Retrofitting Linear Types

ANONYMOUS AUTHOR(S)

Linear type systems have a long and storied history, but not a clear path forward to integrate with existing languages such as OCaml or Haskell. In this paper, we study a linear type system designed with two crucial properties in mind: backwards-compatibility and code reuse across linear and non-linear users of a library. Only then can the benefits of linear types permeate conventional functional programming. Rather than bifurcate types into linear and non-linear counterparts, we instead attach linearity to *function arrows*. Linear functions can receive inputs from linearly-bound values, but can *also* operate over unrestricted, regular values.

To demonstrate the efficacy of our linear type system — both how easy it can be integrated in an existing language implementation and how streamlined it makes it to write programs with linear types — we implemented our type system in GHC, the leading Haskell compiler, and demonstrate two kinds of applications of linear types: mutable data with pure interfaces; and enforcing protocols in I/O-performing functions.

1 INTRODUCTION

Despite their obvious promise, and a huge research literature, linear type systems have not made it into mainstream programming languages, even though linearity has inspired uniqueness typing in Clean, and ownership typing in Rust. We take up this challenge by extending Haskell with linear types. Our design supports many applications for linear types, but we focus on two particular use-cases. First, safe update-in-place for mutable structures, such as arrays; and second, enforcing access protocols for external APIs, such as files, sockets, channels and other resources. Our particular contributions are these:

- We describe a new extension to Haskell, dubbed Hask-LL, using two extended examples (Sec. 2.1-Sec. 2.3). The extension is *non-invasive*: existing programs continue to typecheck, and existing data types can be used as-is even in linear parts of the program. The key to this non-invasiveness is that, in contrast to most other approaches, we focus on *linearity on the function arrow* rather than *linearity in the kinds* (Sec. 6.1).
- Every function arrow can be declared linear, including those of constructor types. This results in data-types which can store both linear values, in addition to unrestricted ones (Sec. 2.4).
- A benefit of linearity-on-the-arrow is that it naturally supports *linearity polymorphism* (Sec. 2.6). This contributes to a smooth extension of Haskell by allowing many existing functions (map, compose, etc) to be given more general types, so they can work uniformly in both linear and non-linear code.
- We formalise our system in a small, statically-typed core calculus that exhibits all these features (Sec. 3). It enjoys the usual properties of progress and preservation.
- We have implemented a prototype of the system as a modest extension to GHC (Sec. 4), which substantiates our claim of non-invasiveness. We use this prototype to implement case-study applications (Sec. 5). Our prototype performs linearity *inference*, but a systematic treatment of type inference for linearity in our system remains open.

Retrofits often involve compromise and ad-hoc choices, but in fact we have found that, as well as fitting into Haskell, our design holds together in its own right. We hope that it may perhaps serve

2017. 2475-1421/2017/1-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

as a template for similar work in other languages. There is a rich literature on linear type systems, as we discuss in a long related work section (Sec. 6).

2 MOTIVATION AND INTUITIONS

Informally, *a function is "linear" if it consumes its argument exactly once.* (It is "affine" if it consumes it at most once.) A linear type system gives a static guarantee that a claimed linear function really is linear. There are many motivations for linear type systems, but they mostly come down to two questions:

- *Is it safe to update this value in-place* (Sec. 2.2)? That depends on whether there are aliases to the value; update-in-place is OK if there are no other pointers to it. Linearity supports a more efficient implementation, by $O(1)$ update rather than $O(n)$ copying.
- *Am I obeying the usage protocol of this external resource* (Sec. 2.3)? For example, an open file should be closed, and should not be used after it it has been closed; a socket should be opened, then bound, and only then used for reading; a malloc'd memory block should be freed, and should not be used after that. Here, linearity does not affect efficiency, but rather eliminates many bugs.

We introduce our extension to Haskell, which we call Hask-LL (Haskell with Linear Logic), by focusing on these two use-cases. In doing so, we introduce a number of ideas that we flesh out in subsequent subsections.

2.1 Operational intuitions

We have said informally that *"a linear function consumes its argument exactly once"*. But what exactly does that mean?

> *Meaning of the linear arrow*: f :: s ⊸ t guarantees that if (f u) is consumed exactly once, then the argument u is consumed exactly once.

To make sense of this statement we need to know what "consumed exactly once" means. Our definition is based on the type of the value concerned:

Definition 2.1 (Consume exactly once).

- To consume a value of atomic base type (like Int or Ptr) exactly once, just evaluate it.
- To consume a function exactly once, apply it to one argument, and consume its result exactly once.
- To consume a pair exactly once, pattern-match on it, and consume each component exactly once.
- In general, to consume a value of an algebraic data type exactly once, pattern-match on it, and consume all its linear components exactly once (Sec. 2.5)[1].

This definition is enough to allow programmers to reason about the typing of their functions, and it drives the formal typing judgements in Sec. 3.

Note that a linear arrow specifies *how the function uses its argument.* It does not restrict *the arguments to which the function can be applied.* In particular, a linear function cannot assume that it is given the unique pointer to its argument. For example, if f :: s ⊸ t, then this is fine:

```
g :: s → t
g x = f x
```

The type of g makes no particular guarantees about the way in which it uses x; in particular, g can pass that argument to f.

[1] You may deduce that pairs have linear components, and indeed they do, as we discuss in Sec. 2.5.

2.2 Safe mutable arrays

The Haskell language provides immutable arrays, built with the function array[2]:

```
array :: Int → [(Int, a)] → Array a
```

But how is array implemented? A possible answer is "it is built-in; don't ask". But in reality GHC implements array using more primitive pieces, so that library authors can readily implement more complex variations — and they certainly do: see for example Sec. 5.1. Here is the definition of array, using library functions whose types are given in Fig. 1.

```
array :: Int → [(Int, a)] → Array a
array size pairs = runST
  (do {ma ← newMArray size
      ; forM_ pairs (write ma)
      ; return (unsafeFreeze ma)})
```

```
type MArray s a
type Array a

newMArray :: Int → ST s (MArray s a)
read :: MArray s a → Int → ST s a
write :: MArray s a → (Int, a) → ST s ()
unsafeFreeze :: MArray s a → ST s (Array a)

forM_ :: Monad m ⇒ [a] → (a → m ()) → m ()
runST :: (∀a. ST s a) → a
```

Fig. 1. Signatures for array primitives (current GHC)

In the first line we allocate a mutable array, of type MArray s a. Then we iterate over the pairs, with forM_, updating the array in place for each pair. Finally, we freeze the mutable array, returning an immutable array as required. All this is done in the ST monad, using runST to securely encapsulate an imperative algorithm in a purely-functional context, as described in (Launchbury and Peyton Jones 1995).

Why is unsafeFreeze unsafe? The result of (unsafeArray ma) is a new immutable array, but to avoid an unnecessary copy, the two are actually *the same array*. The intention is, of course, that that unsafeFreeze should be the last use of the mutable array; but nothing stops us continuing to mutate it further, with quite undefined semantics. The "unsafe" in the function name is a GHC convention meaning "the programmer has a proof obligation here that the compiler cannot check".

The other unsatisfactory thing about the monadic approach to array construction is that it is overly sequential. Suppose you had a pair of mutable arrays, with some updates to perform to each; these updates could be done in *parallel*, but the ST monad would serialise them.

Linear types allow a more secure and less sequential interface. Hask-LL introduces a new kind of function type: the *linear arrow* a ⊸ b. A linear function f :: a ⊸ b must consume its argument *exactly once*. This new arrow is used in a new array API, given in Fig. 2.

```
type MArray a
type Array a

newMArray :: Int → (MArray a ⊸ Unrestricted b) ⊸ b
write :: MArray a ⊸ (Int, a) → MArray a
read :: MArray a ⊸ Int → (MArray a, Unrestricted a)
freeze :: MArray a ⊸ Unrestricted (Array a)
```

Fig. 2. Type signatures for array primitives (linear version), allowing in-place update.

Using this API we can define array thus:

[2] Haskell actually generalises over the type of array indices, but for this paper we will assume that the arrays are indexed, from 0, by Int indices.

```
type File
openFile :: FilePath → IO File
readLine :: File → IO ByteString
closeFile :: File → IO ()
```

Fig. 3. Types for traditional file IO

```
type File
openFile :: FilePath → IO_L 1 File
readLine :: File a ⊸ IO_L 1 (File, Unrestricted ByteString)
closeFile :: File ⊸ IO_L ω ()
```

Fig. 4. Types for linear file IO

```
array :: Int → [(Int, a)] → Array a
array size pairs = newMArray size (λma → freeze (foldl write ma pairs))
```

There are several things to note here:

- The function newMArray allocates a fresh, mutable array of the specified size, and passes it to the function supplied as the second argument to newMArray, as a *linear* value ma.
- Even though linearity is a property of function arrows, not of types (Sec. 6.1), we still disinguish the type of mutable arrays MArray from that of immutable arrays Array, because in this API only immutable arrays are *allowed* to be non-linear (unrestricted). The way to say that results can be freely shared is to use Unrestricted (Sec. 2.4), as in the type of freeze.
- Because freeze consumes its input, there is no danger of the same mutable array being subsequently written to, eliminating the problem with unsafeFreeze.
- Since ma is linear, we can only use it once. Thus each call to write returns a (logically) new array, so that the array is single-threaded, by foldl, through the sequence of writes.
- Above, foldl has the type (a ⊸ b ⊸ a) → a ⊸ [b] ⊸ a, which expresses that it consumes its second argument linearly (the mutable array), while the function it is given as its first argument (write) must be linear. As we shall see in Sec. 2.6 this is not a new foldl, but an instance of a more general, multiplicity-polymorphic version of a single foldl (where "multiplicity" refers to how many times a function consumes its input).

Three factors ensure that a unique MArray is needed in any given application x = newMArray k, and in turn that update-in-place is safe. First, newMArray introduces *only* a linear ma :: MArray. Second, no function that consumes an MArray a returns more than a *single* pointer to it; so k can never obtain two pointers to ma. Third, k must wrap its result in Unrestricted. This third point means that even if x is used in an unrestricted way, it suffices to call k a single time to obtain the result, and in turn no mutable pointer to ma can *escape* when newArray returns (*i.e.* when the b result of newArray is evaluated).

With this mutable array API, the ST monad has disappeared altogether; it is the array *itself* that must be single threaded, not the operations of a monad. That removes the unnecessary sequentialisation we mentioned earlier and opens the possibility of exploiting more parallelism at runtime.

Compared to the *status quo* (using ST and unsafeFreeze), the other major benefit is in shrinking the trusted code base, because more library code (and it can be particularly gnarly code) is statically typechecked. Clients who use only *immutable* arrays do not see the inner workings of the library, and will be unaffected. Our second use-case has a much more direct impact on library clients.

