Destination calculus

A linear λ−calculus for pure, functional memory updates

ARNAUD SPIWACK, Tweag, France
THOMAS BAGREL, LORIA/Inria, France and Tweag, France

We present the destination calculus, a linear λ−calculus for pure, functional memory updates. We introduce the syntax, type system, and operational semantics of the destination calculus, and prove type safety formally in the Coq proof assistant.

We show how the principles of the destination calculus can form a theoretical ground for destination-passing style programming in functional languages. In particular, we detail how the present work can be applied to Linear Haskell to lift the main restriction of DPS programming in Haskell as developed in [1]. We illustrate this with a range of pseudo-Haskell examples.

ACM Reference Format:
Arnaud Spiwack and Thomas Bagrel. 2024. Destination calculus: A linear λ−calculus for pure, functional memory updates. In . ACM, New York, NY, USA, 23 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Destination-passing style programming takes its root in the early days of imperative programming. In such language, the programmer is responsible for managing memory allocation and deallocation, and thus is it often unpractical for function calls to allocate memory for their results themselves. Instead, the caller allocates memory for the result of the callee, and passes the address of this output memory cell to the callee as an argument. This is called an *out parameter*, *mutable reference*, or even *destination*.

But destination-passing style is not limited to imperative settings; it can be used in functional programming as well. One example is the linear destination-based API for arrays in Haskell[2], which enables the user to build an array efficiently in a write-once fashion, without sacrificing the language identity and main guarantees. In this context, a destination points to a yet-unfilled memory slot of the array, and is said to be *consumed* as soon as the associated hole is written to. In this paper, we continue on the same line: we present a linear λ−calculus embedding the concept of *destinations* as first-class values, in order to provide a write-once memory scheme for pure, functional programming languages.

Why is it important to have destinations as first-class values? Because it allows the user to store them in arbitrary control or data structures, and thus to build complex data structures in arbitrary order/direction. This is a key feature of first-class DPS APIs, compared to ones in which destinations are inseparable from the structure they point to. In the latter case, the user is still forced to build the structure in its canonical order (e.g. from the leaves up to the root of the structure when using data constructors).

POPL'25, January 19 – 25, 2025, Denver, Colorado

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 WORKING WITH DESTINATIONS

TODO: Some introductory words

2.1 Building up a vocabulary

In its simplest form, destination passing, much like continuation passing, is the idea of explicitly receiving a location where to return a value to (we say that the destination is filled when the supplied location is written to). Instead of a function with signature $T \to U$, in λ_d you would have $T \to \lfloor U \rfloor \to 1$, where $\lfloor U \rfloor$ is read "destination of type U". For instance, here is a destination-passing version of the identity function:

$$\begin{aligned} &\mathbf{dId} : T \to \lfloor T \rfloor \to 1 \\ &\mathbf{dId}\ x\ d \triangleq d \blacktriangleleft x \end{aligned}$$

We think of a destination as a reference to an uninitialized memory location, and $d \blacktriangleleft x$ (read "fill d with x") as writing x to the memory location.

The form $d \blacktriangleleft x$ is the simplest way to use a destination. But we don't have to fill a destination with a whole value in a single step. Destinations can be filled piecemeal.

$$\begin{aligned} &\mathbf{fillWithInl} : \lfloor T \oplus U \rfloor \to \lfloor T \rfloor \\ &\mathbf{fillWithInl}\ d \triangleq d \triangleleft \mathsf{Inl} \end{aligned}$$

In this example, we're building a value of type $T \oplus U$ by setting the outermost constructor to Inl. We think of $d \triangleleft \mathsf{Inl}$ as allocating memory to store a block of the form $\mathsf{Inl}\ \square$, write the address of that block to the location that d points to, and return a destination pointing to the uninitialized argument of Inl.

Notice that we are constructing the term from the outermost constructor inward: we've built a value of the form $\mathsf{Inl}\ \square$ but we have yet to describe the constructor's payload (we call such incomplete values "hollow cosntructors"). This is opposite to how functional programming usually works, where values are built from the innermost constructors outward: first we make a value v and only then can we use Inl to make an $\mathsf{Inl}\ v$. This will turn out to be a key ingredient in the expressiveness of destination passing.

