Making a Faster Curry with Extensional Types

Paul Downen
Zachary Sullivan
Zena M. Ariola
University of Oregon
Eugene, Oregon, USA
pdownen@cs.uoregon.edu
zsulliva@cs.uoregon.edu
ariola@cs.uoregon.edu

Simon Peyton Jones
Microsoft Research
Cambridge, UK
simonpj@microsoft.com

Abstract

Curried functions apparently take one argument at a time, which is slow. So optimizing compilers for higher-order languages invariably have some mechanism for working around currying by passing several arguments at once, as many as the function can handle, which is 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. Integrating call-by-name with other evaluation strategies in the same intermediate language expresses the arity of a function in its type and gives a principled and compositional account of multi-argument curried functions. An unexpected, but significant, bonus is that our approach is equally suitable for a call-by-value language and a call-by-need language, and it can be readily integrated into an existing compilation framework.

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

Keywords arity, extensionality, type systems

ACM Reference Format:
Paul Downen, Zachary Sullivan, Zena M. Ariola, and Simon Peyton Jones. 2019. Making a Faster Curry with Extensional Types. In *Proceedings of the 12th ACM SIGPLAN International Haskell Symposium (Haskell '19), August 22–23, 2019, Berlin, Germany.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3331545.3342594

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 optimizing compiler to η 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 to run-time performance in practice [Marlow and Peyton Jones 2004]. But this η expansion could 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 optimizing 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 η expand f_1 and replace it with f_2. For this reason GHC—an optimizing compiler for Haskell—keeps track of the *arity* of every in-scope variable, such as h, and uses that information to guide such transformations. This notion of arity is not unique to Haskell or GHC: the same impact of η 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 η 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 by characterizing arity formally in a type system. However, the way we do so differs from previous work [Bolingbroke and Peyton Jones 2009] in integrating types with calling conventions (by which they mean arity) that would require quite a large change for GHC to move from a lazy-by-default to strict-by-default intermediate language. Instead, our idea is to leverage multiple calling conventions (by which we mean, *e.g.,* call-by-name versus call-by-value) to solve the issues surrounding arity and η laws with types. The advantage of this approach is that it is less monolithic, and can be added to a variety of existing systems without disrupting orthogonal parts of their semantics or implementation, making it easier to incorporate into a serious compiler like GHC. Supporting this idea, we make these contributions:

- (Section 3) We describe our approach by starting with a standard call-by-value polymorphic λ-calculus, F_v, and extending it with a new function type that can express higher arities. We call the extended language $\widetilde{\mathrm{F}}_v$, and give a self-contained meaning of arity in terms of the type system and equational theory of $\widetilde{\mathrm{F}}_v$.
- (Section 4) To demonstrate the minimal impact on an existing implementation, we show the addition to a standard call-by-value abstract machine which is needed to support fast higher-arity function calls, but otherwise keeping the same semantics as F_v.
- (Section 5) Since we intend to preserve the original semantics of the above machine, we need to compile $\widetilde{\mathrm{F}}_v$ programs in a way that makes their evaluation strategy explicit to a stock call-by-value implementation. We show how to execute our extended intermediate language by performing a simple translation from $\widetilde{\mathrm{F}}_v$ to $\widetilde{\mathrm{F}}_v^{-}$ based on η expansion, which is shown to be correct.
- (Section 6) This approach does not just apply to call-by-value languages. We explain how the same extension can be made to a call-by-need λ-calculus by likewise formalizing the notion of arity in Haskell programs.

We believe our approach could be integrated well with levity polymorphism [Eisenberg and Peyton Jones 2017] (discussed in Section 7) by extending their solution for polymorphism of non-uniform representations to polymorphism of non-uniform arities. Unlike previous work on optimizing curried functions (which we discuss in Section 8) our approach grows directly from deep roots in type theory; in particular *call-by-push-value* [Levy 2001] and *polarity* [Danos et al. 1997; Munch-Maccagnoni 2013; Zeilberger 2009], which tell 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.

2 The Key Idea

Informally, we say the *arity* of a function is 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`; the only time `f` will compute anything is when it is given three arguments. We begin by explaining why arity is important, before intuitively introducing our new approach.

2.1 Motivation

There are several reasons why an optimizing 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
  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 η equivalence; in general, $\lambda x.ex \neq e$. But η equivalence *does* hold if e has arity greater than 0—in other words, if e is sure to evaluate to a closure—so knowing the arity of an expression can unlock optimizations 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 limiting η 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 η 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, all closures are computed in advance when bound; `loop2 5` evaluates to a closure but `loop1 5` loops forever. So both eager and lazy functional languages can have the same essential problem with restricted η equivalence, which can block optimizations.

- Fast calls for higher-order functions are a problem in GHC and OCaml today. Consider this higher-order function:

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

  The call to `f` in the body of `zipWith` is an "unknown call" [Marlow and Peyton Jones 2004] where the compiler knows nothing about the arity of `f`. 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.

2.2 The Key Idea: A New Function Arrow

Our key idea is to add a new function type $(\sigma \rightsquigarrow \tau)$, introduced with $\widetilde{\lambda} x.e$, alongside the existing one $(\sigma \to \tau)$. With this idea, we can express that `zipWith`'s first argument `f` is a function that takes precisely two arguments with the following type:

```
f :: a ⇝ b ⇝ c
```

Now the call in `zipWith`'s body can be fast, passing two arguments in registers with no runtime arity checks. This new type can be added locally, when compiling `zipWith`, without looking at its call sites (see Section 2.3).

The new type has the following intuitions:

- Terms of type $(\sigma \rightsquigarrow \tau)$ enjoy unconditional η 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 η equivalence holds unconditionally; it is nonsensical to have a program of type $(\sigma \rightsquigarrow \tau)$.
- This η equivalence also preserves *work equivalence*; that is, η expansion does not change the number of reduction steps in a program run. For example, consider these two programs

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

  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 $\widetilde{\lambda}$, 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 intentionally *treated in a call-by-name fashion*, a point we will return to.
- Values of type $(\sigma \rightsquigarrow \tau)$, such as f_1 or f_2, are still fully first-class: they 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. This differs from an ordinary function $g : \tau_1 \to \ldots \tau_n \to \rho$ in a call-by-value language because there is no guarantee that g is bound to a closure which takes n arguments before performing work. And it is even *further* different from an ordinary function in a call-by-need language because g itself might be bound to an unevaluated thunk.
- 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 nonsense of our claim that types statically 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 let type variable be instantiated with an unboxed, or unlifted type [Eisenberg and Peyton Jones 2017]. This restriction is enforced by our type system and our new function arrow fits neatly into this framework.

2.3 Adding Higher-Arity Functions to an Intermediate Language

One of the major advantages to our approach to function arity is that it is non-disruptive to orthogonal concerns of an intermediate language. Rather than a monolithic solution like [Bolingbroke and Peyton Jones 2009], requiring significant investment by changing the entire language, we show here how to seamlessly add extensional higher-arity functions to a λ-based calculus. Worded more formally, our solution to higher-arity functions is a *conservative extension* of an existing base language, meaning that the semantics and implementation of the original features in the base language are unchanged. In practical terms, being a conservative extension is important in the context of a compiler because it means that its other orthogonal jobs, like translating the full source language into the smaller core intermediate representation, do not need to be changed. And optimizations that were performed on the base intermediate language—while they may now make use of arity information provided by types if it is advantageous to do so—are still correct to apply directly as-is in the extended calculus.

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, do higher-arity functions fit into the compiler pipeline? Since there is no need to modify the front-end of the compiler (the translation from the source language to the intermediate language), the program can be first translated without generating any $(\rightsquigarrow)$ functions, pessimistically assuming arity 0. The arity analysis that is already being performed can then be formally captured in the types of functions by replacing $(\to)$ with $(\rightsquigarrow)$. But just arbitrarily changing types is not sound! So how can the optimizer improve function arity when it is entwined with types?

