Retrofitting Linear Types

JEAN-PHILIPPE BERNARDY, Gothenburg University
MATHIEU BOESPFLUG, Tweag I/O
RYAN R. NEWTON, Indiana University
SIMON PEYTON JONES, Microsoft Research
ARNAUD SPIWACK, Tweag I/O

Linear and affine 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 introduce and 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 data types into linear and non-linear counterparts, we instead attach linearity to *binders*. Linear function types can receive inputs from linearly-bound values, but can also operate over unrestricted, regular values.

Linear types are an enabling tool for safe and resource efficient systems programming. We explore the power of linear types with a case study of a large, in-memory data structures that must serve responses with low latency.

CCS Concepts: •**Software and its engineering** → **Language features;** *Functional languages; Formal language definitions;*

Additional Key Words and Phrases: Haskell, laziness, linear logic, Linear types, systems programming

ACM Reference format:
Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2010. Retrofitting Linear Types. *PACM Progr. Lang.* 9, 4, Article 39 (March 2010), 27 pages.
DOI: 0000001.0000001

1 INTRODUCTION

Can we *safely* use Haskell to implement a low-latency server that caches a large dataset in-memory? Today, the answer is "no"[12], because pauses incurred by garbage collection (GC), observed in the order of 50ms or more, are unacceptable. Pauses are in general proportional to the size of the heap. Even if the GC is incremental, the pauses are unpredicable, difficult to control by the programmer and induce furthermore for this particular use case a tax on overall throughput. The problem is: the GC is not *resource efficient*, meaning that resources aren't always freed as soon as they could be. Programmers can allocate such large, long-lived data structures in manually-managed off-heap memory, accessing it through FFI calls. Unfortunately this common technique poses safety risks: space leaks (by failure to deallocate at all), as well use-after-free or double-free errors just like programming in plain C. The programmer has then bought resource efficiency but at the steep price of giving up on resource safety. If the type system was *resource-aware*, a programmer wouldn't have to choose.

This work has received funding from the European Commission through the SAGE project (grant agreement no. 671500).

DOI: 0000001.0000001

It is well known that type systems *can* be useful for controlling resource usage, not just ensuring correctness. Affine types [43], linear types [23, 47], permission types [48] and capabilities [3, 10] enable resource safe as well resource efficient handling of scarce resources such as sockets and file handles. All these approaches have been extensively studied, *yet these ideas have had relatively little effect on programming practice*. Few full-scale languages are designed from the start with such features. Rust is the major exception [31], and in Rust we see one of the attendant complications: advanced resource-tracking features puts a burden on new users, who need to learn how to satisfy the "borrow checker".

We present in this paper the first type system that we believe has a good chance of influencing existing functional programs and libraries to become more resource efficient. Whereas in Rust new users and casual programmers have to buy the whole enchilada, we seek to devise a language that is resource safe always and resource efficient sometimes, only when and where the programmer chooses to accept that the extra burden of proof is worth the better performance. Whereas other languages make resource efficiency *opt-out*, we seek to make it *opt-in*. We do this by retrofitting a *backward-compatible extension* to a Haskell-like language. Existing functions continue to work, although they may now have more refined types that enable resource efficiency; existing data structures continue to work, and can additionally track resource usage. Programmers who do not need resource efficiency should not be tripped up by it. They need not even know about our type system extension. We make the following specific contributions:

- We present a design for Hask-LL, which offers linear typing in Haskell in a fully backward-compatible way (Section 2). In particular, existing Haskell data types can contain linear values as well as non-linear ones; and linear functions can be applied to non-linear values. Most previous designs force an inconveniently sharp distinction between linear and non-linear code (Section 6). Interestingly, our design is fully compatible with laziness, which has typically been challenging for linear systems because of the unpredictable evaluation order of laziness [47].
- We formalise Hask-LL as $\lambda^q_\to$, a linearly-typed extension of the λ-calculus with data types (Section 3). We provide its type system, highlighting how it is compatible with existing Haskell features, including some popular extensions. The type system of $\lambda^q_\to$ has a number of unusual features, which together support backward compatibility with Haskell: linearity appears only in bindings and function arrows, rather than pervasively in all types; we support linearity polymorphism (Section 2.3); and the typing rule for `case` is novel (Section 3.3). No individual aspect is entirely new (Section 6.7), but collectively they add up to an unintrusive system that can be used in practice and scales up to a full programming language implementation with a large type system.
- We provide a dynamic semantics for $\lambda^q_\to$, combining laziness with explicit deallocation of linear data (Section 4). We prove type safety, of course. But we also prove that the type system guarantees the key memory-management properties that we seek: that every linear value is eventually deallocated by the programmer, and is never referenced after it is deallocated.
- Type inference, which takes us from the implicitly-typed source language, Hask-LL, to the explicitly-typed intermediate language $\lambda^q_\to$, is not covered this paper. However, we have implemented a proof of concept, by modifying the leading Haskell compiler GHC to infer linear types. The prototype is freely available[1], and provides strong evidence that it is not hard to extend GHC's full type inference algorithm (despite its complexity) with linear types.

Our work is directly motivated by the needs of large-scale low-latency applications in industrial practice. In Section 5 we show how Hask-LL meets those needs. The literature is dense with related work, which we dicuss in Section 6.

[1] URL suppressed for blind review, but available on request

2 A TASTE OF HASK-LL

We begin with an overview of Hask-LL, our proposed extension of Haskell with linear types. All the claims made in this section are substantiated later on. First, along with the usual arrow type $A \to B$, we propose an additional arrow type, standing for *linear functions*, written $A \multimap B$. In the body of a linear function, the type system tracks that there is exactly one copy of the parameter to consume.

$f :: A \multimap B$
$f\ x =$ {- x has multiplicity 1 here -}

$g :: A \to B$
$g\ y =$ {- y has multiplicity ω here -}

We say that the *multiplicity* of x is 1 in the body of f; and that of y is ω in g. Similarly, we say that unrestricted (non-linear) parameters have multiplicity ω (usable any number of times, including zero). We call a function *linear* if it has type $A \multimap B$ and *unrestricted* if it has type $A \to B$.

The linear arrow type $A \multimap B$ guarantees that any function with that type will consume its argument exactly once. However, the type *places no requirement on the caller* of these functions. The latter is free to pass either a linear or non-linear value to the function. For example, consider these definitions of a function g:

$g1, g2, g3 :: (a \multimap a \to a) \to a \multimap a \to a$
$g1\ k\ x\ y = k\ x\ y$ -- Valid
$g2\ k\ x\ y = k\ y\ x$ -- Invalid: fails x's multiplicity guarantee
$g3\ k\ x\ y = k\ x\ (k\ y\ y)$ -- Valid: y can be passed to linear k

As in *g2*, a linear variable x cannot be passed to the non-linear function $(k\ y)$. But the other way round is fine: *g3* illustrates that the non-linear variable y can be passed to the linear function k. Linearity is a strong contract for the implementation of a function, not its caller.

2.1 Calling contexts and promotion

A call to a linear function consumes its argument once only if the call itself is consumed once. For example, consider these definitions of the same function g:

$f :: a \multimap a$
$g4, g5, g6 :: (a \multimap a \to r) \to a \multimap a \to r$
$g4\ k\ x\ y = k\ x\ (f\ y)$ -- Valid: y can be passed to linear f
$g5\ k\ x\ y = k\ (f\ x)\ y$ -- Valid: k consumes $f\ x$'s result once
$g6\ k\ x\ y = k\ y\ (f\ x)$ -- Invalid: fails x's multiplicity guarantee

In *g5*, the linear x can be passed to f because the result of $(f\ x)$ is consumed linearly by k. In *g6*, x is still passed to the linear function f, but the call $(f\ x)$ is in a non-linear context, so x too is used non-linearly and the code is ill-typed.

In general, any sub-expression is type-checked as if it were to be consumed exactly once. However, an expression which does not contain linear resources, that is an expression whose free variables all have multiplicity ω, like $f\ y$ in *g4*, can be consumed many times. Such an expression is said to be *promoted*. We leave the specifics to Section 3.

2.2 Linear data types

Using the new linear arrow, we can (re-)define Haskell's list type as follows:

$$\begin{array}{l}\mathsf{data}\ [a]\ \mathsf{where}\\ \quad [\,] :: [a]\\ \quad (:) :: a \multimap [a] \multimap [a]\end{array}$$

That is, we give a linear type to the (:) data constructor. Crucially, 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. But we can *also* use the very same list type to contain linear resources (such as file handles) without compromising safety; the type system ensures that resources in a list will eventually be deallocated by the programmer, and that they will not be used after that.

Many list-based functions conserve the multiplicity of data, and thus can be given a more precise type. For example we can write $(+\!\!+)$ as follows:

$$\begin{array}{l}(+\!\!+) :: [a] \multimap [a] \multimap [a]\\ [\,] \quad +\!\!+ ys = ys\\ (x : xs) +\!\!+ ys = x : (xs +\!\!+ ys)\end{array}$$

The type of $(+\!\!+)$ tells us that if we have a list xs with multiplicity 1, appending any other list to it will never duplicate any of the elements in xs, nor drop any element in xs[2].

Giving a more precise type to $(+\!\!+)$ only *strengthens* the contract that $(+\!\!+)$ offers to its callers; *it does not restrict its usage*. For example:

$$\begin{array}{l}sum :: [Int] \multimap Int\\ f :: [Int] \multimap [Int] \to Int\\ f\ xs\ ys = sum\ (xs +\!\!+ ys) + sum\ ys\end{array}$$

Here the two arguments to $(+\!\!+)$ have different multiplicities, but the function f guarantees that it will consume xs precisely 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 consumed ys twice, and so ys must have multiplicity ω, and f needs an unrestricted arrow for that argument.