Yet, everything we've shown so far could have been done with continuations. So it's worth asking: how are destination different from continuations? Part of the answer lies in our intention to represent destinations as pointers to uninitialized memory (see Section 9). But where destinations really differ from continuations is when there are several destinations. Then you can (indeed you must!) fill all the destinations; whereas when you have multiple continuations, you can only return to one of them. Multiple destination arises from filling destination of tuples:

$$\begin{aligned} &\mathbf{fillWithAPair} : \lfloor T \otimes U \rfloor \to \lfloor T \rfloor \otimes \lfloor U \rfloor \\ &\mathbf{fillWithAPair}\ d \triangleq d \triangleleft (,) \end{aligned}$$

To fill a destination for a pair, we must fill both the first field and the second field. In plain English, it sounds obvious, but the key remark is that **fillWithAPair** doesn't exist on continuations.

Values with holes. Let's now turn to how we can use the result made by filling destinations. Observe, as a preliminary remark, that while a destination is used to build a structure, the type of the structure being built might be different from the type of the destination. For instance, **fillWithInl**, above, returns a destination $\lfloor T \rfloor$ while it is used to build a structure of type $T \oplus U$. To represents this, λ_d uses a type $S \ltimes \lfloor T \rfloor$ for a structure of type S missing a value of type T to be complete (we say it has a hole of type T). There can be several holes in S, in which case the right-hand side is a tuple of destinations: $S \ltimes (\lfloor T \rfloor \otimes \lfloor U \rfloor)$.

Maybe it would be wise to explain the notations for sums and tuples, since the linear-logic notations are less standard than the ccc ones

We probably want to give a reading for the fill-with-a-constructor construction.

Introduce the terminology "hollow constructor" here?

For some reason this v is grey. → Thomas : that's the case for non-terminals in the grammar

Somewhere around this point it would be good to explain the difference between holes and destinations.

The form $S \ltimes \lfloor T \rfloor$ is read "S ampar destination of T". The name "ampar" stands for "asymmetric memory par"; the reasons for this name will become apparent as we get into more details of λ_d in Section 5.1. For now, it's sufficient to observe that $S \ltimes \lfloor T \rfloor$ is akin to a $S \parr T^\perp$ in linear logic, indeed you can think of $S \ltimes \lfloor T \rfloor$ as a (linear) function from T to S. That values with holes could be seen a linear functions was first observed in [5], we elaborate on the value of having a par type rather than a function type in Section 4. A similar connective is called `Incomplete` in [1].

Destinations always exist within the context of a structure with holes. A hole of type T inside S represents the fact that S contains uninitialized memory that will have to hold a T for S to be readable without errors; it only denotes the absence of a value and thus cannot be manipulated directly. A destination $\lfloor T \rfloor$, on the other hand, is a first-class value that witnesses the presence of a hole of type T inside S and can be used to complete the structure. To access the destinations, λ_d provides a **map** construction, which lets us apply a function to the right-hand side of an ampar:

DIFF-HOLE-DEST

$$\begin{aligned} &\mathbf{fillWithInl'} : S \ltimes \lfloor T \oplus U \rfloor \to S \ltimes \lfloor T \rfloor \\ &\mathbf{fillWithInl'}\ x \triangleq x \triangleright \mathbf{map}\ d \mapsto d \triangleleft \mathsf{Inl} \end{aligned} \qquad \begin{aligned} &\mathbf{fillWithAPair'} : S \ltimes \lfloor T \otimes U \rfloor \to S \ltimes (\lfloor T \rfloor \otimes \lfloor U \rfloor) \\ &\mathbf{fillWithAPair'}\ x \triangleq x \triangleright \mathbf{map}\ d \mapsto d \triangleleft \mathsf{Inl} \end{aligned}$$

It may be more readable to have fillWithInl' and fillWithAPair' side by side. Maybe they're too wide though

To tie this up, we need a way to introduce and to eliminate values with holes. Values with holes are introduced with **alloc** which creates a value $T \ltimes \lfloor T \rfloor$, which is simply a hole expecting a value of type T. Values with holes are eliminated with[1] $\mathbf{from}'_\ltimes : S \ltimes 1 \to S$: if all the destinations have been filled, then a value with holes is really just a normal, complete value.