The 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], and is especially useful for type-changing transformations. The general technique is amenable to several optimizations, most notably [Peyton Jones and Launchbury 1991]. Here, we use worker/wrapper to improve the arity of types. For example, we can split the standard Haskell `zipWith` function into a worker and wrapper like so (where `\x↝e` means $\widetilde{\lambda} x.e$)

```
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 = wzipWith (\a ↝ \b ↝ f a b)
```

The worker `wzipWith` is an arity 3 function, and its first argument is an arity 2 function. That way, the call to `f` in the body of `wzipWith` can be a fast binary call, even though the definition of `f` is unknown. The purpose of the wrapper is to serve as an interface change, which can update individual call sites by inlining. For example, we can make the following simplifications in a call to `zipWith`, where the boolean function has also been split into the arity 0 wrapper `(||) :: Bool -> Bool -> Bool` and arity 2 worker `|~| :: Bool ↝ Bool ↝ Bool`:

```
    zipWith  (||)                     {- inline -}
==> wzipWith (\a ↝ \b ↝ (||)  a b)    {- inline -}
==> wzipWith (\a ↝ \b ↝ (|~|) a b)    {- eta -}
==> wzipWith (|~|)
```

It may not look as if much has happened, but the new code is much better: the higher-order call to `(|~|)` in the `wzipWith` loop is now a fast call to a known-arity function.

3 Extending an Intermediate Language With Higher-Arity Curried Functions

Since the choice of the beginning base calculus is rather arbitrary, we choose a modest starting point: the call-by-value system F, referred to here as F_v. We make this choice only for illustrative purposes due to the fact that it is simple to state formally, but still serves as a common core for strict functional languages which can speak about important issues such as polymorphism. However, the exact same extension can be applied to a lazy λ-calculus—serving as an intermediate representation for Haskell programs—as well, which we discuss later in Section 6.

3.1 Syntax

We give the syntax of $\widetilde{F}_v$—a simple language based on a conventional system F_v for expressing higher-arity functions—in Fig. 1. Expressions in $\widetilde{F}_v$ include the usual λ-abstractions and applications (of the forms $\lambda x{:}\sigma.e$ and $e\ u$) for functions of

Types

$r, s, t \in \mathit{Kind}$	$::= \mathsf{V}$	Source (CBV)
	$\mid \mathsf{E}$	Extensional (CBN)
$a, b, c \in \mathit{TyVar}$		
$\tau, \sigma, \rho \in \mathit{Type}$	$::= a$	Type variable
	$\mid \sigma \to \tau$	CBV function
	$\mid \forall a.\tau$	CBV polymorphism
	$\mid \sigma \rightsquigarrow \tau$	CBN function
	$\mid \widetilde{\forall} a.\tau$	CBN polymorphism

Expressions

$x, y, z \in \mathit{Var}$		
$e, u, v \in \mathit{Expr}$	$::= x$	Variable
	$\mid e\ g$	Application
	$\mid \lambda \mathbf{x}.e$	CBV abstraction
	$\mid \widetilde{\lambda} \mathbf{x}.e$	CBN abstraction
	$\mid \mathit{error}\ \tau$	Erroneous result
$g \in \mathit{Arg}$	$::= u \mid \sigma$	
$\mathbf{x}, \mathbf{a} \in \mathit{BoundVar}$	$::= x{:}\tau$	Term binding
	$\mid a$	Type binding

Shorthand

$$\boxed{\lambda^r \mathbf{x}.e} \qquad \boxed{\sigma \xrightarrow{r} \tau} \qquad \boxed{\forall^r a.\tau}$$

$$\lambda^{\mathsf{V}} \mathbf{x}.e = \lambda \mathbf{x}.e \qquad \sigma \xrightarrow{\mathsf{V}} \tau = \sigma \to \tau \qquad \forall^{\mathsf{V}} a.\tau = \forall a.\tau$$

$$\lambda^{\mathsf{E}} \mathbf{x}.e = \widetilde{\lambda} \mathbf{x}.e \qquad \sigma \xrightarrow{\mathsf{E}} \tau = \sigma \rightsquigarrow \tau \qquad \forall^{\mathsf{E}} a.\tau = \widetilde{\forall} a.\tau$$

Figure 1. Syntax of $\widetilde{F}_v$: call-by-value system F_v extended with call-by-name λ-abstractions.

some type $\sigma \to \tau$. Expressions also allow for polymorphism with the analogous syntax of the form $\lambda a.e$ for abstracting a type variable a in the polymorphic type $\forall a.\tau$, and $e\ \sigma$ for specializing the polymorphic term e with the specific type σ. We also include the expression *error* τ to allow for the possibility of run-time errors when a value of type τ is expected. Note that this addition is not fundamental, it just shows the robustness of the approach.

Our extension with higher-arity functions is highlighted in the figure, and consists of one new form of expression and two new types. In types, we introduce a new form of function, $\sigma \rightsquigarrow \tau$. The purpose of this new function type is that (unlike $\to$ arrows) multiple $\rightsquigarrow$ arrows *chain together* into a single, higher-arity abstraction. Intuitively, this chaining could be viewed in terms of multiple-argument functions (similar to [Bolingbroke and Peyton Jones 2009]) like so

$$\tau \rightsquigarrow \sigma \rightsquigarrow \rho \approx (\tau, \sigma) \rightsquigarrow \rho$$

however we do not require that all the arguments of a function be grouped together, allowing for the usual currying and partial application even for higher-arity functions in $\widetilde{F}_v$.

We also introduce a new form of polymorphism, $\widetilde{\forall} a.\tau$ that similarly chains together along with $\rightsquigarrow$. Incrementally chaining polymorphism with function arguments is another advantage of our approach over [Bolingbroke and Peyton Jones 2009]. In order to view them both as a polymorphic multi-argument function, we would have to *combine* polymorphic abstraction and functions into a single monolithic type like

$$\widetilde{\forall} a.\tau \rightsquigarrow \sigma \approx (a, \tau) \rightsquigarrow \sigma$$

with the understanding that in $(a, \tau) \rightsquigarrow \sigma$, the type variable a introduced on the left-hand side of the arrow is in scope for *both* τ and σ. Such monolithic types immediately become more complex because $\widetilde{\forall}$s and $\rightsquigarrow$s could be nested in any way, requiring a notion of telescoping in the input of a function that reaches into its output. For example, $\widetilde{\forall} a.\tau \rightsquigarrow \widetilde{\forall} b.\sigma \rightsquigarrow \rho$ chains together as the multi-argument function $(a, \tau, b, \sigma) \rightsquigarrow \rho$, where a is in scope for τ, σ, and ρ, but b is only in scope for σ and ρ. Our approach keeps higher-arity functions and polymorphism separate from one another, completely avoiding the need for telescoping.

To go along with the two new forms of types, we also have a new form of abstraction: $\widetilde{\lambda} x{:}\sigma.e$ for introducing a function of type $\sigma \rightsquigarrow \tau$ and $\widetilde{\lambda} a.e$ for introducing polymorphism of type $\widetilde{\forall} a.\tau$ (where e is of type τ). The two new types share the same syntax for application as in the base system F_v. The primary reason to distinguish λ from $\widetilde{\lambda}$ is for the purpose of determining the types of annotated terms: without the distinction, we don't know whether $\lambda x{:}a.x$ should be of type $a \to a$ or $a \rightsquigarrow a$ absent additional context.

Distinguishing the base language features from the extension is an essential ingredient in our approach: it allows us to preserve the semantics for the original V-subset of the language, which remains untouched while we add the semantics for new E-features. And it is important to know which of the two we are dealing with (the original base language or the extension) because the two semantics need not (and in this case *will not*) be the same. Having the distinction lets us add an entirely new evaluation strategy as a conservative extension to a seemingly incompatible base calculus. To achieve this goal, we just need to distinguish between the new types introduced by the extension and the original types of F_v. This is done by classifying types as either V (for the base F_v types) or E (for the extensional types being added on top), which are two different basic kinds of types. In ordinary system F_v, kinds are not necessary, since every type has the same kind (sometimes written $\star$). Classifying types as just V or E can be seen as a rather rudimentary kind system. But this style of classification makes it possible to extend the kind system with more advanced features in a more full-fledged system.

