety of $\mathcal{L}$)**.** *For all* $\mathcal{L}$ *expressions* e*:*

(1) Progress: *If* e *is a program then either* e *is an answer,* e *is a cycle, or* $e \mapsto e'$ *for some* e'.
(2) Subject reduction: *If* $\Theta; \Gamma \vdash e : \tau$ *and* $e \to e'$ *then* $\Theta; \Gamma \vdash e' : \tau$.

Remark 2. Let's at some examples of how the correct evaluation for higher-arity functions avoids the possibility of arity mismatch at runtime. The purpose of taking arity into account is so that we can pass many values at once to a chain of abstractions (as we will soon do explicitly in section 3.6), effectively performing many individual β-reductions in one step like so:

$$(\beta^n) \qquad\qquad (\lambda^{r_1} x_1 \ldots \lambda^{r_n} x_n.e)\ V_1 \ldots V_n \mapsto e[V_1/x_1 \ldots V_n/x_n]$$

For this n-ary β-reduction to work properly, the number of λs must align with the number of parameters *exactly*. To be efficient and avoid runtime checks, this invariant must be known at compile-time, referred to as a *known* function call. In [Marlow and Peyton Jones 2004], known function calls could only be calls to functions with a known definition. This restriction leaves out higher-order function calls, where the definition of the called function is unknown because it was passed as a parameter, like in the higher-order `map :: (a -> b) -> [a] -> [b]` function.

What could go wrong with higher-arity, higher-order functions? First, there may be too few λs for a desired arity. For example, consider the higher-order function (eliding type annotations):

$$\begin{aligned} &applyToOne : (Int \to Int \to Int) \to Int \to Int \\ &applyToOne = \lambda f.f\ 1 \end{aligned}$$

The problem here is that *applyToOne* has one too few λs in it's definition, which makes it appear as if f is only called with the single argument 1. But at runtime, we expect to pass a binary function like addition, $(+) : Int \to Int \to Int$ along with a second argument like so:

$$applyToOne\ (+)\ 2 = (\lambda f.f\ 1)\ (+)\ 2 \mapsto (+)\ 1\ 2$$

where the $(+)$ operator (formerly f) is called with exactly two arguments. How can we fix too few λs? Eta expansion! Expanding out the definition of *applyToOne* according to its type gives:

$$applyToOne =_\eta \lambda f.\lambda x.f\ 1\ x$$

However, care must be taken with this expansion! Suppose instead we tried to evaluate the call $applyToOne\ (\lambda n.n/0)$. As written,

$$applyToOne\ (\lambda n.n/0) = (\lambda f.f\ 1)\ (\lambda n.n/0) \mapsto (\lambda n.n/0)\ 1 \mapsto 1/0$$

which results in a divide-by-zero exception. Whereas after eta expansion, we would have

$$(\lambda f.\lambda x.f\ 1\ x)\ (\lambda n.n/0) \mapsto (\lambda x.(\lambda n.n/0)\ 1\ x)$$

which does not raise an exception. The mismatch is caused because we passed an argument of only arity 1, not arity 2, so eta expanding *applyToOne* can cause an observable change with both call-by-value and call-by-need evaluation. But with call-by-name evaluation, eta expansion is always valid because there is no way to force evaluation of a function besides calling it. By instead defining *applyToOne* with the type $Int \rightsquigarrow Int \rightsquigarrow Int$, the pre-processing transformation gives back exactly the above expansion

$$\mathcal{E}[\![applyToOne]\!]{:}Int \rightsquigarrow Int \rightsquigarrow Int = \lambda f.\lambda x.f\ 1\ x$$

And the above counter-example cannot happen at runtime, because we can only evaluate a fully saturation application like *applyToOne* $(\lambda n.n/0)$ 2 which raises an exception with or without eta expansion. Unlike the other evaluation strategies, with call-by-name evaluation of higher-order function parameters, there is no way to observe the presence or absence of a λ; a property that holds at any higher arity.

Second, there could be too few arguments for a desired arity. For example, consider:

$$applyToOneTwo : (Int \rightarrow Int \rightarrow Int) \rightarrow Int$$
$$applyToOneTwo = \lambda h{:}\, Int \rightarrow Int \rightarrow Int\,.\,\mathbf{let}\ f{:}\, Int \rightarrow Int = h\ 1\ \mathbf{in}\ f\ 2$$

Based on the type, we would like to assume that h is an arity 2 function. Unfortunately, the two arguments to h (1 and 2, in order) have been broken up into two separate steps. So if we call *applyToOneTwo* with some argument $\lambda x{:}\, Int\,.e$, we get:

$$applyToOneTwo\ (\lambda x{:}\, Int\,.e) \mapsto \mathbf{let}\ f : Int \rightarrow Int = (\lambda x Int\,.e)\ 1\ \mathbf{in}\ f\ 2$$

It is wrong in general to substitute $(\lambda x{:}\, Int\,.e)$ 1 for f during call-by-value or call-by-need evaluation. In both cases, $e[1/x]$ must be evaluated first until a second λ is returned. This is important in call-by-value in case $e[1/x]$ causes any side-effects (including infinite loops or exceptions), and in call-by-need to make sure that the work $e[1/x]$ does before returning the second λ is shared. But by changing the type of *applyToOneTo* to $(Int \rightsquigarrow Int \rightsquigarrow Int) \rightsquigarrow Int$, call-by-name evaluation naturally continues and puts together the complete arity 2 call:

$$applyToOneTwo\ (\lambda x{:}\, Int\,.e) \mapsto \mathbf{let}\ f : Int \rightsquigarrow Int = (\lambda x.e)\ 1\ \mathbf{in}\ f\ 2 \mapsto (\lambda x{:}\, Int\,.e)\ 1\ 2$$

Intuitively, this makes sense if we assume that $\lambda x{:}\, Int\,.e$ is really an arity 2 function: it is represented as a function accepting two arguments simultaneously, anyway, so it doesn't matter what *e does* because effectively *e is* another λ. In this sense, using a call-by-name evaluation order again accurately captures arities higher than 1.

3.6 Abstract machine semantics for $\mathcal{L}X$

The operational semantics for $\mathcal{L}$, while high-level, does not capture the intention that higher-arity function calls are performed all at once in one step. In order to see this optimization in action, an abstract machine for the lower-level $\mathcal{L}X$ is given in Figure 6. The primary goal of this machine is to show how arity is handled at run-time for curried functions using chains of the F arrow and forall types. Furthermore, we would like to use standard implementation techniques, so that arity-aware support is only needed for function application and no where else. As such, the majority of the machine steps are standard, in particular, those involving **let** and **case** expressions as well as variable lookup and update.

The main novelty of the machine can be seen in the first reduction step, which is responsible for performing multiple β reductions in a single step. To do so, there must be a chain of uninterrupted λ-abstractions as well as a matching chain of parameters on the stack. Notice that pushing a parameter P on the stack S has the form $P\ r\ S$, where r denotes the representation of the corresponding function

Runtime state

$$
\begin{array}{rll}
S \in \mathit{Stack} ::= & \bullet & \text{Empty stack} \\
& \mid P\ r\ S & r\text{-represented application} \\
& \mid \mathbf{case}\ \{\overline{p_i \to e_i}\}; S & \text{Case analysis} \\
& \mid\ !x; S & \text{Thunk update} \\
H \in \mathit{Heap} ::= & \bullet & \text{Empty heap} \\
& \mid x{:}\tau := e; H & \text{Allocated binding} \\
& \mid x{:}\tau := \bullet; H & \text{Black hole}
\end{array}
$$

Machine steps $\boxed{\langle e|S|H\rangle \mapsto \langle e'|S'|H'\rangle}$

$$
\begin{array}{rll}
(\beta^n) & \langle \lambda^r \mathbf{x_0}.\widetilde{\lambda}\mathbf{x_1}...\widetilde{\lambda}\mathbf{x_n}.e \mid P_0\ r\ P_1\ \mathsf{F}\ ...P_n\ \mathsf{F}\ S \mid H\rangle \mapsto \langle e[P_0/\mathbf{x_0}...P_n/\mathbf{x_n}] \mid S \mid H\rangle & \\
& \qquad (\text{if } e \neq \widetilde{\lambda}\mathbf{x'}.e' \text{and } S \neq P'\ \mathsf{F}\ S') & \\
(\mathit{match}) & \langle W \mid \mathbf{case}\ \{\overline{p_i \to u_i}; p \to e; \ldots\}; S \mid H\rangle \mapsto \langle e[\overline{P/x}] \mid S \mid H\rangle & (\text{if } W \hookrightarrow p[\overline{P/x}]) \\
(\mathit{default}) & \langle W \mid \mathbf{case}\ \{\ldots; x{:}\tau \to e\}; S \mid H\rangle \mapsto \langle e[y/x] \mid S \mid y{:}\tau := W; H\rangle & (\text{if } W \not\hookrightarrow p_i) \\
(\mathit{lookup}) & \langle x \mid S \mid x{:}\tau := W; H\rangle \mapsto \langle W \mid S \mid x{:}\tau := W; H\rangle & \\
(\mathit{apply}) & \langle e\ P \mid S \mid H\rangle \mapsto \langle e \mid P\ t\ S \mid H\rangle & \\
& \qquad (\text{if } e : \tau, \tau : t, \text{ and } e\ P \notin \mathit{WHNF}) & \\
(\mathit{case}) & \langle \mathbf{case}\ e\ \mathbf{of}\ \{\overline{p_i \to u_i}\} \mid S \mid H\rangle \mapsto \langle e \mid \mathbf{case}\ \{\overline{p_i \to u_i}\}; S \mid H\rangle & \\
(\mathit{alloc}) & \langle \mathbf{let}\ x{:}\tau = u\ \mathbf{in}\ e \mid S \mid H\rangle \mapsto \langle e[y/x] \mid S \mid y{:}\tau := u; H\rangle & \\
(\mathit{rec}) & \langle \mathbf{let\ rec}\ x{:}\tau = u\ \mathbf{in}\ e \mid S \mid H\rangle \mapsto \langle e[y/x] \mid S \mid y{:}\tau := u[y/x]; H\rangle & \\
(\mathit{force}) & \langle x \mid S \mid x{:}\tau := e; H\rangle \mapsto \langle e \mid\ !x; S \mid x{:}\tau := \bullet; H\rangle & (\text{if } e \notin \mathit{WHNF}) \\
(\mathit{memo}) & \langle W \mid\ !x; S \mid x{:}\tau := \bullet; H\rangle \mapsto \langle W \mid S \mid x{:}\tau := W; H\rangle &
\end{array}
$$

Fig. 6. Abstract machine for $\mathcal{L}X$

abstraction that is expected. This information is used when performing a multi-arity β reduction and is statically known at compile time via type checking, which is pushed along onto the stack with a new parameter in the first refocusing step. Observe how the first application of a reduction can be of any representation r (indeed, we can call any kind of function we'd like as normal), but once that initial application is triggered there is a cascade of several F applications that all take place at once. And these followup chains of F λ-abstractions and applications must match exactly; there must be the same number of $\widetilde{\lambda}$s ready as there are F parameters on the stack, no more or less. In this sense, the machine in Figure 6 models multi-arity function calls through currying.

Multi-arity β reduction puts some limitations on the expressions that this machine can evaluate. In particular, some expressions that would result in an answer using single-step β reduction can now get stuck when the number of λs and applications don't align correctly. For example, we could have too many $\widetilde{\lambda}$s, as in the stuck state $\langle \lambda x{:}\mathit{Bool}.\widetilde{\lambda} y{:}\mathit{Bool}.x \mid \mathit{True}\ \mathsf{L}\ \bullet\ \mid \bullet\rangle$ or too few as in the stuck state $\langle \lambda x{:}\mathit{Bool} \rightsquigarrow \mathit{Bool}.x \mid (\widetilde{\lambda} x{:}\mathit{Bool}.x)\ \mathsf{L}\ \mathit{True}\ \mathsf{F}\ \bullet\ \mid \bullet\rangle$. Both of these machine configurations appear to be type correct, but have an arity mismatch between the parameters on the call stack and the number of $\widetilde{\lambda}$s available. This apparent mismatch can also happen when some "serious work" interrupts the chain of call-by-name $\widetilde{\lambda}$-abstractions. For example, if the expression happens to be $\lambda x{:}\tau.\ \mathbf{let}\ z{:}\rho = \ldots\ \mathbf{in}\ \widetilde{\lambda} y{:}\sigma.e$ instead of the more accurate $\lambda x{:}\tau.\widetilde{\lambda} y{:}\sigma.\ \mathbf{let}\ z{:}\rho = \ldots\ \mathbf{in}\ e$.