Equipped with these, we can, for instance, derive traditional constructors from piecemeal filling. In fact, λ_d doesn't have primitive constructor forms, they are syntactic sugar. We show here the definition of Inl and (,), but the other constructors are derived similarly.

$$\begin{aligned} &\mathsf{Inl} : T \to T \oplus U \\ &\mathsf{Inl}\ x \triangleq \mathbf{from}'_\ltimes\ (\mathbf{alloc} \triangleright \mathbf{map}\ d \mapsto d \triangleleft \mathsf{Inl} \blacktriangleleft x) \end{aligned}$$

$$\begin{aligned} &(,) : T \to U \to T \otimes U \\ &(x, y) \triangleq \mathbf{from}'_\ltimes\ (\mathbf{alloc} \triangleright \mathbf{map}\ d \mapsto (d \triangleleft (,)) \triangleright \mathbf{case}\ (d_1, d_2) \mapsto d_1 \blacktriangleleft x \fatsemi d_2 \blacktriangleleft y) \end{aligned}$$

Wording slightly awkward. Revisit at some point. Maybe point out that it makes destinations strictly more expressive than traditional constructor. Maybe discuss as Thomas did at some point that this means that the order in which a structure is built is actually arbitrary, not just outside-in.

Purity. At this point, the reader may be forgiven for feeling distressed at all the talk of filling by mutation and uninitialized memory. How is it consistent with our claim to be building a pure language?

Indeed, unrestricted use of destination can lead to rather catastrophic results. The simplest such issue happens when we forget to fill a destination:

$$\begin{aligned} &\mathbf{forget} : T \\ &\mathbf{forget} \triangleq \mathbf{from}'_\ltimes\ (\mathbf{alloc} \triangleright \mathbf{map}\ d \mapsto ()) \end{aligned}$$

Here, **forget** claims to be a value of type T, but it's really just a hole, just uninitialized memory. So any attempt to use **forget** will read uninitialized memory.

A similar situation can happen if we reuse a destination.

TODO: Syntactic sugar: let → Thomas: I added **let** $x := t$ **in** u and its linebreak-allowing versions

$$\begin{aligned} &\mathbf{overwrite} : T \to T \oplus U \\ &\mathbf{overwrite}\ x \triangleq \mathbf{from}'_\ltimes\ (\mathbf{alloc} \triangleright \mathbf{map}\ d \mapsto (\lambda d' \mapsto d \triangleleft \mathsf{Inr} \fatsemi d' \blacktriangleleft x)\ (d \triangleleft \mathsf{Inl})) \end{aligned}$$

Thomas: I do not really like this paragraph. I would like this paragraph to be the other way around: we want a pure model, in particular a model that doesn't break observable immutability. So write-(exactly)-once memory model, with no way to read before every hole has been writ-

These programs are rejected by λ_d , using a linear type system, ensuring that λ_d is a pure and safe language. This justifies the claim that we can think of destinations as write-once memory locations.

Linearity of destination isn't the only measure that we need to take in order to ensure that uninitialized memory is never read. Indeed consider a value of type $v : S \ltimes T$. In memory v is a

[1] As the name suggest, there is a more general elimination $\mathbf{from}_\ltimes$. It will be discussed in Section 6.

value of type S except that it has some uninitialized memory. So we have really careful when we read anything in v. In λ_d we take the bluntest restriction: we simply do not give any way to read v at all.

This is quite incorrect, v is rather a pair, and only its left side cannot be read. At least, if we stick to a general T on the right hand side, it would be too much of a lie to say that there isn't anything else than a S in memory

2.2 Functional queues, with destinations

If we suppose equirecursive types and a fixed-point operator, then λ_d becomes expressive enough to build any usual data structure.