3.2 Type System

We present the type system for $\widetilde{F}_v$ in Fig. 2. Again, the extension to the base language is highlighted in the figure. In

Typing environments

$$\Gamma ::= x_1 : \tau_1 \ldots x_n : \tau_n$$

Classifying types $\boxed{\tau : r}$

$$a : \mathsf{V} \qquad \sigma \to \tau : \mathsf{V} \qquad \forall a.\tau : \mathsf{V} \qquad \sigma \rightsquigarrow \tau : \mathsf{E} \qquad \widetilde{\forall} a.\tau : \mathsf{E}$$

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

$$\dfrac{}{\Gamma, x : \tau \vdash x : \tau}\ \mathit{Var} \qquad \dfrac{}{\Gamma \vdash \mathit{error}\ \tau : \tau}\ \mathit{Error}$$

$$\dfrac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x{:}\sigma.e : \sigma \to \tau}\ {\to}I \qquad \dfrac{\Gamma \vdash e : \sigma \to \tau \quad \Gamma \vdash u : \sigma}{\Gamma \vdash e\ u : \tau}\ {\to}E$$

$$\dfrac{\Gamma \vdash e : \tau \quad a \notin FV(\Gamma)}{\Gamma \vdash \lambda a.e : \forall a.\tau}\ \forall I \qquad \dfrac{\Gamma \vdash e : \forall a.\tau \quad \sigma : \mathsf{V}}{\Gamma \vdash e\ \sigma : \tau[\sigma/a]}\ \forall E$$

$$\dfrac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \widetilde{\lambda} x{:}\sigma.e : \sigma \rightsquigarrow \tau}\ {\rightsquigarrow}I \qquad \dfrac{\Gamma \vdash e : \sigma \rightsquigarrow \tau \quad \Gamma \vdash u : \sigma}{\Gamma \vdash e\ u : \tau}\ {\rightsquigarrow}E$$

$$\dfrac{\Gamma \vdash e : \tau \quad a \notin FV(\Gamma)}{\Gamma \vdash \widetilde{\lambda} a.e : \widetilde{\forall} a.\tau}\ \widetilde{\forall} I \qquad \dfrac{\Gamma \vdash e : \widetilde{\forall} a.\tau \quad \sigma : \mathsf{V}}{\Gamma \vdash e\ \sigma : \tau[\sigma/a]}\ \widetilde{\forall} E$$

Figure 2. Type system of $\widetilde{F}_v$.

the base system F_v language, there is only one kind of type (denoted here by V). However, since our extension has added a new kind of types (denoted by E), checking whether $\tau : r$ holds points out the dichotomy between the two. And this distinction can be done by just reading the head connective of the type. As such, the base language features (type variables a, arrows $\to$, and quantifiers $\forall$) are all assigned the kind V. In contrast, the extension (arrows $\rightsquigarrow$ and quantifiers $\widetilde{\forall}$) are assigned the kind E.

Also note that, for now, we choose not to add E type variables in addition to the existing V ones inherited from the base system F, since this extension is not necessary for capturing arity in types. We will further discuss the ramifications of this choice later in Section 7.

Next, Fig. 2 shows the typing rules. The rules for the base language are conventional for system F_v, showing how to type check variables as well as abstractions and applications for both functions and polymorphism. The highlighted typing rules are essentially the same as the ones for system F_v, except that they refer to the new forms of arrow and quantifier.

3.3 Equational Theory

In order to reason about programs in terms of high-level syntactic rewriting, we give an equational theory for $\widetilde{F}_v$ with the axioms presented in Fig. 3. Notice that the first two axioms, β_{V} and $\forall$ are the usual rules for resolving applications of function and type abstraction in the call-by-value system F_v. In particular, the β_{V} rule is limited to substituting only arguments which are *values* (syntactically, a variable or a λ-abstraction). That's because in call-by-value, arguments

$$V \in \mathit{Value} ::= x \mid \lambda \mathbf{x}.e$$

$$\begin{array}{rlr}
(\beta_{\mathsf{V}}) & (\lambda^r x{:}\sigma.e)\ V = e[V/x] & \\
(\forall) & (\lambda^r a.e)\ \sigma = e[\sigma/a] & \\
(\beta_{\mathsf{E}}) & (\lambda^r x{:}\sigma.e)\ u = e[u/x] & (\sigma : \mathsf{E}) \\
(\eta_{\rightsquigarrow}) & \widetilde{\lambda} x{:}\sigma.(e\ x) = e : \sigma \rightsquigarrow \tau & (x \notin FV(e)) \\
(\eta_{\widetilde{\forall}}) & \widetilde{\lambda} a.(e\ a) = e : \widetilde{\forall} a.\tau & (a \notin FV(e))
\end{array}$$

Figure 3. Equational theory of $\widetilde{\mathrm{F}}_v$.

are evaluated first before the call is resolved, so that only values may be passed to the function. For example,

$$(\lambda z{:}a.\lambda x{:}a.x)\ (\mathit{error}\ a) \neq \lambda x{:}a.x$$

is not a valid equation because the two sides give vastly different results: evaluating the left-hand side causes an error at run-time when the argument *error a* is encountered, whereas the right-hand side is already a value.

Besides the usual call-by-value system F_v axioms, we have additional axioms to handle higher-arity functions in $\widetilde{\mathrm{F}}_v$. First of all, we have the full version of the β axiom, denoted here by β_{E}, which can substitute expressions which are not values. However, just allowing such a rule to apply whenever would be dangerous—as illustrated by the inequality above. How can we reconcile the difference between call-by-name and call-by-value functions coexisting in the same language? We can use the type of the argument to determine when the full call-by-name β axiom is safe. If the parameter of a function has type $\sigma : \mathsf{E}$, then any expression may be substituted since the argument should be handled in a call-by-name way. Otherwise, if the function's parameter has type $\sigma : \mathsf{V}$, then this is a call-by-value application, and only values may be substituted. This side condition allows us to safely mix both β axioms in the same language.

The primary motivation, though, is the addition of the full η axioms $\eta_{\rightsquigarrow}$ and $\eta_{\widetilde{\forall}}$ in Fig. 3. These are the equations that allow us to use types to statically expand any expression with the appropriate number of $\widetilde{\lambda}$s to match its arity. As usual, the η axioms only apply to expressions of the correct type (for example, expanding 5 to $\lambda x.(5\ x)$ is nonsensical).

We really do need to use a call-by-name understanding for expressions of type $\sigma \rightsquigarrow \tau$ and $\widetilde{\forall} a.\tau$ in order to support these η axioms. For example, applying the analogous η axiom to an erroneous expression of type $a \rightarrow b$ is not an equality

$$\mathit{error}\ (a \rightarrow b) \neq \lambda x{:}a.(\mathit{error}\ (a \rightarrow b)\ x)$$

because the left-hand side evaluates to an error while the right-hand side is a value. The safe version of the η axiom in the call-by-value λ-calculus must be restricted like so

$$(\eta_{\rightarrow}) \qquad \lambda x{:}\sigma.(V\ x) = V : \sigma \rightarrow \tau \qquad (x \notin FV(V))$$

thereby avoiding the above counter example. However, even admitting this restricted η axiom does not solve the problem of explicating arity by expanding out every λ allowed by a type. For example, if we have some unknown function $z : a \rightarrow a \rightarrow a$, then the above $\eta_{\rightarrow}$ axiom gets stuck after the first expansion:

$$z =_{\eta_{\rightarrow}} \lambda x{:}a.(z\ x) \neq_{\eta_{\rightarrow}} \lambda x{:}a.\lambda y{:}a.(z\ x\ y)$$

This is because the partial application $z\ x$ might already result in an error, so $\eta_{\rightarrow}$ no longer applies.

In order to fully η-expand a function of *any* higher arity, we must ensure that even erroneous expressions of type $\sigma \rightsquigarrow \tau$ can be safely η-expanded without changing the result of a program. The consequence of this constraint is that the only context in which we are ever allowed to evaluate an expression of type $\sigma \rightsquigarrow \tau$ is a calling context that already has an argument ready. This restriction on evaluation follows the call-by-name strategy, making it safe to η-expand any number of $\widetilde{\lambda}$s like for the expression $e : \widetilde{\forall} a.a \rightsquigarrow a \rightsquigarrow a$

$$\begin{aligned}
e &=_{\eta_{\widetilde{\forall}}} \widetilde{\lambda} a.(e\ a) \\
&=_{\eta_{\rightsquigarrow}} \widetilde{\lambda} a.\widetilde{\lambda} x{:}a.(e\ a\ x) \\
&=_{\eta_{\rightsquigarrow}} \widetilde{\lambda} a.\widetilde{\lambda} x{:}a.\widetilde{\lambda} y{:}a.(e\ a\ x\ y)
\end{aligned}$$

Note that as a side constraint necessary for satisfying the full η axiom: expressions of type $\tau : \mathsf{E}$ are not programs and cannot be evaluated. Allowing such expressions to be programs would let us observe the difference between $\mathit{error}\ (a \rightsquigarrow b)$ and $\widetilde{\lambda} x{:}a.(\mathit{error}\ (a \rightsquigarrow b)\ x)$ which, again, must be equivalent.