Generalising from lists to arbitrary algebraic data types, we designed Hask-LL so that when in a traditional Haskell (non-linear) calling context, linear constructors degrade to regular Haskell data types. Thus our radical position is that data types in Hask-LL should have *linear fields by default*, including all standard definitions, such as pairs, tuples, *Maybe*, lists, and so on. More precisely, when defined in old-style Haskell-98 syntax, all fields are linear; when defined using GADT syntax, the programmer can explicitly choose. For example, in our system, pairs defined as

$$\mathsf{data}\ (,)\ a\ b = (,)\ a\ b$$

would use linear arrows. This becomes explicit when defined in GADT syntax:

$$\mathsf{data}\ (a, b)\ \mathsf{where}\ (,) :: a \multimap b \multimap (a, b)$$

We will see in Section 2.5 when it is also useful to have contstructors with unrestricted arrows.

[2]This follows from parametricity. In order to *free* linear list elements, we must pattern match on them to consume them, and thus must know their type (or have a type class instance). Likewise to copy them.

2.3 Linearity polymorphism

As we have seen, implicit conversions between multiplicities make first-order linear functions *more general*. But the higher-order case thickens the plot. Consider that the standard *map* function over (linear) lists:

$$map\ f\ [\,] \qquad = [\,]$$
$$map\ f\ (x : xs) = f\ x : map\ f\ xs$$

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 most general type: $\forall \rho.\ (a \to_\rho b) \to [a] \to_\rho [b]$. Likewise, function composition can be given the following general type:

$$(\circ) :: \forall \pi\ \rho.\ (b \to_\pi c) \multimap (a \to_\rho b) \to_\pi a \to_{\rho\,\pi} c$$
$$(f \circ g)\ x = f\ (g\ x)$$

That is: two functions that accept arguments of arbitrary multiplicities (ρ and π respectively) can be composed to form a function accepting arguments of multiplicity $\rho\pi$ (*i.e.* the product of ρ and π — see Definition 3.1). 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 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.

2.4 Operational intuitions

Suppose that a linear function takes as its argument a *resource*, such as a file handle, channel, or memory block. Then the function guarantees:

- that the resource will be consumed by the time it returns;
- that the resource can only be consumed once; so it will never be used after being destroyed.

In this way, the linear type system of Hask-LL ensures both that prompt resource deallocation happens, and that no use-after-free error occurs.

But wait! We said earlier that a non-linear value can be passed to a linear function, so it would absolutely *not* be safe for the function to always deallocate its argument when it is consumed! To understand this we need to explain our operational model. In our model there there are two heaps: the familiar *dynamic heap* managed by the garbage collector, and a *linear heap* managed by the programmer supported by statically-checked guarantees. Assume for a moment two primitives to allocate and free objects on the linear heap:

$$alloc_T :: (T \multimap IO\ a) \multimap IO\ a$$
$$free_T :: T \multimap ()$$

Here $alloc_T\ k$ allocates a value of type T on the linear heap, and passes it to k. The continuation k must eventually free T by calling $free_T$. If there are any *other* ways of making a value of type T on the dynamic heap (e.g. $mkT :: Int \to T$), then $free_T$ might be given either a value on the linear heap or the dynamic heap. It can only free the former, so it must make a dynamic test to tell which is the case.

A consequence of the above design is that unrestricted values never contain (point to) linear values (but the converse is possible). This makes sense: after all, if the GC deallocates a value in the dynamic heap that points off-heap, then the off-heap data will be left dangling (being off-heap, the GC cannot touch it) with no means to free it manually. Conversely, a pointer from a resource to the heap can simply act as a new programmer-controlled GC root. We prove this invariant in Section 4.

We have said repeatedly that "a linear function guarantees to consume its argument exactly once if the call is consumed exactly once". But what does *"consuming a value exactly once"* mean? We can now give a more precise operational intuition:

- To consume exactly once a value of an atomic base type, like *Int* or *Ptr*, just evaluate it.
- To consume a function exactly once, call it, and consume its result exactly once.
- To consume a pair exactly once, evaluate it and consume each of its components excatly once.
- More generally, to consume exactly once a value of an algebraic data type, evaluate it and consume all its linear components exactly once.

A salient point of this definition is that "linear" emphatically does not imply "strict". Our linear function space behaves as Haskell programmers expect.

2.5 Linearity of constructors: the usefulness of unrestricted constructors

We saw in Section 2.2 that data types in Hask-LL have linear arguments by default. Do we ever need data constructors unrestricted arguments? Yes, we do.

Using the type T of Section 2.4, suppose we wanted a primitive to copy a T from the linear heap to the dynamic heap. We could define it in CPS style but a direct style is more convenient:

$$copy_T :: (T \to r) \multimap T \multimap r \qquad vs. \qquad copy_T :: T \multimap Unrestricted\ a$$

where *Unrestricted* is a data type with a non-linear constructor[3]:

```
data Unrestricted a where Unrestricted :: a → Unrestricted a
```

The *Unrestricted* data type is used to indicate that when a value (*Unrestricted* x) is consumed once (see Section 2.4) we have no guarantee about how often x is consumed. With our primitive in hand, we can now use ordinary code to copy a linear list of T values into the dynamic heap (we mark patterns in let and where with !, Haskell's syntax for strict pattern bindings: Hask-LL does not support lazy pattern bindings of linear values, case on the other hand, is always strict):

```
copy :: (a ⊸ Unrestricted a) → [a] ⊸ Unrestricted [a]
copy copyElt (x : xs) = Unrestricted (x' : xs') where !(Unrestricted xs') = copy xs
                                                      !(Unrestricted x')  = copyElt x
```

2.6 Running example: zero-copy packets

Imagine a server application which receives data and then stores it for a while before sending it out to receivers, perhaps in a different order. This general pattern characterizes a large class of low-latency servers of in-memory data, such as Memcached [13] or burst buffers [28].

As an example, consider a network router with software defined forwarding policies. First, we need to read packets from, and send them to network interfaces. Linearity can help with *copy-free* hand-off of packets between network interfaces and in-memory data structures. Assume that the user can acquire a linear handle on a *mailbox* of packets:

```
data MB
withMailbox :: (MB ⊸ IO a) ⊸ IO a
close       :: MB ⊸ ()
```

The mailbox handle must be eventually passed to *close* in order to release it. For each mailbox, *get* yields the next available packet, whereas *send* forwards packet on the network.

[3]The type constructor *Unrestricted* is in fact an encoding of the so-called *exponential* modality written ! in linear logic.

$$
\begin{aligned}
&\mathit{get} \;\; :: \mathit{MB} \multimap (\mathit{Packet}, \mathit{MB}) \\
&\mathit{send} :: \mathit{Packet} \multimap ()
\end{aligned}
$$

In the simplest case, we can read a message and send it immediately — without any filtering or buffering. When calling *get* and *send*, *Packet*s never need to be copied: they can be passed along from the network interface card to the mailbox and then to the linear calling context of *send*, all by reference.

The above API assumes that mailboxes are independent: the order of packets is ensured within a mailbox queue, but not accross mailboxes (and not even between the input and output queue of a given mailbox). This assumption enables us to illustrate the ability of our type system to finely track dependencies between various kinds of effects: in this case effects (*get* and *send*) related to separate mailboxes commute.

Buffering data in memory. So what can our server do with the packets once they are retrieved from the network? To support software-defined routing policies that dictate what packet to forward when, we introduce a priority queue.

$$
\begin{aligned}
&\mathsf{data}\ \mathit{PQ}\ a \\
&\mathit{empty} :: \mathit{PQ}\ a \\
&\mathit{insert} :: \mathit{Int} \to a \multimap \mathit{PQ}\ a \multimap \mathit{PQ}\ a \\
&\mathit{next} \;\; :: \mathit{PQ}\ a \multimap \mathit{Maybe}\ (a, \mathit{PQ}\ a)
\end{aligned}
$$

The interface is familiar, except that since the priority queue must store packets it must have linear arrows. In this way, the type system *guarantees* that despite being stored in a data structure, every packet is eventually sent.

In a router, priorities could represent deadlines associated to each packet[4]. So we give ourselves a function to infer a priority for a packet:

$$\mathit{priority} :: \mathit{Packet} \multimap (\mathit{Unrestricted}\ \mathit{Int}, \mathit{Packet})$$

The take-home message is that the priority queue can be implemented as a Hask-LL data type. Here is a naive implementation as a sorted list:

$$
\begin{aligned}
&\mathsf{data}\ \mathit{PQ}\ a\ \mathsf{where} \\
&\quad \mathit{Empty} :: \mathit{PQ}\ a \\
&\quad \mathit{Cons} :: \mathit{Int} \to a \multimap \mathit{PQ}\ a \multimap \mathit{PQ}\ a \\
&\mathit{empty} = \mathit{Empty} \\
&\mathit{insert}\ p\ x\ q\ \mathit{Empty} = \mathit{Cons}\ p\ x\ q \\
&\mathit{insert}\ p\ x\ (\mathit{Cons}\ p'\ x'\ q') \mid p < p' \quad\;\; = \mathit{Cons}\ p\ x\ (\mathit{Cons}\ p'\ x'\ q') \\
&\qquad\qquad\qquad\qquad\qquad\quad \mid \mathit{otherwise} = \mathit{Cons}\ p'\ x'\ (\mathit{insert}\ p\ x\ q') \\
&\mathit{next}\ \mathit{Empty} = \mathit{Nothing} \\
&\mathit{next}\ (\mathit{Cons}\ \_\ x\ q) = \mathit{Just}\ (x, q)
\end{aligned}
$$

In Section 5.1, we return to the implementation of the priority queue and discuss the implications for garbage collection overheads.

Finally, here is a tiny router that forwards all packets on the network once three packets are available.

[4] whereas in a caching application, the server may be trying to evict the bigger packets first in order to leave more room for incoming packets

```
sendAll :: PQ Packet ⊸ ()
sendAll q | Just (p, q') ← next q = case send p of () → sendAll q'
sendAll q | Nothing      ← next q = ()

enqueue :: MB ⊸ PQ a ⊸ (PQ a, MB)
enqueue mb q = let !(p, mb')                    = get mb
                   !(!(Unrestricted prio), p')  = priority p
               in (mb', insert prio p' q)

main :: IO ()
main = withMailbox $ λmb0 → let !(mb1, q1) = enqueue mb0 empty
                                !(mb2, q2) = enqueue mb1 q1
                                !(mb3, q3) = enqueue mb2 q2
                                !()        = close mb3
                            in return $ sendAll q3
```

Note that lifetimes of two packets *can intersect arbitrarily*, ruling out region-based approaches. The ability to deal with deallocation in a different order to allocation, which can be very important in practice, is a crucial feature of approaches based on linear types.