Linked lists. For starters, we can define lists as the fixpoint of the functor $X \mapsto 1 \oplus (\mathsf{T} \otimes X)$ where T is the type of list items. Following the recipe that we've outline so far, instead of defining the usual "nil" [] and "cons" (::) constructors, we define the more general "fillNil" ◃[] and "fillCons" ◃(::) operators, as presented in Figure 1.

$$\mathsf{List\ T} \overset{\text{rec}}{\triangleq} 1 \oplus (\mathsf{T} \otimes (\mathsf{List\ T}))$$

$$\triangleleft[] \ : \ \lfloor \mathsf{List\ T} \rfloor \to 1$$
$$d \triangleleft [] \triangleq d \triangleleft \mathsf{Inl} \triangleleft ()$$

$$\triangleleft(::) \ : \ \lfloor \mathsf{List\ T} \rfloor \to \lfloor \mathsf{T} \rfloor \otimes \lfloor \mathsf{List\ T} \rfloor$$
$$d \triangleleft (::) \triangleq d \triangleleft \mathsf{Inr} \triangleleft (,)$$

Fig. 1. List implementation in equirecursive destination calculus

The related work must mention that Minamide has this same restriction. Whereas as SNAX (semi-axiomatisable sequent calculus) doesn't, in exchange of having the ability to wait on unfilled destination and requiring a parallel execution semantics. → Thomas: in SNAX we thus get involutive dest, and symetric ampar, as everything is now a promise

Just like we did in Section 2.1 for the primitive constructors, we can recover the "cons" constructor: $\mathsf{T} \otimes (\mathsf{List\ T}) \to \mathsf{List\ T}$:

$$(::) \ : \ \mathsf{T} \otimes (\mathsf{List\ T}) \to \mathsf{List\ T}$$
$$(::)\ x\ xs \triangleq \mathbf{from}'_{\ltimes}\ (\mathbf{alloc} \triangleright \mathbf{map}\ d \mapsto (d \triangleleft (::)) \triangleright \mathbf{case}\ (dx, dxs) \mapsto dx \blacktriangleleft x \fatsemi dxs \blacktriangleleft xs)$$

Going from a "fill" operator to the associated constructor is completely generic, and with more metaprogramming tools, we could build this transformation into the language.

Difference lists. While linked lists are optimized for the prepend operation ("cons"), they are not efficient for appending or concatenation, as it requires a full copy (or traversal at least) of the first list before the last cons cell can be changed to point to the head of the second list.

Difference lists are a data structure that allows for efficient concatenation. In functional languages, difference lists are often encoded using a function that take a tail, and returns the previously-unfinished list with the tail appended to it. For example, the difference list $x_1{:}x_2{:}\ldots{:}x_k{:}\square$ is represented by the linear function $\lambda xs \mapsto x_1{:}x_2{:}\ldots{:}x_k{:}xs$. This encoding shines when list concatenation calls are nested to the left, as the function encoding delays the actual concatenation so that it happens in a more optimal, right-nested fashion.

I wanted to have an example here, but I couldn't find a natural way to write something bad since we truly don't have a tool to write them.

In destination calculus, we can go even further, and represent difference lists much like we would do in an imperative programming language (although in a safe setting here), as a pair of an incomplete list who is missing its tail, and a destination pointing to the missing tail's location. This is exactly what an ampar is designed to allow: thanks to an ampar, we can handle incomplete structures safely, with no need to complete them immediately. The incomplete list is represented by the left side of the ampar, and the destination is represented by its right side. Creating an empty difference list is exactly what the **alloc** primitive already does when specialized to type List T : it returns an incomplete list with no items in it, and a destination pointing to that cell so that the list can be built later through destination-filling primitives, together in an ampar: $\mathsf{List\ T} \ltimes \lfloor \mathsf{List\ T} \rfloor$. Type definition and operators for difference lists in destination calculus are presented in Figure 2.