4 Adding Higher-Arity Calls to an Abstract Machine

Now, lets consider how to run programs, with the goal of performing higher-arity function calls to a stock implementation [Peyton Jones 1992]. As is the running theme, we begin with a standard call-by-value abstract machine for system F_v and only add an extension to function applications and calls which is capable of passing multiple parameters at once. The machine in question is specified by the operational semantics in Fig. 4, and eagerly evaluates all function arguments in a right-to-left order. Evaluated function parameters can be either types (for specializing polymorphism) or closures, and the stack consists of a pending function waiting for its argument ($eB \circ S$) or a chain of parameters which are annotated with the calling convention of their function ($P\ r\ P'\ s\ P''\ t\ S$). The purpose of keeping track of the calling convention of function arguments is to remember the difference from the base system F_v (for which r will always be V) and the new form of extensional functions during β-reduction. Our only significant extension to this base semantics is found in the β^* rule. When limited to just expressions from the base call-by-value system F_v, the β^* performs the usual single-argument

Runtime state

$$
\begin{array}{rcll}
W \in \mathit{WHNF} & ::= & \lambda^r \mathbf{x}.e & \text{Abstraction value} \\
P \in \mathit{Param} & ::= & WB & \text{Closure parameter} \\
 & \mid & \sigma & \text{Type parameter} \\
S \in \mathit{Stack} & ::= & \varepsilon & \text{Empty stack} \\
 & \mid & eB \circ S & \text{Function application} \\
 & \mid & P\ r\ S & \text{Application chain} \\
B \in \mathit{Env} & ::= & \varepsilon & \text{Empty environment} \\
 & \mid & [WB/x]B & \text{Closure binding} \\
 & \mid & [\sigma/a]B & \text{Type binding}
\end{array}
$$

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

$$
\begin{array}{lll}
(\beta^*) & \langle \lambda^r \mathbf{x}.e \mid B \mid P\ r\ S\rangle \mapsto \langle e' \mid B' \mid S'\rangle & \\
 & \qquad (\mathit{app}^*(e, [P/\mathbf{x}]B, S) = (e', B', S')) & \\
(\mathit{var}) & \langle x \mid B \mid S\rangle \mapsto \langle W \mid B' \mid S\rangle & ([WB'/x] \in B) \\
(\mathit{arg}) & \langle e\ u \mid B \mid S\rangle \mapsto \langle u \mid B \mid (eB) \circ S\rangle & \\
(\mathit{fun}) & \langle W \mid B \mid (eB') \circ S\rangle \mapsto \langle e \mid B' \mid (WB)\ t\ S\rangle & (e : \tau : t) \\
(\mathit{spec}) & \langle e\ \sigma \mid B \mid S\rangle \mapsto \langle e \mid B \mid \sigma\ t\ S\rangle & (e : \tau : t)
\end{array}
$$

Multi-arity application $\boxed{\mathit{app}^*(e, B, S) = (e', B', S')}$

$$
\begin{array}{ll}
\mathit{app}^*(\widetilde{\lambda}\mathbf{x}.e, B, P \mathrel{\mathsf{E}} S) = \mathit{app}^*(e, [P/\mathbf{x}]B, S) & \\
\mathit{app}^*(e, B, S) = (e, B, S) & (e \neq \widetilde{\lambda}\mathbf{x}.e' \text{ and } S \neq P \mathrel{\mathsf{E}} S')
\end{array}
$$

Figure 4. Abstract machine for $\widetilde{\mathrm{F}}_v^-$

function application like so

$$\langle \lambda \mathbf{x}.e \mid B \mid P \mathrel{\mathsf{V}} S\rangle \mapsto \langle e \mid [P/\mathbf{x}]B \mid S\rangle$$

However, β^* is also capable of passing multiple arguments at once during a single step, as defined by $\mathit{app}^*(e, B, S)$. The idea is that, once the first argument is being passed to a function closure (of any r), then *all* the following arguments for the next E-function calls are passed, too, at the same time. For example, a ternary function call beginning with a V closure and followed by two E calls is performed by the following single β^* step:

$$
\begin{aligned}
&\langle \lambda\mathbf{x}.\widetilde{\lambda}\mathbf{y}.\widetilde{\lambda}\mathbf{z}.(f\ z\ y\ x) \mid B \mid P \mathrel{\mathsf{V}} P' \mathrel{\mathsf{E}} P'' \mathrel{\mathsf{E}} \varepsilon\rangle \\
&\mapsto \langle f\ z\ y\ x \mid [P''/\mathbf{z}][P'/\mathbf{y}][P/\mathbf{x}]B \mid \varepsilon\rangle
\end{aligned}
$$

All three parameters are substituted in exactly one step. In this sense, the semantics in Fig. 4 explicitly models how to add higher-arity function calls to a standard abstract machine. But if we want to run programs of $\widetilde{\mathrm{F}}_v$ on this stock call-by-value machine, we will need to compile $\widetilde{\mathrm{F}}_v$ programs first to implement the semantics of $\widetilde{\mathrm{F}}_v$ that was given in Fig. 3. In particular, in order to reflect $\widetilde{\mathrm{F}}_v$ semantics with this stock call-by-value machine, the compilation we describe next in Section 5 will have to resolve the following two issues.

4.1 Differences in Evaluation Order

If we just run an arbitrary $\widetilde{\mathrm{F}}_v$ expression, then the result might be wrong! That's because the machine in Fig. 4 uses just a call-by-value evaluation order, and has no idea that some expressions should be treated in a call-by-name way as specified by the axioms in Fig. 3. For example, we have

$$(\lambda x{:}\sigma{\rightsquigarrow}\tau.I)\ (\mathit{error}\ \sigma{\rightsquigarrow}\tau) =_{\beta_{\mathsf{E}}} I$$

where $I = \lambda a.\widetilde{\lambda}x{:}a.x$ is the identity function, but instead at run-time we get an error (i.e. a stuck state):

$$
\begin{aligned}
&\langle (\lambda x{:}\sigma{\rightsquigarrow}\tau.I)\ (\mathit{error}\ \sigma{\rightsquigarrow}\tau) \mid \varepsilon \mid \varepsilon\rangle \\
&\mapsto \langle \mathit{error}\ \sigma{\rightsquigarrow}\tau \mid \varepsilon \mid (\lambda x{:}\sigma{\rightsquigarrow}\tau.I)\varepsilon \circ \varepsilon\rangle \\
&\not\mapsto
\end{aligned}
$$

In this case, the equational theory gives the intended result and the machine is wrong. But we don't want to modify the existing machine, since it is already perfectly serviceable for the base language.

Instead, we should elaborate the input programs themselves so that they give the correct result even on a purely call-by-value machine. Thankfully we don't need to do anything as drastic as continuation-passing style in order to make the evaluation order explicit. Rather, η expansion comes in to solve the problem for us. As a pleasant side benefit, just η-expanding the call-by-name expression to make the arity explicit in the syntax also makes the evaluation order explicit! So there is no need to change the implementation of the machine; strategically η expanding the program gives the correct result as in:

$$
\begin{aligned}
&\langle (\lambda x{:}\sigma{\rightsquigarrow}\tau.I)\ (\widetilde{\lambda}y{:}\sigma.\mathit{error}\ \sigma{\rightsquigarrow}\tau\ y) \mid \varepsilon \mid \varepsilon\rangle \\
&\mapsto \langle \widetilde{\lambda}y{:}\sigma.\mathit{error}\ \sigma{\rightsquigarrow}\tau\ y \mid \varepsilon \mid (\lambda x{:}\sigma{\rightsquigarrow}\tau.I)\varepsilon \circ \varepsilon\rangle \\
&\mapsto \langle \lambda x{:}\sigma{\rightsquigarrow}\tau.I \mid \varepsilon \mid (\widetilde{\lambda}y{:}\sigma.\mathit{error}\ \sigma{\rightsquigarrow}\tau\ y)\varepsilon \mathrel{\mathsf{V}} \varepsilon\rangle \\
&\mapsto \langle I \mid [(\widetilde{\lambda}y{:}\sigma.\mathit{error}\ \sigma{\rightsquigarrow}\tau\ y)\varepsilon/x]\varepsilon \mid \varepsilon\rangle
\end{aligned}
$$

4.2 Run-Time Arity Mismatch

There is a new kind of error that might occur at run-time: it may be the case that the machine gets stuck if the arity of a function and the call stack do not match. For example, there could be too many E-parameters and not enough $\widetilde{\lambda}$s like so

$$\langle \lambda x{:}\sigma{\rightsquigarrow}\tau.\widetilde{\lambda}y{:}\rho.x \mid B \mid P_1 \mathrel{\mathsf{V}} P_2 \mathrel{\mathsf{E}} P_3 \mathrel{\mathsf{E}} \varepsilon\rangle \not\mapsto$$