3 $\lambda^{Q}_{\to}$ STATICS

In this section we turn to the calculus at the core of Hask-LL, which we refer to as $\lambda^{q}_{\to}$, and for which we provide a step-by-step account of its syntax and typing rules.

As we discussed in Section 2.4, our operational model for $\lambda^{q}_{\to}$ is that of two heaps: the dynamic heap and the linear heap. Values in the linear heap are managed by the programmer, hence *must* be consumed *exactly once*, while values in the dynamic heap are managed by the garbage collector hence *may* freely be consumed any number of times (including just once or none at all). The role of the type system of $\lambda^{q}_{\to}$ is to enforce this very property.

Let us point out that closures (partial applications or lazy thunks) may reside in the linear heap. Indeed, as we explained in Section 2.4, values from the dynamic heap do not point to values in the linear heap, so if any member of a closure resides in the linear heap, so must the closure itself.

3.1 Syntax

The term syntax (Figure 1) is that of a type-annotated (*à la* Church) simply typed λ-calculus with let-definitions. Binders in λ-abstractions and type definitions are annotated both with their type and their multiplicity. Multiplicity abstraction and application are explicit.

In our static semantics for $\lambda^{q}_{\to}$ the familiar judgement $\Gamma \vdash t : A$ has a non-standard reading: it asserts that consuming the term $t : A$ *exactly once* will consume Γ exactly once (see Section 2.4).

The types of $\lambda^{q}_{\to}$ (see Figure 1) are simple types with arrows (albeit multiplicity-annotated ones), data types, and multiplicity polymorphism. The annotated function type is a generalisation of the intuitionistic arrow and the linear arrow. We use the following notations: $A \to B \stackrel{\text{def}}{=} A \to_{\omega} B$ and $A \multimap B \stackrel{\text{def}}{=} A \to_{1} B$.

The intuition behind the multiplicity-annotated arrow $A \to_q B$ is that consuming $f\ u : B$ exactly once will consume q times the value $u{:}A$. Therefore, a function of type $A \to B$ *must* be applied to an argument residing in the dynamic heap, while a function of type $A \multimap B$ *may* be applied to an argument residing on either heap. One might, thus, expect the type $A \multimap B$ to be a subtype of $A \to B$. This is however, not so, because there is no notion of subtyping in $\lambda^{q}_{\to}$. This is a salient choice in our design. Our objective is to integrate with existing typed functional languages such as Haskell and the ML family, which are based

Multiplicities

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

Contexts

$$\begin{array}{lr} \Gamma, \Delta ::= & \\ \quad \mid x :_q A, \Gamma & \text{multiplicity-annotated binder} \\ \quad \mid & \text{empty context} \end{array}$$

Type declarations

$$\mathsf{data}\ D\ \mathsf{where} \left(c_k : A_1 \to_{q_1} \cdots\ A_{n_k} \to_{q_{n_k}} D \right)_{k=1}^{m}$$

Types

$$\begin{array}{lr} A, B ::= & \\ \quad \mid A \to_q B & \text{function type} \\ \quad \mid \forall \rho . A & \text{multiplicity-dependent type} \\ \quad \mid D & \text{data type} \end{array}$$

Terms

$$\begin{array}{lr} e, s, t, u ::= & \\ \quad \mid x & \text{variable} \\ \quad \mid \lambda (x :_q A).t & \text{abstraction} \\ \quad \mid t\ s & \text{application} \\ \quad \mid \lambda \pi . t & \text{multiplicity abstraction} \\ \quad \mid t\ p & \text{multiplicity application} \\ \quad \mid c\ t_1 \ldots t_n & \text{data construction} \\ \quad \mid \mathsf{case}_p\ t\ \mathsf{of}\ \{c_k\ x_1 \ldots x_{n_k} \to u_k\}_{k=1}^{m} & \text{case} \\ \quad \mid \mathsf{let}\ x_1 :_{q_1} A_1 = t_1 \ldots x_n :_{q_n} A_n = t_n\ \mathsf{in}\ u & \text{let} \end{array}$$

Fig. 1. Syntax of the linear calculus

on Hindley-Milner-style polymorphism. Hindley-Milner-style polymorphism, however, does not mesh well with subtyping as the extensive exposition by Pottier [37] witnesses. Therefore $\lambda^q_{\to}$ uses multiplicity polymorphism for the purpose of reuse of higher-order function as we described in Section 2.3.

Data type declarations (see Figure 1) are of the following form:

$$\mathsf{data}\ D\ \mathsf{where} \left(c_k : A_1 \to_{q_1} \cdots\ A_{n_k} \to_{q_{n_k}} D \right)_{k=1}^{m}$$

The above declaration means that D has m constructors c_k (where $k \in 1 \ldots m$), each with n_k arguments. Arguments of constructors have a multiplicity, just like arguments of functions: an argument of multiplicity ω means that the data type can store, at that position, data which *must* reside in the dynamic heap;

$$\frac{}{\omega\Gamma + x :_1 A \vdash x : A}\text{var} \qquad \frac{\Gamma, x :_q A \vdash t : B}{\Gamma \vdash \lambda(x :_q A).t : A \to_q B}\text{abs} \qquad \frac{\Gamma \vdash t : A \to_q B \qquad \Delta \vdash u : A}{\Gamma + q\Delta \vdash t\ u : B}\text{app}$$

$$\frac{\Delta_i \vdash t_i : A_i \qquad c_k : A_1 \to_{q_1} \ldots \to_{q_{n-1}} A_n \to_{q_n} D \text{ constructor}}{\omega\Gamma + \sum_i q_i\Delta_i \vdash c_k\ t_1 \ldots t_n : D}\text{con}$$

$$\frac{\Gamma \vdash t : D \qquad \Delta, x_1 :_{pq_i} A_i, \ldots, x_{n_k} :_{pq_{n_k}} A_{n_k} \vdash u_k : C \qquad \text{for each } c_k : A_1 \to_{q_1} \ldots \to_{q_{n-1}} A_{n_k} \to_{q_{n_k}} D}{p\Gamma + \Delta \vdash \mathsf{case}_p\ t\ \mathsf{of}\ \{c_k\ x_1 \ldots x_{n_k} \to u_k\}_{k=1}^m : C}\text{case}$$

$$\frac{\Gamma_i \vdash t_i : A_i \qquad \Delta, x_1 :_{q_1} A_1 \ldots x_n :_{q_n} A_n \vdash u : C}{\Delta + \sum_i q_i\Gamma_i \vdash \mathsf{let}\ x_1 :_{q_1} A_1 = t_1\ \ldots\ x_n :_{q_n} A_n = t_n\ \mathsf{in}\ u : C}\text{let} \qquad \frac{\Gamma \vdash t : A \qquad \pi \text{ fresh for } \Gamma}{\Gamma \vdash \lambda\pi.t : \forall\pi.A}\text{m.abs}$$

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

Fig. 2. Typing rules

while a multiplicity of 1 means that data at that position *can* reside in either heap. A further requirement is that the multiplicities q_i must be concrete (*i.e.* either 1 or ω).

For most purposes, c_k behaves like a constant with the type $A_1 \to_{q_1} \cdots A_{n_k} \to_{q_{n_k}} D$. As the typing rules of Figure 2 make clear, this means in particular that from a value d of type D with multiplicity ω, pattern matching extracts the elements of d with multiplicity ω. Conversely, if all the arguments of c_k have multiplicity ω, c_k constructs D with multiplicity ω.

Note that, as discussed in Section 2.2, constructors with arguments of multiplicity 1 are not more general than constructors with arguments of multiplicity ω, because if, when constructing $c\ u$ with the argument of c of multiplicity 1, u *may* be either of multiplicity 1 or of multiplicity ω; dually when pattern-matching on $c\ x$, x *must* be of multiplicity 1 (if the argument of c had been of multiplicity ω, on the other hand, then x could be used either as having multiplicity ω or 1).

3.2 Contexts

Many of the typing rules scale contexts by a multiplicity, or add contexts together. We will explain the why very soon in Section 3.3, but first, let us focus on the how.

In $\lambda^q_\to$, each variable binding, in a typing context, is annotated with a multiplicity. These multiplicity annotations are the natural counterpart of the multiplicity annotation on abstractions and arrows.

For multiplicities we need the concrete multiplicities 1 and ω as well as multiplicity variables (ranged over by the metasyntactic variables π and ρ) for the sake of polymorphism. However, we are going to need to multiply and add multiplicities together, therefore we also need formal sums and products of multiplicities. Multiplicity expressions are quotiented by the following equivalence relation:

Definition 3.1 (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 as follows.

Definition 3.2 (Context addition).

$$\begin{aligned}(x :_p A, \Gamma) + (x :_q A, \Delta) &= x :_{p+q} A, (\Gamma + \Delta) \\ (x :_p A, \Gamma) + \Delta &= x :_p A, \Gamma + \Delta && (x \notin \Delta) \\ () + \Delta &= \Delta\end{aligned}$$

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 3.3 (Context scaling).

$$p(x :_q A, \Gamma) = x :_{pq} A, p\Gamma$$

Lemma 3.4 (Contexts form a module). *The following laws hold:*

$$\begin{aligned}\Gamma + \Delta &= \Delta + \Gamma & p(\Gamma + \Delta) &= p\Gamma + p\Delta \\ (p + q)\Gamma &= p\Gamma + q\Gamma \\ (pq)\Gamma &= p(q\Gamma) & 1\Gamma &= \Gamma\end{aligned}$$

3.3 Typing rules

We are now ready to understand the typing rules of Figure 2. Remember that the typing judgement $\Gamma \vdash t : A$ reads as: consuming the term $t : A$ once consumes Γ once. But what if we want to consume t more than once? This is where context scaling comes into play, like in the application rule:

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

The idea is that consuming u an arbitrary number of times also consumes Δ an arbitrary number of times, or equivalently, consumes $\omega\Delta$ exactly once. We call this the *promotion principle*[5]: to know how to consume a value any number of times it is sufficient (and, in fact, necessary) to know how to consume said value exactly once.