2.3 I/O protocols

Consider the API for files in Fig. 3, where a File is a cursor in a physical file. Each call to readLine returns a ByteString (the line) and moves the cursor one line forward. But nothing stops us reading a file after we have closed it, or forgetting to close it. An alternative API using linear types is given in Fig. 4. Using it we can write a simple file handling program, firstLine, shown here.

```
firstLine :: FilePath → IO_L ω Bytestring
firstLine fp =
  do {f ← open fp
     ; (f, Unrestricted bs) ← readLine f
     ; close f
     ; return bs}
```

Notice several things:

- Operations on files remain monadic, unlike the case with mutable arrays. I/O operations affect the world, and hence must be sequenced. It is not enough to sequence operations on files individually, as it was for arrays.
- We generalise the IO monad so that it expresses whether or not the returned value is linear. We add an extra *multiplicity* type parameter p to the monad IO_L, where p can be 1 or ω, indicating a linear or unrestricted result, respectively. Now openFile returns IO_L 1 (File ByteString), the "1" indicating that the returned File must be used linearly. We will return to how IO_L is defined in Sec. 2.7.
- As before, operations on linear values must consume their input and return a new one; here readLine consumes the File and produces a new one.
- Unlike the File, the ByteString returned by readLine is unrestricted, and the type of readLine indicates this.

It may seem tiresome to have to thread the File as well as sequence operations with the IO monad. But in fact it is often useful do to do so, because we can use types to witness the state of the resource, *e.g.*, with separate types for an open or closed File. We show applications in Sec. 5.1 and Sec. 5.2.

2.4 Linear data types

With the above intutions in mind, what type should we assign to a data constructor such as the pairing constructor (,)? Here are two possibilities:

$$(,) :: a \multimap b \multimap (a, b) \qquad\qquad (,) :: a \to b \to (a, b)$$

Using the definition in Sec. 2.1, the former is clearly the correct choice: if the result of (,) e1 e2 is consumed exactly once, then (by Def. 2.1), e1 and e2 are each consumed exactly once; and hence (,) is linear it its arguments.

So much for construction; what about pattern matching? Consider f1 and f2 defined here; f1 is an ordinary Haskell function. Even though the data constructor (,) has a linear type, that does *not* imply that the pattern-bound variables a and b must be consumed exactly once; and indeed they are not. Therefore, f1 does not have the linear type (Int, Int) ⊸ (Int, Int). Why not? If the result of (f1 t) is consumed once, is t guaranteed to be consumed once? No: t is guaranteed to be evaluated once, but its first component is then consumed twice and its second component not at all, contradicting Def. 2.1. In contrast, f2 *does* have a linear type: if (f2 t) is consumed exactly once, then indeed t is consumed exactly once. The key point here is that *the same pair constructor works in both functions; we do not need a special non-linear pair.*

```
f1 :: (Int, Int) → (Int, Int)
f1 x = case x of (a, b) → (a + a, 0)

f2 :: (Int, Int) ⊸ (Int, Int)
f2 x = case x of (a, b) → (b, a)
```

The same idea applies to all existing Haskell data types: in Hask-LL we treat all data types defined using legacy Haskell-98 (non-GADT) syntax as defining constructors with linear arrows. For example here is a declaration of Hask-LL's list type, whose constructor (:) uses linear arrows:

Just as with pairs, this is not a new, linear list type: this *is* Hask-LL's list type, and all existing Haskell functions will work over it perfectly well. Even better, many list-based functions are in fact linear, and can be given a more precise type. For example we can write (⅋) as follows:

```
data [a] = [] | a : [a]

(⅋) :: [a] ⊸ [a] ⊸ [a]
[]       ⅋ ys = ys
(x : xs) ⅋ ys = x : (xs ⅋ ys)
```

This type says that if (xs ⅋ ys) is consumed exactly once, then xs is consumed exactly once, and so is ys, and indeed our type system will accept this definition.

As before, giving a more precise type to (⅋) only *strengthens* the contract that (⅋) offers to its callers; *it does not restrict its usage*. For example:

```
sum :: [Int] ⊸ Int
f :: [Int] ⊸ [Int] → Int
f xs ys = sum (xs ⅋ ys) + sum ys
```

Here the two arguments to (⅋) have different multiplicities, but the function f guarantees that it will consume xs exactly once if (f xs ys) is consumed exactly once.

For an existing language, being able to strengthen (⅋), and similar functions, in a *backwards-compatible* way is a huge boon. Of course, not all functions are linear: a function may legitimately demand unrestricted input. For example, the function f above consumes ys twice, and so f needs an unrestricted arrow for that argument.

Finally, we can use the very same pairs and lists types to contain linear values (such as mutable arrays) without compromising safety. For example:

```
upd :: (MArray Char, MArray Char) ⊸ Int → (MArray Char, MArray Char)
upd (a1, a2) n | n ⩾ 10    = (write a1 n 'x', a2)
               | otherwise = (write a2 n 'o', a1)
```

2.5 Unrestricted data constructors

Suppose I want to pass a linear MArray and an unrestricted Int to a function f. We could give f the signature f :: MArray Int ⊸ Int → MArray Int. But suppose we wanted to uncurry the function; we could then give it the type

```
f :: (MArray Int, Int) ⊸ MArray Int
```

But this is no good: now f is only allowed to use the Int linearly, but it might actually use it many times. For this reason it is extremely useful to be able to declare data constructors with non-linear types, like this:

```
data PLU a b where {PLU :: a ⊸ b → PLU a b}

f :: PLU (MArray Int) Int ⊸ MArray Int
```

Here we use GADT-style syntax to give an explicit type signature to the data constructor PLU, with mixed linearity. Now, when *constructing* a PLU pair the type of the constructor means that we must always supply an unrestricted second argument; and dually when *pattern-matchinng* on PLU we are therefore free use the second argument in an unrestricted way, even if the PLU value itself is linear.

Instead of defining a pair with mixed linearity, we can also write

```
data Unrestricted a where {Unrestricted :: a → Unrestricted a}

f :: (MArray Int, Unrestricted Int) ⊸ MArray Int
```

The type (Unrestricted t) is very much like "!t" in linear logic, but in our setting it is just an ordinary user-defined data type. We saw it used in Fig. 2, where the result of read was a pair of a linear MArray and an unrestricted array element:

```
read :: MArray a ⊸ Int → (MArray a, Unrestricted a)
```

Note that, according to the definition in Sec. 2.1, if a value of type (Unrestricted t) is consumed exactly once, that tells us nothing about how the argument of the data constructor is consumed: it may be consumed many times or not at all.

2.6 Multiplicity polymorphism

A linear function provides more guarantees to its caller than a non-linear one — it is more general. But the higher-order case thickens the plot. Consider the standard map function over (linear) lists:

```
map f []       = []
map f (x : xs) = f x : map f xs
```

It can be given the two following incomparable types: $(a \multimap b) \to [a] \multimap [b]$ and $(a \to b) \to [a] \to [b]$. Thus, Hask-LL features quantification over multiplicities and parameterised arrows ($A \to_q B$). Using these, map can be given the following more general type: $\forall p.\ (a \to_p b) \to [a] \to_p [b]$. Likewise, function composition and foldl (*cf.* Section 2.2) can be given the following general types:

```
foldl :: ∀p q. (a →p b →q a) → a →p [b] →q a
(∘) :: ∀p q. (b →p c) ⊸ (a →q b) →p a →p·q c
(f ∘ g) x = f (g x)
```

The type of (∘) says that two functions that accept arguments of arbitrary multiplicities (p and q respectively) can be composed to form a function accepting arguments of multiplicity $p \cdot q$ (*i.e.* the product of p and q — see Def. 3.4). Finally, from a backwards-compatibility perspective, all of these subscripts and binders for multiplicity polymorphism can be ignored. Indeed, in a context where client code does not use linearity, all inputs will have unlimited multiplicity, ω, and transitively all expressions can be promoted to ω. Thus in such a context the compiler, or indeed documentation tools, can even altogether hide linearity annotations from the programmer when this language extension is not turned on.

2.7 Linear input/output

In Sec. 2.3 we introduced the IO_L monad.[3] But how does it work? IO_L is just a generalisation of the IO monad, thus:

```
type IOL p a
returnIOL :: a →p IOL p a
bindIOL :: IO p a ⊸ (a →p IOL q b) ⊸ IOL q b
```

The idea is that if m :: IO_L 1 t, then m is an input/output computation that returns a linear value of type t. But what does it mean to "return a linear value" in a world where linearity applies only to function arrows? Fortunately, in the world of monads each computation has an explicit continuation, so we just need to control the linearity of the continuation arrow. More precisely, in an application m `` `bind_IOL` `` k where m :: IO_L 1 t, we need the continuation k to be linear, k :: t ⊸ IO_L q t'. And that is captured by the multiplicity-polymorphic type of $\mathsf{bind}_{\mathsf{IO}_\mathsf{L}}$.

[3] IO_L p is not a monad in the strict sense, p and q can be different in $\mathsf{bind}_{\mathsf{IO}_\mathsf{L}}$, it is however a relative monad (Altenkirch et al. 2010). The details, involving the functor **data** Mult p a = Mult :: a →p Mult p a and linear arrows, are left as an exercise to the reader

Even though they have a different type than usual, the bind and return combinators of IO_L can be used in the familiar way. The difference with the usual monad is that multiplicities may be mixed, but this poses no problem in practice. Consider

```
do {f ← openFile s    -- openFile :: FilePath → IOL 1 (File ByteString)
   ;d ← getDate       -- getDate :: IOL ω Date
   ;e [f,d]}
```

Here openFile returns a linear File that should be closed, but getDate returns an ordinary non-linear Date. So this sequence of operations has mixed linearity. Nevertheless, we can combine them with $\mathsf{bind}_{\mathsf{IO}_\mathsf{L}}$ in the usual way:

```
openFile s `bindIOL` λf →
getData `bindIOL` λd →
e [f,d]
```

Such an interpretation of the **do**-notation requires Haskell's `-XRebindableSyntax` extension, but if linear I/O becomes commonplace it would be worth considering a more robust solution.

Internally, hidden from clients, GHC actually implements IO as a function, and that implementation too is illuminated by linearity, like so:

```
data World
newtype IOL p a = IOL (unIOL :: World ⊸ IORes p a)
data IORes p a where
  IOR :: World ⊸ a →p IOR p a
bindIOL :: IOL p a ⊸ (a →p IOL q b) ⊸ IOL q b
bindIOL (IOL m) k = IOL (λw → case m w of
                            IOR w' r → unIOL (k r) w')
```

A value of type World represents the state of the world, and is threaded linearly through I/O computations. The linearity of the result of the computation is captured by the p parameter of IO_L, which is inherited by the specialised form of pair, IORes that an IO_L computation returns. All linearity checks are verified by the compiler, further reducing the size of the trusted code base.