This is a stuck configuration because the multiple-argument application operation is not defined in this case, since there are not enough $\widetilde{\lambda}$s to match the last E-parameter remaining on the stack:

$$
\begin{aligned}
&\mathit{app}^*((\widetilde{\lambda}y{:}\rho.x), [P_1/x]B, (P_2 \mathrel{\mathsf{E}} P_3 \mathrel{\mathsf{E}} \varepsilon)) \\
&= \mathit{app}^*(x, [P_2/y][P_1/x]B, (P_3 \mathrel{\mathsf{E}} \varepsilon))
\end{aligned}
$$

In this case, too, applying the missing η-expansion to supply all the $\widetilde{\lambda}$s allowed by the type fixes the dynamic arity

Type-driven $\boxed{\mathcal{E}\llbracket e \rrbracket \overline{g} : \tau}$ *(invariant, $e\ \overline{g} : \tau$)*

$$
\begin{array}{lll}
(\widetilde{\lambda}) & \mathcal{E}\llbracket \widetilde{\lambda}\mathbf{x}.e \rrbracket \varepsilon : \tau = \widetilde{\lambda}\mathbf{x}.\mathcal{E}\llbracket e \rrbracket \varepsilon : \sigma & (e : \sigma) \\
(\eta_{\rightsquigarrow}) & \mathcal{E}\llbracket e \rrbracket \overline{g} : \sigma \rightsquigarrow \tau = \widetilde{\lambda}x{:}\sigma.\mathcal{E}\llbracket e \rrbracket \overline{g}, x : \tau & (e \neq \widetilde{\lambda}x{:}\sigma.e') \\
(\eta_{\widetilde{\forall}}) & \mathcal{E}\llbracket e \rrbracket \overline{g} : \widetilde{\forall}a.\tau = \widetilde{\lambda}a.\mathcal{E}\llbracket e \rrbracket \overline{g}, a : \tau & (e \neq \widetilde{\lambda}a.e') \\
(\mathsf{V}) & \mathcal{E}\llbracket e \rrbracket \overline{g} : \tau = \mathcal{C}\llbracket e \rrbracket \overline{g} & (\tau : \mathsf{V})
\end{array}
$$

Type-generic $\boxed{\mathcal{C}\llbracket e \rrbracket \overline{g}}$

$$
\begin{array}{lll}
(var) & \mathcal{C}\llbracket x \rrbracket \overline{g} = x\ \overline{g} & \\
(err) & \mathcal{C}\llbracket error\ \tau \rrbracket \overline{g} = error\ \tau\ \overline{g} & \\
(\lambda) & \mathcal{C}\llbracket \lambda^r\mathbf{x}.e \rrbracket \overline{g} = (\lambda^r\mathbf{x}.\mathcal{E}\llbracket e \rrbracket \varepsilon : \tau)\ \overline{g} & (e : \tau) \\
(app) & \mathcal{C}\llbracket e\ u \rrbracket \overline{g} = \mathcal{C}\llbracket e \rrbracket (\mathcal{E}\llbracket u \rrbracket \varepsilon : \sigma), \overline{g} & (u : \sigma) \\
(spec) & \mathcal{C}\llbracket e\ \sigma \rrbracket \overline{g} = \mathcal{C}\llbracket e \rrbracket \sigma, \overline{g} &
\end{array}
$$

Figure 5. Pre-processing transformation from $\widetilde{\mathrm{F}}_v$ to $\widetilde{\mathrm{F}}_v^-$

Restricted η-long typing rules $\boxed{\Gamma \vdash_\eta e : \tau}$

$$
\frac{}{\Gamma, x : \tau \vdash_\eta x : \tau}\ \eta Var
\qquad
\frac{}{\Gamma \vdash_\eta error\ \tau : \tau}\ \eta Error
$$

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

$$
\frac{\Gamma \vdash_\eta e : \sigma \xrightarrow{r} \tau \quad \Gamma \vdash_\eta^{\mathsf{E}} u : \sigma}{\Gamma \vdash_\eta e\ u : \tau}\ \eta{\to}E
$$

$$
\frac{\Gamma \vdash_\eta^{\mathsf{E}} e : \tau \quad a \notin FV(\Gamma)}{\Gamma \vdash_\eta \lambda^r a.e : \forall^r a.\tau}\ \eta\forall I
\qquad
\frac{\Gamma \vdash e : \forall a.\tau \quad \sigma : \mathsf{V}}{\Gamma \vdash e\ \sigma : \tau[\sigma/a]}\ \eta\forall E
$$

Checking η-expansion $\boxed{\Theta; \Gamma \vdash_\eta^{\mathsf{E}} e : \tau}$

$$
\frac{\Gamma, x : \tau \vdash_\eta^{\mathsf{E}} e : \sigma}{\Gamma \vdash_\eta^{\mathsf{E}} \widetilde{\lambda}x{:}\tau.e : \tau \rightsquigarrow \sigma}\ \eta{\rightsquigarrow}I
\qquad
\frac{\Gamma \vdash_\eta^{\mathsf{E}} e : \tau \quad a \notin FV(\Gamma)}{\Gamma \vdash_\eta^{\mathsf{E}} \widetilde{\lambda}a{:}k.e : \widetilde{\forall}a{:}k.\tau}\ \eta\widetilde{\forall}
$$

$$
\frac{\tau : \mathsf{V} \quad \Gamma \vdash_\eta e : \tau}{\Gamma \vdash_\eta^{\mathsf{E}} e : \tau}\ \mathsf{V}
$$

Figure 6. Type system of $\widetilde{\mathrm{F}}_v^-$.

mismatch problem:

$$
\begin{aligned}
&\langle \lambda x{:}\sigma{\rightsquigarrow}\tau.\widetilde{\lambda}y{:}\rho\widetilde{\lambda}z{:}\sigma.(x\ z) \mid B \mid P_1\ \mathsf{V}\ P_2\ \mathsf{E}\ P_3\ \mathsf{E}\ \varepsilon \rangle \\
&\mapsto \langle x\ z \mid [P_3/z][P_2/y][P_1/x]B \mid \varepsilon \rangle
\end{aligned}
$$

The opposite case is when there are too few E-parameters on the stack to fill all of the $\widetilde{\lambda}$s. For example, we could have

$$
\begin{aligned}
&\langle (\lambda x{:}a{\rightsquigarrow}a.\widetilde{\lambda}y{:}a.x\ y)\ I_a \mid \varepsilon \mid \varepsilon \rangle \\
&\mapsto^* \langle \lambda x{:}a{\rightsquigarrow}a.\widetilde{\lambda}y{:}a.x\ y \mid \varepsilon \mid I_a \varepsilon\ \mathsf{V}\ \varepsilon \rangle \\
&\not\mapsto
\end{aligned}
$$

where $I_a = \widetilde{\lambda}x{:}a.x : a \rightsquigarrow a$. But notice that the type of the input program was $a \rightsquigarrow a$, which is call-by-name (E), not call-by-value (V)! As pointed out previously in the context of the equational theory, validating the η axioms requires us to restrict programs to expressions of V-types. This condition already rejects the term $(\lambda x{:}a{\rightsquigarrow}a.\widetilde{\lambda}y{:}a.x\ y)\ I_a$ as a valid input program because it has the type $a \rightsquigarrow a$. As it turns out, this same condition on the final type of result that well-typed programs are allowed to produce also eliminates the possibility that there are not enough parameters on the stack to complete a higher-arity function call.

5 Compiling Extensional Functions

We just saw some examples of how strategic η expansion can let us run extensional, higher-arity functions on a stock call-by-value machine. But is there a general procedure for compiling any $\widetilde{\mathrm{F}}_v$ expression? Here, we demonstrate a transformation for inserting all the missing $\widetilde{\lambda}$s allowed by the types of sub-expressions via $\eta_{\rightsquigarrow}$ and $\eta_{\widetilde{\forall}}$ expansion, thereby making the arity implied by types explicit in the syntax of the program. This transformation is given in Fig. 5 and consists of two parts. $\mathcal{E}\llbracket e \rrbracket \varepsilon : \tau$ uses the type τ of e to insert any $\widetilde{\lambda}$ missing from e (in the $\eta_{\rightsquigarrow}$ and $\eta_{\widetilde{\lambda}}$ steps). $\mathcal{C}\llbracket e \rrbracket \varepsilon$ handles all the other cases by descending down a chain of applications (in the *app* and *spec* steps). To do so, both transformations are parameterized by a list, $\overline{g}$, of the arguments surrounding the expression. So in their full generality, we have the η-expansion transformation $\mathcal{E}\llbracket e \rrbracket \overline{g} : \tau$ (where $e\ \overline{g} : \tau$) and the helping $\mathcal{C}\llbracket e \rrbracket \overline{g}$.