To get a better grasp of the application rule and the promotion principle, you may want to consider how it indeed validates following judgement. In this judgement, π is a multiplicity variable; that is, the judgement is multiplicity-polymorphic:

$$f :_\omega A \to_\pi B, x :_\pi A \vdash f\ x$$

This implicit use of the promotion principle in rules such as the application rule is the technical device which makes the intuitionistic λ-calculus a subset of $\lambda^q_\to$. Specifically the subset where all variables are

[5]The name *promotion principle* is a reference to the promotion rule of linear logic. In $\lambda^q_\to$, however, promotion is implicit.

annotated with the multiplicity ω:

$$\dfrac{\dfrac{\dfrac{\overline{x :_\omega A \vdash x : A}^{\text{var}} \quad \overline{x :_\omega A \vdash x : A}^{\text{var}}}{x :_\omega A \vdash Tensor\ x\ x : Tensor\ A\ A}^{\text{con}}}{\vdash \lambda(x :_\omega A).Tensor\ x\ x : A \to_\omega Tensor\ A\ A}^{\text{abs}} \quad \dfrac{\vdots}{\vdash id_\omega\ 42 : A}}{() + \omega() \vdash (\lambda(x :_\omega A).Tensor\ x\ x)_\omega\ (id_\omega\ 42)}^{\text{app}}$$

This latter fact is, in turn, why Hask-LL is an extension of Haskell (provided unannotated bindings are understood as having multiplicity ω). The variable rule, as used above, may require some clarification:

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

The variable rule implements weakening of unrestricted variables: that is, it lets us ignore variables with multiplicity ω[6]. Note that the judgement $x :_\omega A \vdash x : A$ is an instance of the variable rule, because $(x :_\omega A) + (x :_1 A) = x :_\omega A$. The constructor rule has a similar $\omega\Gamma$ context: it is necessary to support weakening at the level of constant constructors.

Most of the other typing rules are straightforward, but let us linger for a moment on the unusual, yet central to our design, case rule, and specifically on its multiplicity annotation:

$$\dfrac{\Gamma \vdash t : D \qquad \Delta, x_1 :_{pq_i} A_i, \ldots, x_{n_k} :_{pq_{n_k}} A_{n_k} \vdash u_k : C \qquad \text{for each } c_k : A_1 \to_{q_1} \ldots \to_{q_{n-1}} A_{n_k} \to_{q_{n_k}} D}{p\Gamma + \Delta \vdash \mathsf{case}_p\ t\ \mathsf{of}\ \{c_k\ x_1 \ldots x_{n_k} \to u_k\}_{k=1}^m : C}^{\text{case}}$$

The interesting case is when $p = \omega$, which reads as: if we can consume t an arbitrary number of time, then so can we of its constituents. Or, in terms of heaps: if t is on the dynamic heap, so are its constituents (see 2.4). As a consequence, the following program, which asserts the existence of projections, is well-typed (note that, both in *first* and *snd*, the arrow is — and must be — unrestricted).

$$\begin{array}{ll} first :: (a, b) \to a & snd :: (a, b) \to b \\ first\ \ (a, b) = a & snd\ \ (a, b) = b \end{array}$$

This particular formulation of the case rule is not implied by the rest of the system: only the case $p = 1$ is actually necessary. Yet, providing the case $p = \omega$ is the design choice which makes it possible to consider data-type constructors as linear by default, while preserving the semantics of the intuitionistic λ-calculus (as we already stated in Section 2.2). For Hask-LL, it means that types defined in libraries which are not aware of linear type (*i.e.* libraries in pure Haskell) can nevertheless be immediately useful in a linear context. Inheritance of multiplicity is thus crucial for backwards compatibility, which is a design goal of Hask-LL.

4 $\lambda^Q_\to$ DYNAMICS

We wish to give a dynamic semantics for $\lambda^q_\to$ which accounts for the packet forwarding example of Section 2.6 where packets are kept out of the garbage collected heap, and freed immediately upon send. To that effect we follow Launchbury [24] who defines a semantics for lazy computation. We will need also need to account for the *IO* monad, which occurs in the API for packets.

[6]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 [2].

4.1 The IO monad

Linear typing allows to safely and easily express world-passing semantics. Launchbury and Peyton Jones [25] defines $IO\ a$ as $World \to (World, a)$, for an abstract type $World$ representing the state of the entire world. The idea is that every time some IO action is undertaken, the world has possibly changed so we *consume* the current view of the world and return the new version.

The above technique gives a pure interface to I/O. However, it leaves the possibility for the programmer to access and old version of the world, as well as the current one, which is expensive to implement. In practice, one does not want to perform such a duplication, and thus Haskell solves the issue by forcing the programmer to use IO via its monadic interface.

Linear typing gives a much more direct solution to the problem: if the $World$ is kept linear, then there is no way to observe two different $World$s. Namely, it is enough to define IO as

$$\begin{aligned}&\mathsf{data}\ IO_0\ a\ \mathsf{where}\ IO_0 : World \multimap a \to IO_0\ a\\ &IO\ a = World \multimap IO_0\ a\end{aligned}$$

Notice that the a of $IO\ a$ is always unrestricted, so that IO has the same semantics as in Haskell (it is also the semantics which we need to ensure that *withMailbox* is safe).

The last missing piece is to inject a $World$ to start the computation. Haskell relies on a $main :: IO\ ()$ function, of which there must be a single one at link time. In $\lambda^q_{\to}$ the simplest way to achieve the same result is to start computation with a $world :_1 World$ in the context.

In general, a top-level definition of multiplicity 1 corresponds to something which must be consumed exactly once at link time, which generalises the concept of the *main* function (if only slightly).

4.2 Modelling network traffic

We are not going to give an accurate, non-deterministic, model of I/O for the purpose of this section. Instead, we are going to consider the semantics as a Laplace demon: the entirety of the events past and future are pre-ordained, and the semantics has access to this knowledge.

Because the only interaction with the world which we need to model in order to give a semantics to the packet example of Section 2.6 is to obtain a packet, it is sufficient for this section to consider all the packets. Because there are several mailboxes and each can get their own streams of packets, we suppose implicitly a given collection of packets $(p_i^j)_{j,i\in\mathbb{N}}$. Where the packet p_i^j represents the i-th package which will be received by the j-th mailbox.

Instead of using the world token as a proxy for an abstract world, we are using it to keep track of how many mailboxes have been opened. So the (unique) world token in the stack holds this number. Similarly, the mailbox tokens are pairs $\langle j, i\rangle$ of integers where j is the mailbox number and i the number of packets the mailbox has received. In effect, the world and mailbox tokens are pointers into the infinite matrix of potential packets. We define these constants as having the same typing rules as zero-ary constructors (but without the pattern-matching rule): *e.g.*:

$$\frac{}{\omega\Gamma \vdash j : World}\text{world}$$

In addition to the abstract types $World$, $Packet$ and MB, and the concrete types IO_0, IO, $(,)$, and $()$, $\lambda^q_{\to}$ is extended with three primitives (see also Section 2.6):

- $withMailbox : (MB \multimap IOa) \multimap IOa$
- $close : MB \multimap ()$
- $get : MB \multimap (Packet, MB)$
- $send : Packet \multimap ()$

Translation of typed terms

$$
\begin{aligned}
(\lambda(x :_q A).t)^* &= \lambda(x :_q A).(t)^* \\
x^* &= x \\
(t\ x)^* &= (t)^*\ x \\
(t\ u)^* &= \mathsf{let}\ y :_q A = (u)^*\ \mathsf{in}\ (t)^*\ y \qquad \text{with } \Gamma \vdash t : A \to_q B
\end{aligned}
$$

$$
c_k\ t_1 \dots t_n = \mathsf{let}\ x_1 :_{q_1} A_1 = (t_1)^*, \dots, x_n :_{q_n} A_n = (t_n)^*\ \mathsf{in}\ c_k\ x_1 \dots x_n \quad \text{with } c_k : A_1 \to_{q_1} \dots A_n \to_{q_n} D
$$

$$
\begin{aligned}
(\mathsf{case}_p\ t\ \mathsf{of}\ \{c_k\ x_1 \dots x_{n_k} \to u_k\}_{k=1}^m)^* &= \mathsf{case}_p\ (t)^*\ \mathsf{of}\ \{c_k\ x_1 \dots x_{n_k} \to (u_k)^*\}_{k=1}^m \\
(\mathsf{let}\ x_1 :_{q_1} A_1 = t_1\ \dots\ x_n :_{q_n} A_n = t_n\ \mathsf{in}\ u)^* &= \mathsf{let}\ x_1 :_{q_1} A_1 = (t_1)^*, \dots, x_n :_{q_n} A_n =_{q_n} (t_n)^*\ \mathsf{in}\ (u)^*
\end{aligned}
$$

Fig. 3. Syntax for the Launchbury-style semantics

Packets p_i^j are considered as constants. We do not model packet priorities in the semantics, for concision.

4.3 Operational semantics

Launchbury's semantics is a big-step semantics where variables play the role of pointers to the heap (hence represent sharing, which is the cornerstone of a lazy semantics).

To account for a foreign heap, we have a single logical heap with bindings of the form $x :_p A = e$ where $p \in \{1, \omega\}$ a multiplicity: bindings with multiplicity ω represent objects on the regular, garbage-collected, heap, while bindings with multiplicity 1 represent objects on a foreign heap, which we call the *linear heap*. The linear heap will hold the *World* and *MB* tokens as well as packets. Launchbury [24]'s semantics relies on a constrained λ-calculus syntax which we remind in Figure 3. We assume, in addition, that the primitives are η-expanded by the translation.

The dynamic semantics is given in Figure 4. Let us review the new rules:

Linear variable In the linear variable rule, the binding in the linear heap is removed. In conjunction with the rule for *send*, it represents deallocation of packets.

WithMailbox A new *MB* is created with a fresh name j. Because it has not received any message yet, the mailbox token is $\langle j, 0\rangle$, and the world token is incremented. The body k is an IO action, so it takes the incremented world as an argument and returns a new one, which is then returned as the final world after the entire $withMailbox$ action.