2.8 Linearity and strictness

It is tempting to assume that, since a linear function consumes its argument exactly once, then it must also be strict. But not so! For example

```
f :: a ⊸ (a, Bool)
f x = (x, True)
```

Here f is certainly linear according to Sec. 2.1, and given the type of (,) in Sec. 2.4. That is, if (f x) is consumed exactly once, then each component of its result pair is consumed exactly once, and hence x is consumed exactly once. But f is certainly not strict: f ⊥ is not ⊥.

3 $\lambda^{Q}_{\to}$: A CORE CALCULUS FOR HASK-LL

It would be impractical to formalise all of Hask-LL. So instead we formalise a core calculus, $\lambda^{q}_{\to}$, which exhibits all the key features of Hask-LL, including data types and multiplicity polymorphism. In this way we make precise much of the informal discussion above.

Multiplicities

$$\pi, \mu ::= 1 \mid \omega \mid p \mid \pi + \mu \mid \pi \cdot \mu$$

Contexts

$$\Gamma, \Delta ::= (x :_\mu A), \Gamma \quad \mid \quad -$$

Datatype declaration

$$\text{data } D\ p_1\ \ldots\ p_n \text{ where } \left(c_k : A_1 \to_{\pi_1} \cdots\ A_{n_k} \to_{\pi_{n_k}} D\right)_{k=1}^{m}$$

Types

$$A, B ::= A \to_\pi B \quad \mid \quad \forall p.A \quad \mid \quad D\ p_1\ \ldots\ p_n$$

Terms

$$\begin{array}{rr}
e, s, t, u ::= x & \text{variable} \\
\mid \lambda_\pi (x{:}A).t & \text{abstraction} \\
\mid t\ s & \text{application} \\
\mid \lambda p.t & \text{multiplicity abstraction} \\
\mid t\ \pi & \text{multiplicity application} \\
\mid c\ t_1 \ldots t_n & \text{data construction} \\
\mid \text{case}_\pi\ t \text{ of } \{c_k\ x_1 \ldots x_{n_k} \to u_k\}_{k=1}^{m} & \text{case} \\
\mid \text{let}_\pi\ x_1 : A_1 = t_1 \ldots x_n : A_n = t_n \text{ in } u & \text{let}
\end{array}$$

Fig. 5. Syntax of $\lambda^q_\to$

3.1 Syntax

The term syntax of $\lambda^q_\to$ is that of a type-annotated (*à la* Church) simply-typed λ-calculus with let-definitions (Fig. 5). It includes multiplicity polymorphism, but to avoid clutter we omit ordinary type polymorphism.

$\lambda^q_\to$ is an explicitly-typed language: each binder is annotated with its type and multiplicity; and multiplicity abstraction and application are explicit. Hask-LL will use type inference to fill in much of this information, but we do not address the challenges of type inference here.

The types of $\lambda^q_\to$ (see Fig. 5) are simple types with arrows (albeit multiplicity-annotated ones), data types, and multiplicity polymorphism. We use the following abbreviations: $A \to B \stackrel{\text{def}}{=} A \to_\omega B$ and $A \multimap B \stackrel{\text{def}}{=} A \to_1 B$.

Data type declarations (see Fig. 5) are of the following form:

$$\text{data } D\ p_1\ \ldots\ p_n \text{ where } \left(c_k : A_1 \to_{\pi_1} \cdots\ A_{n_k} \to_{\pi_{n_k}} D\right)_{k=1}^{m}$$

The above declaration means that D is parameterized over n multiplicities p_i and has m constructors c_k, each with n_k arguments. Arguments of constructors have a multiplicity, just like arguments of functions: an argument of multiplicity ω means that consuming the data constructor once makes no claim on how often that argument is consumed (Def. 2.1). All the variables in the multiplicities π_i must be among $p_1 \ldots p_n$; we write $\pi[\pi_1 \ldots \pi_n]$ for the substitution of p_i by π_i in π.

3.2 Static semantics

The static semantics of $\lambda^q_\to$ is given in Fig. 6. Each binding in Γ, of form $x :_\pi A$, includes a multiplicity π (Fig. 5). The familiar judgement $\Gamma \vdash t : A$ should be read as follows

$$\frac{}{\omega\Gamma + x :_1 A \vdash x : A}\text{var} \qquad \frac{\Gamma, x :_\pi A \vdash t : B}{\Gamma \vdash \lambda_\pi(x{:}A).t : A \to_q B}\text{abs} \qquad \frac{\Gamma \vdash t : A \to_\pi B \qquad \Delta \vdash u : A}{\Gamma + \pi\Delta \vdash t\ u : B}\text{app}$$

$$\frac{\Delta_i \vdash t_i : A_i \qquad c_k : A_1 \to_{\mu_1} \ldots \to_{\mu_{n-1}} A_n \to_{\mu_n} D\ p_1\ \ldots\ p_n \text{ constructor}}{\omega\Gamma + \sum_i \mu_i[\pi_1 \ldots \pi_n]\Delta_i \vdash c_k\ t_1 \ldots t_n : D\ \pi_1\ \ldots\ \pi_n}\text{con}$$

$$\frac{\Gamma \vdash t : D\ \pi_1\ \ldots\ \pi_n \qquad \Delta, x_1 :_{\pi\mu_i[\pi_1\ldots\pi_n]} A_i, \ldots, x_{n_k} :_{\pi\mu_{n_k}[\pi_1\ldots\pi_n]} A_{n_k} \vdash u_k : C \quad \text{for each } c_k : A_1 \to_{\mu_1} \ldots \to_{\mu_{n_k-1}} A_{n_k} \to_{\mu_{n_k}} D\ p_1\ \ldots\ p_n}{\pi\Gamma + \Delta \vdash \mathsf{case}_\pi\ t\ \mathsf{of}\ \{c_k\ x_1 \ldots x_{n_k} \to u_k\}_{k=1}^m : C}\text{case}$$

$$\frac{\Gamma_i \vdash t_i : A_i \qquad \Delta, x_1 :_\pi A_1 \ldots x_n :_\pi A_n \vdash u : C}{\Delta + q\sum_i \Gamma_i \vdash \mathsf{let}_\pi\ x_1 : A_1 = t_1\ \ldots\ x_n : A_n = t_n\ \mathsf{in}\ u : C}\text{let} \qquad \frac{\Gamma \vdash t : A \qquad p \text{ fresh for } \Gamma}{\Gamma \vdash \lambda p.t : \forall p.A}\text{m.abs}$$

$$\frac{\Gamma \vdash t : \forall p.A}{\Gamma \vdash t\ \pi : A[\pi/p]}\text{m.app}$$

Fig. 6. Typing rules

> $\Gamma \vdash t : A$ asserts that consuming the term $t : A$ exactly once will consume each binding $(x :_\pi A)$ in Γ with its multiplicity π.

One may want to think of the *types* in Γ as inputs of the judgement, and the *multiplicities* as outputs.

The rule (abs) for lambda abstraction adds $(x :_\pi A)$ to the environment Γ before checking the body t of the abstraction. Notice that in $\lambda^q_\to$, the lambda abstraction $\lambda_\pi(x{:}A).t$ is explicitly annotated with multiplicity π. Remember, this is an explicitly-typed intermediate language; in Hask-LL this multiplicity is inferred.

The dual application rule (app) is more interesting:

$$\frac{\Gamma \vdash t : A \to_\pi B \qquad \Delta \vdash u : A}{\Gamma + \pi\Delta \vdash t\ u : B}\text{app}$$

To consume (t u) once, we consume t once, yielding the multiplicities in Γ, and u once, yielding the multiplicies in Δ. But if the multiplicity π on u's function arrow is ω, then the function consumes its argument not once but ω times, so all u's free variables must also be used with multiplicity ω. We express this by taking the *product* of the multiplicities in Δ and π, thus $\pi\Delta$. Finally we need to add together all the multiplicities in Γ and $\pi\Delta$; hence the context $\Gamma + \pi\Delta$ in the conclusion of the rule.

In writing this rule we needed to "scale" a context by a multiplicity, and "add" two contexts. We pause to define these operations.

Definition 3.1 (Context addition).

$$\begin{aligned}(x :_\pi A, \Gamma) + (x :_\mu A, \Delta) &= x :_{\pi+\mu} A, (\Gamma + \Delta)\\ (x :_\pi A, \Gamma) + \Delta &= x :_\pi A, \Gamma + \Delta \qquad (x \notin \Delta)\\ () + \Delta &= \Delta\end{aligned}$$

Context addition is total: if a variable occurs in both operands the first rule applies (with possible re-ordering of bindings in Δ), if not the second or third rule applies.

Definition 3.2 (Context scaling).

$$\pi(x :_{\mu} A, \Gamma) = x :_{\pi\mu} A, \pi\Gamma$$

Lemma 3.3 (Contexts form a module). *The following laws hold:*

$$\Gamma + \Delta = \Delta + \Gamma \qquad \pi(\Gamma + \Delta) = \pi\Gamma + \pi\Delta$$
$$(\pi + \mu)\Gamma = \pi\Gamma + \mu\Gamma$$
$$(\pi\mu)\Gamma = \pi(\mu\Gamma) \qquad 1\Gamma = \Gamma$$

These operations depend, in turn, on addition and multiplication of multiplicities. The syntax of multiplicities is given in Fig. 5. We need the concrete multiplicities 1 and ω and, to support polymorphism, multiplicity variables (ranged over by the metasyntactic variables p and q) as well as formal sums and products of multiplicities. Multiplicity expressions are quotiented by the following equivalence relation:

Definition 3.4 (equivalence of multiplicities). The equivalence of multiplicities is the smallest transitive and reflexive relation, which obeys the following laws:

- $+$ and $\cdot$ are associative and commutative
- 1 is the unit of $\cdot$
- $\cdot$ distributes over $+$
- $\omega \cdot \omega = \omega$
- $1 + 1 = 1 + \omega = \omega + \omega = \omega$

Thus, multiplicities form a semi-ring (without a zero), which extends to a module structure on typing contexts.

Returning to the typing rules in Fig. 6, the rule (let) is like a combination of (abs) and (app). Again, each let binding is explicitly annotated with its multiplicity. The variable rule (var) uses a standard idiom:

$$\frac{}{\omega\Gamma + x :_1 A \vdash x : A}\text{var}$$

This rule allows us to ignore variables with multiplicity ω (usually called weakening), so that, for example $x :_1 A, y :_{\omega} B \vdash x : A$ holds [4]. Note that the judgement $x :_{\omega} A \vdash x : A$ is an instance of the variable rule, because $(x :_{\omega} A) + (x :_1 A) = x :_{\omega} A$.