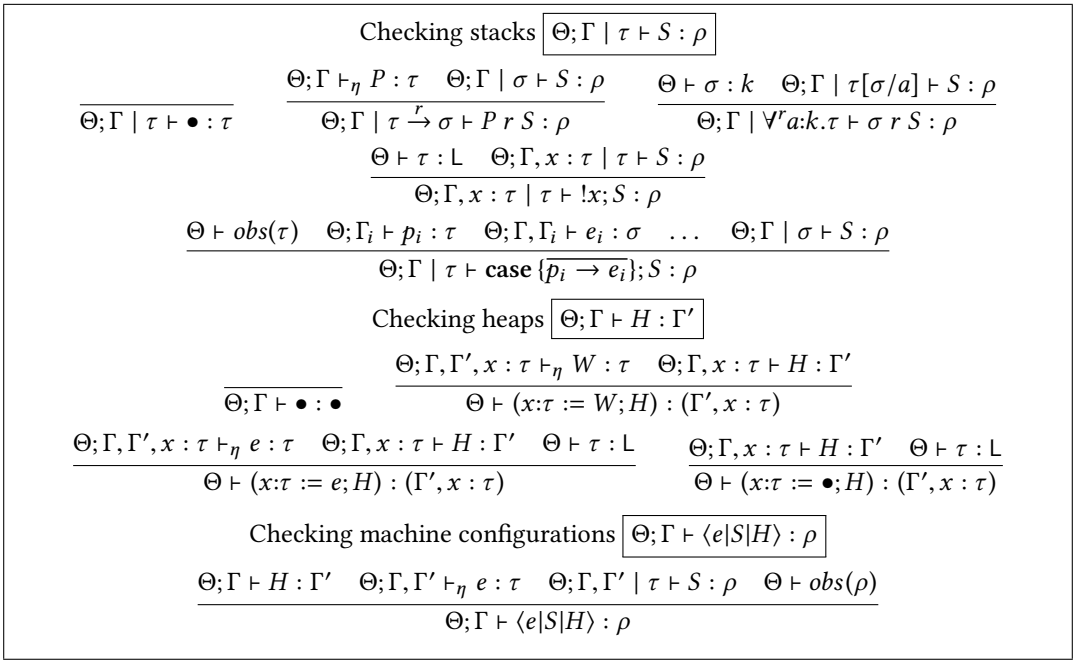

Fig. 7. Type system for $\mathcal{L}X$ machine configurations

Thankfully, the type system for $\mathcal{L}X$ from Figure 3 addresses and rules out the potential for arity mismatches at runtime. This arity check is done in one of two places. First, the results of **let** and **case** expressions must be of observable types. This makes expressions of like $\mathbf{let}\ z{:}\rho = \ldots\ \mathbf{in}\ \widetilde{\lambda} y{:}\sigma.e$ untypable, so that we *must* write the $\widetilde{\lambda}$ first before the **let**. Second, once the first λ-abstraction (of any representation) is encountered, then *every possible* $\widetilde{\lambda}$-abstraction allowed by the type must immediately follow. So $\lambda x{:}Bool \rightsquigarrow Bool.x$ must be instead written in the expanded form $\lambda x{:}Bool \rightsquigarrow Bool.\widetilde{\lambda} y{:}Bool.x\ y$. Both of these restrictions ensure that there are enough $\widetilde{\lambda}$s available as promised by the type of the expression, and can be understood as a form of mandatory η-expansion.

But will there be enough parameters on the stack? After all, a chain of applications might be interrupted by a **case** or a thunk update. For example, instead of the stack $x\ \mathsf{F}\ y\ \mathsf{F}\ \bullet$ might be interrupted as $x\ \mathsf{F}\ \mathbf{case}\ \{z{:}\tau \rightsquigarrow \sigma \to z\}; y\ \mathsf{F}\ \bullet$ or $x\ \mathsf{F}\ !z; y\ \mathsf{F}\ \bullet$. Again, the runtime typing rules help prevent both of these arity errors, too. An expression of type $\tau \rightsquigarrow \sigma$ or $\widetilde{\forall} a{:}k.\tau$ cannot be examined by case analysis, due to the restriction on the type of patterns: neither are observable types, so $x{:}\tau \rightsquigarrow \sigma$ and $x{:}\widetilde{\forall} a{:}k.\tau$ are ill-typed default patterns. And the memoization mechanisms are only in place for variable bindings of L types. This is possible because **let** expressions cannot bind variables of kind V (these are only bound by a **case**), and can only bind variables of kind F to weak head-normal (due to the η-expansion restrictions in place).[7] The one last way there may not be enough parameters to complete a multi-arity function call is if the stack just ends too early. For this reason, we will only run expressions of observable types, and never run expressions of type $\tau \rightsquigarrow \sigma$ or $\widetilde{\forall} a{:}k.\tau$; in order to evaluate expressions of these types, they must first be applied to enough parameters until an observable type of result is achieved.

[7] As a pleasant side-effect of these restrictions on **case** and **let** bindings, the standard abstract machine rules in Figure 6 implement the correct evaluation strategy for each of the three kinds of types (L, V, and F) without requiring runtime type inspection or additional support for call-by-value and -name reduction.

These restrictions all ensure type safety of multi-arity function calls, which means that the arities of functions and callers must always match at runtime. In particular, we can extend type checking to full machine configurations in addition to just $\mathcal{L}X$ expressions, as shown in Figure 7. This lets us approach type safety using the standard technique of progress and preservation.

Definition 2 (Initial, Final, and Cyclic)**.** An *initial* $\mathcal{L}$ machine configuration has the form $\Theta; \bullet \vdash \langle e | \bullet | \bullet \rangle : \tau$ with an empty stack and heap. A *final* machine configuration has the form $\langle W | \bullet | H \rangle$ where W is any weak-head normal form and the stack is empty. A *cyclic* machine configuration has the form $\langle x | S | H \rangle$ where $x{:}\tau := \bullet$ is in H.

Definition 3 (Memory Safety)**.** An $\mathcal{L}X$ machine configuration $\langle e | S | H \rangle$ is *memory safe* if, for every black hole ($x{:}\tau = \bullet$) in H, there is exactly one matching update ($!x$) in S, and vice versa.

Proposition 4 (Type Safety of $\mathcal{L}X$)**.** *For any memory safe $\mathcal{L}X$ machine configuration $\langle e | S | H \rangle$:*

- Progress: *If* $\Theta; \bullet \vdash \langle e | S | H \rangle : \tau$ *and* $\Theta \vdash obs(\tau)$ *then either* $\langle e | S | H \rangle$ *is final,* $\langle e | S | H \rangle$ *is cyclic, or* $\langle e | S | H \rangle \mapsto \langle e' | S' | H' \rangle$ *for some* e', S', *and* H'.
- Preservation: *If* $\Theta; \Gamma \vdash \langle e | S | H \rangle : \tau$ *and* $\langle e | S | H \rangle \mapsto \langle e' | S' | H' \rangle$ *then* $\Theta; \Gamma \vdash \langle e' | S' | H' \rangle : \tau$ *and* $\langle e' | S' | H' \rangle$ *is memory safe.*

3.7 Semantic correctness

With the pre-processing transformation bridging between $\mathcal{L}$ and $\mathcal{L}X$, we effectively have two different operational semantics for our intermediate language—a high-level semantics with incremental function calls and a low-level arity-aware semantics—and we should expect that they correspond to one another. The first check is that both semantics always give the same answer.

Proposition 5 (Operational soundness)**.** *For any $\mathcal{L}$ program e, if $e \mapsto^* A$ then there is some W and H such that $\langle C[\![e]\!] \bullet | \bullet | \bullet \rangle \mapsto^* \langle W | \bullet | H \rangle \leftarrow^* \langle C[\![A]\!] \bullet | \bullet | \bullet \rangle$.*

We don't just care about the final answer, though; we also care about how much it costs to calculate it. In order to model a notion of "cost," we introduce a primitive "tick" operation which serves as a programmable profiling mechanism. The idea is that an $\mathcal{L}$ program can include manual applications the primitive function $tick^r : \widetilde{\forall} a{:}r.a \to a$ which behaves like the identity function. However, we will track the number of times $tick^r$ is called in the $\mathcal{L}X$ machine with the following additional rule:

$$\langle tick^r | \tau \mathsf{F} x \mathsf{F} S | H \rangle \mapsto_1 \langle x | S | H \rangle$$

The subscript 1 denotes the number of times *tick* was called (here, just once), so all the other individual steps of the machine from Figure 6 count as 0. This number gives an approximation of the cost for executing a program, where we just add together the cost of each individual step, so that if $\langle e | S | H \rangle \mapsto_m^* \langle e' | S' | H' \rangle \mapsto_n^* \langle e'' | S'' | H'' \rangle$ then $\langle e | S | H \rangle \mapsto_{m+n}^* \langle e'' | S'' | H'' \rangle$.

Definition 4 (Cost)**.** For any $\mathcal{L}X$ configuration $\Theta; \bullet \vdash \langle e | S | H \rangle : \tau$, the *cost* of $\langle e | S | H \rangle$ (written $cost\,\langle e | S | H \rangle$) is the number n such that $\langle e | S | H \rangle \mapsto_n^* \langle W | \bullet | H' \rangle$. If $\langle e | S | H \rangle$ does not reduce to a final configuration, then $cost\,\langle e | S | H \rangle$ is undefined. Furthermore, for any $\Theta \vdash obs(\tau)$ and $\mathcal{L}$ expression $\Theta; \bullet \vdash e : \tau$, the *cost* of e (written $cost(e)$) is $cost\,\langle C[\![e]\!] \bullet | \bullet | \bullet \rangle$.

The net result is that any two equal programs (where no *tick*s have been preemptively reduced) have the same cost. We can use this fact to justify that the call-by-name semantics of F types, and the aggressive work duplication associated with it, does not interfere or change the cost of call-by-value and call-by-need expressions. In other words, selective use of call-by-name does not harm the algorithmic complexity of programs.

Proposition 6 (Cost soundness). *For any $\mathcal{L}$ programs e, e', if $\Theta; \bullet \vdash e = e' : \tau$ then $cost(e) = cost(e')$.*

4 EMBRACING POLYMORPHISM: KINDS AS CALLING CONVENTIONS

The use of explicit runtime representations in a high-level intermediate language also appears in Eisenberg and Peyton Jones' notion of *levity polymorphism* [Eisenberg and Peyton Jones 2017]. Levity polymorphism builds upon previous work on unboxed types [Peyton Jones and Launchbury 1991] in order to achieve two goals:

- Systematically characterize the representation of non-uniform data (*e.g.,* 8-bit characters are represented differently from 64-bit floating point numbers) in a type system.
- Decouple the separate issues of representation and polymorphism, thereby avoiding the *uniform representation* restriction: that polymorphic types must be uniformly representable (*e.g.,* as a pointer to an object).

The result of these two points is that anything can be treated polymorphically, because the kind of a polymorphic type a gives enough compile-time information to know how to store and move values of type a. In other words, *kinds are calling conventions*, not types. This shift then moves the question away from *type* polymorphism to *levity* polymorphism: when can a piece of code operate uniformly over different data representations. For example, a function like *error*—which immediately ends execution and prints an error message—can safely promise to return a result of any type *and* of any representation, because it will never return.

4.1 Combining levity polymorphism and arity: Higher-order representations

In section 3, we required the somewhat unsatisfying restriction that F types had to be monomorphic. How do we "scale up" the notion of arity presented in the small $\mathcal{L}$ calculus, and integrate it into a larger intermediate language with more advanced type features like levity polymorphism and higher kinds (*i.e.,* type functions like $List : \forall r.r \to r$ instead of the compound $List\ ^r\tau$)? The main obstacle is that the simple representation constant F for higher-arity functions is just not informative enough: nothing about its actual calling convention (*i.e.,* how many and what kinds of arguments can it be passed) is described by F! In this richer setting where kinds are *really* calling conventions, we must in turn give more structure to the kind F. In particular, the kind of a higher-arity function should express:

- the number of arguments the function accepts before doing serious work (the traditional notion of arity),
- the representation of each argument (so that the appropriate parameter-passing code is known the function call), and
- the representation of its result (so that higher-arity functions can be composed).

With all this information available in the kind of a of higher-arity function, a compiler can correctly generate code for direct, higher-arity function calls of potentially unboxed parameters; even that function has an unknown polymorphic type!

Formally, the above enrichment of higher-arity representations can be expressed by the following extension of $\mathcal{L}$:

$$
\begin{aligned}
k, l \in \mathit{Kind} &::= \mathtt{TYPE}\ r \mid k \to l \mid a \mid \forall a.k \\
r, s, t \in \mathit{Rep} &::= p \mid \mathtt{Tuple}\ [\overline{s}] \mid s \rightsquigarrow r && (\text{if } s \neq \mathtt{Tuple}\ \overline{t}) \\
p \in \mathit{PrimRep} &::= \mathtt{V} \mid \mathtt{L} \mid \mathtt{IntRep} \mid \mathtt{FloatRep} \mid \mathtt{UnitRep} \mid \ldots
\end{aligned}
$$

The biggest change here is that representations (r, s, t) have been separated from kinds (k, l). The kind `TYPE` r classifies the types of terms and values, while the other forms of kinds k classify type constructors (e.g `Array : TYPE V → TYPE V`). The kind `TYPE` r, for types of terms, carries r which describes the memory layout and/or calling convention of the values of that type.