Get The *get* primitive receives the next packet as is determined by the $(p_i^j)_{j,i\in\mathbb{N}}$ matrix, and the number of packets received by the *MB* is incremented.

Send The *send* primitive does not actually change the world, because all the messages that will ever be received are preordained, by assumption. So, from the point of view of this semantics, *send* simply frees its argument: the packet is stored in a linear variable, so it is removed from the heap with the linear variable rule; then the send rule drops it.

Close The *close* primitive consumes the mailbox. Like *send*, for the purpose of this semantics, *close* simply frees the mailbox.

4.4 Type safety

While the semantics of Figure 4 describes quite closely our plans for implementation in GHC, it is not convenient for proving properties. There are two reasons to that fact: first the semantics follows a

$$\frac{}{\Gamma : \lambda\pi.t \Downarrow \Gamma : \lambda\pi.t}\text{m.abs} \qquad \frac{\Gamma : e \Downarrow \Delta : \lambda\pi.e' \qquad \Delta : e'[q/\pi] \Downarrow \Theta : z}{\Gamma : e\ q \Downarrow \Theta : z}\text{m.app}$$

$$\frac{}{\Gamma : \lambda x :_p A.e \Downarrow \Gamma : \lambda x :_p A.e}\text{abs} \qquad \frac{\Gamma : e \Downarrow \Delta : \lambda y :_p A.e' \qquad \Delta : e'[x/y] \Downarrow \Theta : z}{\Gamma : e\ x \Downarrow \Theta : z}\text{application}$$

$$\frac{\Gamma : e \Downarrow \Delta : z}{(\Gamma, x :_\omega A = e) : x \Downarrow (\Delta; x :_\omega Az) : z}\text{shared variable} \qquad \frac{\Gamma : e \Downarrow \Delta : z}{(\Gamma, x :_1 A = e) : x \Downarrow \Delta : z}\text{linear variable}$$

$$\frac{(\Gamma, x_1 :_\omega A_1 = e_1, \ldots, x_n :_\omega A_n e_n) : e \Downarrow \Delta : z}{\Gamma : \mathsf{let}\ x_1 :_{q_1} A_1 = e_1 \ldots x_n :_{q_n} A_n = e_n\ \mathsf{in}\ e \Downarrow \Delta : z}\text{let} \qquad \frac{}{\Gamma : c\ x_1 \ldots x_n \Downarrow \Gamma : c\ x_1 \ldots x_n}\text{constructor}$$

$$\frac{\Gamma : e \Downarrow \Delta : c_k\ x_1 \ldots x_n \qquad \Delta : e_k[x_i/y_i] \Downarrow \Theta : z}{\Gamma : \mathsf{case}_q\ e\ \mathsf{of}\ \{c_k\ y_1 \ldots y_n \mapsto e_k\}_{k=1}^m \Downarrow \Theta : z}\text{case}$$

$$\frac{\Gamma, x :_1 MB = \langle j, 0\rangle : k\ x\ (j+1) \Downarrow \Delta : z}{\Gamma, w :_1 World = j : withMailbox\ k\ w \Downarrow \Delta : z}\text{withMailbox} \qquad \frac{\Gamma : x \Downarrow \Delta : \langle j, i\rangle}{\Gamma : close\ x \Downarrow \Delta : ()}\text{close}$$

$$\frac{\Gamma : x \Downarrow \Delta : \langle j, i\rangle}{\Gamma : get\ x \Downarrow \Delta, x :_1 MB = \langle j, i+1\rangle, y :_1 Packet = p_i^j : (y, z)}\text{get} \qquad \frac{\Gamma : x \Downarrow \Delta : p_i^j}{\Gamma : send\ x \Downarrow \Delta : ()}\text{send}$$

Fig. 4. Dynamic semantics

different structure than the type system and, also, there are pointers from the garbage-collected heap to the linear heap. Such pointers occur, for instance, in the priority queue from Section 2.6: the queue itself is allocated on the garbage collected heap while packets are kept in the linear heap.

This is not a problem in and on itself: pointers to packets may be seen as opaque by the garbage collector, which does not collect them, so that their lifetime is still managed explicitly by the programmer. However, in order to prevent use-after-free bugs, we must be sure that by the time a packet is sent (hence freed), every extant object in the garbage-collected heap which points to that packet must be dead.

In order to prove such a property, let us introduce a stronger semantics with the lifetime of objects more closely tracked. The strengthened semantics differs from Figure 4 in two aspects: the evaluation states are typed, and values with statically tracked lifetimes (linear values) are put on the linear heap.

In order to define the strengthened semantics, we introduce a few notations. First we need a notion of product annotated with the multiplicity of its first component.

Definition 4.1 (Weighted tensors). We use $A\ {}_\rho\otimes B$ ($\rho \in \{1, \omega\}$) to denote one of the two following types:

- $\mathsf{data}\ A\ {}_1\otimes B = (,) : A \multimap B \multimap A\ {}_1\otimes B$
- $\mathsf{data}\ A\ {}_\omega\otimes B = (,) : A \to B \multimap A\ {}_\omega\otimes B$

Weighted tensors are used to internalise a notion of stack that keeps track of multiplicities for the sake of the following definition, which introduces the states of the strengthened evaluation relation.

Definition 4.2 (Annotated state). An annotated state is a tuple $\Xi \vdash (\Gamma | t :_\rho A), \Sigma$ where

- Ξ is a typing context
- Γ is a *typed heap*, *i.e.* a collection of bindings of the form $x :_\rho A = e$
- t is a term
- $\rho \in \{1, \omega\}$ is a multiplicity
- A is a type
- Σ is a typed stack, *i.e.* a list of triple $e :_\omega A$ of a term, a multiplicity and an annotation.

Definition 4.3 (Well-typed state). We say that an annotated state is well-typed if the following typing judgement holds:

$$\Xi \vdash \mathsf{let}\ \Gamma\ \mathsf{in}\ (t, terms(\Sigma)) : (A\ {}_\rho\otimes\ multiplicatedTypes(\Sigma))$$

Where $\mathsf{let}\ \Gamma\ \mathsf{in}\ e$ stands for the grafting of Γ as a block of bindings, $terms(e_1 :_{\rho_1} A_1, \dots, e_n :_{\rho_n} A_n)$ for $(e_1\ {}_{\rho_1}, (\dots, (e_n\ {}_{\rho_n}, ())))$, and $multiplicatedTypes(e_1 :_{\rho_1} A_1, \dots, e_n :_{\rho_n} A_n)$ for $A_1\ {}_{\rho_1}\otimes (\dots (A_n\ {}_{\rho_n}\otimes ()))$.

Definition 4.4 (Strengthened reduction relation). We define the strengthened reduction relation, also written $\Downarrow$, as a relation on annotated states. Because Ξ, ρ, A and Σ are always the same for related states, we abbreviate

$$(\Xi \vdash \Gamma | t :_\rho A, \Sigma) \Downarrow (\Xi \vdash \Delta | z :_\rho A, \Sigma)$$

as

$$\Xi \vdash (\Gamma | t \Downarrow \Delta | z) :_\rho A, \Sigma$$

The strengthened reduction relation is defined inductively by the rules of Figure 5.

A few noteworthy remarks about the semantics in Figure 5 can be made. First, the let rule does not necessarily allocate in the garbage collected heap anymore — this was the goal of the strengthened semantics to begin with — but nor does it systematically allocate bindings of the form $x :_1 A = e$ in the linear heap either: the heap depends on the multiplicity ρ. The reason for this behaviour is promotion: an ostensibly linear value can be used in an unrestricted context. In this case the ownership of x must be given to the garbage collector: there is no static knowledge of x's lifetime. For the same reason, the linear variable case requires ρ to be 1 (Corollary 4.6 proves this restriction to be safe).

The other important rule is the *withMailbox* rule: it requires a result of the form $IO_0\ x\ w$. This constraint is crucial, because the *withMailbox* rule must ensure that the allocated mailbox is deallocated (with *close*) before it scope returns. The reason why it is possible is that, by definition, in $IO_0\ xw$, x *must* be in the dynamic heap. In other words, when an expression $e : IO_0\ A$ is forced to the form $IO_0\ x\ w$, it will have consumed all the pointers to the linear heap (except w). The crucial safety property of the strengthened relation is that it preserves well-typing of states.

Lemma 4.5 (Strengthened reduction preserves typing). *If* $\Xi \vdash (\Gamma | t \Downarrow \Delta | z) :_\rho A, \Sigma$, *then*

$$\Xi \vdash (\Gamma | t :_\rho A), \Sigma \quad \textit{implies} \quad \Xi \vdash (\Delta | z :_\rho A), \Sigma.$$

Proof. By induction on the typed-reduction. □

Thanks to this property we can freely consider the restriction of the strengthened relation to well-typed states. For this reason, from now on, we only consider well-typed states.