Finally, abstraction and application for multiplicity polymorphism are handled straightforwardly by (m.abs) and (m.app).

3.3 Data constructors and case expressions

The handling of data constructors and case expressions is a distinctive aspect of our design. For constructor applications, the rule (con), everything is straightforward: we treat the data constructor in precisely the same way as an application of a function with that data constructor's type. This includes weakening via the $\omega\Gamma$ context in the conclusion. The (case) rule is more interesting:

$$\frac{\Gamma \vdash t : D\ \pi_1\ \ldots\ \pi_n \qquad \Delta, x_1 :_{\pi\mu_i[\pi_1 \ldots \pi_n]} A_i, \ldots, x_{n_k} :_{\pi\mu_{n_k}[\pi_1 \ldots \pi_n]} A_{n_k} \vdash u_k : C \quad \text{for each } c_k : A_1 \to_{\mu_1} \ldots \to_{\mu_{n_k-1}} A_{n_k} \to_{\mu_{n_k}} D\ p_1\ \ldots\ p_n}{\pi\Gamma + \Delta \vdash \text{case}_{\pi}\ t\ \text{of}\ \{c_k\ x_1 \ldots x_{n_k} \to u_k\}_{k=1}^{m} : C}\text{case}$$

[4] Pushing weakening to the variable rule is classic in many λ-calculi, and in the case of linear logic, dates back at least to Andreoli's work on focusing (Andreoli 1992).

First, notice that the case keyword is annotated with a multiplicity π; this is analogous to the explicit multiplicity on a let binding. It says how often the scrutinee (or, for a let, the right hand side) will be consumed. Just as for let, we expect π to be inferred from an un-annotated case in Hask-LL.

The scrutinee t is consumed π times, which accounts for the $\pi\Gamma$ in the conclusion. Now consider the bindings $(x_i :_{\pi\mu_i[\pi_1 \ldots \pi_n]} A_i)$ in the environment for typechecking u_k. That binding will be linear only if *both* π *and* π_i are linear; that is, only if we specify that the scrutinee is consumed once, and the i'th field of the data constructor c_k specifies that is it consumed once if the constructor is (Def. 2.1). To put it another way, suppose one of the linear fields[5] of c_k is used non-linearly in u_k. Then, $\mu_i = 1$ (it is a linear field), so π must be ω, so that $\pi\mu_i = \omega$. In short, using a linear field non-linearly forces the scrutinee to be used non-linearly, which is just what we want. Here are some concrete examples:

```
fst :: (a, b) → a          swap :: (a, b) ⊸ (b, a)
fst   (a, b) = a           swap   (a, b) = (b, a)
```

Recall that both fields of a pair are linear (Sec. 2.4). In fst, the second component of the pair is used non-linearly (by being discarded) which forces the use of case_ω, and hence a non-linear type for fst. But swap uses the components linearly, so we can use case_1, giving swap a linear type.

3.4 Metatheory

In order to prove that our type system meets its stated goals, we introduce an operational semantics. The details are deferred to Appendix A, included in the anonymous supplementary material submitted with the article.

Of consuming exactly once. The operational semantics is a big-step operational semantics for lazy evaluation in the style of Launchbury (1993). Following Gunter and Rémy (1993), starting from a big-step evaluation relation $a \Downarrow b$, we define *partial derivations* and from there a *partial evaluation* relation $a \Downarrow^* b$ (see Sec. A.1). Progress is then expressed as the fact that a derivation of $a \Downarrow^* b$ can always be extended.

The operational semantics differs from Launchbury's in two major respects:

- The reduction states are heavily annotated with type information. These type annotations are used for the proofs.
- Reduction is indexed by whether we intend to consume the term under consideration exactly once or an arbitrary number of times
- Variables in the environments are annotated by a multiplicity (1 or ω), ω-variables are ordinary variables. When such a variable is forced it is replaced by its value (to model lazy sharing), but 1-variables *must be consumed exactly once*: when they are forced, they are removed from the environment. So reduction would get stuck if a 1-variable was used more than once.

Because the operational semantics gets stuck if a 1-variable is used more than once, the progress theorem (Theorem 3.6) shows that linear functions do indeed consume their argument at most once if their result is consumed exactly once. The 1-variables are in fact used exactly once: it is a consequence of type preservation that evaluation of a closed term of a basic type (say Bool) returns an environment with no 1-variables.

Our preservation and progress theorems then look like this, writing a, b for states of the evaluation:

[5] Recall Sec. 2.5, which described how each constructor can have a mixture of linear and non-linear fields.

Theorem 3.5 (Type preservation). *If a is well typed, and $a \Downarrow b$, or $a \Downarrow^* b$ then b is well-typed.*

Theorem 3.6 (Progress). *Evaluation does not block. That is, for any partial evaluation $a \Downarrow^* b$, where a is well-typed, the derivation can be extended.*

These theorems are proved in Sec. A.3.

In-place update & typestate. Furthermore, linear types can be used to implement some operations as in-place updates, and typestates (like whether an array is mutable or frozen) are actually enforced by the type system.

To prove this, we introduce a second, distinct, semantics. It is also a Launchbury style semantics. It differs from Launchbury (1993) in the following ways:

- Environments are enriched with mutable references (for the sake of concreteness, they are all references to arrays but they could be anything)
- Typestates are implemented by mutating the type of such references, functions can block if the type of the references isn't correct: that is, we track typestates dynamically

The idea behind the latter is that progress will show that we are never blocked by typestates. In other words, they are enforced statically and can be erased at runtime.

It is hard to reason on a lazy language with mutation. But what we show is that we are using mutation carefully enough so that they behave as pure data. To formalise this, we relate this semantics with mutation to our pure semantics above. Specifically, we show that they are *bisimilar*. This is similar to Amani et al. (2016), who also have a language with linear types with both a pure and imperative semantics.

Bisimilarity allows us to lift the type-preservation and progress from the pure semantics. That is, writing σ, τ for states of this evaluation with mutation:

Theorem 3.7 (Type preservation). *For any well-typed σ, if $\sigma \Downarrow \tau$ or $\sigma \Downarrow^* \tau$, then τ is well-typed.*

Theorem 3.8 (Progress). *Evaluation does not block. That is, for any partial evaluation $\sigma \Downarrow^* \tau$, for σ well-typed, the evaluation can be extended.*
In particular, typestates need not be checked dynamically.

Just as importantly, we can prove that, indeed, we cannot observe mutations. More precisely, we prove that the pure semantics and the semantics with mutation are observationally equivalent: any observation, which we reduce to a boolean test, is identical in either semantics.

Theorem 3.9 (Observational equivalence). *The semantics with in-place mutation is observationally equivalent to the pure semantics.*
That is, for any closed term of type Bool, if e evaluates to the value z with the pure semantics, and to the value z' with the semantics with mutation, then $z = z'$.

These three theorems are proved in Sec. A.4.

3.5 Design choices & trade-offs

We could as well have picked different points in the design space for $\lambda^q_{\to}$. We review some of the choices we made in this section.

Case rule. It is possible to do without case_ω, and have only case_1. Consider fst again. We could instead have

```
data Pair p q a b where
  Pair :: a →p  b →q  Pair p q a b
```

```
fst :: Pair 1 ω a b ⊸ a
fst x = case _1 x of Pair a b → a
```

But now multiplicity polymorphism infects all basic data types (such as pairs), with knock-on consequences. Moreover, **let** is annotated so it seems reasonable to annotate **case** in the same way.

To put it another way, case_ω allows us to meaningfully inhabit ∀a b. Unrestricted (a, b) ⊸ (Unrestricted a, Unrestricted b), while linear logic does not.

Subtyping. Because the type $A \multimap B$ only strengthens the contract of its elements compared to $A \to B$, one might expect the type $A \multimap B$ to be a subtype of $A \to B$. But while $\lambda^q_\to$ has *polymorphism*, it does not have *subtyping*. For example, if

```
f :: Int ⊸ Int
g :: (Int → Int) → Bool
```

then the call (g f) is ill-typed, even though f provides more guarantees than g requires. However, g might well be multiplicity-polymorphic, with type ∀p. (Int $\to_p$ Int) → Bool; in which case (g f) is, indeed, typeable. Alternatively, η-expansion to g (λx. f x) makes the expression typeable, as the reader may check.

The lack of subtyping is a deliberate choice in our design: it is well known that Hindley-Milner-style type inference does not mesh well with subtyping (see, for example, the extensive exposition by Pottier (1998), but also Dolan and Mycroft (2017) for a counterpoint). Hask-LL has limited support for subtyping: calls like (g f) are well-typed. But these are elaborated to their η-expansions in $\lambda^q_\to$.

Polymorphism. Consider the definition: "id x = x". Our typing rules would validate both id :: Int → Int and id :: Int ⊸ Int. So, since we think of multiplicities ranging over $\{1, \omega\}$, surely we should also have id :: ∀p. Int $\to_p$ Int? But as it stands, our rules do not accept it. To do so we would need $x :_p Int \vdash x : Int$. Looking at the (var) rule in Fig. 6, we can prove that premise by case analysis, trying $p = 1$ and $p = \omega$. But if we had a richer domain of multiplicities, including 0 (see Sec. 7.2), we would be able to prove $x :_p Int \vdash x : Int$, and rightly so because it is not the case that id :: Int $\to_0$ Int.

For now, we accept more conservative rules, in order to keep open the possiblity of extending the multiplicity domain later. There is an up-front cost to this: we have less polymorphism than we might expect.

4 IMPLEMENTING HASK-LL

We implement Hask-LL on top of the leading Haskell compiler, GHC, version 8.2[6]. The implementation modifies type inference and type-checking in the compiler. Neither the intermediate language (Sulzmann et al. 2007) nor the run-time system are affected. Our implementation of multiplicity polymorphism is incomplete, but the current prototype is sufficient for the examples and case studies presented in in this paper (see Sec. 5).

In order to implement the linear arrow, we added a multiplicity annotation to function arrows as in $\lambda^q_\to$. The constructor for arrow types is constructed and destructed frequently in GHC's type checker, and this accounts for most of the modifications to existing code.

As suggested in Sec. 3.2, the multiplicities are an output of the algorithm. In order to infer the multiplicities of variables in the branches of a **case** expression we need a way to join the output of the branch. We use a supremum operation on multiplicities where $1 \vee 0 = \omega$ (0 stands for a variable absent in a branch).

[6]URL suppressed for anonymous review

Implementing Hask-LL affects 1,152 lines of GHC (in subsystems of the compiler that together amount to more than 100k lines of code), including 444 net extra lines. These figures support our claim that Hask-LL is easy to integrate into an existing implementation: despite GHC being 25 years old, we implement a first version of Hask-LL with reasonable effort.