We allow for many primitive representations (like `IntRep` for real, 64-bit integers and `UnitRep` for unit values which take up no space). On top of that, representations in general may be compound to describe unboxed tuples (`Tuple` $[\overline{s}]$) or higher-arity functions ($s \rightsquigarrow r$). As in the previous work on levity polymorphism [Eisenberg and Peyton Jones 2017], `Tuple` $[\overline{s}]$ representations do not nest (*i.e.,* each s cannot be another `Tuple` representation) is because unboxed tuples are flattened out in memory. For example, the same representation $TrRep$ = `Tuple [IntRep, L, FloatRep]` describes each of the following different types[8]

$$(\#\, Int\, , (\#\, List\ \mathtt{L}\ Integer\, , Float\, \#)\#) : \mathtt{TYPE}\ TrRep$$
$$(\#(\#\, Int\, , List\ \mathtt{L}\ Integer\, \#), Float\, \#) : \mathtt{TYPE}\ TrRep$$
$$(\#\, Int\, , List\ \mathtt{L}\ Integer\, , Float\, \#) : \mathtt{TYPE}\ TrRep$$

where we write the type of lazy (*i.e.,* lifted) integers as *Integer* . Even though the types are different, their values all have the same representation in memory and can be used interchangeably. The net effect is that the `Tuple` $[\overline{s}]$ representation captures the aggressive flattening of unboxed tuples similar to [Bergstrom and Reppy 2010].

By the same token, the $s \rightsquigarrow r$ representation captures aggressive currying of higher-arity functions, where s is again not a `Tuple` representation. Intuitively, the curried type $\tau \rightsquigarrow \sigma \rightsquigarrow \rho$ is represented exactly the same as the uncurried type $(\#\tau, \sigma\#) \rightsquigarrow \rho$, and so they can be used interchangeably. For example, the same higher-order representation

$$HORep = \mathtt{IntRep} \rightsquigarrow (\mathtt{IntRep} \rightsquigarrow \mathtt{FloatRep} \rightsquigarrow \mathtt{FloatRep}) \rightsquigarrow \mathtt{L} \rightsquigarrow \mathtt{FloatRep}$$

describes each of the following types with different amounts of currying:

$$(\#\, Int\, , (Int \rightsquigarrow Float \rightsquigarrow Float\,), List\ \mathtt{L}\ Integer\, \#) \rightsquigarrow Float : \mathtt{TYPE}\ HORep$$
$$(\#\, Int\, , (Int \rightsquigarrow Float \rightsquigarrow Float\,)\, \#) \rightsquigarrow List\ \mathtt{L}\ Integer \rightsquigarrow Float : \mathtt{TYPE}\ HORep$$
$$Int \rightsquigarrow (Int \rightsquigarrow Float \rightsquigarrow Float\,) \rightsquigarrow List\ \mathtt{L}\ Integer \rightsquigarrow Float : \mathtt{TYPE}\ HORep$$

This more expressive form of higher-order representations can allow us to overcome two challenges of integrating arity with levity polymorphism.

4.2 Challenge 1: Polymorphism

Recall from remark 1 that polymorphism is forbidden for types of kind `F` due to issues surrounding unknown η-expansion, as in the rejected definition

$$applyToOne : \widetilde{\forall} a{:}\mathtt{F}.(Int \rightsquigarrow a) \rightsquigarrow a$$
$$applyToOne = \widetilde{\lambda} a{:}\mathtt{F}.\widetilde{\lambda} f{:}(Int \rightsquigarrow a).\ f\ 1$$

With the more informative arity representations ($s \rightsquigarrow r$ instead of the constant `F`), we can rely on the *kind* of the definition, rather than the type, to tell us how many parameters f will really be passed at runtime. The different possible arities of f now show up as explicitly different types,

[8] Following GHC's convention, we write an unboxed pair type as $(\#a, b\#)$ and an unboxed triple type as $(\#a, b, c\#)$.

because the $\widetilde{\forall}$ introducing a must specify the full arity of the type variable like so:

$$applyToOne_2 : \widetilde{\forall} a{:}(\mathtt{IntRep} \rightsquigarrow \mathtt{L}).\ (Int \rightsquigarrow a) \rightsquigarrow a$$
$$applyToOne_3 : \widetilde{\forall} a{:}(\mathtt{IntRep} \rightsquigarrow \mathtt{L} \rightsquigarrow \mathtt{L}).\ (Int \rightsquigarrow a) \rightsquigarrow a$$

Now there is no ambiguity: the runtime arity of f is made clear (in $applyToOne_2$ if is passed exactly two arguments at once, whereas in $applyToOne_3$ it is passed three) even though f is polymorphic. So type-driven pre-processing from Figure 4 would instead be *representation*-driven ($\mathcal{E}[\![e]\!]\overline{P}{:}r$).

4.3 Challenge 2: Type Erasure

Higher-arity representations handily solve the monomorphism restriction, but doing so introduces a second wrinkle. When elaborating a higher-order polymorphic definition, what are the extra input and output types? Consider the following η-expanded definition for $applyToOne_2$:

$$applyToOne_2 = \widetilde{\lambda} a{:}(\mathtt{IntRep} \rightsquigarrow \mathtt{L}).\widetilde{\lambda} f{:}(Int \rightsquigarrow a).\widetilde{\lambda} x{:}???.f\ 1\ x$$

The type of $f\ 1$ is a, so only the caller knows what the type of the second argument x should be (and for that matter, only the caller knows the return type of $f\ 1\ x$). The type annotations, and type checking in general, is now ambiguous.

Unlike the previous problem of ambiguous η-expansion—which is used during compilation for the purpose of code generation—type annotations are irrelevant for execution. In reality, types are removed before runtime so that

$$applyToOne_2 = \widetilde{\lambda} f.\widetilde{\lambda} x.f\ 1\ x$$

is a suitable, executable definition; we don't need to know the type of x, just how it is represented in memory (in this case, L) in order to pass it to f. Therefore, we can compile polymorphic, higher-arity definitions by using the existing *type erasure* process.

By erasing types, in effect we end up erasing $\forall$s—along with type abstractions and applications in expressions—too. So to account for type erasure, we should express that $\forall$s don't appear in the representation of any runtime value. This is captured by the following revision to the kinding rule of $\forall$s from Figure 2:

$$\frac{\Theta, a{:}k \vdash \tau : \mathtt{TYPE}\ r \quad delay(r)}{\Theta \vdash \forall a{:}k.\tau : \mathtt{TYPE}\ r}$$

Notice how the representation of an erased $\forall$ type just inherits the representation of it's body (τ) without modifying it in any way.

The extra constraint that r must be delayable (*i.e.,* $delay(r)$) is due to the semantics of polymorphism in call-by-value languages. For example, consider the undefined call-by-value definition

$$undefined : \forall a{:}\mathtt{TYPE\ V}.a$$
$$undefined = \lambda a{:}\mathtt{TYPE\ V}.error\ \text{“}\mathtt{oops}\text{”}$$

In call-by-value, the polymorphic abstraction $\lambda a{:}\mathtt{TYPE\ V}.error$ “oops” is observably different from the underlying *error* “oops”: one is a value and the other is not, so the two cannot be equated. The usual solution to call-by-value type erasure is to leave a trace of type abstractions afterward: the abstraction $\lambda a{:}k.e$ is erased to an explicit delay $\lambda().e$ and the specialization $e\ \sigma$ to an explicit force $e\ ()$, where $()$ is some unit value. The net effect of the above constraint on $\forall$ types says that this delay/force shadow of polymorphism must be *already present* in the original program: the naïve type erasure for lazy λ-calculus is also sound for call-by-value so long as polymorphism is only ever

introduced over delayed types. For example, instead of the troublesome definition of a call-by-value *undefined* value above, the type system tells us to write the explicated

$$\mathit{undefined} : \forall a{:}\mathsf{TYPE}\ \mathsf{V}.\mathtt{Unit} \rightsquigarrow a$$

$$\mathit{undefined} = \lambda a{:}\mathsf{TYPE}\ \mathsf{V}.\widetilde{\lambda} x{:}\mathtt{Unit}.\ \mathit{error}\ \text{“}\mathtt{oops}\text{”}$$

With this definition, just erasing the ∀ and type abstraction still gives an equivalent definition, in call-by-name, call-by-need, *and* call-by-value.

5 RELATED WORK

5.1 Uncurrying

The classical way to handle multi-argument function calls is to uncurry the function [Bolingbroke and Peyton Jones 2009; Dargaye and Leroy 2009; Hannan and Hicks 1998]. Uncurrying can be done as a whole-program transformation, but using worker/wrapper is a more incremental approach. For example, here is the worker/wrapper for uncurrying `zipWith` in Haskell:

```
$wzipWith :: (# ((# a, b#) -> c), [a], [b] #) -> [c]
$wzipWith (# f, (x:xs), (y:ys) #) = f (# x,y #) : $wzipWith (# f,xs,ys #)
$wzipWith (# f, _,      _      #) = []

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f xs ys = $zipWith (# \ (# x,y #) -> f x y, xs, ys #)
```

Notice that both `$wzipWith` *and* its argument `f` are uncurried — this is referred to as *higher order uncurrying* by [Dargaye and Leroy 2009; Hannan and Hicks 1998].

Uncurrying techniques and our polarity inspired approach are similar in their desire to encode arity in types, resulting in the introduction of new function types. Both [Dargaye and Leroy 2009] and [Bolingbroke and Peyton Jones 2009] use types that represent multiple-argument functions $(a_0, \ldots, a_n) \to b$. [Hannan and Hicks 1998]'s approach looks more similar to ours. They introduce an intermediate representation for their translation that involves the annotated function types $a \to_{\Uparrow} b$ where b is a function whose call will be fused with this one and $(\to_{\epsilon})$ which is a standard curried function type. Our $(a \rightsquigarrow b)$ functions will also fuse when applied, but we do not require b to be function. Instead, we use a call-by-name evaluation strategy to safely support η-expansion.

Forcing an uncurried representation complicates languages with polymorphic types. For example, how do you uncurry the function $\forall a{:}k.\forall b{:}k.a \to b \to b \times a$? Since both the arguments and outputs of arrows can depend on type variables bound earlier, the uncurried version will require some sort of dependent product $a{:}k \times \tau$ so that types can refer to variables that were introduced by the ∀. However, this is not so straight forward, since the type variables are *also* in scope for the return type of the function. Trying to rewrite the above polymorphic function type as $(a{:}k \times b{:}k \times a \times b) \to b \times a$ doesn't quite work, since a and b have escaped in the return type $b \times a$ of the function. And this situation is even worse if the type doesn't happen to have this special ∀s-first form, like $\forall a{:}k.a \to \forall b{:}k.b \to b \times a$. By using our kind-based solution, we completely avoid this problem since we have no need to re-associate a function type the way that uncurrying does; we just replace one arrow $(\to)$ with another $(\rightsquigarrow)$. [Hannan and Hicks 1998]'s work also avoids this problem by sticking to the Hindley-Milner type system, where quantifiers only appear in type schemes, but this is significantly less expressive than the unrestricted quantifiers of System F [Girard et al. 1989].

Another difference in the two solutions is with respect to sharing in non-strict languages, *e.g.*, Haskell. [Bolingbroke and Peyton Jones 2009] use an ad-hoc approach by treating all functions whose uncurried argument is an *empty tuple* as thunks. Their combination of thunks and functions can make programs slower if the nullary function was intentionally used to avoid sharing. Our kinds

as calling conventions approach cleanly distinguishes call-by-name and call-by-need functions avoiding this extra allocation.

5.2 Arity in compilation

The importance of function arity became apparent in the once pervasive categorical abstract machine [Cousineau et al. 1985] wherein curried functions would allocate many intermediate closures. With the goal of fast multi-arity function calls, the Krivine [Krivine 2007] and ZINC abstract machines [Leroy 1990] repeatedly push arguments onto the stack and an n-ary function will just consume the n arguments it needs. Another approach to speeding up curried programs is presented by [Marlow and Peyton Jones 2004] which combines both statically and dynamically known arity information to make fast calls. Statically, the compiler knows that a function declared with 2 manifest lambdas has the arity 2. If a function of known arity being fully applied, then we can generate code that fuses the applications. For the cases where the arity is unknown at compile time, *e.g.*, higher-order functions like `zipWith`, there is a dynamic check for when calls can be fused. The crux of these approaches is that either the compiler must propagate known arity information to the call site or the runtime must check arity information.