The purpose of this compilation process is to transform an $\widetilde{\mathrm{F}}_v$ program into an equivalent one that gives the correct answer. The first fact about the transformation is that it just elaborates the original expression by selectively applying η.

Proposition 1 (η Elaboration)**.** *For a typed $\widetilde{\mathrm{F}}_v$ expression e such that $\Gamma \vdash e : \tau$,*

1. *$e =_{\eta_{\rightsquigarrow}\eta_{\widetilde{\forall}}} (\mathcal{E}\llbracket e \rrbracket \varepsilon : \tau)$, and*
2. *$e =_{\eta_{\rightsquigarrow}\eta_{\widetilde{\forall}}} (\mathcal{C}\llbracket e \rrbracket \varepsilon)$.*

The invariants created by the η-elaboration process can be captured in the form of a type system, shown in Fig. 6. This type system is a restriction of Fig. 2 by mandating η-expansion in the arguments to function calls and underneath any λ (either λ or $\widetilde{\lambda}$). We refer to the strict subset of $\widetilde{\mathrm{F}}_v$ expressions which are well-typed according to this restricted system (*i.e.*, expressions e such that $\Gamma \vdash_\eta e : \tau$) as the executable sub-language $\widetilde{\mathrm{F}}_v^-$.

Proposition 2 (Compilation)**.** *For a typed $\widetilde{\mathrm{F}}_v$ expression e such that $\Gamma \vdash e : \tau$,*

1. *$\Gamma \vdash_\eta^{\mathsf{E}} (\mathcal{E}\llbracket e \rrbracket \varepsilon : \tau) : \tau$ is a $\widetilde{\mathrm{F}}_v^-$ expression, and*
2. *$\Gamma \vdash_\eta (\mathcal{C}\llbracket e \rrbracket \varepsilon) : \tau$ is a $\widetilde{\mathrm{F}}_v^-$ expression when $\tau : \mathsf{V}$.*

Runtime State

$$
\begin{array}{rcll}
W \in \mathit{WHNF} & ::= & \lambda^r\mathbf{x}.e & \\
P \in \mathit{Param} & ::= & x \mid \sigma & \\
S \in \mathit{Stack} & ::= & \varepsilon & \text{Empty stack} \\
 & \mid & P\ r\ S & \text{Application Chain} \\
 & \mid & !x; S & \text{Thunk update} \\
H \in \mathit{Heap} & ::= & \varepsilon & \text{Empty heap} \\
 & \mid & x := e; H & \text{Thunk allocation}
\end{array}
$$

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

$$
\begin{array}{rl}
(\beta^*) & \langle \lambda^r\mathbf{x}.e \mid H \mid P\ r\ S\rangle \mapsto \langle e' \mid H \mid S'\rangle \\
 & \qquad (\mathit{app}^*(e[P/\mathbf{x}], S) = (e', S')) \\
(\mathit{push}) & \langle e\ P \mid H \mid S\rangle \mapsto \langle e \mid H \mid P\ t\ S\rangle \qquad (e : \tau : t) \\
(\mathit{alloc}) & \langle \mathbf{let}\, x = u\, \mathbf{in}\, e \mid H \mid S\rangle \mapsto \langle e[y/x] \mid y := u; H \mid S\rangle \\
(\mathit{force}) & \langle x \mid x := e; H \mid S\rangle \mapsto \langle e \mid x := e; H \mid !x; S\rangle \\
(\mathit{memo}) & \langle W \mid x := e; H \mid !x; S\rangle \mapsto \langle W \mid x := W; H \mid S\rangle
\end{array}
$$

Multi-arity application $\boxed{\mathit{app}^*(e, S) = (e', S')}$

$$
\begin{array}{rcll}
\mathit{app}^*(\widetilde{\lambda}\mathbf{x}.e, P \mathsf{E} S) & = & \mathit{app}^*(e[P/\mathbf{x}], S) & \\
\mathit{app}^*(e, S) & = & (e, S) & (e \neq \widetilde{\lambda}\mathbf{x}.e' \text{ and } S \neq P \mathsf{E} S')
\end{array}
$$

Figure 7. A lazy abstract machine extended with higher-arity function calls

We are now equipped to properly run $\widetilde{\mathrm{F}}_v$ expressions by first compiling to the sub-language $\widetilde{\mathrm{F}}_v^-$. Doing so lets us use the same implementation of the base call-by-value system F_v (with support for faster multiple-argument function calls) while still satisfying the mixed call-by-value and call-by-name semantics of $\widetilde{\mathrm{F}}_v$.

Definition 1 (Programs)**.** An expression e is an $\widetilde{\mathrm{F}}_v^-$ *program* if $\varepsilon \vdash_\eta e : \tau$ and $\tau : \mathsf{V}$.

Proposition 3 (Operational soundness)**.** *For any $\widetilde{\mathrm{F}}_v^-$ program e, if $e = V$ (for some value V according to Fig. 3) then $\langle e|\varepsilon|\varepsilon\rangle \mapsto^* \langle W|B|\varepsilon\rangle$ (for some W and B).*

6 Sharing and Lazy Evaluation

So far, we showed how to extend an existing language with higher-arity using extensional functions to get the $\widetilde{\mathrm{F}}_v$ calculus. But the starting point—the call-by-value system F_v—was not important; we needed to begin somewhere for the purpose of illustration. Because we formally distinguish the base language (F_v) in the extended calculus ($\widetilde{\mathrm{F}}_v$) in order to preserve two different semantics, we can choose a base language with any evaluation strategy and apply the same extension. Of particular interest, we could have instead started with an intermediate language that uses lazy evaluation, and achieve higher-arity functions using the same technique. The question then becomes "how do higher-arity, extensional functions interact with sharing and memoization in a call-by-need language like Haskell?"

6.1 A Higher-Arity Lazy Machine

To avoid repetition, we will only discuss the interesting differences that laziness poses when compared with Sections 3 to 5. In order to model laziness and sharing, we can use the standard construct $\mathbf{let}\, x{:}\tau = u\, \mathbf{in}\, e$ which should be read as "first evaluate e and remember $x = u$; only evaluate u if x is needed in e." This understanding of let-bindings is formalized by the call-by-need abstract machine in Fig. 7. In this machine, function parameters cannot be arbitrary expressions; they are pointers into the heap H which are allocated by the *alloc* step. As such, they are only ever evaluated when the variable is referenced (by the *force* step) after which its binding will be updated with its value (by the *memo* step).

Just like with the call-by-value machine of Section 4, this lazy machine requires some pre-processing step for elaborating the evaluation strategy of extensional functions. The elaboration for lazy evaluation is the same sort of η-expansion as the one in Section 5, with a few minor differences. First, since function parameters must be variables, we should take care to name arguments with a **let**. Second, to compile a **let**, we should η-expand the bound expression, and push all the surrounding arguments into its body. These additions are expressed by the following new lines for the transformation (where we use a list of parameters $\overline{P}$ instead of arguments):

$$
\begin{array}{rcll}
\mathcal{C}[\![e\ x]\!]\overline{P} & = & \mathcal{C}[\![e]\!]x, \overline{P} & \\
\mathcal{C}[\![e\ u]\!]\overline{P} & = & \mathcal{C}[\![\mathbf{let}\, x{:}\tau = u\, \mathbf{in}\, e\ x]\!]\overline{P} & (x \neq u : \tau) \\
\mathcal{C}[\![\mathbf{let}\, x{:}\tau = u\, \mathbf{in}\, e]\!]\overline{P} & = & \mathbf{let}\, x{:}\tau = \mathcal{E}[\![u]\!]\varepsilon : \tau\, \mathbf{in}\, \mathcal{C}[\![e]\!]\overline{P} &
\end{array}
$$

6.2 Forcing Thunks

Haskell-like languages have the capability of forcing thunks even when their value is not yet needed using an operation like `seq` to evaluate them to weak-head normal form. As mentioned in Section 2, `seq` is precisely the operation that invalidates η expansion for lazy function closures. This is unacceptable for our approach, since we heavily rely on the η law to make the evaluation strategy explicit at run-time.