5 EVALUATION AND CASE STUDIES

While many linear type systems have been proposed, a *retrofitted* linear type system for a mature language like Haskell offers the opportunity to implement non-trivial applications mixing linear and non-linear code, I/O, etc., and observe how linear code interacts with existing libraries and the optimiser of a sophisticated compiler.

Our first method for evaluating the implementation is to simply compile a large existing code base together with the following changes: (1) all (non-GADT) data constructors are linear by default, as implied by the new type system; and (2) we update standard list functions to have linear types (++, concat, uncons). Under these conditions, we verified that the base GHC libraries and the nofib benchmark suites compile successfully: 195K lines of Haskell, providing preliminary evidence of backwards compatibility.

In the remainder of section, we describe case-studies implementated with the modified GHC of Sec. 4. In Sec. 7.3, we propose further applications for Hask-LL, which we have not yet implemented, but which motivate this work.

5.1 Computing directly with serialised data

While Sec. 2.2 covered simple mutable arrays, we now turn to a related but more complicated application: operating directly on binary, serialised representations of algebraic data-types (as in Vollmer et al. (2017)). The motivation is that programs are increasingly decoupled into separate (cloud) services that communicate via serialised data in text or binary formats, carried by remote procedure calls. The standard approach is to deserialise data into an in-heap, pointer-based representation, process it, and then serialise the result for transmission. This process is inefficient, but nevertheless tolerated, because the alternative — computing directly with serialised data — is far too difficult to program. Nevertheless, the potential performance gain of working directly with serialised data has motivated small steps in this direction: libraries like "Cap'N Proto" [7] enable unifying in-memory and on-the-wire formats for simple product types (protobufs).

Here is an unusual case where advanced types can yield *performance* by making it practical to code in a previously infeasible style: accessing serialised data at a fine grain without copying it.

The interface on the right gives an example of type-safe, *read-only* access to serialised data for a particular datatype. A Packed value is a pointer to raw bits (a bytestring), indexed by the types of the values contained within. We define a *type-safe* serialisation layer as one which *reads* byte-ranges only at the type and size they were originally *written*. This is a small extension of the memory safety we already expect of Haskell's heap — extended to include the contents of bytestrings containing serialised data[8]. To preserve this type safety, the

```
data Tree = Leaf Int | Branch Tree Tree
pack     :: Tree ⊸ Packed [Tree]
unpack   :: Packed [Tree] ⊸ Tree
caseTree :: Packed (Tree : r) →p
              (Packed (Int : r) →p  a) ⊸
              (Packed (Tree : Tree : r) →p  a) ⊸ a
```

[7] https://capnproto.org/

[8] The additional safety ensured here is lower-stakes than typical memory-safety, as, even it is violated, the serialised values do not contain pointers and cannot segfault the program reading them.

Packed type *must* be abstract. Consequently, a client of the module defining Tree need not be privy to the memory layout of its serialisation.

If we cannot muck about with the bits inside a Packed directly, then we can still retrieve data with unpack, *i.e.*, the traditional, *copying*, approach to deserialisation. Better still is to read the data *without* copying. We can manage this feat with caseTree, which is analogous to the expression "**case** e **of** {Leaf…; Branch…}". Lacking built-in syntax, (caseTree p k1 k2) takes two continuations corresponding to the two branches of the case expression. Unlike the case expression, caseTree operates on the packed byte stream, reads a tag byte, advances the pointer past it, and returns a type-safe pointer to the fields (*e.g.* Packed [Int] in the case of a leaf).

It is precisely to access multiple, consecutive fields that Packed is indexed by a *list* of types as its phantom type parameter. Individual atomic values (Int, Double, etc) can be read one at a time with a lower-level read primitive, which can efficiently read out scalars and store them in registers:

```
read :: Storable a ⇒ Packed (a : r) ⊸ (a, Packed r)
```

Putting it together, we can write a function that consumes serialised data, such as sumLeaves, shown on the right. Indeed, we can even use caseTree to implement unpack, turning it into safe "client code" – sitting outside the module that defines Tree and the trusted code establishing its memory representation.

```
sumLeaves :: Packed [Tree] → Int
sumLeaves p = fst (go p)
  where go p = caseTree p
                 read    -- Leaf case
                 (λp2 → let (n, p3) = go p2
                            (m, p4) = go p3
                        in (n + m, p4))
```

In this read-only example, linearity was not essential, only phantom types. Next we consider an API for writing Packed [Tree] values bit by bit, where linearity is key. In particular, can we also implement pack using a public interface?

5.1.1 Writing serialised data. To create a serialised data constructor, we must write a tag, followed by the fields. A *linear* write pointer can ensure all fields are initialised, in order. We use a type "Needs" for write pointers, parameterised by (1) a list of remaining things to be written, and (2) the type of the final value which will be initialised once those writes are performed. For example, after we write the tag of a Leaf we are left with: "Needs [Int] Tree" — an *obligation* to write the Int field, and a *promise* to receive a Tree value at the end (albeit a packed one).

To write an individual number, we provide a primitive that shaves one element off the type-level list of obligations (a counterpart to read, above): As with mutable arrays, this write operates in-place on the buffer, in spite being a pure function.

```
write :: Storable a ⇒ a ⊸ Needs (a : r) t ⊸ Needs r t
```

When the list of outstanding writes is empty, we can retrive a readable packed buffer. Just as when we froze arrays (Sec. 2.2), the immutable value is *unrestricted*, and can be used multiple times:

```
finish :: Needs [ ] t ⊸ Unrestricted (Packed [t])
```

Finalizing written values with finish works hand in hand with allocating new buffers in which to write data (similar to newMArray from Sec. 2.2):

```
newBuffer :: (Needs [a] a ⊸ Unrestricted b) ⊸ b
```

We also need to explicitly let go of linear input buffers we've exhausted.

```
done :: Packed [ ] ⊸ ()
```

The primitives write, read, newBuffer, done, and finish are *general* operations for serialised data, whereas caseTree is datatype-specific. Further, the module that defines Tree exports a datatype-specific way to *write* each serialised data constructor:

```
startLeaf   :: Needs (Tree : r) t ⊸ Needs (Int : r) t
startBranch :: Needs (Tree : r) t ⊸ Needs (Tree : Tree : r) t
```

Operationally, start∗ functions write only the tag, hiding the exact tag-encoding from the client, and leaving field-writes as future obligations. With these building blocks, we can move pack and unpack outside of the private code that defines Trees, which has this minimal interface:

```
module TreePrivate (Tree (..), caseTree, startLeaf, startBranch)
module Data.Packed (Packed, Needs, read, write, newBuffer, finish, done)
```

On top of the safe interface, we can of course define higher-level construction routines, such as for writing a complete Leaf:

```
writeLeaf n = write n ∘ startLeaf
```

Now we can allocate and initialize a complete tree — equivalent to Branch (Leaf 3) (Leaf 4), but without ever creating the non-serialised values — as follows:

```
newBuffer (finish ∘ writeLeaf 4 ∘ writeLeaf 3 ∘ startBranch) :: Packed [Tree]
```

Finally, we have what we need to build a map function that logically operates on the leaves of a tree, but reads serialised input and writes serialised output. Indeed, in our current Hask-LL implementation "mapLeaves (+1) tree" touches *only* packed buffers — it performs zero Haskell heap allocation! We will return to this map example and benchmark it in Sec. 5.1.3. With the safe interface to serialised data, functions like sumLeaves and mapLeaves are not burdensome to program. The code for mapLeaves is shown below.

```
module TreePublic (pack, unpack, writeLeaf, sumLeaves, mapLeaves)
 ...
mapLeaves :: (Int → Int) → Packed Tree ⊸ Packed Tree
mapLeaves fn pt = newBuffer (extract ∘ go pt)
  where
    extract (inp, outp) = case done inp of () → finish outp
    go :: Packed (Tree : r) ⊸ Needs (Tree : r) t ⊸ (Packed r, Needs r t)
    go p = caseTree p (λp o → let (x, p') = read p in (p', writeLeaf (fn x) o))
                      (λp o → let (p', o') = go p (writeBranch o) in go p' o')
```

5.1.2 A version without linear types. How would we build the same thing in Haskell without linear types? It may appear that the ST monad is a suitable choice:

```
writeST :: Storable a ⇒ a → Needs' s (a : r) t → ST s (Needs' s r t)
```

Here we use the same typestate associated with a Needs pointer, while also associating its mutable state with the ST session indexed by s. Unfortunately, not only do we have the same trouble with freezing in the absence of linearity (unsafeFreeze, Sec. 2.2), we also have an *additional* problem not present with arrays: namely, a non-linear use of a Needs pointer can ruin our type-safe deserialisation guarantee! For example, we can write a Leaf and a Branch to the same pointer in an interleaved fashion. Both will place a tag at byte 0; but the leaf will place an integer in bytes 1-9, while the branch will place another tag at byte 1. We can receive a corrupted 8-byte integer, clobbered by a tag from an interleaved "alternate future".

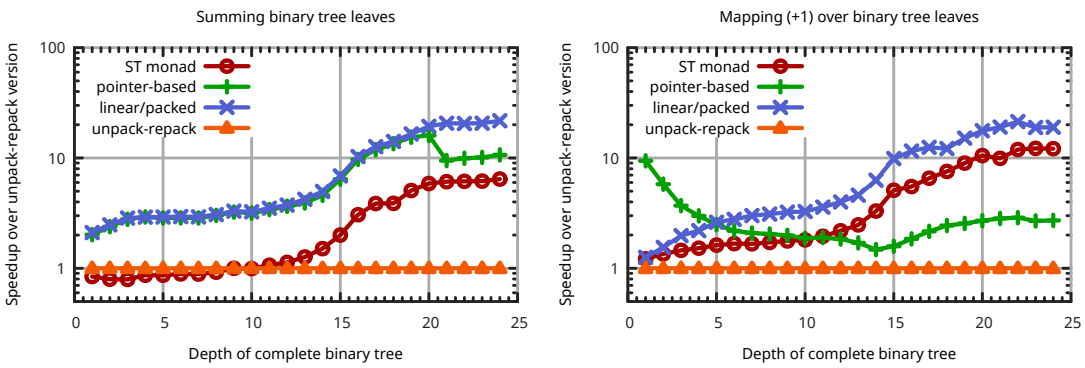

Fig. 7. Speedup of operating directly on serialised data, either using linear-types or the ST monad, as compared to fully unpacking, processing, and repacking the data. For reference, a "pointer-based" version is also included, which doesn't operate on serialised data at all, but instead normal heap objects — it represents the hypothetical performance of "unpack-repack" if (de)serialisation were instantaneous.