The usage of arity information has long been an important in compilers leading to the development of complex arity analyzers [Breitner 2014; Xu and Peyton Jones 2005] which use the information to eta expand and to float lambda abstractions into and out-of contexts without increasing work. More recent work performs cardinality analysis (which checks the usage of expressions) to apply the same transformations [Sergey et al. 2014] and also to generate non-updateable thunks at runtime if it will only be evaluated once. Since our goal is to improve the frequency of mult-arity function calls, our arity analysis is simply based on visible lambda abstractions and applications making it much simpler than previously mentioned analyses. We seek to secure arity information in the type of our intermediate language's programs; a place where it will be preserved while optimizations freely manipulate the structure of terms. For worker/wrapper transformation (Section 2.3), the type is derived from counting λs and this is used to create the worker and wrapper. The wrapper eta expand higher-order functions to produce call-by-name ($\rightsquigarrow$) functions. These functions must appear fully applied in the wrapper which may require eta expansion. We exploit that call-by-name functions always appear fully applied in code generation to generate n-ary applications.

5.3 Polarity

Our inspiration for the evaluation strategy conscious treatment of arity came from the study of polarity and type-based mixed evaluation strategies in programming languages [Levy 2001; Munch-Maccagnoni 2013; Zeilberger 2009]. These based calculi enjoy the strongest possible extensionality laws (*a.k.a.* η axioms) for every type, even in the presence of recursion and computational effects. To this end, every type has an innate evaluation strategy which allows for mixing of call-by-value constructs (like data types) with call-by-name constructs (like functions).

Polarity has been used in the past to address other issues relevant to intermediate languages and optimization, like resolving the conflict between "strong sum" types and non-termination [Munch-Maccagnoni and Scherer 2015]. Our intermediate languages $\mathcal{L}$ and $\mathcal{L}X$ closely follow a polarized calculus extended with call-by-need constructs [Downen and Ariola 2018] used to support Haskell (for call-by-value languages, adding call-by-name functions is sufficient). Investigating the consequences of these polarized in a practical setting has lead us to find that the practical concerns of a function's arity and representation are actually semantic properties of the function's type.

Interestingly, types with an innate evaluation strategy has arisen independently in practice. In order to get the performance gain from passing primitives in registers instead of pointers to thunks on the heap, [Peyton Jones and Launchbury 1991] introduce unboxed types. Passing an

unboxed integer in a register means it is already a value, so it must be evaluated ahead of time (call-by-value). Their implementation uses worker/wrapper transformations to change the interface of a function from slow boxed values to faster unboxed ones. Unfortunately, these new type representations limited polymorphism because every type did not have the same runtime structure; a restriction which was later solved by levity polymorphism [Eisenberg and Peyton Jones 2017]. Levity polymorphism also allows for polymorphism over representation, not just types, which is similar to evaluation order polymorphism [Dunfield 2015]. Our implementation works nicely with this machinery, where we introduce a new higher-order representation $r \rightsquigarrow s$ that expresses arity in kinds, too, which enables new forms of arity-specific polymorphism.

6 CONCLUSION

This work follows the tradition of designing calculi that faithfully capture practical issues: here, optimizing curried function calls and unboxed types. On one hand, higher-order functions are very important for expressiveness and currying can dramatically reduce code size [Arvind and Ekanadham 1988]. On the other, efficiency is also a concern; no one wants to be penalized for writing elegant programs! Compiler writers have many techniques to solve these problems, and e think some of these can be put on solid ground with polarization. Interestingly, the solutions for making fast calls across higher-order functions and for unboxing tuples appear to be logically dual to one another. As a pleasant side effect, we arrive at an elusive goal: a common and uncompromised intermediate language suitable for both strict and lazy functional languages. We have implemented the ideas in $\mathcal{L}$ as an extension of GHC's Core intermediate language as a proof of concept. Our next step is to use the new, fully extensional function type and higher-order representations to improve code generation for fast function calls.

REFERENCES

Zena M. Ariola, John Maraist, Martin Odersky, Matthias Felleisen, and Philip Wadler. 1995. A Call-By-Need Lambda Calculus. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. ACM, New York, NY, USA, 233–246.

Arvind and Kattamuri Ekanadham. 1988. Future Scientific Programming on Parallel Machines. *J. Parallel Distrib. Comput.* 5, 5 (1988), 460–493.

Lars Bergstrom and John Reppy. 2010. Arity Raising in Manticore. In *Proceedings of the 21st International Conference on Implementation and Application of Functional Languages (IFL'09)*. Springer-Verlag, Berlin, Heidelberg, 90–106.

Maximilian C. Bolingbroke and Simon L. Peyton Jones. 2009. Types Are Calling Conventions. In *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell (Haskell '09)*. 1–12.

Joachim Breitner. 2014. Call Arity. In *Trends in Functional Programming - 15th International Symposium, TFP 2014, Soesterberg, The Netherlands, May 26-28, 2014. Revised Selected Papers*. 34–50.

Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. 1985. The Categorical Abstract Machine. In *Functional Programming Languages and Computer Architecture, FPCA 1985, Nancy, France, September 16-19, 1985, Proceedings*. 50–64.

Pierre-Louis Curien and Hugo Herbelin. 2000. The Duality of Computation. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. ACM, New York, NY, USA, 233–243.

Zaynah Dargaye and Xavier Leroy. 2009. A verified framework for higher-order uncurrying optimizations. *Higher-Order and Symbolic Computation* 22, 3 (2009), 199–231.

Paul Downen and Zena M. Ariola. 2014a. Compositional Semantics for Composable Continuations: From Abortive to Delimited Control. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, New York, NY, USA, 109–122.

Paul Downen and Zena M. Ariola. 2014b. The Duality of Construction. In *Programming Languages and Systems: 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014*. Lecture Notes in Computer Science, Vol. 8410. Springer Berlin Heidelberg, 249–269.

Paul Downen and Zena M. Ariola. 2018. Beyond Polarity: Towards a Multi-Discipline Intermediate Language with Sharing. In *27th EACSL Annual Conference on Computer Science Logic, CSL 2018, September 4-7, 2018, Birmingham, UK*. 21:1–21:23.

Joshua Dunfield. 2015. Elaborating evaluation-order polymorphism. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. 256–268.

Richard A. Eisenberg and Simon Peyton Jones. 2017. Levity polymorphism. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 525–539.

Andrzej Filinski. 1989. *Declarative Continuations and Categorical Duality.* Master's thesis. Computer Science Department, University of Copenhagen.

Andy Gill and Graham Hutton. 2009. The Worker/Wrapper Transformation. *J. Funct. Program.* 19, 2 (March 2009), 227–251.

Jean-Yves Girard, Paul Taylor, and Yves Lafont. 1989. *Proofs and Types.* Cambridge University Press, New York, NY, USA.

John Hannan and Patrick Hicks. 1998. Higher-Order unCurrying. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*. 1–11.

Jean-Louis Krivine. 2007. A Call-By-Name Lambda-Calculus Machine. *Higher-Order and Symbolic Computation* 20, 3 (2007), 199–207.

Xavier Leroy. 1990. *The ZINC experiment: an economical implementation of the ML language.* Technical report 117. INRIA.

Paul Blain Levy. 2001. *Call-By-Push-Value.* Ph.D. Dissertation. Queen Mary and Westfield College, University of London.

Simon Marlow and Simon L. Peyton Jones. 2004. Making a fast curry: push/enter vs. eval/apply for higher-order languages. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, Snow Bird, UT, USA, September 19-21, 2004*. 4–15.

Guillaume Munch-Maccagnoni. 2009. Focalisation and Classical Realisability. In *Computer Science Logic: 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL (CSL 2009)*, Erich Grädel and Reinhard Kahle (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 409–423.

Guillaume Munch-Maccagnoni. 2013. *Syntax and Models of a non-Associative Composition of Programs and Proofs.* Ph.D. Dissertation. Université Paris Diderot.

Guillaume Munch-Maccagnoni and Gabriel Scherer. 2015. Polarised Intermediate Representation of Lambda Calculus with Sums. In *2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2015)*. 127–140.

Simon L. Peyton Jones and John Launchbury. 1991. Unboxed Values As First Class Citizens in a Non-Strict Functional Language. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*. Springer-Verlag, London, UK, UK, 636–666.

Amr Sabry and Matthias Felleisen. 1993. Reasoning About Programs in Continuation-Passing Style. *Lisp and Symbolic Computation* 6, 3-4 (Nov. 1993), 289–360.

Ilya Sergey, Dimitrios Vytiniotis, and Simon Peyton Jones. 2014. Modular, Higher-order Cardinality Analysis in Theory and Practice. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. 335–347.

Philip Wadler. 2003. Call-By-Value is Dual to Call-By-Name. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*. ACM, New York, NY, USA, 189–201.

Dana N. Xu and Simon L. Peyton Jones. 2005. Arity Analysis. (2005). Working notes.

Noam Zeilberger. 2008. On the Unity of Duality. *Annals of Pure and Applied Logic* 153, 1 (2008), 660–96.

Noam Zeilberger. 2009. *The Logical Basis of Evaluation Order and Pattern-Matching.* Ph.D. Dissertation. Carnegie Mellon University.

A TYPE SAFETY

Lemma 1 (Replacement). *For all $\mathcal{L}$ contexts C and expressions $\Theta;\Gamma \vdash e_i : \tau$ (with $i \in \{1,2\}$), $\Theta';\Gamma' \vdash C[e_1] : \tau'$ if and only if $\Theta';\Gamma' \vdash C[e_2] : \tau'$.*

Proof. By induction on the given typing derivation of the expression $C[e_i]$. □

Lemma 2 (Renaming). (1) *If $\Theta;\Gamma, x : \sigma \vdash e : \tau$ then $\Theta;\Gamma, y : \sigma \vdash e[y/x] : \tau$.*
(2) *If $\Theta;\Gamma, x : \sigma \vdash_\eta e : \tau$ then $\Theta;\Gamma, y : \sigma \vdash^{\mathsf{F}}_\eta e[y/x] : \tau$.*
(3) *If $\Theta;\Gamma, x : \sigma \vdash_\eta e : \tau$ then $\Theta;\Gamma, y : \sigma \vdash^{\mathsf{F}}_\eta e[y/x] : \tau$.*

Proof. By induction on the given typing derivation of the expression e. □

Lemma 3 (Specialization). *For all $\Theta \vdash \sigma : k$:*

(1) *If $\Theta, a : k;\Gamma \vdash e : \tau$ then $\Theta;\Gamma[\sigma/a] \vdash e[\sigma/a] : \tau[\sigma/a]$.*
(2) *If $\Theta, a : k;\Gamma \vdash_\eta e : \tau$ then $\Theta;\Gamma[\sigma/a] \vdash_\eta e[\sigma/a] : \tau[\sigma/a]$.*
(3) *If $\Theta, a : k;\Gamma \vdash^{\mathsf{F}}_\eta e : \tau$ then $\Theta;\Gamma[\sigma/a] \vdash^{\mathsf{F}}_\eta e[\sigma/a] : \tau[\sigma/a]$.*

Proof. By induction on the given typing derivation of the expression e. □

A.1 Safety of $\mathcal{L}$

The goal here is to show that every $\mathcal{L}$ program either converges or diverges but cannot get stuck.

Definition 5 (Convergence and Divergence). An expression e *converges* when it reaches an answer in a finite number of steps, *i.e.,* $e \mapsto^* A$, and *diverges* either when it takes an infinite number of steps, *i.e.,* $e \mapsto e_1 \mapsto e_2 \mapsto \ldots$, or when it reaches a cycle, *i.e.,* $e \mapsto^* E_1[\mathbf{letrec}\, x{:}\tau = E_3[x]\, \mathbf{in}\, E_2[x]]$ where $x \notin BV(E_2) \cup BV(E_3)$.

Lemma 4 (Open Progress). *If $\Theta;\Gamma \vdash e : \tau$ then either e is an answer, e is a cycle, $e \mapsto e'$ for some e', or $e = E[x]$ for some E and $x{:}\tau \in \Gamma$.*

Proof. By induction on the given typing derivation for $\Theta;\Gamma \vdash e : \tau$.

- The typing rule for variables checks an expression of the form $x = \Box[x]$, that is, a variable x inside an empty evaluation context.
- Derivations ending in an inference rule for numbers n, constructors K, and abstractions $\lambda^r \mathbf{x}.e$ all type check answers.
- Derivations ending in an inference rule for applications $e\ g$ proceed by the inductive hypothesis on the sub-derivation for e. Since $\Box\ g$ is a frame context, we have the following possibilities:
  - If $e \mapsto e'$ then $e\ g \mapsto e'\ g$.
  - If e is cyclic then so is $e\ g$.
  - If $e = E[x]$ then $E\ g$ is also an evaluation context surrounding x.
  - If e is an answer, then we can proceed by cases on the form of answer. If $e = W$, then W must be an appropriate abstraction for the application (due to the restrictions of the typing rules), and so $W\ g \mapsto e'$ by the β_P or *name* rule for some e'. If $e = B[A]$, then $B[A]\ g \mapsto B[A\ g]$ by the *float* rule.
- Derivations ending in an inference rule for **case** expressions are similar to the case for applications above.
- Derivations ending in an inference rule for delayable $\mathbf{let(rec)}\ x{:}\tau = u\ \mathbf{in}\ e$ expressions (binding a variable of type $delay(\tau)$), proceed by the induction hypothesis of e.
  - If $e \mapsto e'$ then $\mathbf{let(rec)}\ x{:}\tau = u\ \mathbf{in}\ e \mapsto \mathbf{let\ rec}\ x{:}\tau = u\ \mathbf{in}\ e'$.