Thankfully, the same solution we used to differentiate the evaluation strategy of the base language and the higher-arity function space—that is, distinguishing the two types of programs—also solves the issue of inappropriate forcing due to `seq`. In Section 3, there wasn't even the possibility of a call-by-name type variable, so the type of `seq` : $\forall a.\forall b.a \to b \to b$ prevents it from being applied to higher-arity functions, thereby rescuing their η law. Even if we were to go to a more expressive system with arity polymorphism (discussed next in Section 7), the type variables a and b only range over lazy (call-by-need) types, which rules out extensional (call-by-name) ones. Note that if our lazy base calculus has a polymorphic case expression of the form $\mathbf{case}\, e\, \mathbf{of}\, x \to u$

which forces e, then it too needs to follow the interface of `seq` described above; e cannot have a call-by-name type.

6.3 Work Preservation

Even though both call-by-need and call-by-name are non-strict evaluation strategies, they still have a crucial difference: call-by-need shares the work done to evaluate bindings, whereas call-by-need repeats work every time the binding is used. So we must take care that adding call-by-name to a call-by-need language does not accidentally cause work duplication, as that could have a severe performance penalty.

With our methodology, work is preserved automatically because we distinguish work-sharing bindings (call-by-need) from ones that are intended to be repeated (call-by-name). If we bind a variable of a call-by-need type, then we intend to share its work, so the η law does not apply in general which prevents duplication. But if an expression e has a call-by-name type, then we do not plan to share its work anyway because it must be treated as $\widetilde{\lambda}x.(e\ x)$. This gives an optimizer more freedom to η expand, which is something call-by-need compilers are sensitive about doing because it may duplicate work. Moreover, distinguishing the two kinds of function types allows us to simplify the runtime representation of functions. Call-by-name functions can be a direct pointer to a closure instead of a pointer to a thunk like call-by-need functions, which avoids the need to dynamically check whether the closure has been evaluated yet.

7 Arity Polymorphism

In Section 3, we began with a calculus (system F_v) that had a notion of polymorphism, and added to it higher-arity curried functions. The base language and the extension were distinguished by different kinds of types (written V versus E, respectively). A consequence of this decision meant that there was only polymorphism over the original V types, but not over the new E types. For example, if we just added a pairing constructor $\mathit{Pair} : \forall a.\forall b.a \to b \to (a, b)$, then we could only instantiate Pair with call-by-value (V) types, but not extensional types (E). On the surface, putting higher-arity functions inside a pair seems reasonable. But to quantify over higher-arity types, there are some serious issues surrounding arity polymorphism to consider.

Notice that the polymorphism already available in $\widetilde{\mathrm{F}}_v$ does allow for polymorphism over functions that might have different arity. For example, consider the identity function

$$\mathit{id} : \forall a.a \rightsquigarrow a$$
$$\mathit{id} = \lambda a.\widetilde{\lambda}x{:}a.x$$

Even though a is a V-type variable, it can still be instantiated with call-by-value function types that are called with multiple arguments. For example, id can be applied to unary (like id) and binary function (like const) as follows:

$$\mathit{const} : \forall a.\widetilde{\forall}b.a \rightsquigarrow b \rightsquigarrow a$$
$$\mathit{const} = \lambda a.\widetilde{\lambda}b.\widetilde{\lambda}x{:}a.\widetilde{\lambda}y{:}b.x$$

$$\mathit{id}\ (\forall a.a \rightsquigarrow a)\ \mathit{id} : (\forall a.a \rightsquigarrow a)$$
$$\mathit{id}\ (\forall a.\widetilde{\forall}b.a \rightsquigarrow b \rightsquigarrow a)\ \mathit{const} : \forall a.\widetilde{\forall}b.a \rightsquigarrow b \rightsquigarrow a$$

In this sense, the polymorphism allowed by $\widetilde{\mathrm{F}}_v$ is analogous to [Bolingbroke and Peyton Jones 2009]. But $\widetilde{\mathrm{F}}_v$ is more expressive, because in addition to call-by-value function closures (which might have a higher arity by starting with $\forall$ or $\to$ and continuing with multiple $\widetilde{\forall}$s or $\rightsquigarrow$s) it *also* allows for "naked" higher-arity functions of kind E.

Now suppose that we just added type variables of kind E (we will distinguish the two kinds of type variables by $a : \mathsf{V}$ versus $a : \mathsf{E}$ at their binding site à la system F_ω). What goes wrong? We could now write the following alternative identity and constant functions over E types:

$$\begin{aligned} \mathit{id}_{\mathsf{E}} &: \widetilde{\forall}a{:}\mathsf{E}.a \rightsquigarrow a & &= \widetilde{\lambda}a{:}\mathsf{E}.\widetilde{\lambda}x{:}a.x \\ \mathit{const}_{\mathsf{E}} &: \widetilde{\forall}a{:}\mathsf{E}.\widetilde{\forall}b{:}\mathsf{E}.a \rightsquigarrow b \rightsquigarrow a & &= \widetilde{\lambda}a{:}\mathsf{E}.\widetilde{\lambda}b{:}\mathsf{E}.\widetilde{\lambda}x{:}a.\widetilde{\lambda}y{:}b.x \end{aligned}$$

But remember, in order to be able to evaluate programs with a stock implementation, we need to be able to fully η-expand them! This makes both the evaluation strategy and the arity of each function call fully explicit in the syntax of the program. But how much expansion is necessary? In both of the above cases, it depends on the instantiation of a, which can include any number of additional $\rightsquigarrow$ arrows which must "fuse" with the preceding ones. For example, if we specialize the a in id_{E} to the type of the identity function, we learn that we are missing one $\widetilde{\lambda}$, and so that special case should be expanded once, but if we specialize a to the type of the constant function, it is instead missing two $\widetilde{\lambda}$s, like so:

$$\mathit{id}_{\mathsf{E}}\ (\widetilde{\forall}a{:}\mathsf{E}.a{\rightsquigarrow}a) =_{\eta_{\rightsquigarrow}} \widetilde{\lambda}x{:}\widetilde{\forall}a{:}\mathsf{E}.a{\rightsquigarrow}a.\widetilde{\lambda}y{:}a.x\ y$$
$$\mathit{id}_{\mathsf{E}}\ (\widetilde{\forall}a{:}\mathsf{E}.\widetilde{\forall}a{:}\mathsf{E}.a{\rightsquigarrow}b{\rightsquigarrow}a)$$
$$=_{\eta_{\rightsquigarrow}} \widetilde{\lambda}x{:}\widetilde{\forall}a{:}\mathsf{E}.\widetilde{\forall}b{:}\mathsf{E}.a{\rightsquigarrow}b{\rightsquigarrow}a.\widetilde{\lambda}y{:}a.\widetilde{\lambda}z{:}b.x\ y\ z$$

In each case, depending on the type of the higher-order parameter x, x will be called at a different arity, which is no longer known at compile-time in the definition site of id_{E}.

The problem with polymorphic types a of kind E is that E types posses a form of non-uniformity: they may vary in their calling convention in terms of the number of arguments they expect. Unlike V types for which arity checking is done dynamically at run-time ("unknown" functions in the parlance of [Marlow and Peyton Jones 2004]), the different arity of E types is "known" at compile-time. This property is essential for fast code generation: the call site of an E function type will just assume a particular arity for passing some number of arguments at once and call that function without consulting its meta-data. As such, the arity of E types must be statically predictable, which conflicts with generic type variables $a : \mathsf{E}$.

There is another, better-known, non-uniformity problem with polymorphism: data representation. For example, consider the projection function: $snd : (a, b) \to b = \lambda(x, y).y$. When both a and b are unknown polymorphic types, how can we generate code for extracting the second component of the pair? This is quite a complicated question when the values of different types may be represented by different sizes and layouts in memory. If the pair (x, y) is stored contiguously in memory, then the offset to the beginning of y depends on the size of the value x: when x is a character it may 8-bits, but when it is a double-precision floating point number it takes up 64-bits. Likewise, we need to know the size of y to know how many bytes to move. And the situation is further exacerbated when certain types of values, like floating-points, require special code for parameter passing.

A common solution to these problems with polymorphism is to enforce *uniform representation*: every type is constrained to values represented by pointer-sized data. With this solution, all of the above problems vanish, but at the cost of extra *boxing*: if a value cannot fit within the constraints of a pointer-like shape, then there is an extra level of indirection by replacing the value with a pointer to the value. The indirection of boxing is costly in practice, and so an optimizing compiler should use unboxed values when possible, but this may come at the cost of disabling polymorphism.