Corollary 4.6 (Never stuck on the linear variable rule). $\Xi \vdash (\Gamma, x :_1 A = e | x) :_\omega B, \Sigma$ *is not reachable.*

Proof. Remember that we consider only well-typed states because of Lemma 4.5. By unfolding the typing rules it is easy to see that $\Xi \vdash (\Gamma, x :_1 A = e | x) :_\omega B, \Sigma$ is not well-typed: it would require $x :_1 A = \omega\Delta$ for some Δ, which cannot be. □

$$\frac{}{\Xi \vdash (\Gamma|\lambda x :_q A.e \Downarrow \Gamma|\lambda x :_q A.e) :_\rho A \to_q B}\text{abs}$$

$$\frac{\Xi \vdash (\Gamma|e \Downarrow \Delta|\lambda(y :_q A).u) :_\rho A \to_q B, x :_{q\rho} A, \Sigma \qquad \Xi \vdash (\Delta|u[x/y] \Downarrow \Theta|z) :_\rho B, \Sigma}{\Xi \vdash (\Gamma|e_q\ x \Downarrow \Theta|z) :_\rho B, \Sigma}\text{app}$$

$$\frac{\Xi, x :_\omega B \vdash (\Gamma|e \Downarrow \Delta|z) :_\rho A, \Sigma}{\Xi \vdash (\Gamma, x :_\omega B = e|x \Downarrow \Delta, x :_\omega B = z|z) :_\rho A, \Sigma}\text{shared variable}$$

$$\frac{\Xi \vdash (\Gamma|e \Downarrow \Delta|z) :_1 A, \Sigma}{\Xi \vdash (\Gamma, x :_1 B = e|x \Downarrow \Delta|z) :_1 A, \Sigma}\text{linear variable}$$

$$\frac{\Xi \vdash (\Gamma, x_1 :_{\rho q_1} A_1 = e_1 \ldots x_n :_{pq_n} A_n = e_n|t \Downarrow \Delta|z) :_\rho C, \Sigma}{\Xi \vdash (\Gamma|\mathsf{let}\ x_1 :_{q_1} A_1 = e_1 \ldots x_n :_{q_n} A_n = e_n\ \mathsf{in}\ t \Downarrow \Delta|z) :_\rho C, \Sigma}\text{let}$$

$$\frac{}{\Xi \vdash (\Gamma|c\ x_1 \ldots x_n \Downarrow \Gamma|c\ x_1 \ldots x_n) :_\rho A, \Sigma}\text{constructor}$$

$$\frac{\begin{array}{c}\Xi, y_1 :_{p_1 q\rho} A_1 \ldots, y_n :_{p_n q\rho} A_n \vdash (\Gamma|e \Downarrow \Delta|c_k\ x_1 \ldots x_n) :_{q\rho} D, u_k :_\rho C, \Sigma \\ \Xi \vdash (\Delta|u_k[x_i/y_i] \Downarrow \Theta|z) :_\rho C, \Sigma\end{array}}{\Xi \vdash (\Gamma|\mathsf{case}_q\ e\ \mathsf{of}\ \{c_k\ y_1 \ldots y_n \mapsto u_k\}_{k=1}^m \Downarrow \Theta|z) :_\rho C, \Sigma}\text{case}$$

$$\frac{\Xi \vdash (\Gamma, x :_1 MB = \langle j, 0\rangle|k\ x\ (j+1) \Downarrow \Delta|z) :_1 IO_0\ A, \Sigma}{\Xi \vdash (\Gamma, w :_1 \mathit{World} = j|\mathit{withMailbox}\ k\ w \Downarrow \Delta|z) :_1 IO_0\ A, \Sigma}\text{withMailbox}$$

$$\frac{\Xi \vdash (\Gamma|x \Downarrow \Delta|\langle j, i\rangle) :_1 MB, \Sigma}{\Xi \vdash (\Gamma|\mathit{get}\ x \Downarrow \Delta, x :_1 MB = \langle j, i+1\rangle, y :_1 \mathit{Packet} = p_i^j|(y, z)) :_1 (\mathit{Packet}, MB), \Sigma}\text{get}$$

$$\frac{\Xi \vdash (\Gamma|x \Downarrow \Delta|\langle j, i\rangle) :_1 MB, \Sigma}{\Xi \vdash (\Gamma|\mathit{close}\ x \Downarrow \Delta|()) :_1 (), \Sigma}\text{close} \qquad \frac{\Xi \vdash (\Gamma|x \Downarrow \Delta|p_i^j) :_1 \mathit{Packet}, \Sigma}{\Xi \vdash (\Gamma|\mathit{send}\ x \Downarrow \Delta|()) :_1 (), \Sigma}\text{send}$$

Fig. 5. Strengthened operational semantics (Omitting the obvious m.abs and m.app for concision)

We are now ready to prove properties of the ordinary semantics by transfer of properties of the strengthened semantics. Let us start by defining a notion of type assignment for states of the ordinary semantics.

Definition 4.7 (Type assignment). A well-typed state is said to be a type assignment for an ordinary state, written $\gamma(\Gamma : e)(\Xi \vdash \Gamma'|e' :_\rho A, \Sigma)$, if $e = e' \wedge \Gamma' \leqslant \Gamma$.

That is, Γ' is allowed to strengthen some ω bindings to be linear, and to drop unnecessary ω bindings.

Note that for a closed term, type assigment reduces to the fact that e has a type. So we can see type assignment to state as a generalisation of type assignment to terms which is preserved during the reduction. Let us turn to prove that fact, noticing that type assignment defines a relation between ordinary states and well-typed states.

Lemma 4.8 (Type safety). *The refinement relation defines a simulation of the ordinary reduction by the strengthened reduction.*

That is for all $\gamma(\Gamma : e)(\Xi \vdash (\Gamma'|e) :_\rho A, \Sigma)$ *such that* $\Gamma : e \Downarrow \Delta : z$*, there exists a well-typed state* $\Xi \vdash (\Delta'|z) :_\rho A, \Sigma$ *such that* $\Xi \vdash (\Gamma|t \Downarrow \Delta|z) :_\rho A, \Sigma$ *and* $\gamma(\Delta : z)(\Xi \vdash (\Delta'|z) :_\rho A, \Sigma)$*.*

Proof. This is proved by a straightforward induction over the ordinary reduction. The case of let may be worth considering for the curious reader. □

From type-safety, it follows that a completely evaluated program has necessarily deallocated all the linear heap. This is a form of safety from resource leaks (of course, resource leaks can always be programmed in, but the language itself does not leak resources).

Corollary 4.9 (Eventual deallocation of linear values). *Let* $w :_1 \mathit{World} \vdash t : \mathit{World}$ *be a well-typed term. If* $w :_1 \mathit{World} = 0 : t \Downarrow \Delta : j$*, then* Δ *only contains* ω*-bindings.*

Proof. By Lemma 4.8, we have $\vdash (\Delta|j :_1 \mathit{World}), \cdot$. Then the typing rules of let and j (see Section 4.2) conclude: in order for j to be well typed, the environment introduced by let Δ must be of the form $\omega\Delta'$. □

For the absence of use-after-free errors, let us invoke a liveness property: namely that the type assignment is also a simulation of the strengthened semantics by the ordinary semantics (making type assignment a bisimulation). There is not a complete notion of progress which follows from this as big step semantics such as ours do not distinguish blocking from looping: we favour clarity of exposition over a completely formal argument for progress.

Lemma 4.10 (Liveness). *The refinement relation defines a simulation of the strengthened reduction by the ordinary reduction.*

That is for all $\gamma(\Gamma : e)(\Xi \vdash (\Gamma'|e) :_\rho A, \Sigma)$ *such that* $\gamma(\Delta : z)(\Xi \vdash (\Delta'|z) :_\rho A, \Sigma)$*, there exists a state* $\Delta : z$ *such that* $\Gamma : e \Downarrow \Delta : z$ *and* $\gamma(\Delta : z)(\Xi \vdash (\Delta'|z) :_\rho A, \Sigma)$*.*

Proof. This is proved by a straightforward induction over the ordinary reduction. □

In conjunction with Corollary 4.6, Lemma 4.10 shows that well-typed programs do not get blocked, in particular that garbage-collected objects which point to the linear objects are not dereferenced after the linear object has been freed: $\lambda^q_\to$ is safe from use-after-free errors.

5 APPLICATIONS

There is a wealth of literature regarding the application of linear typing to many practical problems, for instance: explicit memory management [1, 17, 23], array computations [7, 26], protocol specification [18], privacy guarantees [14] and graphical interfaces [22].

This section develops a few examples which are directly usable in Hask-LL, that is simply by changing Haskell's type system, and using the dynamic semantics of Section 4 to justify the memory safety of a foreign heap implemented using a foreign function interface.

5.1 Lowering the GC pressure

In a practical implementation of the zero-copy packet example of Section 2.6, the priority queue can easily become a bottleneck, because it will frequently stay large [12]. We can start by having a less obnoxiously naive implementation of queues, but this optimisation would not solve the issue which we are concerned with in this section: garbage collection latency. Indeed, the variance in latency incurred by GC pauses can be very costly in a distributed application. Indeed, having a large number of processes that may

decide to run a long pause increases the probability that at least one is running a pause. Consequently, waiting on a large number of processes is slowed down (by the slowest of them) much more often than a sequential application. This phenomenon is known as Little's law [27].

A radical solution to this problem, yet one that is effectively used in practice, is to allocate the priority queue with *malloc* instead of using the garbage collector's allocator [29, Section IV.C]. Our own benchmarks [7], consistent with Pusher's findings, indicate that peak latencies encountered with large data-structures kept in GHC's GC heap are two orders of magnitude higher than using foreign function binding to an identical data-structure allocated with *malloc*; furthermore the latency distribution of GC latency consistently degrades with the size of the heap, while *malloc* has a much less flat distribution. Of course, using *malloc* leaves memory safety in the hand of the programmer.

Fortunately, it turns out that the same arguments that we use to justify proper deallocation of mailboxes in Section 4 can be used to show that, with linear typing, we can allocate a priority queue with *malloc* safely (in effect considering the priority queue as a resource). We just need to replace the *empty* queue function from Section 2.6 by a $withQueue :: (PQ\ a \multimap IO\ a) \multimap IO\ a$ primitive, in the same style as *withMailbox*.

We can go even further and allow *malloc*'d queues to build pure values: it is enough to replace the type of *withQueue* as $withQueue :: (PQ\ a \multimap Unrestricted\ a) \multimap Unrestricted\ a$. Justifying the safety of this type requires additional arguments as the result of *withQueue* may be promoted (*IO* actions are never promoted because of their *World* parameter), hence one must make sure that the linear heap is properly emptied, which is in fact implied by the typing rules for *Unrestricted*.

5.2 Safe streaming

The standard for writing streaming applications (*e.g.* reading from a file) in Haskell is to use a combinator library such as Conduits [40] or Machines [21]. Such libraries have many advantages: they are fast, they release resources promptly, and they are safe.