  - If e is an answer or then so too is $\mathbf{let(rec)}\ x{:}\tau = u\ \mathbf{in}\ e$. Likewise when e is cyclic or when $e = E[y]$ where $y \neq x$.
  - If $e = E[x]$ where x is not bound by E, then we may proceed by cases on the representation of τ:
    * If $\tau : \mathsf{F}$ then $\mathbf{let(rec)}\ x{:}\tau = u\ \mathbf{in}\ E[x] \mapsto \mathbf{let(rec)}\ x{:}\tau = u\ \mathbf{in}\ E[u]$.
    * If $\tau : \mathsf{L}$ then we continue by the inductive hypothesis on u.
      · If $u \mapsto u'$ then
      $$\mathbf{let(rec)}\ x{:}\tau = u\ \mathbf{in}\ E[x] \mapsto \mathbf{let(rec)}\ x{:}\tau = u'\ \mathbf{in}\ E[x]$$
      · If u is an answer of the form W, then
      $$\mathbf{let(rec)}\ x{:}\tau = W\ \mathbf{in}\ E[x] \mapsto \mathbf{let(rec)}\ x{:}\tau = W\ \mathbf{in}\ E[W]$$
      · If u is an answer of the form $B[A]$, then
      $$\mathbf{let(rec)}\ x{:}\tau = B[A]\ \mathbf{in}\ E[x] \mapsto B[\mathbf{let(rec)}\ x{:}\tau = A\ \mathbf{in}\ E[x]]$$
      · If u is cyclic, then so is the whole expression.
      · And if $u = E'[x]$ where x is not bound by E' then $\mathbf{let(rec)}\ x{:}\tau = E'[x]\ \mathbf{in}\ E[x]$ is cyclic.
- The case for non-recursive strict **let** expressions (binding a variable of type $\tau : \mathsf{V}$) first proceed by the induction hypothesis on the bound expression similar to application and **case** expressions and, in the sub-case where the bound expression is a weak head-normal form, continue by the induction hypothesis on the body of the **let** similar to the above case for delayable **let** bindings. □

Lemma 5 (Progress). *If e is a program then either e is an answer, e is a cycle, or $e \mapsto e'$ for some e'*

Proof. Follows from lemma 4, since $\Theta; \bullet \vdash e : \tau$ rules out the possibility that $e = E[x]$. □

Lemma 6 (Subject reduction). *If $\Theta; \Gamma \vdash e : \tau$ and $e \to e'$ then $\Theta; \Gamma \vdash e' : \tau$.*

Proof. By cases on the possible reductions $e \to e'$:

- The *lookup* rule follows by lemma 1.
- The β_P and *match* follow by lemmas 2 and 3.
- The *float* rule follows by cases on the form of the frame context F and the binding context B.
- The *name* rule is immediate.

Finally, reduction inside an evaluation context follows by lemma 1. □

Corollary 1 (Type safety). *Every $\mathcal{L}$ program either converges or diverges.*

Proof. As a consequence of lemmas 5 and 6. □

A.2 Safety of $\mathcal{L}X$

The goal here is to show that every initial configuration of $\mathcal{L}X$ either converges or diverges.

Definition 6 (Convergence and Divergence). A machine configuration $\langle e|S|H\rangle$ *converges* when it reaches a final configuration in a finite number of steps, *i.e.,* $\langle e|S|H\rangle \mapsto^* \langle W| \bullet |H'\rangle$, and *diverges* either when it takes an infinite number of steps, *i.e.,* $\langle e|S|H\rangle \mapsto \langle e_1|S_1|H_1\rangle \mapsto \langle e_2|S_2|H_2\rangle \mapsto \ldots$, or when it reaches a cyclic configuration, *i.e.,* $\langle e|S|H\rangle \mapsto^* \langle x|S|\ x{:}\tau := \bullet; H\rangle$.

Property 2 (Stack Arity). *For any $\Theta \vdash obs(\rho)$:*

(1) *If $\Theta; \Gamma \mid \sigma \rightsquigarrow \tau \vdash S : \rho$ then there is an x, a Γ', and a stack $\Theta; \Gamma \mid \tau \vdash S' : \rho$ such that $\Gamma = \Gamma', x : \sigma$ and $S = x \mathsf{F} S'$.*

(2) *If* $\Theta; \Gamma \mid \tilde{\forall} a{:}k.\tau \vdash S : \rho$ *then there is a type* $\Theta \vdash \sigma : k$ *and a stack* $\Theta; \Gamma \mid \tau[\sigma/a] \vdash S' : \rho$ *such that* $S = \sigma \mathsf{F} S'$.

Proof. By inversion on the possible typing rules. □

Lemma 7 (Progress). *If* $\Theta; \bullet \vdash \langle e|S|H\rangle : \rho$ *is memory safe, then either* $\langle e|S|H\rangle$ *is finished,* $\langle e|S|H\rangle$ *is cyclic, or* $\langle e|S|H\rangle \mapsto \langle e'|S'|H'\rangle$ *for some* e', S', H'.

Proof. By induction on the derivation of $\Theta; \Gamma \vdash \langle e|S|H\rangle$. The most interesting case is for higher-arity function application, for example, when we have a configuration of the form $\langle W|S|H\rangle$ where $\Theta; \Gamma \vdash_\eta W : \sigma_0 \xrightarrow{r} \sigma_1 \rightsquigarrow \ldots \sigma_n \rightsquigarrow \tau$ and $\Theta; \Gamma \mid \sigma_0 \xrightarrow{r} \sigma_1 \rightsquigarrow \ldots \sigma_n \rightsquigarrow \tau \vdash S : \rho$ for some observable $\Theta \vdash obs(\tau)$. The typing derivation for W must have the form

$$\dfrac{\dfrac{\dfrac{\Theta; \Gamma, x_0 : \sigma_0, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash_\eta e : \tau \quad \Theta \vdash obs(\tau)}{\Theta; \Gamma, x_0 : \sigma_0, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash^{\mathsf{F}}_\eta e : \tau}}{\vdots}}{\dfrac{\Theta; \Gamma, x_0 : \sigma_0, x_1 : \sigma_1 \vdash^{\mathsf{F}}_\eta \widetilde{\lambda} x_n{:}\sigma_n.e : \sigma_n \rightsquigarrow \tau}{\Theta; \Gamma, x_0 : \sigma_0 \vdash^{\mathsf{F}}_\eta \widetilde{\lambda} x_1{:}\sigma_1 \ldots \widetilde{\lambda} x_n{:}\sigma_n.e : \sigma_1 \rightsquigarrow \ldots \sigma_n \rightsquigarrow \tau}}{\Theta; \Gamma \vdash_\eta \lambda^r x_0{:}\sigma_0.\widetilde{\lambda} x_1{:}\sigma_1 \ldots \widetilde{\lambda} x_n{:}\sigma_n.e : \sigma_0 \xrightarrow{r} \sigma_1 \rightsquigarrow \ldots \sigma_n \rightsquigarrow \tau}$$

where $e \neq \widetilde{\lambda}\mathbf{x}.e'$. Furthermore, by property 2, we must also have a stack $S = P_0\ r\ P_1\ \mathsf{F}\ \ldots\ \mathsf{F}\ P_n\ \mathsf{F}\ S'$ of the matching arity, where $S' \neq P'\ \mathsf{F}\ S''$. Therefore, the β^n reduction step applies. Likewise, the β^n will also apply for similar types beginning with $\forall^r$ instead of $\xrightarrow{r}$ and/or with $\widetilde{\forall}$ quantifiers interspersed within the chain of $\rightsquigarrow$ arrows.

The remaining cases for configurations of the form $\langle e|S|H\rangle$ proceed as follows:

- If e is not a *WHNF* then either $\langle e|S|H\rangle$ steps by one of the *apply*, *case*, *alloc*, or *force* rules, or $x{:}\tau := \bullet$ in H which is a cyclic configuration.
- If e is *WHNF* of some data type (Int, $List^{\,r}\,\tau$, or $Box^{\,r}_{\,s}\,\tau$) or polymorphic type (a), then either $\langle e|S|H\rangle$ steps by one of the *match*, *default*, or *memo* rules depending on S, or $\langle e|S|H\rangle$ is final if S is the empty stack. Note that in the case for a *memo* step, the assumption that $\langle e|S|H\rangle$ is memory safe ensures that there is already a black hole allocated in the heap that matches the variable being updated. □

Lemma 8 (Preservation). *If* $\Theta; \Gamma \vdash \langle e|S|H\rangle : \rho$ *is memory safe and* $\langle e|S|H\rangle \mapsto \langle e'|S'|H'\rangle$ *then* $\Theta; \Gamma \vdash \langle e'|S'|H'\rangle : \rho$ *is memory safe.*

Proof. By cases on the possible reductions $\langle e|S|H\rangle \mapsto \langle e'|S'|H'\rangle$, and then inversion on the derivation of $\Theta; \Gamma \vdash \langle e|S|H\rangle : \rho$, using lemma 1 for the cases for β^n and *match* steps and lemma 2 for the *default* and *alloc* steps. □

Corollary 2 (Type Safety). *Every initial* $\mathcal{L}X$ *configuration either converges or diverges.*

Proof. As a consequence of lemmas 7 and 8. □

B COMPILATION

As shortand, we will write just $C[\![e]\!]$ for $C[\![e]\!]\bullet$ and $\mathcal{E}[\![e]\!]{:}\tau$ for $\mathcal{E}[\![e]\!]\bullet{:}\tau$.

The completeness of the compilation process relies on the following property about coverage of the constraints on representations:

Property 3 (Non-observable and Non-delayable). *For all types* $\Theta \vdash \tau : r$,

(1) *either* $obs(r)$ *or* $r = \mathsf{F}$, *and*

(2) *either* $delay(r)$ *or* $r = \vee$.

Together with property 1, this means that every well-formed type $\Theta \vdash \tau : r$ either has the form $\tau = \sigma \rightsquigarrow \rho$ or $\tau = \widetilde{\forall} a{:}k.\sigma$, or is observable $\Theta \vdash obs(\tau)$.

Lemma 9 (Open Compilation). *For any* $\Theta; \Gamma \vdash e : \sigma$ *and* $\overline{P}$ *such that* $\Theta; \Gamma \vdash e\ \overline{P} : \tau$ *then*

(1) $\Theta; \Gamma \vdash^{\mathsf{F}}_{\eta} (\mathcal{E}[\![e]\!]\overline{P}{:}\tau) : \tau$ *if either* $\overline{P}$ *is empty or* e *is not a* $\widetilde{\lambda}$, *and*

(2) $\Theta; \Gamma \vdash_{\eta} (C[\![e]\!]\overline{P}) : \tau$ *if* $\Theta \vdash obs(\tau)$.

Proof. Both parts are proved simultaneously by induction on the typing derivation of $\Theta; \Gamma \vdash e : \sigma$ and the type τ. Part (1) proceeds by cases on the derivation of $\Theta; \Gamma \vdash e : \sigma$ and τ (and never increases the derivation of $\Theta; \Gamma \vdash e : \sigma$) as follows:

- If the derivation concludes with with a $\rightarrow I$ or $\forall I$ inference then e is a $\widetilde{\lambda}$ which forces $\overline{P}$ to be empty. The concluding inference is replaced by $\eta{\rightarrow}I$ or $\eta\forall I$, respectively, and the remaining derivation is given by the inductive hypothesis on typing derivation for the body of e.
- If $\tau = \sigma \rightsquigarrow \rho$ where e is not a $\widetilde{\lambda}$, then the derivation is given by the inductive hypothesis and an additional concluding $\eta{\rightsquigarrow}I$ inference.
- If $\tau = \widetilde{\forall} a{:}k.\sigma$ where e is not a $\widetilde{\lambda}$, then the derivation is given by the inductive hypothesis and an additional concluding $\eta\widetilde{\forall}$ inference.
- If $obs(\tau)$, then the derivation is given directly by part (2) of the inductive hypothesis.