An interesting approach to reconciling unboxing with polymorphism was introduced by [Eisenberg and Peyton Jones 2017] called "levity polymorphism." One can view levity polymorphism as a conservative extension of a conventional functional language relying on uniform representation with the following novel features:

1. unboxed types (*e.g.,* integer and floating-point numbers) with a variety of different representations,
2. polymorphism over types with a known (*i.e.,* monomorphic) representation, and
3. polymorphism over representations themselves.

The key idea to make levity polymorphism work is to enrich *kinds* to be informative enough to determine the representations of values, even when the specific type of those values are yet unknown. Pleasantly, points 1 and 2 together do not cause issue for compilability. Only point 3 could be trouble, which encapsulates all of the above issues we discussed about compiling non-uniform polymorphism. And thankfully, [Eisenberg and Peyton Jones 2017] gives a simple, fundamental, requirement to ensure compilability:

Never move or store a levity-polymorphic value.

We believe that the same basic idea can be applied to enhance the polymorphism over functions with non-uniform arity. As future work, we would like to further enrich the language of kinds to fill in arity information even when the specific type is unknown. This extension could give a rich language of kinds with the goal of statically tracking:

1. the specific evaluation strategy of function arguments (*e.g.,* call-by-value versus call-by-name) as in Section 3,
2. the number of arguments the function accepts before doing serious work (the traditional notion of arity),
3. the representation of each argument (so that specific parameter-passing code is known at the call site), and
4. the representation of its result (so that higher-arity functions can be composed).

The purpose of such an extension would be to maintain information about both representations and arity in a compositional way: arities and representations should still be inferred from kinds no matter how deeply nested functions and data structures become.

8 Related Work

8.1 Uncurrying

The classic 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]. Similar to the work here, they too focus on solving fast higher-arity calls inside higher-order functions.

Like us, uncurrying techniques aim 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 a compound, multiple-argument function type $(\sigma_0, \ldots, \sigma_n) \to \tau$ for uncurrying. As discussed in Section 3, multi-argument functions requires special support in the presence of polymorphism, which ends up entwining the two different language features into one. This involves a telescoped introduction of type variables in the (many) arguments of the function which remain in scope for the return type. Our approach based on extensional functions completely avoids this technical problem since the structure of our higher-arity types $\widetilde{\forall} a.\tau$ and $\sigma \rightsquigarrow \tau$ are exactly the same as the original curried definition of functions.

In an approach more similar to ours, [Hannan and Hicks 1998] introduce an intermediate representation for their uncurrying transformation that involves adding annotated function types. The function type $\sigma \to_{\Uparrow} \tau$ fuses with τ when it is also a function, and $(\sigma \to_{\epsilon} \tau)$ is a standard curried function type. Our $(\sigma \rightsquigarrow \tau)$ functions also have a fusion property, but we do not require the codomain τ be function. [Hannan and Hicks 1998]'s work avoids the problem of polymorphism that other uncurrying techniques have 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].

Thus far, uncurrying-based arity presents a monolithic approach which fuses several distinct language features and can require switching to an entirely different evaluation strategy like [Bolingbroke and Peyton Jones 2009], which can interfere with the rest of the compiler pipeline. As a general design goal, our technique aims to be as non-disruptive as possible, maintaining the same overall structure of programs and types for both lazy and strict languages (avoiding both explicit uncurrying and thunking), and only requiring that types inform us of the evaluation strategy. Thus, our technique is better suited for integrating into existing implementations because of its lower initial investment. As a pleasant side benefit, our approach based on integrating multiple evaluation strategies introduces opportunities for optimizations unrelated to arity. For instance, in Haskell, a call to `f :: Int -> Int` must first perform a dynamic check to determine if `f` needs to be evaluated. In contrast, calling `f :: Int ⇝ Int` can just immediately call `f` without that dynamic check, because it must be bound to a closure. Just uncurrying a higher-arity function does not express this difference of evaluation strategy.

8.2 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 [Leroy 1990] abstract machines repeatedly push arguments onto the stack and an n-ary function will consume all 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 is fully applied, then the compiler can generate code that fuses the applications. For the cases where the arity is unknown at compile time, *e.g.,* inside 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 statically-known arity information to the call site at compile-time, or there must be a dynamic check for arity information at run-time.

The usage of arity information has long been an issue in compilers leading to the development of complex arity analyzers [Breitner 2014; Xu and Peyton Jones 2005] which use the information to η expand and to float λ-abstractions into and out of contexts; all while being careful to not duplicate 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 run-time if they will only be evaluated once. Our goal is to improve the frequency of multi-arity function calls, which is orthogonal to analysis. We can use either simple analysis like syntactically-visible λ-abstractions and applications, or those more thorough methods.

We seek to secure arity information inside the types of our intermediate language; a place where it will be preserved while optimizations freely manipulate the structure of terms. To apply the standard worker/wrapper transformation, the type is derived from counting λs and this is used to create the worker and wrapper. The wrapper η expands higher-order functions to produce higher-arity ($\rightsquigarrow$) functions. These functions must appear fully applied in the wrapper which may require further η expansion. We exploit the fact that call-by-name functions can always appear fully applied due to the η law in order to generate saturated applications.

8.3 Polarity

Our inspiration for the treatment of arity came from *polarity* and *call-by-push-value* [Levy 2001; Munch-Maccagnoni 2013; Zeilberger 2009], which study a type-based mixture of evaluation strategies in programming languages from the fields of logic and type theory. These systems enjoy the strongest possible extensionality laws (*a.k.a.* η axioms) for every type [Munch-Maccagnoni 2009], 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]. Recently, they have been extended with call-by-need constructs [Downen and Ariola 2018; McDermott and Mycroft 2019] used to support Haskell-like languages. Investigating the consequences of these polarized languages 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 have arisen independently in practice. 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? Since a value of type `Int#` is represented by a 64-bit machine integer, there is no way to delay the call to `g`. Passing an unboxed integer in a register means it is already a value, so it must be evaluated ahead of time (call-by-value). This insight lets GHC handle unboxed values [Peyton Jones and Launchbury 1991].

There are different ways of formalizing polarity in a system, and the closest one to our approach here is [Munch-Maccagnoni 2009; Munch-Maccagnoni and Scherer 2015], wherein type constructors can be applied to any types. Other systems [Levy 2001; Zeilberger 2008] impose stronger constraints on types. In the notation here, we could restrict $\sigma \rightsquigarrow \tau : \mathsf{E}$ so that $\sigma : \mathsf{V}$ and $\tau : \mathsf{E}$. This style requires "polarity shifts:" conversions between V and E kinds of types. With shifts, we could remove the original type $\sigma \rightarrow \tau : \mathsf{V}$, but that would go against our desire for a conservative extension; it fundamentally changes the semantics of the base language, changing the front-end translation into the intermediate language. However, this alternative could have its own merits. For example, GHC already uses monomorphic versions of these shifts to represent the boxing and unboxing explicitly in programs (*e.g.,* the unboxed `Int#` type is primitive, and the lazy `Int` is derived from it). Further investigation is required to evaluate the relative practical merits of these two styles.

9 Conclusion

This work follows the tradition of designing calculi that faithfully capture the practical issue of optimizing curried function calls. 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 we think some of these can be put on solid ground with polarization and extensionality. Interestingly, the solutions for making fast calls across higher-order functions and for unboxing tuples appear to be dual to one another. As a pleasant side effect, this approach is suitable for both strict and lazy functional languages. We have implemented the ideas presented here as an extension of GHC's Core intermediate language as a proof of concept.(available at https://github.com/zachsully/ghc/tree/eta-arity).

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CCF-1719158 and Grant No. CCF-1423617. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

References

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

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.

Vincent Danos, Jean-Baptiste Joinet, and Harold Schellinx. 1997. A New Deconstructive Logic: Linear Logic. *Journal of Symbolic Logic* 62, 3 (1997), 755—-807.

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. 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.

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.

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.

Dylan McDermott and Alan Mycroft. 2019. Extended Call-by-Push-Value: Reasoning About Effectful Programs and Evaluation Order. In *Programming Languages and Systems*, Luís Caires (Ed.). Springer International Publishing, Cham, 235–262.

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. 1992. Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-machine. *Journal of Functional Programming* 2, 2 (1992), 127—-202.

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.

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.

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.