Fixing this problem would require switching to an indexed monad with additional type-indices that model the typestate of all accessible pointers, which would in turn need to have static, type-level identifiers. That is, it would require *encoding* linearity after all, but in a way which would become very cumbersome as soon as several buffers are involved.

5.1.3 Benchmarking compiler optimisations. Finally, as shown in Fig. 7, there are some unexpected performance consequences from using a linear versus a monadic, ST style in GHC. Achieving allocation-free loops in GHC is always a challenge — tuple types and numeric types are lazy and "boxed" as heap objects by default. As we saw in the sumLeaves and mapLeaves examples, each recursive call returned a tuple of a result and a new pointer. In a monadic formulation, an expression of type m a, for Monad m, implies that the "result" type a, of kind $*$, must be a *lifted* type. Nevertheless, in some situations, for some monads, the optimiser is able to deforest data constructors returned by monadic actions. In the particular case of fold and map operations over serialised trees, unfortunately, we are currently unable to eliminate all allocation from ST-based implementations of the algorithms.

For the linearly-typed code, however, we have more options. GHC has the ability to directly express unboxed values such as a tuple (#Int#, Double # #), which fills two registers and inhabits an unboxed kind distinct from $*$. In fact, the type of a combinator like caseTree is a good fit for the recent "levity polymorphism" addition to GHC (Eisenberg and Peyton Jones 2017). Thus we permit the branches of the **case** to return unlifted, unboxed types, and give caseTree a more general type:

```
caseTree :: ∀(rep :: RuntimeRep) (res :: TYPE rep) b.
  Packed (Tree : b) → (Packed (Int : b) ⊸ res) → (Packed (Tree : Tree : b) ⊸ res) → res
```

This works because we do not need to *call* a function with res as argument (and thus unknown calling conventions) only return it. Using this approach, we were able to ensure by construction that the "linear/packed" implementations in Fig. 7 were completely non-allocating, rather than depending on the optimiser. This results in better performance for the linear, compared to monadic version of the serialised-data transformations.

The basic premise of Fig. 7 is that a machine in the network receives, processes, and transmits serialized data (trees). We consider two simple benchmarks: sumLeaves and mapTree (+1). The

baseline is the traditional approach: deserialise, transform, and reserialise, the "unpack-repack" line in the plots. Compared to this baseline, *processing the data directly in its serialised form results in speedups of over 20× on large trees*. What linear types makes safe, is also efficient.

The experiment was conducted on a Xeon E5-2699 CPU (2.30GHz, 64GB memory) using our modified version of GHC 8.2 (Sec. 4). Each data point was measured by performing many trials and taking a linear regression of iteration count against time[9]. This process allows for accurate measurements of both small and large times. The baseline unpack-repack tree-summing times vary from 25ns to 1.9 seconds at depths 1 and 24 respectively. Likewise, the baseline mapping times vary from 215ns to 2.93 seconds. We use a simple contiguous implementation of buffers for serialisation[10]. At depth 20, one copy of the tree takes around 10MB, and towards the right half of the plot we see tree size exceeding cache size.

5.2 Sockets with type-level state

The BSD socket API is a standard, if not *the* standard, through which computers connect over networks. It involves a series of actions which must be performed in order: on the server-side, a freshly created socket must be *bound* to an address, then start *listening* incoming traffic, then *accept* connection requests; said connection is returned as a new socket, this new socket can now *receive* traffic. One reason for having that many steps is that the precise sequence of actions is protocol-dependent. For TCP traffic you would do as described, but for UDP, which does not need connections, you would not accept a connection but receive messages directly.

The `socket` library for Haskell, exposes precisely this sequence of actions. Programming with it is exactly as clumsy as socket libraries for other languages: after each action, the state of the socket changes, as do the permissible actions, but these states are invisible in the types. Better is to track the state of sockets in the type, akin to a typestate analysis (Strom 1983). In the File API of Sec. 2.3, we made files safer to use at the cost of having to thread a file handle explicitely: each function consumes a file handle and returns a fresh one. We can make this cost into an opporunity: we have the option of returning a handle *with a different type* from that of the handle we consumed! So by adjoining a phantom type to sockets to track their state, we can effectively encode the proper sequencing of socket actions.

As an illustration, we implemented wrapper around the API of the `socket` library. For concision, this wrapper is specialised for the case of TCP.

```
data State = Unbound | Bound | Listening | Connected
data Socket (s :: State)
data SocketAddress

socket :: IO_L 1 (Socket Unbound)
bind :: Socket Unbound ⊸ SocketAddress → IO_L 1 (Socket Bound)
listen :: Socket Bound ⊸ IO_L 1 (Socket Listening)
accept :: Socket Listening ⊸ IO_L 1 (Socket Listening, Socket Connected)
connect :: Socket Unbound ⊸ SocketAddress → IO_L 1 (Socket Connected)
send :: Socket Connected ⊸ ByteString → IO_L 1 (Socket Connected, Unrestricted Int)
receive :: Socket Connected ⊸ IO_L 1 (Socket Connected, Unrestricted ByteString)
close :: ∀s. Socket s → IO_L ω ()
```

[9] using the criterion library (O'Sullivan 2013)
[10] A full, practical implementation should include growable or doubling buffers.

This linear socket API is very similar to that of files: we use the IO_L monad in order to enforce linear use of sockets. The difference is the argument to Socket, which represents the current state of the socket and is used to limit the functions which apply to a socket at a given time.

Implementing the linear socket API. Our socket API has been tested by writing a small echo-server. The API is implemented as a wrapper around the `socket` library. Each function wrapped takes half-a-dozen lines of code, of type annotation and coercions between IO and IO_L[11]. There is no computational behaviour besides error recovery.

It would have been too restrictive to limit the typestate to enforce the usage protocol of TCP. We do not intend for a new set of wrapper functions to be implemented for each protocol. Instead the wrappers are implemented with a generic type-state evolving according to the rules of a deterministic automaton. Each protocol can define it's own automaton, which is represented as a set of instances of a type class.

5.3 Pure bindings to impure APIs

In Haskell SpriteKit, Chakravarty and Keller (2017) have a different kind of problem. They build a pure interface for graphics, in the same style as the Elm programming language (Czaplicki 2012), but implement it in terms of an existing imperative graphical interface engine.

Basically, the pure interface takes an update function $u : \mathsf{Scene} \to \mathsf{Scene}$ which is tasked with returning the next state that the screen will display. The scene is first converted to a pure tree where each node keeps, along with the pure data, a pointer to its imperative counterpart when it applies, or Nothing for new nodes.

```
data Node = Node {payload :: Int, ref :: Maybe (IORef ImperativeNode), children :: [Node]}
```

On each frame, SpriteKit applies u to the current scene, and checks if a node n was updated. If it was, it applies the update directly onto ref n or creates a new imperative node.

Things can go wrong though: if the update function *duplicates* any proxy node, one gets the situation where two nodes n and n' can point to the same imperative source ref n = ref n', but have different payloads. In this situation the Scene has become inconsistent and the behaviour of SpriteKit is unpredictable.

In the API of Chakravarty and Keller (2017), the burden of checking non-duplication is on the programmer. Using linear types, we can switch that burden to the compiler: we change the update function to type Scene $\multimap$ Scene, and the ref field is made linear too. Thanks to linearity, no reference can be duplicated: if a node is copied, the programmer must choose which one will correspond to the old imperative counterpart and which will be new.

We implemented such an API in our implementation of Hask-LL. The library-side code does not use any linear code, the Nodes are actually used unrestrictedly. Linearity is only imposed on the user of the interface, in order to enforced the above restriction.

6 RELATED WORK

6.1 Linearity via arrows vs. linearity via kinds

There are two possible choices to indicate the distinction between linear and unrestricted objects. Our choice is to use the arrow type. That is, we have both a linear arrow to introduce linear objects in the environment, and an unrestricted arrow to introduce unrestricted objects. This choice is featured in the work of McBride (2016) and Ghica and Smith (2014) and is ultimately inspired by

[11] Since our implementation of Hask-LL does not yet have multiplicity-polymorphism, we had to fake it with type families and GADTs

Girard's presentation of linear logic, which features only linear arrows, and where the unrestricted arrow $A \to B$ is encoded as $!A \multimap B$.

Another popular choice (Mazurak et al. 2010; Morris 2016; Tov and Pucella 2011; Wadler 1990) is to separate types into two kinds: a linear kind and an unrestricted kind. Values with a type whose kind is linear are linear, and the others are unrestricted. (Thus in particular such systems feature "linear arrows", but they have a completely different interpretation from ours.) This choice is attractive on the surface because, intuitively, some types are inherently linear (file handles, updateable arrays, etc.) and some types are inherently unrestricted (Int, Bool, etc.). However, after scratching the surface we have discovered that "linearity via arrows" has an edge over "linearity via kinds".

Better code reuse. When retrofitting linear types in an existing language, it is important to share has much code as possible between linear and non-linear code. In a system with linearity on arrows, the subsumption relation (linear arrows subsume unrestricted arrows) and the scaling of context in the application rule mean that much linear code can be used as-is from unrestricted code, and be properly promoted. Indeed, assuming lists as defined in Sec. 2.4 and:

```
(++) :: [a] ⊸ [a] ⊸ [a]    -- Append two lists
cycle :: [a] → [a]          -- Repeat a list, infinitely
```

The following definition type-checks, even though ++ is applied to unrestricted values and used in an unrestricted context.

```
f :: [a] → [a] → [a]
f xs ys = cycle (xs ++ ys)
```

In contrast, in a two-kind system, a function must declare the *exact* linearity of its return value. Consequently, to make a function promotable from linear to unrestriced, its declaration must use polymorphism over kinds. We show how this may look like below; but first we need to discuss data types.

As seen in Sec. 2, in Hask-LL the reuse of linear code extends to data types: the usual parametric data types (lists, pairs, etc.) work both with linear and unrestricted values. On the contrary, if linearity depends on the kind, then if a linear value is contained in a type, the container type must be linear too. (Indeed, an unrestricted container could be discarded or duplicated, and its contents with it.) Consequently, sharing data types also requires polymorphism. For example, in a two-kinds system, the List type may look like so, if one assumes a that Type 1 is the kind of linear types and Type ω is the kind of unrestricted types.

```
data List (p :: Multiplicity) (a :: Type p) :: Type p = [] | a : (List p m a)
```

The above declaration ensures that the linearity of the list inherits the linearity of the contents. A linearity-polymorphic (++) function could have the definition, assuming the $(\wedge)$ operator takes the minimum of multiplicities.

```
(++) :: List p (a p) → List q (a q) → List (p ∧ q) (a (p ∧ q))
[] ++ xs = xs
(x : xs) ++ ys = x : (xs ++ ys)
```

The above type ensures that one can mix multiplicities freely between the arguments; but the result must be linear if any argument is linear. However, the definition is valid only if a q is a subtype of a (p ∧ q) for any type family a :: (p :: Multiplicitiy) → Type p. Thus, code-sharing requires not only polymorphism, but a non-trivial subtyping and subkinding system.