Part (2) proceeds by cases on the concluding inference of the derivation of $\Theta; \Gamma \vdash e : \sigma$ (and may increase the type τ) as follows:

- *Var* and *Num* are replaced with ηVar and ηNum, respectively, followed by a chain of $\eta{\rightarrow}E$ and $\eta\forall E$ inferences for each parameter in $\overline{P}$.
- *Constr* is replaced with $\eta Constr$, where there is enough parameters to fully apply the constructor due to the assumption that $\Theta \vdash obs(\tau)$.
- $\rightarrow I$ and $\forall I$ are either replaced by $\eta{\rightarrow}I$ and $\eta\forall I$, respectively.
- $\rightarrow E$ and $\forall E$ either follows directly from the inductive hypothesis (if the argument is a parameter P') or requires an additional ηLet or $\eta Case$ (if the argument is not a parameter) depending on the type of the argument.
- *Let* and *LetRec* for a binding of type $\Theta \vdash delay(\sigma)$ are replaced with ηLet and $\eta LetRec$, respectively.
- *Let* for a binding of type $\Theta \vdash \tau : \vee$ is replaced with a $\eta Case$.
- *Case* is replaced with $\eta Case$. □

Corollary 3 (Compilation). *Given* $\Theta; \Gamma \vdash e : \tau$ *then*

(1) $\Theta; \Gamma \vdash^{\mathsf{F}}_{\eta} (\mathcal{E}[\![e]\!]{:}\tau) : \tau$, *i.e.,* $\mathcal{E}[\![e]\!]{:}\tau$ *is a* $\mathcal{L}X$ *expression, and*

(2) $\Theta; \Gamma \vdash_{\eta} (C[\![e]\!]) : \tau$ *if* $\Theta \vdash obs(\tau)$, *i.e.,* $C[\![e]\!]$ *is a* $\mathcal{L}X$ *program.*

Proof. As a consequence of lemma 9. □

Lemma 10 (Open η Elaboration). *If* $\Theta; \Gamma \vdash e : \sigma$ *and* $\Theta; \Gamma \vdash e\ \overline{P} : \tau$, *then*

(1) $\Theta; \Gamma \vdash (\mathcal{E}[\![e]\!]\overline{P}{:}\tau) = e\ \overline{P} : \tau$, *and*

(2) $\Theta; \Gamma \vdash (C[\![e]\!]\overline{P}) = e\ \overline{P} : \tau$.

Proof. Both parts are proved simultaneously by mutual induction on the typing derivation of $\Theta; \Gamma e : \sigma$ and the type τ. Part (1) proceeds by cases on the derivation of $\Theta; \Gamma e : \sigma$ and the type τ (and never increases the derivation) as follows:

- If e is a $\widetilde{\lambda}$, so the derivation ends in a $\to I$ or $\forall I$ inference, then the equality follows by congruence.
- $\tau = \sigma \rightsquigarrow \rho$: the equality follows by the $\eta_{\rightsquigarrow}$ extensional rule.
- $\tau = \widetilde{\forall} a{:}k.\sigma$: the equality follows by the $\eta_{\widetilde{\forall}}$ extensional rule.
- $obs(\tau)$: the equality follows by part (1) of the inductive hypothesis.

Part (2) proceeds by cases on the concluding inference of the derivation (but may increase the type τ) as follows:

- *Var*, *Num*, and *Constr*: follows by reflexivity.
- $\to I$ and $\forall I$: the equality follows by part (2) of the inductive hypothesis on the sub-expression in the body of the abstraction
- $\to E$ and $\forall E$: if the argument is a parameter (*i.e.,* a type or a variable), then the equality follows by the inductive hypothesis. Otherwise, the equality also requires the *name* operational rule.
- *Let* and *LetRec* where the bound variable is of a delayable type $delay(\sigma)$: follows from the *lift* extensional rule and the inductive hypothesis.
- *Let* where the bound variable is of type $\tau : \forall$: follows by the *lift* and $case/let_{\forall}$ extensional rules along with the inductive hypothesis.
- *Case*: follows by the *lift* extensional rule and the inductive hypothesis. □

Corollary 4 (η Elaboration)**.** *If* $\Theta; \Gamma \vdash e : \tau$*, then*

(1) $\Theta; \Gamma \vdash (\mathcal{E}[\![e]\!]{:}\tau) = e : \tau$*, and*
(2) $\Theta; \Gamma \vdash (C[\![e]\!]) = e : \tau$.

Proof. As a consequence of lemma 10. □

C SIMULATION

We break down the bisimulation between $\mathcal{L}$ and $\mathcal{L}X$ into two separate steps:

- A simulation between $\mathcal{L}$ programs and their compiled $\mathcal{L}X$ forms, both in terms of the high-level $\mathcal{L}$ operational semantics.
- A simulation between $\mathcal{L}X$ programs using the $\mathcal{L}$ operational semantics and $\mathcal{L}X$ machine configurations using the abstract machine semantics.

And since simulation compose, we get a complete simulation between the operational semantics of $\mathcal{L}$ and the abstract machine of $\mathcal{L}X$. Note that in order for the second simulation, it is important that the heap H be treated as an *unordered* list of bindings.

For the purpose of this bisimulation, we will consider $\mathcal{L}X$ extended with the following additional form of well-typed term:

$$\frac{\Theta; \Gamma \vdash_{\eta} W : \tau \quad \Theta; \Gamma, x : \tau \vdash_{\eta} e : \rho \quad \Theta \vdash obs(\rho)}{\Theta; \Gamma \vdash_{\eta} \mathbf{let}\, x{:}\tau = W \,\mathbf{in}\, e : \rho}$$

In other words, non-recursive **let**s are allowed to bind any type of variable (even non-delayable ones) to a weak-head normal form. Note that this extension to $\mathcal{L}X$ is still type safe. And further, this extension makes the $\mathcal{L}X$ sub-syntax closed under $\mathcal{L}$ operational reductions (in particular, *default*).

Definition 7 (Context typing)**.** For any $\mathcal{L}$ expression context C, the judgement $\Theta; \Gamma \mid \Gamma'; \tau \vdash C : \rho$ means that, for any $\Theta; \Gamma, \Gamma' \vdash e : \tau$, it follows that $\Theta; \Gamma \vdash C[e] : \rho$. The judgements $\Theta; \Gamma \mid \Gamma'; \tau \vdash_{\eta} C : \rho$ and $\Theta; \Gamma \mid \Gamma'; \tau \vdash^{\mathsf{F}}_{\eta} C : \rho$ are defined similarly.

C.1 Compilation simulation

Definition 8 (Simulation)**.** For any $\mathcal{L}$ program e and $\mathcal{L}X$ expression e', the *compilation simulation* relation $e \sim_\eta e'$ is

$$e \sim_\eta e' \text{ iff } C[\![e]\!] \mapsto^*_{default} e' \text{ for some } \mathcal{L}X\ e''$$

Lemma 11 (Context Compilation)**.** *Given $\Theta \vdash \sigma : s$, $\Theta \vdash \tau : t$, $\Theta \vdash obs(\rho)$, and $\Theta \vdash obs(\rho')$:*

(1) *Let $\Theta; \Gamma \mid \Gamma'; \sigma \vdash B : \sigma$ be any $\mathcal{L}$ binding context such that $\Theta; \Gamma \mid \Gamma'; \sigma \vdash B\ \overline{P} : \rho$. There is a $\mathcal{L}X$ evaluation context $\Theta; \Gamma \mid \Gamma'; \sigma \vdash_\eta B' : \rho$ such that $C[\![B[e]]\!]\overline{P} \mapsto^*_{default} B'[C[\![e]\!]\overline{P}]$ for any $\Theta; \Gamma, \Gamma' \vdash e : \sigma$.*

(2) *Let $\Theta; \Gamma \mid \Gamma'; \sigma \vdash F : \tau$ be any $\mathcal{L}$ frame context such that $\Theta; \Gamma \mid \Gamma'; \sigma \vdash F\ \overline{P} : \rho$. Either:*
   (a) *there is a parameter P' such that $C[\![F[e]]\!]\overline{P} = C[\![e]\!]P'\overline{P}$, or*
   (b) *There is a $\mathcal{L}X$ frame context $\Theta; \Gamma \mid \Gamma'; \sigma \vdash_\eta F' : \rho$ such that $C[\![F[e]]\!]\overline{P} \mapsto^*_{default} F'[C[\![e]\!]]$ for any $\Theta; \Gamma, \Gamma' \vdash e : \sigma$, or*

(3) *Let $\Theta; \Gamma \mid \Gamma'; \sigma \vdash E : \tau$ be any $\mathcal{L}$ evaluation context such that $\Theta; \Gamma \mid \Gamma'; \sigma \vdash E\ \overline{P} : \rho$. There is a $\mathcal{L}X$ evaluation context $\Theta; \Gamma \mid \Gamma'; \rho' \vdash_\eta E' : \rho$ and $\overline{P'}$ such that $C[\![E[e]]\!]\overline{P} \mapsto^*_{default} E'[C[\![e]\!]\overline{P'}]$ for any $\Theta; \Gamma, \Gamma' \vdash e : \sigma$.*

Proof. By mutual induction on the contexts B, F, and E:

- $\mathbf{let}\ x{:}\tau' = W\ \mathbf{in}\ \square$ where $\Theta \vdash \tau' : \vee$: We have $B' = \mathbf{let}\ x{:}\tau' = C[\![W]\!]\ \mathbf{in}\ \square$ where

$$C[\![\mathbf{let}\ x{:}\tau' = W\ \mathbf{in}\ e]\!]\overline{P} = \mathbf{case}\ C[\![W]\!]\ \mathbf{of}\ \{x{:}\tau' \to C[\![e]\!]\overline{P}\}$$
$$\mapsto_{default} \mathbf{let}\ x{:}\tau' = C[\![W]\!]\ \mathbf{in}\ C[\![e]\!]\overline{P}$$

- $\mathbf{let(rec)}\ x{:}\tau' = u\ \mathbf{in}\ \square$ where $\Theta \vdash delay(\tau')$: We have $B' = \mathbf{let(rec)}\ x{:}\tau' = \mathcal{E}[\![u]\!]{:}\tau'\ \mathbf{in}\ \square$

$$C[\![\mathbf{let}\ x{:}\tau' = u\ \mathbf{in}\ e]\!]\overline{P} = \mathbf{let(rec)}\ x{:}\tau' = \mathcal{E}[\![u]\!]{:}\tau'\ \mathbf{in}\ C[\![e]\!]\overline{P}$$

- $\square\ P'$: We have case (a) since $C[\![e\ P']\!]\overline{P} = C[\![e]\!]P'\overline{P}$.
- $\mathbf{case}\ \square\ \mathbf{of}\ \{\overline{p \to u}\}$: We have case (b) with $F' = \mathbf{case}\ \square\ \mathbf{of}\ \{\overline{p \to C[\![u]\!]\overline{P}}\}$ since

$$C[\![\mathbf{case}\ e\ \mathbf{of}\ \{\overline{p \to u}\}]\!]\overline{P} = \mathbf{case}\ C[\![e]\!]\ \mathbf{of}\ \{\overline{p \to C[\![u]\!]\overline{P}}\}$$

- $\mathbf{let}\ x{:}\tau' = \square\ \mathbf{in}\ u$ where $\Theta \vdash \tau' : \vee$: We have case (b) with $F' = \mathbf{case}\ \square\ \mathbf{of}\ \{x{:}\tau' \to C[\![u]\!]\overline{P}\}$ since

$$C[\![\mathbf{let}\ x{:}\tau' = e\ \mathbf{in}\ u]\!]\overline{P} = \mathbf{case}\ C[\![e]\!]\ \mathbf{of}\ \{\overline{p \to C[\![u]\!]\overline{P}}\}$$

- $\mathbf{let(rec)}\ x{:}\tau' = \square\ \mathbf{in}\ E_1[x]$ where $\Theta \vdash \tau' : \mathsf{L}$ and $x \notin BV(E')$: by the inductive hypothesis on E_1, we have an E_1' and $\overline{P_1'}$ such that

$$\begin{aligned} C[\![\mathbf{let(rec)}\ x{:}\tau' = e\ \mathbf{in}\ E_1[x]]\!]\overline{P} &= \mathbf{let(rec)}\ x{:}\tau' = \mathcal{E}[\![e]\!]{:}\tau'\ \mathbf{in}\ C[\![E_1[x]]\!]\overline{P} \\ &= \mathbf{let(rec)}\ x{:}\tau' = C[\![e]\!]\ \mathbf{in}\ C[\![E_1[x]]\!]\overline{P} \\ &\mapsto^*_{default} \mathbf{let(rec)}\ x{:}\tau' = C[\![e]\!]\ \mathbf{in}\ E_1'[C[\![x]\!]\overline{P_1'}] \qquad (IH) \\ &= \mathbf{let(rec)}\ x{:}\tau' = C[\![e]\!]\ \mathbf{in}\ E_1'[x\ \overline{P_1'}] \end{aligned}$$

  So we have case (b) with $F' = \mathbf{let(rec)}\ x{:}\tau' = \square\ \mathbf{in}\ E_1'[x\ \overline{P_1'}]$.
- $\square$: we have $E' = \square$ and $\overline{P'} = \overline{P}$
- $B[E]$: follows from the inductive hypothesis by transitivity of reduction and composition of evaluation contexts.