However, they come at a significant cost: they are difficult to use. As a result the authors have observed industrial users walking back from this type of library to use the simpler, but unsafe `streaming` [41] library. The lack of safety of the stream library stems from the *uncons* function (in *Streaming.Prelude*):

$$uncons :: Monad\ m \Rightarrow Stream\ (Of\ a)\ m\ () \to m\ (Maybe\ (a, Stream\ (Of\ a)\ m\ ()))$$

Note the similarity with the *IO* monad: a stream is consumed and a new one is returned. Just like the *World* of the *IO* monad, the initial stream does not make sense anymore and reusing it will result in incorrect behaviour. We have observed this very mistake in our own code in industrial projects, and it proved quite costly to hunt down. Lippmeier et al. [26, Section 2.2] describe a very similar example of unsafety in the `repa-flow`.

Provided a sufficiently linear notion of monad (see Morris [34, Section 3.3 (Monads)] for a discussion on the interaction of monads and linear typing), we can make *uncons* safe merely by changing the arrow to a linear one:

$$uncons :: Monad\ m \Rightarrow Stream\ (Of\ a)\ m\ () \multimap m\ (Maybe\ (a, Stream\ (Of\ a)\ m\ ()))$$

5.3 Protocols

Honda [18] introduces the idea of using types to represent and enforce protocols. Wadler [46] showed that Honda's system is isomorphic to (classical) linear logic. The high-level idea is that one end of a communication channel is typed with the protocol P and the other end with the dual protocol $P^{\perp}$; for

[7] URL suppressed for blind review, but available on request

instance: if A denotes "I expect an A", the dual $A^\perp$ denotes "I shall send an A". Then, protocols can be composed using pairs: the protocol $(A, B^\perp)$ means "I expect an A, and I shall send a B".

In our intuitionistic setting, we can represent the dual $P^\perp$ by using continuation passing style: $P^\perp = P \multimap \bot$, where $\bot$ represents a type of effects ($\bot = IO\ ()$ would be a typical choice in Haskell). This encoding is the standard embedding of classical (linear) logic in intuitionistic (linear) logic. Using $P^\perp = P \to \bot$ would not be sufficient to enforce the protocol P, because a process can skip sending a required value or expect two values where one ought to be sent, potentially causing deadlocks.

Thus there are two reasons why $\lambda^q_\to$ does not have a built-in additive product ($A\&B$), dual to (linear) sum: 1. uniformity commands to use the general dualisation pattern instead and 2. the definition would depend on the choice of effects. The following example (a linear `if` combinator) shows a glimpse of linear CPS code:

$$\mathsf{data}\ A \oplus B = \mathit{Left} :: A \multimap A \oplus B \mid \mathit{Right} :: B \multimap A \oplus B$$
$$\mathsf{type}\ A\ \&\ B = ((A \multimap \bot) \oplus (B \multimap \bot)) \multimap \bot$$
$$\mathit{if}' :: \mathit{Bool} \multimap (a\ \&\ a) \multimap (a \multimap \bot) \multimap \bot$$
$$\mathit{if}'\ \mathit{True}\ \ p\ k = p\ (\mathit{Left}\quad k)$$
$$\mathit{if}'\ \mathit{False}\ p\ k = p\ (\mathit{Right}\ k)$$

6 RELATED WORK

6.1 Regions

Haskell's ST monad [25] taught us a conceptually simple approach to lifetimes. The ST monad has a phantom type parameter s that is instantiated once at the beginning of the computation by a $runST$ function of type:

$$runST :: (\forall s.\ ST\ s\ a) \to a$$

In 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.

This apparent simplicity (one only needs rank-2 polymorphism) comes at the cost of strong limitations in practice:

- ST-like regions confine to a stack-like allocation discipline. Scopes cannot intersect arbitrarily, limiting the applicability of this technique. In our running example, if unused mailboxes have to be kept until all mailboxes opened in their scope have been closed, they would be hoarding precious resources (like file descriptors).
- Kiselyov and Shan [20] 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. Moreover, computations cannot live directly in IO, but instead in a wrapper monad. The HaskellR project [8] 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, force otherwise pure functions to be written monadically and rule out useful syntactic facilities like view patterns. In contrast, with linear types, values in two regions (in our running example packets from different mailboxes) have the same type hence can safely be mixed: any data structure containing packet of a mailbox will be forced to be consumed before the mailbox is closed.

6.2 Finalisers

A garbage collector can be relied on for more than just cleaning up no longer extant memory. By registering finalizers (*IO* callbacks) with values, such as a file handle, one can make it the job of the garbage collector to make sure the file handle is eventually closed. But "eventually" can mean a very long time. File descriptors and other system resources are particularly scarce: operating systems typically only allow a small number of descriptors to be open at any one time. Kiselyov [19] argues that such system resources are too scarce for eventual deallocation. Prompt deallocation is key.

6.3 Uniqueness types

The literature is awash with enforcing linearity not via linear types, but via uniqueness (or ownership) types. The most prominent representatives of languages with such uniqueness types are perhaps Clean [4] and Rust [31]. Hask-LL, on the other hand, is designed around linear types based on linear logic [16].

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 expressions context even if the function can work with the argument as it pleases.

From a compiler's perspective, uniqueness type provide a *non-aliasing analysis* while linear types 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. [6], see also the discussion in Section 7.2.

Because of this weak duality, we perhaps could as well 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 Section 5; (c) and decisively, linear type systems are conceptually simpler than uniqueness type systems, giving a clearer path to implementation in GHC.

6.4 Linearity as a property of types vs. a property of bindings

In several presentations [32, 34, 45] programming languages incorporate linearity by dividing types into two kinds. A type is either linear or unrestricted.

In effect, this distinction imposes a clean separation between the linear world and the unrestricted world. An advantage of this approach is that it instantiates both to linear types and to uniqueness types depending on how they the two worlds relate, and even have characteristics of both [11].

Such approaches have been very successful for theory: see for instance the line of work on so-called *mixed linear and non-linear logic* (usually abbreviated LNL) started by Benton [5]. However, for practical language design, code duplication between the linear an unrestricted worlds quickly becomes costly. So language designers try to create languages with some kind of kind polymorphism to overcome this limitation. This usually involves a subkinding relation and bounded polymorphism, and these kind polymorphic designs are complex. See Morris [34] for a recent example. We argue that by contrast, the type system of $\lambda^q_{\to}$ is simpler.

The complexity introduced by kind polymorphism and subtyping relations makes retrofitting a rich core language such as GHC's an arduous endeavour. GHC already supports impredicative dependent types and a wealth of unboxed or otherwise primitive types that cannot be substituted for polymorphic type arguments. It is not clear how to support linearity in GHC by extending its kind system. In contrast, our

design inherits many features of McBride's, including its compatibility with dependent types, and such compatibility is pretty much necessary to accommodate the dependently-typed kinds of GHC.

6.5 Alms

Alms [43] is an ML-like language based on affine types (a variant of linear types where values can be used *at most* once). It is uses the kinds to separate affine from unrestricted arguments.

It is a case in point for kind-based systems being more complex: for the sake polymorphism, Alms deploys an elaborate dependent kind system. Even if such a kind system could be added to an existing language implementation, Alms does not attempt to be backwards compatible with an ML dialect. In fact Morris notes:

> Despite the (not insignificant) complexity of [Alms], it is still not clear that it fully supports the expressiveness of traditional functional programming languages. For example, [Alms] has distinct composition operators with distinct types. These types are not related by the subtyping relation, as subtyping is contravariant in function arguments.

6.6 Ownership typing à la Rust

Rust [31] features ownership (aka uniqueness) types. But like the original formulation of linear logic, in Rust A stands for linear values, unrestricted values at type A are denoted $!A$, and duplication is explicit.

Rust quite beautifully addresses the problem of being mindful about memory, resources, and latency. But this comes at a heavy price: Rust, as a programming language, is specifically optimised for writing programs that are structured using the RAII pattern[8] (where resource lifetimes are tied directly or indirectly to stack allocated objects that are freed when the control flow exits the current lexical scope). Ordinary functional programs seldom fit this particular resource acquisition pattern so end up being second class citizens. For instance, tail-call optimization, crucial to the operational behaviour of many functional programs, is not usually sound. This is because resource liberation must be triggered when the tail call returns.

Hask-LL aims to hit a different point in the design space where regular non-linear expressions are the norm yet gracefully scaling up to latency-sensitive and resource starved programs is still possible.

6.7 Related type systems

The $\lambda^q_\to$ type system is heavily inspired from the work of Ghica and Smith [15] and McBride [33]. 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 into the judgement $\vdash_\omega \lambda x.(x, x) : A \multimap (A, A)$ being derivable. This derivation is not desirable: it means that there cannot be reusable definitions of linear functions. In terms of linear logic [16], 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. What we keep from McBride, is the typing rule of case (see Section 3), which can be phrased in terms of linear logic as making the natural function of type $!A \otimes !B \Longrightarrow !(A \otimes B)$ into an isomorphism. This choice is unusual from a linear logic perspective, but it is the key to be able to use types both linearly an unrestrictedly without intrusive multiplicity polymorphic annotation on all the relevant types.

[8] https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization

The literature on so-called coeffects [9, 35] uses type systems similar to Ghica and Smith, but with a linear arrow and multiplicities carried by the exponential modality instead. Brunel et al. [9], 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 it is not directly applicable to $\lambda^q_{\to}$.

6.8 Operational aspects of linear languages

Recent literature is surprisingly quiet on the operational aspects of linear types, and rather concentrates on uniqueness types [31, 38].

Looking further back, Wakeling and Runciman [47] produced a complete implementation of a language with linear types, with the goal of improving the performance. Their implementation features a separate linear heap, as Section 4 where they allocate as much as possible in the linear heap, as modelled by the strengthened semantics. However, Wakeling and Runciman did not manage to obtain consistent performance gains. On the other hand, they still manage to reduce GC usage, which may be critical in distributed and real-time environments, where latency that matters more than throughput.

Wakeling and Runciman propose to not attempt prompt free of thunks and only taking advantage of linearity for managing the lifetimes of large arrays. Our approach is similar: we advocate exploiting linearity for operational gains on large data structures (but not just arrays) stored off-heap. we go further and leave the management of external (linear) data to external code, only accessing it via an API. Yet, our language supports an implementation where each individual constructor with multiplicity 1 can be allocated on a linear heap, and deallocated when it is pattern matched. Implementing this behaviour is left for future work.

7 CONCLUSION AND FUTURE WORK

This article demonstrated 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 semantics, and
- 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 without being duplicated.