Note that, in the above, we parameterize over multiplicities instead of parameterizing over kinds directly, as is customary in the literature. We do so because it fits better GHC, whose kinds are already parameterized over a so-called levity (Eisenberg and Peyton Jones 2017).

Dependent types. Linearity on the arrow meshes better with dependent types (see Sec. 7.2). Indeed, consider a typical predicate over files (P : File → ∗). It will need to mention its argument several times to relate the file at the start and at the start of a sequence of operations. While this is not a problem in our system, the function P is not expressible if File is intrinsically linear. Leaving the door open to dependent types is crucial to us, as this is currently explored as a possible extension to GHC.

Yet, an advantage of "linearity via kinds" is the possibility to directly declare the linearity of values returned by a function – not just that of the argument of a function. In contrast, in our system if one wants to indicate that a returned value is linear, we have to use a double-negation trick. That is, given $f : A \to (B \multimap !r) \multimap r$, then B can be used a single time in the (single) continuation, and effectively f "returns" a single B. One can obviously declare a type for linear values Linear a = (a ⊸ !r) ⊸ r and chain Linear-returning functions with appropriate combinators. In fact, as explained in Sec. 2.7, the cost of the double negation almost entirely vanishes in the presence of an ambient monad.

6.2 Other variants of "linearity on the arrow"

The $\lambda^q_\to$ type system is heavily inspired from the work of Ghica and Smith (2014) and McBride (2016). Both of them present a type system where arrows are annotated with the multiplicty of the the argument that they require, and where the multiplicities form a semi-ring.

In contrast with $\lambda^q_\to$, McBride uses a multiplicity-annotated type judgement $\Gamma \vdash_\rho t : A$, where ρ represents the multiplicity of t. So, in McBride's system, when an unrestricted value is required, instead of computing $\omega\Gamma$, it is enough to check that $\rho = \omega$. The problem is that this check is arguably too coarse, and results in the judgement $\vdash_\omega \lambda x.(x, x) : A \multimap (A, A)$ being derivable. This derivation is not desirable: it implies that there cannot be reusable definitions of linear functions. In terms of linear logic (Girard 1987), McBride makes the natural function of type $!(A \multimap B) \Longrightarrow !A \multimap !B$ into an isomorphism. In that respect, our system is closer to Ghica and Smith's.

The essential differences between our system and that of Ghica and Smith is that we support multiplicity-polymorphism and datatypes. In particular our **case** rule is novel.

The literature on so-called coeffects (Brunel et al. 2014; Petricek et al. 2013) uses type systems similar to Ghica and Smith, but with a linear arrow and multiplicities carried by the exponential modality instead. Brunel et al. (2014), in particular, develops a Krivine-style realisability model for such a calculus. We are not aware of an account of Krivine realisability for lazy languages, hence this work is not directly applicable to $\lambda^q_\to$.

6.3 Uniqueness and ownership typing

The literature contains many proposals for uniqueness (or ownership) types (in contrast with linear types). Prominent representative languages with uniqueness types include Clean (Barendsen and Smetsers 1996) and Rust (Matsakis and Klock 2014). Hask-LL, on the other hand, is designed around linear types based on linear logic (Girard 1987).

Linear types and uniqueness types are, at their core, dual: whereas a linear type is a contract that a function uses its argument exactly once even if the call's context can share a linear argument as many times as it pleases, a uniqueness type ensures that the argument of a function is not used anywhere else in the expression's context even if the callee can work with the argument as it pleases. Seen as a system of constraints, uniqueness typing is a *non-aliasing analysis* while

linear typing provides a *cardinality analysis*. The former aims at in-place updates and related optimisations, the latter at inlining and fusion. Rust and Clean largely explore the consequences of uniqueness on in-place update; an in-depth exploration of linear types in relation with fusion can be found in Bernardy et al. (2015); see also the discussion in Sec. 7.1.

Because of this weak duality, we could have retrofitted uniqueness types to Haskell. But several points guided our choice of designing Hask-LL around linear logic rather than uniqueness types: (a) functional languages have more use for fusion than in-place update (if the fact that GHC has a cardinality analysis but no non-aliasing analysis is any indication); (b) there is a wealth of literature detailing the applications of linear logic — see Sec. 5; (c) and decisively, linear type systems are conceptually simpler than uniqueness type systems, giving a clearer path to implementation in GHC.

Rust & Borrowing. In Hask-LL we need to thread linear variables throughout the program (consider using several functions of type $\mathsf{T} \multimap \mathsf{T}$). Even though this burdend could be alleviated using syntactic sugar, Rust uses instead a type-system feature for this purpose: *borrowing*.

Borrowed values differ from owned values in that they can be used in an unrestricted fashion, albeit in a *delimited scope*.

Borrowing does not come without a cost, however: if a function `f` borrows a value `v` of type `T`, then the caller of the function *must* retain `v` alive until `f` has returned; the consequence is that Rust cannot, in general, perform tail-call elimination, crucial to the operation behaviour of many functional programs, as some resources must be released *after* `f` has returned.

The reason that Rust programs depend so much on borrowing is that unique values are the default. Hask-LL aims to hit a different point in the design space where regular non-linear expressions are the norm, yet gracefully scaling up investing extra effort to enforce linearity invariants is possible. Nevertheless, we discuss in Sec. 7.2 how to extend Hask-LL with borrowing.

6.4 Linearity via monads

Launchbury and Peyton Jones (1995) taught us a conceptually simple approach to lifetimes: the ST monad. It has a phantom type parameter `s` (the *region*) that is instantiated once at the beginning of the computation by a `runST` function of type:

```
runST :: (∀s. ST s a) → a
```

This way, resources that are allocated during the computation, such as mutable cell references, cannot escape the dynamic scope of the call to `runST` because they are themselves tagged with the same phantom type parameter.

Region-types. With region-types such as ST, we cannot express typestates, but this is sufficient to offer a safe API for freezing array or ensuring that files are eventually closed. This simplicity (one only needs rank-2 polymorphism) comes at a cost: we've already mentionned in Sec. 2.2 that it forces operations to be more sequentialised than need be, but more importantly, it does not support *prima facie* the interaction of nested regions.

Kiselyov and Shan (2008) show that it is possible to promote resources in parent regions to resources in a subregion. But this is an explicit and monadic operation, forcing an unnatural imperative style of programming where order of evaluation is explicit. The HaskellR project (Boespflug et al. 2014) uses monadic regions in the style of Kiselyov and Shan to safely synchronise values shared between two different garbage collectors for two different languages. Boespflug et al. report that custom monads make writing code at an interactive prompt difficult, compromises code reuse, forces otherwise pure functions to be written monadically and rules out useful syntactic facilities like view patterns.

In contrast, with linear types, values in two regions hence can safely be mixed: elements can be moved from one data structure (or heap) to another, linearly, with responsibility for deallocation transferred along.

Idris's dependent indexed monad. To go beyond simple regions, Idris introduces a generic way to add typestate on top of a monad, the ST indexed monad transformer[12]. The basic idea is that everything which must be single-threaded – and that we would track with linearity – become part of the state of the monad. For instance, coming back to the sockets of Sec. 5.2, the type of bind would be as follows:

```
bind :: (sock :: Var) → SocketAddress → ST IO () [sock ::: Socket Unbound :↦ Socket Bound]
```

Where sock is a reference into the monads's state, and Socket Unbound is the type of sock before bind, and Socket Bound, the type of sock after bind.

Idris uses its dependent types to associate a state to the value of its first argument. Dependent types are put to even greater use for error management where the state of the socket depends on whether bind succeeded or not:

```
  -- In Idris, bind uses a type-level function (or) to handle errors
bind :: (sock :: Var) → SocketAddress →
      ST IO (Either () ()) [sock ::: Socket Unbound :↦ (Socket Bound `or` Socket Unbound)]
  -- In Hask-LL, by contrast, the typestate is part of the return type
bind :: Socket Unbound ⊸ SocketAddress → Either (Socket Bound) (Socket Unbound)
```

The support for dependent types in GHC is not as comprehensive as Idris's. But it is conceivable to implement such an indexed monad transformer in Haskell. However, this is not an easy task, and we can anticipate that the error messages would be hard to stomach.

7 FUTURE WORK

7.1 Controlling program optimisations

Inlining is a cornerstone of program optimisation, exposing opportunities for many program transformations. Yet not every function can be inlined without negative effects on performance: inlining a function with more than one use sites of the argument may result in duplicating a computation. For example one should avoid the following reduction: $(\lambda x \to x + x)$ expensive $\longrightarrow$ expensive + expensive.

Many compilers can discover safe inlining opportunities by analysing source code and determine how many times functions use their arguments. (In GHC it is called the cardinality analysis (Sergey et al. 2014)). A limitation of such an analysis is that it is necessarily heuristic (the problem is undecidable for Haskell). Because inlining is crucial to efficiency, programmers find themselves in the uncomfortable position of relying on a heuristic to obtain efficient programs. Consequently, a small, seemingly innocuous change can prevent a critical inlining opportunity and have rippling catastrophic effects throughout the program. Such unpredictable behaviour justifies the folklore that high-level languages should be abandoned to gain precise control over program efficiency.

A remedy is to use the multiplicity annotations of $\lambda^q_\to$ as cardinality *declarations*. Formalising and implementing the integration of multiplicity annotations in the cardinality analysis is left as future work.

[12] See *e.g.* http://docs.idris-lang.org/en/latest/st/index.html. Where you will also discover that ST is actually defined in terms of a more primitive STrans

7.2 Extending multiplicities

For the sake of this article, we use only multiplicities 1 and ω. But in fact $\lambda^q_{\to}$ can readily be extended to more, following Ghica and Smith (2014) and McBride (2016). The general setting for $\lambda^q_{\to}$ is an ordered-semiring of multiplicities (with a join operation for type inference). In particular, in order to support dependent types, we additionally need a 0 multiplicity. We may may want to add a multiplicity for affine arguments (*i.e.* arguments which can be used *at most once*).

The typing rules are mostly unchanged with the *caveat* that case_π must exclude $\pi = 0$ (in particular we see that we cannot substitute multiplicity variables by 0). The variable rule becomes:

$$\frac{x :_1 A \leqslant \Gamma}{\Gamma \vdash x : A}$$

Where the order on contexts is the point-wise extension of the order on multiplicities.