- $F[E]$: also follows by the inductive hypothesis. In case (a) we have $C[\![F[E[e]]]\!]\overline{P} = C[\![E[e]]\!]P'\overline{P} = E'[C[\![e]\!]\overline{P''}]$. In the case (b) we have $C[\![F[E[e]]]\!]\overline{P} = F'[C[\![E[e]]\!]] = F'[E'[C[\![e]\!]\overline{P'}]]$. □

Corollary 5 (Decomposition). *For any $\mathcal{L}$ evaluation context $\Theta;\Gamma \mid \Gamma';\sigma \vdash E : \rho$ with $\Theta \vdash obs(\rho)$, there is a $\mathcal{L}X$ evaluation context E' and parameter list $\overline{P'}$ such that $C[\![E[e]]\!] \mapsto^*_{default} E'[C[\![e]\!]\overline{P'}]$ for any $\Theta;\Gamma,\Gamma' \vdash e : \sigma$.*

Proof. As a consequence of lemma 11. □

Corollary 6 (Answer Preservation). *For any $\mathcal{L}$ answer A, $C[\![A]\!] \mapsto^*_{default} A'$ for some $\mathcal{L}X$ answer A'.*

Proof. As a consequence of lemma 11. □

Lemma 12 (Substitution). (1) $(C[\![e]\!]\overline{P})[P'/\mathbf{x}] = C[\![e[P'/\mathbf{x}]]\!]\overline{P}$.
(2) $(\mathcal{E}[\![e]\!]\overline{P}{:}\tau)[P'/\mathbf{x}] = \mathcal{E}[\![e[P'/\mathbf{x}]]\!]\overline{P}{:}\tau$.

Proof. By the fact that the $C[\![-]\!]\overline{P}$ and $\mathcal{E}[\![-]\!]\overline{P}{:}\tau$ translations are compositional [Downen and Ariola 2014a]. □

Lemma 13. *If $\Theta;\Gamma \vdash e : \tau$ and $\Theta;\Gamma \vdash e\ \overline{P} : \rho$ where $\Theta \vdash obs(\rho)$ then $(\mathcal{E}[\![e]\!]{:}\tau)\ \overline{P} \mapsto^*_{\beta_P} C[\![e]\!]\ \overline{P}$.*

Proof. By induction on the derivation of $\Theta;\Gamma \vdash e : \tau$. If e is a $\widetilde{\lambda}$ then $\mathcal{E}[\![e]\!]{:}\tau = C[\![e]\!]$. Otherwise, the number of η expansions inserted by $\mathcal{E}[\![e]\!]{:}\tau$ is equal to the number of parameters $\overline{P}$, so that β_P reduction cancels out the η expansion. □

Lemma 14 (Forward simulation). *Given $e \sim_\eta u$:*

(1) *If e is an answer then $u \mapsto^*_{default} A$ for some answer A.*
(2) *If $e \mapsto e'$ then $u \mapsto^* u'$ and $e' \sim_\eta u'$ for some $\mathcal{L}X$ expression u'.*

Proof. (1) We know $C[\![e]\!] \mapsto^*_{default} A \not\mapsto$ by corollary 6 and further $u \mapsto^*_{default} A$ because $C[\![e]\!] \mapsto^*_{default} u$ and $\mapsto_{default}$ is deterministic.

(2) By cases on the reduction $e \mapsto e'$. In each non-*float* case, we show that

$$C[\![e]\!] \mapsto_{default} u_1 \mapsto^* u_1' \leftarrow^*_{default} C[\![e']\!]$$

such that $u_1 \mapsto^* u_1'$ by non-*default* reductions. From the assumption $e \sim u$ we know that $C[\![e]\!] \mapsto_{default} u$. It then follows that either $e' \sim_\eta u_1$ (in the reflexive case that $u_1 = u_1'$) or that $u \mapsto_{default} u_1 \mapsto^* u_1'$ such that $e' \sim_\eta u_1'$ (in the non-reflexive case).

- β_P: Given $E[(\lambda^r\mathbf{x}.e)\ P] \mapsto E[e[P/\mathbf{x}]]$ then

$$\begin{aligned}
C[\![E[(\lambda^r\mathbf{x}.e)\ P]]\!] &\mapsto^*_{default} E'[C[\![(\lambda^r\mathbf{x}.e)\ P]\!]\overline{P'}] && (corollary\ 5)\\
&= E'[(\lambda^r\mathbf{x}.\mathcal{E}[\![e]\!]{:}\tau)\ P\ \overline{P'}] && (e : \tau)\\
&\mapsto E'[C[\![e]\!][P/\mathbf{x}]\ \overline{P'}] && (\beta_P)\\
&= E'[C[\![e[P/\mathbf{x}]]\!]\ \overline{P'}] && (lemma\ 12)\\
&\leftarrow^*_{default} C[\![E[e[P/\mathbf{x}]]]\!] && (corollary\ 5)
\end{aligned}$$

- *name*: Given $E[e\ u] \mapsto E[\mathbf{let}\ x{:}\tau = u\ \mathbf{in}\ e\ x]$ because $u : \tau$ and $u \notin Param$ then

$$\begin{aligned}
C[\![E[e\ u]]\!] &\mapsto^*_{default} E'[C[\![e\ u]\!]\overline{P'}] && (corollary\ 5)\\
&= E'[C[\![\mathbf{let}\ x{:}\tau = u\ \mathbf{in}\ e\ x]\!]\overline{P'}]\\
&\leftarrow^*_{default} C[\![E[\mathbf{let}\ x{:}\tau = u\ \mathbf{in}\ e\ x]]\!]
\end{aligned}$$

- *match*: Given $E[\mathbf{case}\ W\ \mathbf{of}\ \{\overline{p_i \to u_i}\}] \mapsto E[u_i[\overline{P/x}]]$ then $W = K\ \overline{P}$ so that

$$\begin{aligned}
C[\![E[\mathbf{case}\ W\ \mathbf{of}\ \{\overline{p_i \to u_i}\}]\!] &\mapsto^*_{default} E'[C[\![\mathbf{case}\ W\ \mathbf{of}\ \{\overline{p_i \to u_i}\}]\!]\overline{P'}] && (corollary\ 5)\\
&= E'[\mathbf{case}\ C[\![W]\!]\ \mathbf{of}\ \{\overline{p_i \to C[\![u_i]\!]\overline{P'}}\}]\\
&= E'[\mathbf{case}\ W\ \mathbf{of}\ \{\overline{p_i \to C[\![u_i]\!]\overline{P'}}\}] && (C[\![W]\!] = W)\\
&\mapsto E'[C[\![u_i]\!]\overline{P'}[\overline{P/x}]] && (match)\\
&= E'[C[\![u_i[\overline{P/x}]]\!]\overline{P'}] && (lemma\ 12)\\
&\leftarrow^*_{default} C[\![E[u_i[\overline{P/x}]]]\!]
\end{aligned}$$

- *default*: Given $E[\mathbf{case}\ W\ \mathbf{of}\ \{\ldots; x{:}\tau \to u\}] \mapsto E[\mathbf{let}\ x{:}\tau = W\ \mathbf{in}\ u]$ then

$$\begin{aligned}
C[\![E[\mathbf{case}\ W\ \mathbf{of}\ \{\ldots; x{:}\tau \to u\}]\!] &\mapsto^*_{default} E'[C[\![\mathbf{case}\ W\ \mathbf{of}\ \{\ldots; x{:}\tau \to u\}]\!]\overline{P'}] && (corollary\ 5)\\
&= E'[\mathbf{case}\ C[\![W]\!]\ \mathbf{of}\ \{\ldots; x{:}\tau \to C[\![u]\!]\overline{P'}\}]\\
&\mapsto E'[\mathbf{let}\ x{:}\tau = C[\![W]\!]\ \mathbf{in}\ C[\![u]\!]\overline{P'}] && (default)\\
&= E'[C[\![\mathbf{let}\ x{:}\tau = W\ \mathbf{in}\ u]\!]\overline{P'}]\\
&\leftarrow^*_{default} C[\![E[\mathbf{let}\ x{:}\tau = W\ \mathbf{in}\ u]]\!] && (corollary\ 5)
\end{aligned}$$

- *lookup*: Given $E[\mathbf{let(rec)}\ x{:}\tau = u\ \mathbf{in}\ E_1[x]] \mapsto E[\mathbf{let(rec)}\ x{:}\tau = u\ \mathbf{in}\ E_1[u]]$ where $\tau : \mathsf{F}$ and $x \notin BV(E_1)$, then

$$\begin{aligned}
C[\![E[\mathbf{let(rec)}\ x{:}\tau = u\ \mathbf{in}\ E_1[x]]\!] &\mapsto^*_{default} E'[C[\![\mathbf{let(rec)}\ x{:}\tau = u\ \mathbf{in}\ E_1[x]]\!]\overline{P'}]\\
&\mapsto^*_{default} E'[\mathbf{let(rec)}\ x{:}\tau = \mathcal{E}[\![u]\!]{:}\tau\ \mathbf{in}\ C[\![E_1[x]]\!]\overline{P'}] && (corollary\ 5)\\
&\mapsto^*_{default} E'[\mathbf{let(rec)}\ x{:}\tau = \mathcal{E}[\![u]\!]{:}\tau\ \mathbf{in}\ E_1'[x\ \overline{P_1'}]] && (lemma\ 11)\\
&\mapsto E'[\mathbf{let(rec)}\ x{:}\tau = \mathcal{E}[\![u]\!]{:}\tau\ \mathbf{in}\ E_1'[\mathcal{E}[\![u]\!]{:}\tau\ \overline{P_1'}]] && (lookup)\\
&\mapsto^*_{\beta_P} E'[\mathbf{let(rec)}\ x{:}\tau = \mathcal{E}[\![u]\!]{:}\tau\ \mathbf{in}\ E_1'[C[\![u]\!]\overline{P_1'}]] && (lemma\ 13)\\
&\leftarrow^*_{default} C[\![E[\mathbf{let(rec)}\ x{:}\tau = u\ \mathbf{in}\ E_1[u]]]\!] && (lemma\ 11)
\end{aligned}$$

  The case is similar when instead $obs(\tau) : \mathsf{F}$ and u is a weak head-normal form, because $\mathcal{E}[\![u]\!]{:}\tau = C[\![u]\!]$ is also a weak head-normal form.
- *float*: Given $E[F[B[A]]] \mapsto E[B[F[A]]]$, then there are two cases depending on F from lemma 11. For case (a), the result is immediate because $C[\![E[F[B[A]]]]\!] = C[\![E[F[B[A]]]]\!]$.

In case (b), we have

$$\begin{aligned}
C[\![E[F[B[A]]]]\!] &\mapsto^*_{default} E'[C[\![F[B[A]]]\!]\overline{P'}] && (corollary\ 5)\\
&\mapsto^*_{default} E'[F'[C[\![B[A]]\!]]] && (lemma\ 11)\\
&\mapsto^*_{default} E'[F'[B'[C[\![A]\!]]]] && (lemma\ 11)\\
&\mapsto^*_{default} E'[F'[B'[A']]] && (corollary\ 6)\\
&\mapsto E'[B'[F'[A']]] && (float)\\
&\leftarrow^*_{default} E'[B'[F'[C[\![A]\!]]]] && (corollary\ 6)\\
&\leftarrow^*_{default} E'[B'[C[\![F[A]]\!]\overline{P'}]] && (lemma\ 11)\\
&\leftarrow^*_{default} E'[C[\![B[F[A]]]\!]\overline{P'}] && (lemma\ 11)\\
&\leftarrow^*_{default} C[\![E[B[F[A]]]]\!] && (corollary\ 5)
\end{aligned}$$

□

C.2 Machine simulation

Definition 9 (Internal vs external reduction)**.** The *internal reduction* steps of the $\mathcal{L}X$ abstract machine (written $\mapsto_i$) are given by the β^n, *apply*, *case*, *alloc*, *rec*, and *force* rules. All other steps are *external reductions* (written $\mapsto_e$) of $\mathcal{L}X$.

Definition 10 (Simulation)**.** For any $\mathcal{L}X$ program e' and $\mathcal{L}X$ machine configuration $\langle u|S|H\rangle$, the *machine simulation* relation $e \sim_{\langle\rangle} \langle u|S|H\rangle$ is

$$e \sim_{\langle\rangle} \langle u|S|H\rangle \text{ iff } \langle e| \bullet |\bullet\rangle \mapsto_i^* \langle u|S|H\rangle$$

Definition 11 (Stack and heap composition)**.** We will write the composition of two stacks as $S + S'$ (that is, replacing the empty stack $\bullet$ inside S with S'), and the composition of to disjoint heaps as $H + H'$.