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. The main difference between the implementation and $\lambda^q_{\to}$is that the implementation is adapted to bidirectionality: typing contexts go in, inferred multiplicities come out (and are compared to their expected values). As we hoped, this design integrates very well in GHC.

It is worth stressing that, in order to implement foreign data structures like we advocate as a means to provide safe access to resources or reduce GC pressure and latency, we only need to modify the type system: primitives to manipulate foreign data can be implemented in user libraries using the foreign function interface. This helps keeping the prototype lean, since GHC's runtime system (RTS) is unaffected.

7.1 Dealing with exceptions

Exceptions run afoul of linearity. Consider for instance the expression *error* `"oops"` $+ x$, for some linear x. Evaluating x may be required in order to free resources, but x will not be evaluated, hence the resources will linger.

Haskell program can raise exceptions even during the evaluation of pure code [36], so we have to take it into account in order to demonstrate the eventual deallocation of resources. Both Thrippleton and Mycroft [42] and Tov and Pucella [44] develop solutions, but they rely on effect type systems, which are intrusive changes to make to an existing compiler. Moreover, effect type systems would not be compatible with Haskell's asynchronous exception mechanism [30].

Because we are using explicit allocators for resources such as $\mathit{withMailbox} :: (\mathit{MB} \multimap \mathit{IO}\ a) \multimap \mathit{IO}\ a$, these allocators can be responsible for safe deallocation in response to exceptions, internally making use of the bracket operation [30, Section 7.1]. A full justification of this hypothesis is left for future work.

7.2 Fusion

Inlining is a staple of program optimisation, exposing opportunities for many program transformation including fusion. Not every function can be inlined without negative effects on performance: inlining a function with two use sites of the argument may result in duplicating a computation.

In order to discover inlining opportunities GHC deploys a cardinality analysis [39] which determines how many times functions use their arguments. The limitation of such an analysis is that it is necessarily heuristic (the problem is undecidable). Consequently, it can be hard for the programmer to rely on such optimisations: a small, seemingly innocuous change can prevent a critical inlining opportunity and have rippling effects throughout the program. Hunting down such a performance regression proves painful in practice.

Linear types address this issue and serve as a programmer-facing interface to inlining: because it is always safe to inline a linear function, we can make it part of the *semantics* of linear functions that they are always inlined. In fact, the system of multiplicity annotation of $\lambda^q_\to$ can be faithfully embedded the abstract domain presented by Sergey et al. [39]. This gives confidence in the fact that multiplicity annotation can serve as cardinality *declarations*.

Formalising and implementing the integration of multiplicity annotation in the cardinality analysis is left as future work.

7.3 Extending multiplicities

For the sake of this article, we use only 1 and ω as possibilities. But in fact $\lambda^q_\to$ can readily be extended to more multiplicities: we can follow Ghica and Smith [15] and McBride [33], which work with abstract sets of multiplicities. In particular, in order to support dependent types, we additionally need a 0 multiplicity.

Applications of multiplicities beyond linear logic seem to often have too narrow a focus to have their place in a general purpose language such as Haskell. Ghica and Smith [15] propose to use multiplicities to represent real time annotations, and Petricek et al. [35] show how to use multiplicities to track either implicit parameters (*i.e.* dynamically scoped variables) or the size of the history that a dataflow program needs to remember.

To go further still, more multiplicities may prove useful. For instance we may want to consider a multiplicity for affine arguments (*i.e.* arguments which can be used *at most once*).

The general setting for $\lambda^q_\to$ is an ordered-semiring of multiplicities (with a join operation for type inference). The rules are mostly unchanged with the *caveat* that case_q must exclude $q = 0$ (in particular

we see that we cannot substitute multiplicity variables by 0). The variable rule is modified as:

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

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

REFERENCES

[1] Amal Ahmed, Matthew Fluet, and Greg Morrisett. 2007. L^3: A Linear Language with Locations. *Fundamenta Informaticae* 77, 4 (2007), 397–449.

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

[3] Thibaut Balabonski, François Pottier, and Jonathan Protzenko. 2016. The Design and Formalization of Mezzo, a Permission-Based Programming Language. *ACM Trans. Program. Lang. Syst.* 38, 4, Article 14 (Aug. 2016), 94 pages. DOI:http://dx.doi.org/10.1145/2837022

[4] 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.

[5] P. N. Benton. 1995. *A mixed linear and non-linear logic: Proofs, terms and models*. Springer Berlin Heidelberg, Berlin, Heidelberg, 121–135. DOI:http://dx.doi.org/10.1007/BFb0022251

[6] 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.

[7] Jean-Philippe Bernardy and Josef Svenningsson. 2015. On the Duality of Streams. (2015). https://github.com/jyp/organ/blob/master/Organ.lhs.

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

[9] 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

[10] Arthur Charguéraud and François Pottier. 2008. Functional translation of a calculus of capabilities. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*. 213–224. DOI:http://dx.doi.org/10.1145/1411204.1411235

[11] Edsko de Vries. 2017. Linearity, Uniqueness, and Haskell. (2017). http://edsko.net/2017/01/08/linearity-in-haskell/.

[12] James Fisher. 2016. Low latency, large working set, and GHC's garbage collector: pick two of three. (2016). https://blog.pusher.com/latency-working-set-ghc-gc-pick-two/.

[13] Brad Fitzpatrick. 2004. Distributed Caching with Memcached. *Linux J.* 2004, 124 (Aug. 2004), 5–. http://dl.acm.org.proxyiub.uits.iu.edu/citation.cfm?id=1012889.1012894

[14] M. Gaboardi, A. Haeberlen, J. Hsu, A. Narayan, and B.C. Pierce. 2013. Linear Dependent Types for Differential Privacy. (2013).

[15] 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

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

[17] Martin Hofmann. 2000. In-place update with linear types or How to compile functional programs into malloc()-free C. (2000). Preprint.

[18] Kohei Honda. 1993. *Types for dyadic interaction*. Springer Berlin Heidelberg, Berlin, Heidelberg, 509–523. DOI: http://dx.doi.org/10.1007/3-540-57208-2_35

[19] Oleg Kiselyov. 2012. Iteratees. In *Functional and Logic Programming - 11th International Symposium, FLOPS 2012, Kobe, Japan, May 23-25, 2012. Proceedings*. 166–181. DOI:http://dx.doi.org/10.1007/978-3-642-29822-6_15

[20] 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

[21] Edward A. Kmett, Rúnar Bjarnason, and Josh Cough. 2015. The machines package. (2015). https://github.com/ekmett/machines/

[22] Neelakantan R. Krishnaswami and Nick Benton. 2011. A Semantic Model for Graphical User Interfaces. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*. ACM, New York, NY, USA, 45–57. DOI:http://dx.doi.org/10.1145/2034773.2034782
[23] Yves Lafont. 1988. The linear abstract machine. *Theoretical Computer Science* 59, 1 (1988), 157–180.
[24] John Launchbury. 1993. A Natural Semantics for Lazy Evaluation. In *POPL*. 144–154.
[25] 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
[26] 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.
[27] John D. C. Little. 1961. A Proof for the Queuing Formula: L = λW. *Operations Research* 9, 3 (1961), 383–387. DOI: http://dx.doi.org/10.1287/opre.9.3.383 arXiv:http://dx.doi.org/10.1287/opre.9.3.383
[28] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn. 2012. On the role of burst buffers in leadership-class storage systems. In *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*. 1–11. DOI:http://dx.doi.org/10.1109/MSST.2012.6232369
[29] O. C. Marcu, A. Costan, G. Antoniu, and M. S. Pérez-Hernández. 2016. Spark Versus Flink: Understanding Performance in Big Data Analytics Frameworks. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. 433–442. DOI:http://dx.doi.org/10.1109/CLUSTER.2016.22
[30] Simon Marlow, Simon Peyton Jones, Andrew Moran, and John Reppy. 2001. Asynchronous Exceptions in Haskell. *SIGPLAN Not.* 36, 5 (May 2001), 274–285. DOI:http://dx.doi.org/10.1145/381694.378858
[31] 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
[32] 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.
[33] 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
[34] 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
[35] 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
[36] Simon Peyton Jones, Alastair Reid, Fergus Henderson, Tony Hoare, and Simon Marlow. 1999. A Semantics for Imprecise Exceptions. *SIGPLAN Not.* 34, 5 (May 1999), 25–36. DOI:http://dx.doi.org/10.1145/301631.301637
[37] 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
[38] François Pottier and Jonathan Protzenko. 2013. Programming with permissions in Mezzo. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Functional Programming (ICFP'13)*. 173–184.
[39] 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
[40] Michael Snoyman. 2015. The conduit package. (2015). http://www.stackage.org/package/conduit
[41] Michael Thompson. 2017. The streaming package. (2017). https://github.com/michaelt/streaming
[42] Richard Thrippleton and Alan Mycroft. 2007. Memory safety with exceptions and linear types. (2007).
[43] Jesse A Tov and Riccardo Pucella. 2011. Practical affine types. In *POPL*. ACM, 447–458.
[44] Jesse A. Tov and Riccardo Pucella. 2011. A theory of substructural types and control. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*. 625–642. DOI:http://dx.doi.org/10.1145/2048066. 2048115
[45] Philip Wadler. 1990. Linear types can change the world. In *Programming Concepts and Methods*, M Broy and C B Jones (Eds.). North-Holland.
[46] Philip Wadler. 2012. Propositions as Sessions. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional Programming (ICFP '12)*. ACM, New York, NY, USA, 273–286.
[47] David Wakeling and Colin Runciman. 1991. Linearity and laziness. In *Functional Programming Languages and Computer Architecture*. Springer, 215–240. https://www.cs.york.ac.uk/plasma/publications/pdf/WakelingRuncimanFPCA91.pdf
[48] Edwin Westbrook, Jisheng Zhao, Zoran Budimlić, and Vivek Sarkar. 2012. *Practical Permissions for Race-Free Parallelism*. Springer Berlin Heidelberg, Berlin, Heidelberg, 614–639. DOI:http://dx.doi.org/10.1007/978-3-642-31057-7_27

Received February 2017