In Sec. 6.3, we have considered the notion of *borrowing*: delimiting life-time without restricting to linear usage. This seems to be a useful pattern, and we believe it can be encoded as an additional multiplicity as follows: let β be an additional multiplicity with the following characteristics:

- $1 < \beta < \omega$
- $\beta + \beta = 1 + \beta = 0 + \beta = 1 + 1 = \beta$

That is, β supports contraction and weakening but is smaller than ω. We can then introduce a value with an explicit lifetime with the following pattern

```
borrow :: (T →β  Unrestricted a) →β  Unrestricted a
```

The borrow function makes the life-time manifest in the structure of the program. In particular, it is clear that calls within the argument of borrow are not tail: a shortcoming of borrowing that we mentioned in Sec. 6.3.

7.3 Future industrial applications

Our own work in an industrial context triggered our efforts to add linear types to GHC. We were originally motivated by precisely typed protocols for complex interactions and by taming GC latencies in distributed systems. But we have since noticed other potential applications of linearity in a variety of other industrial projects.

Streaming I/O Program inputs and outputs are frequently much larger than the available RAM on any single node. Rather than building complex pipelines with brittle explicit loops copying data piecemeal to spare our precious RAM, one approach is to compose combinators that transform, split and merge data wholemeal but in a streaming fashion. These combinators manipulate first-class *streams* and guarantee bounded memory usage, as in the below infinitely running echo service:

```
receive :: Socket → IOStream Msg
send :: Socket → IOStream Msg → IO ()

echo isock osock = send osock (receive isock)
```

However, reifying sequences of IO actions (socket reads) in this way runs the risk that effects might be duplicated inadvertently. In the above example, we wouldn't want to inadvertently hand over the receive stream to multiple consumers, or the abstraction of wholemeal I/O programming would be broken (like in Lippmeier et al. (2016, Section 2.2)), because neither consumer would ultimately see the same values from the stream. If say one consumer reads in the stream first, the second consumer would see an empty stream — not what the first consumer saw.

We have seen this very error several times in industrial projects, where the symptoms are bugs whose root cause are painful to track down. A linear type discipline would prevent such bugs.

Programming foreign heaps Complex projects with large teams invariably involve a mix of programming languages. Reusing legacy code is often much cheaper than reimplementing it. A key to successful interoperation between languages is performance. If all code lives in the same address space, then data need not be copied as it flows from function to function implemented in multiple programming languages. Trouble is, language A needs to tell language B what objects in language A's heap still have live references in the call stack of language B to avoid too eager garbage collection.

For instance, users of `inline-java` call the JVM from Haskell via the JNI. The JVM implicitly creates so-called *local references* any time we request a Java object from the JVM. The references count as GC roots that prevent eager garbage collection. For performance, local references have a restricted scope: they are purely thread-local and never survive the call frame in which they were created. Both restrictions to their use can be enforced with linear types.

Remote direct memory access Section 5.1 is an example of an API requiring destination-passing style. This style often appears in performance-sensitive contexts. One notable example from our industrial experience is RDMA (Remote Direct Memory Access), which enables machines in high-performance clusters to copy data from the address space in one process to that of a remote process directly, bypassing the kernel and even the CPU, thereby avoiding any unnedded copy in the process.

One could treat a remote memory location as a low-level resource, to be accessed using an imperative API. Using linear types, one can instead treat it as a high-level value which can be written to directly (but exactly once). Using linear types the compiler can ensure that, as soon as the writing operation is complete, the destination computer is notified.

8 CONCLUSION

This article demonstrates how an existing lazy language, such as Haskell, can be extended with linear types, without compromising the language, in the sense that:

- existing programs are valid in the extended language *without modification*,
- such programs retain the same operational semantics, and in particular
- the performance of existing programs is not affected,
- yet existing library functions can be reused to serve the objectives of resource-sensitive programs with simple changes to their types, and no code duplication.

In other words: regular Haskell comes first. Additionally, first-order linearly typed functions and data structures are usable directly from regular Haskell code. In such a setting their semantics is that of the same code with linearity erased.

Hask-LL was engineered as an unintrusive design, making it tractable to integrate to an existing, mature compiler with a large ecosystem. We have developed a prototype implementation extending GHC with multiplicities. As we hoped, this design integrates well in GHC.

Even though we change only GHC's type system, we found that the compiler and runtime already had the features necessary for unboxed, off-heap, and in-place data structures. That is, GHC has the low-level compiler primitives and FFI support to implement, for example, mutable arrays, mutable cursors into serialised data, or off-heap foreign data structures without garbage collection. These features could be used *before* this work, but their correct use put some burden-of-proof on the programmers. Linearity unlocks these capabilities for safe, compiler-checked use, within pure code.

REFERENCES

Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. 2010. Monads Need Not Be Endofunctors. In *Foundations of Software Science and Computational Structures, 13th International Conference, FOSSACS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. 297–311. DOI: http://dx.doi.org/10.1007/978-3-642-12032-9_21

Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. 2016. Cogent: Verifying High-Assurance File System Implementations. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. Atlanta, GA, USA, 175–188. DOI: http://dx.doi.org/10.1145/2872362.2872404

Jean-Marc Andreoli. 1992. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation* 2, 3 (1992), 297–347.

Erik Barendsen and Sjaak Smetsers. 1996. Uniqueness Typing for Functional Languages with Graph Rewriting Semantics. *Mathematical Structures in Computer Science* 6, 6 (1996), 579–612.

Jean-Philippe Bernardy, Víctor López Juan, and Josef Svenningsson. 2015. Composable Efficient Array Computations Using Linear Types. (2015). Submitted to ICFP 2015. http://www.cse.chalmers.se/ josefs/publications/vectorcomp.pdf.

Mathieu Boespflug, Facundo Dominguez, Alexander Vershilov, and Allen Brown. 2014. Project H: Programming R in Haskell. (2014). Talk at IFL 2014.

Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. 2014. A Core Quantitative Coeffect Calculus. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*. Springer-Verlag New York, Inc., New York, NY, USA, 351–370. DOI: http://dx.doi.org/10.1007/978-3-642-54833-8_19

Manuel M. T. Chakravarty and Gabriele Keller. 2017. Haskell SpriteKit – Transforming an Imperative Object-oriented API into a Purely Functional One. (2017). http://www.cse.unsw.edu.au/~chak/papers/CK17.html

Evan Czaplicki. 2012. *Elm: Concurrent FRP for functional guis*. Senior thesis. Harvard University.

Stephen Dolan and Alan Mycroft. 2017. Polymorphism, Subtyping, and Type Inference in MLsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 60–72. DOI: http://dx.doi.org/10.1145/3009837.3009882

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. DOI: http://dx.doi.org/10.1145/3062341.3062357

Dan R. Ghica and Alex I. Smith. 2014. Bounded Linear Types in a Resource Semiring. 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, Grenoble, France, April 5-13, 2014, Proceedings*. 331–350. DOI: http://dx.doi.org/10.1007/978-3-642-54833-8_18

Jean-Yves Girard. 1987. Linear logic. *Theoretical Computer Science* 50, 1 (1987), 1–101.

Carl A. Gunter and Didier Rémy. 1993. *A proof-theoretic assessment of runtime type errors*. Technical Report. AT&T Bell laboratories. Technical Memo 11261-921230-43TM.

Oleg Kiselyov and Chung-chieh Shan. 2008. Lightweight Monadic Regions. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell (Haskell '08)*. ACM, New York, NY, USA, 1–12. DOI: http://dx.doi.org/10.1145/1411286.1411288

John Launchbury. 1993. A Natural Semantics for Lazy Evaluation. In *POPL*. 144–154.

John Launchbury and Simon L. Peyton Jones. 1995. State in Haskell. *LISP and Symbolic Computation* 8, 4 (1995), 293–341. DOI: http://dx.doi.org/10.1007/BF01018827

Ben Lippmeier, Fil Mackay, and Amos Robinson. 2016. Polarized data parallel data flow. In *Proceedings of the 5th International Workshop on Functional High-Performance Computing*. ACM, 52–57.

Nicholas D. Matsakis and Felix S. Klock, II. 2014. The Rust Language. *Ada Lett.* 34, 3 (Oct. 2014), 103–104. DOI: http://dx.doi.org/10.1145/2692956.2663188

Karl Mazurak, Jianzhou Zhao, and Steve Zdancewic. 2010. Lightweight linear types in system f. In *Proceedings of the 5th ACM SIGPLAN workshop on Types in language design and implementation*. ACM, 77–88.

Conor McBride. 2016. *I Got Plenty o' Nuttin'*. Springer International Publishing, Cham, 207–233. DOI: http://dx.doi.org/10.1007/978-3-319-30936-1_12

J. Garrett Morris. 2016. The best of both worlds: linear functional programming without compromise. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. 448–461. DOI: http://dx.doi.org/10.1145/2951913.2951925

Bryan O'Sullivan. 2013. The Criterion benchmarking library. (2013). http://github.com/bos/criterion

Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2013. *Coeffects: Unified Static Analysis of Context-Dependence*. Springer Berlin Heidelberg, Berlin, Heidelberg, 385–397. DOI: http://dx.doi.org/10.1007/978-3-642-39212-2_35

François Pottier. 1998. *Type Inference in the Presence of Subtyping: from Theory to Practice.* Research Report RR-3483. INRIA. https://hal.inria.fr/inria-00073205

Ilya Sergey, Dimitrios Vytiniotis, and Simon Peyton Jones. 2014. Modular, Higher-order Cardinality Analysis in Theory and Practice. *SIGPLAN Not.* 49, 1 (Jan. 2014), 335–347. DOI: http://dx.doi.org/10.1145/2578855.2535861

Robert E Strom. 1983. Mechanisms for compile-time enforcement of security. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages.* ACM, 276–284.

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. 2007. System F with Type Equality Coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI '07).* ACM, New York, NY, USA, 53–66. DOI: http://dx.doi.org/10.1145/1190315.1190324

Jesse A Tov and Riccardo Pucella. 2011. Practical affine types. In *POPL*. ACM, 447–458.

Michael Vollmer, Sarah Spall, Buddhika Chamith, Laith Sakka, Chaitanya Koparkar, Milind Kulkarni, Sam Tobin-Hochstadt, and Ryan R. Newton. 2017. Compiling Tree Transforms to Operate on Packed Representations. In *31st European Conference on Object-Oriented Programming (ECOOP 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Peter Müller (Ed.), Vol. 74. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 26:1–26:29. DOI: http://dx.doi.org/10.4230/LIPIcs.ECOOP.2017.26

Philip Wadler. 1990. Linear types can change the world. In *Programming Concepts and Methods*, M Broy and C B Jones (Eds.). North-Holland.