Lemma 15 (Refocusing)**.** (1) *For all $\mathcal{L}X$ binding contexts $\Theta;\Gamma \mid x\tau;\sigma \vdash_\eta B : \sigma$, there is a heap binding $x{:}\tau := u$ such that $\langle B[e]|S_0|H_0\rangle \mapsto_i \langle e|S_0|x{:}\tau := u; H_0\rangle$ for any $\Theta;\Gamma, x : \tau \vdash_\eta e : \sigma$, $\Theta;\Gamma \mid \sigma \vdash S_0 : \rho$, and $\Theta;\bullet \vdash H_0 : \Gamma$.*

(2) *For all $\mathcal{L}X$ frame contexts $\Theta;\Gamma \mid \Gamma';\sigma \vdash F : \tau$, there is a heap $\Theta;\Gamma \vdash H : \Gamma'$ and non-empty stack $\Theta;\Gamma \mid \sigma \vdash S : \tau$ such that $\langle F[e]|S_0|H_0\rangle \mapsto_i^+ \langle e|S + S_0|H + H_0\rangle$ for any $\Theta;\Gamma,\Gamma' \vdash_\eta e : \sigma$, $\Theta;\Gamma,\Gamma' \mid \tau \vdash S_0 : \rho$, and $\Theta;\bullet \vdash H_0 : \Gamma$.*

(3) *For all $\mathcal{L}X$ evaluation contexts and $\Theta;\Gamma \mid \Gamma';\sigma \vdash E : \tau$, there is a heap $\Theta;\Gamma \vdash H : \Gamma'$ and stack $\Theta;\Gamma,\Gamma' \mid \sigma \vdash S : \tau$ such that $\langle E[e]|S_0|H_0\rangle \mapsto_i^* \langle e|S + S_0|H + H_0\rangle$ for any $\Theta;\Gamma,\Gamma' \vdash_\eta e : \sigma$, $\Theta;\Gamma \mid \tau \vdash S_0 : \rho$, and $\Theta;\bullet \vdash H_0 : \Gamma$.*

Proof. By mutual induction on E, B, and F contexts.

- $\Box$: $S = \bullet$ and $H = \bullet$ by reflexivity.
- $B[E']$ and $F[E']$: follows by the inductive hypothesis.
- $\mathbf{let(rec)}\ x{:}\tau = u\ \mathbf{in}\ \Box$ where $delay(\tau)$: the binding is $x{:}\tau = u$ as follows:

$$\langle \mathbf{let(rec)}\ x{:}\tau = u\ \mathbf{in}\ e|S_0|H_0\rangle \mapsto_i \langle e|S_0|x{:}\tau := u; H_0\rangle \qquad (alloc, rec)$$

- $\Box\ P$: $e\ P$ cannot be a weak head-normal form (since that would be ill-typed), so $S = P\ s \bullet$ and $H = \bullet$ as follows:

$$\langle e\ P|S_0|H_0\rangle \mapsto_i \langle e|P\ s\ S_0|H_0\rangle \qquad (apply)$$

- **case** $\square$ **of** $\{\overline{p \to u}\}$: $S =$ **case** $\{\overline{p \to u}\}$; $\bullet$ and $H = \bullet$ as follows:

$$\langle \textbf{case}\ e\ \textbf{of}\ \{\overline{p \to u}\} | S_0 | H_0 \rangle \mapsto_i \langle e |\ \textbf{case}\ \{\overline{p \to u}\}; S_0 | H_0 \rangle \qquad (case)$$

- **let(rec)** $x{:}\tau = \square$ **in** $E'[x]$ where $\tau : \mathsf{L}$ and x is not bound by E': by the inductive hypothesis on E, there is an appropriate S' and H' such that

$$\begin{aligned} \langle \textbf{let(rec)}\ x{:}\tau = e\ \textbf{in}\ E'[x] | S_0 | H_0 \rangle &\mapsto_i \langle E'[x] | S_0 | x{:}\tau := e; H_0 \rangle && (alloc, rec) \\ &\mapsto_i^* \langle x | S' + S_0 | H' + x{:}\tau := e; H_0 \rangle && (IH) \\ &\mapsto_i \langle e | !x; S' + S_0 | H' + x{:}\tau := \bullet; H_0 \rangle && (force) \end{aligned}$$

  So $S = !x; S'$ and $H = H' + x{:}\tau := \bullet$.

□

Lemma 16 (Decomposition). *For any $\mathcal{L}X$ evaluation context $\Theta; \bullet \mid \Gamma; \sigma \vdash_\eta E : \rho$ with $\Theta \vdash obs(\rho)$, there is a heap $\Theta; \bullet \vdash H : \Gamma$ and stack $\Theta; \Gamma \mid \sigma \vdash S : \rho$ such that, for any expression $\Theta; \Gamma \vdash_\eta e : \sigma$, $\langle E[e] | \bullet | \bullet \rangle \mapsto_i^* \langle e | S | H \rangle$.*

Proof. As a consequence of lemma 15. □

Lemma 17 (Forward simulation). *Given $e \sim_{\langle\rangle} \langle u | S | H \rangle$:*

(1) *If e is an answer then $\langle u | S | H \rangle \mapsto^* \langle W | \bullet | H \rangle$ for some W and H'.*
(2) *If $e \mapsto e'$ then $\langle u | S | H \rangle \mapsto^* \langle u' | S' | H' \rangle$ and $e' \sim_{\langle\rangle} \langle u' | S' | H' \rangle$ for some $\langle u' | S' | H' \rangle$.*

Proof. (1) By assumption $\langle e | \bullet | \bullet \rangle \mapsto_i^* \langle u | S | H \rangle$. And since e is an answer, then $e = B_1[\ldots[B_n[W]]]$, and so $\langle e | \bullet | \bullet \rangle \mapsto_i^* \langle W | \bullet | H' \rangle \not\mapsto$ by lemma 15. And because $\mapsto_i$ is deterministic, we have that $\langle u | S | H \rangle \mapsto_i^* \langle W | \bullet | H' \rangle$.

(2) By cases on the reduction $e \mapsto e'$. In each case, we show that

$$\langle e | \bullet | \bullet \rangle \mapsto_i^* \langle u_1 | S_1 | H_1 \rangle \mapsto_e^* \langle u_1' | S_1' | H_1' \rangle \leftarrow_i^* \langle e' | \bullet | \bullet \rangle$$

It then follows from the assumption $\langle e | \bullet | \bullet \rangle \mapsto_i^* \langle u | S | H \rangle$ and the determinism of $\mapsto_i$ that $\langle u | S | H \rangle \mapsto_i^* \langle u' | S' | H' \rangle \leftarrow_i^* \langle u_1 | S_1 | H_1 \rangle$ for some $\langle u' | S' | H' \rangle$, and therefore either

$$\langle u | S | H \rangle \mapsto_i^* \langle u' | S' | H' \rangle \leftarrow_i^* \langle u_1 | S_1 | H_1 \rangle = \langle u_1' | S_1' | H_1' \rangle \leftarrow_i^* \langle e' | \bullet | \bullet \rangle$$

in the reflexive case or

$$\langle u | S | H \rangle \mapsto_i^* \langle u' | S' | H' \rangle = \langle u_1 | S_1 | H_1 \rangle \mapsto_e^+ \langle u_1' | S_1' | H_1' \rangle \leftarrow_i^* \langle e' | \bullet | \bullet \rangle$$

in the non-reflexive case (because $\langle u_1 | S_1 | H_1 \rangle \not\mapsto_i$ due to the fact that $\mapsto_i$ and $\mapsto_e$ redexes are disjoint).

- (β_P) Given $E[(\lambda^r \mathbf{x}.\widetilde{\lambda}\overline{\mathbf{y}}.e)\ P'\ \overline{P}] \mapsto E[(\widetilde{\lambda}\overline{\mathbf{y}}.e[P'/\mathbf{x}])\ \overline{P}]$ then

$$\begin{aligned} \langle E[(\lambda^r \mathbf{x}.\widetilde{\lambda}\overline{\mathbf{y}}.e)\ P'\ \overline{P}] | \bullet | \bullet \rangle &\mapsto_i^* \langle \lambda^r \mathbf{x}.\widetilde{\lambda}\overline{\mathbf{y}}.e | P'\ r\ \overline{P\ \mathsf{F}}\ S | H \rangle && (lemma\ 16) \\ &\mapsto_i \langle e[P'/\mathbf{x}, \overline{P/\mathbf{y}}] | S | H \rangle && (\beta^n) \\ &\leftarrow_i \langle \widetilde{\lambda}\overline{\mathbf{y}}.e[P'/\mathbf{x}] | \overline{P\ \mathsf{F}}\ S | H \rangle && (\beta^n) \\ &\leftarrow_i^* \langle E[\widetilde{\lambda}\overline{\mathbf{y}}.e[P'/\mathbf{x}]] | \bullet | \bullet \rangle && (lemma\ 16) \end{aligned}$$

- (*match*) Given $E[\textbf{case}\ W\ \textbf{of}\ \overline{p_i \to u_i}] \mapsto E[u_i[\overline{P/x}]]$ then

$$\begin{aligned} \langle E[\textbf{case}\ W\ \textbf{of}\ \overline{p_i \to u_i}] | \bullet | \bullet \rangle &\mapsto_i^* \langle W |\ \textbf{case}\ \{\overline{p_i \to u_i}\}; S | H \rangle && (lemma\ 16) \\ &\mapsto_e \langle u_i[\overline{P/x}] | S | H \rangle && (match) \\ &\leftarrow_i^* \langle E[u_i[\overline{P/x}]] | \bullet | \bullet \rangle && (lemma\ 16) \end{aligned}$$

- (*default*) Given $E[\mathbf{case}\ W\ \mathbf{of}\ \overline{\ldots ; x{:}\tau \to u}] \mapsto E[\mathbf{let}\ x{:}\tau = W\ \mathbf{in}\ u]$ then

$$\begin{aligned}
\langle E[\mathbf{case}\ W\ \mathbf{of}\ \overline{\ldots ; x{:}\tau \to u}] | \bullet | \bullet \rangle &\mapsto_i^* \langle W | \mathbf{case}\ \{\overline{\ldots ; x{:}\tau \to u}\}; S | H \rangle && (\textit{lemma } 16)\\
&\mapsto_e \langle u | S | x{:}\tau := W; H \rangle && (\textit{default})\\
&\leftarrow_i \langle \mathbf{let}\ x{:}\tau = W\ \mathbf{in}\ u | S | H \rangle && (\textit{alloc})\\
&\leftarrow_i^* \langle E[\mathbf{let}\ x{:}\tau = W\ \mathbf{in}\ u] | \bullet | \bullet \rangle && (\textit{lemma } 16)
\end{aligned}$$

- (*lookup*) Given $E[\mathbf{let(rec)}\ x{:}\tau = W\ \mathbf{in}\ E'[x]] \mapsto E[\mathbf{let(rec)}\ x{:}\tau = W\ \mathbf{in}\ E'[u]]$ then

$$\begin{aligned}
\langle E[\mathbf{let(rec)}\ x{:}\tau = W\ \mathbf{in}\ E'[x]] | \bullet | \bullet \rangle &\mapsto_i^* \langle E'[x] | S | x{:}\tau := W; H \rangle && (\textit{lemma } 16)\\
&\mapsto_i^* \langle x | S' + S | H' + x{:}\tau := W; H \rangle && (\textit{lemma } 15)\\
&\mapsto_e \langle W | S' + S | H' + x{:}\tau := W; H \rangle && (\textit{lookup})\\
&\leftarrow_i^* \langle E'[W] | S | x{:}\tau := W; H \rangle && (\textit{lemma } 15)\\
&\leftarrow_i^* \langle E[\mathbf{let}\ x{:}\tau = W\ \mathbf{in}\ E'[W]] | \bullet | \bullet \rangle && (\textit{lemma } 16)
\end{aligned}$$

- (*float*) Given $E[F[B[A]]] \mapsto E[B[F[A]]]$ then

$$\begin{aligned}
\langle E[F[B[A]]] | \bullet | \bullet \rangle &\mapsto_i^* \langle F[B[A]] | S | H \rangle && (\textit{lemma } 16)\\
&\mapsto_i^+ \langle B[A] | S' + S | H' + H \rangle && (\textit{lemma } 15)\\
&\mapsto_i \langle A | S' + S | x{:}\tau := u; H' + H \rangle && (\textit{lemma } 15)\\
&= \langle A | S' + S | H' + x{:}\tau := u; H \rangle\\
&\leftarrow_i^+ \langle F[A] | S | x{:}\tau := u; H \rangle && (\textit{lemma } 15)\\
&\leftarrow_i \langle B[F[A]] | S | H \rangle && (\textit{lemma } 15)\\
&\leftarrow_i^* \langle E[B[F[A]]] | \bullet | \bullet \rangle && (\textit{lemma } 16)
\end{aligned}$$

□

C.3 Operational soundness

Corollary 7 (Operational soundness)**.** *For any $\mathcal{L}$ program e, if $e \mapsto^* A$ then there is some W and H such that $\langle C[\![e]\!] | \bullet | \bullet \rangle \mapsto \langle W | \bullet | H \rangle \leftarrow^* \langle C[\![A]\!] | \bullet | \bullet \rangle$.*

Proof. Follow from lemmas 14 and 17, and the fact that the $\mathcal{L}$ operational semantics and $\mathcal{L}X$ abstract machine are deterministic. □