Arnaud hasn't used in the previous section small-caps name for constructors and operations. If we are to use them in the next few sections, it would be good to introduce this vocabulary early.

$$\mathsf{DList}\ \mathsf{T} \triangleq (\mathsf{List}\ \mathsf{T}) \ltimes \lfloor \mathsf{List}\ \mathsf{T} \rfloor$$

$$\textbf{append} : \mathsf{DList}\ \mathsf{T} \to \mathsf{T} \to \mathsf{DList}\ \mathsf{T}$$
$$ys\ \textbf{append}\ y \triangleq ys \triangleright \textbf{map}\ dys \mapsto (dys \triangleleft (::)) \triangleright \textbf{case}\ (dy, dys') \mapsto dy \blacktriangleleft y \fatsemi dys'$$

$$\textbf{concat} : \mathsf{DList}\ \mathsf{T} \to \mathsf{DList}\ \mathsf{T} \to \mathsf{DList}\ \mathsf{T}$$
$$ys\ \textbf{concat}\ ys' \triangleq ys \triangleright \textbf{map}\ d \mapsto d \mathbin{\triangleleft\!\bullet} ys'$$

$$\textbf{to}_{\mathsf{List}} : \mathsf{DList}\ \mathsf{T} \to \mathsf{List}\ \mathsf{T}$$
$$\textbf{to}_{\mathsf{List}}\ ys \triangleq \textbf{from}'_{\ltimes}\ (ys \triangleright \textbf{map}\ d \mapsto d \triangleleft [])$$

Fig. 2. Difference list implementation in equirecursive destination calculus

The **append** simply appends an element at the end of the list. It uses "fillCons" to link a new hollow "cons" cell at the end of the list, and then handles the two associated destinations dy and dt. The former, representing the item slot, is fed with the item to append, while the latter, representing the slot for the tail of the resulting difference list, is returned and so stored back in the right side of the ampar. If that second destination was consumed, and not returned, we would end up with a regular linked list, instead of a difference list.

The **concat** operator concatenates two difference lists by writing the head of the second one to the hole left at the end of the first one. This is done using the "fillComp" primitive $\triangleleft\!\bullet : \lfloor_n U_1 \rfloor \to U_1 \ltimes U_2 \upharpoonright_{1 \cdot n} \to U_2$[2]. It takes a destination on its left-hand side, and an ampar on its right-hand side. The left side of the ampar (type U_1) is fed to the destination (so the incomplete structure is written to a larger incomplete structure from which the destination originated from), and the right side of the ampar (type U_2) is returned.

Here the left side of the second ampar is the second incomplete list, which is pasted at the end of the first incomplete list, consuming the destination of the first difference list in the process. Then the right side of the second ampar, that is to say the destination to the yet-unspecified tail of the second difference list, is returned, and stored back in the resulting ampar (thus serves as the new destination to the tail of the the resulting difference list).

Finally, the $\textbf{to}_{\mathsf{List}}$ operator converts a difference list to a regular list by writing the "nil" constructor to the hole left in the incomplete list using "fillNil".

We can note that although this exemple is typical of destination-style programming, it doesn't use the first-class nature of destinations that our calculus allows, and thus can be implemented in other destination-passing style frameworks such as [3] and [4]. We will see in the next sections what kind of programs can be benefit from first-class destinations.

Efficient queue using previously defined structures. The usual functional encoding for a queue is two use a pair of lists, one representing the front of the queue, and keeping the element in order, while the second list represent the back of the queue, and is kept in reversed order (e.g the latest inserted element will be at the front of the second list).

With such a queue implementation, dequeueing the front element is efficient (just pattern-match on the first cons cell of the first list, $O(1)$), and enqueuing a new element is efficient too (just add a new "cons" cell at the front of the second list, $O(1)$ too). However, when the first list is depleted, one has to transfer elements from the second list to the first one, and as such, has to reverse the second list, which is a $O(n)$ operation (although it is amortized).

[2]In this particular context, $U_1 = \mathsf{List}\ \mathsf{T}$ and $U_2 = \lfloor \mathsf{List}\ \mathsf{T} \rfloor$, so "fillComp" has signature $\triangleleft\!\bullet : \lfloor \mathsf{List}\ \mathsf{T} \rfloor \to \mathsf{DList}\ \mathsf{T} \upharpoonright_1 \to \lfloor \mathsf{List}\ \mathsf{T} \rfloor$

With access to efficient difference lists, as shown in the previous paragraph, we can replace the second list by a difference list, to maintain a quick **enqueue** operation (still $O(1)$), but remove the need for a **reverse** operation (as $\mathbf{to}_{\mathsf{List}}$ is $O(1)$ for difference lists). Nothing needs to change for the first list. The corresponding implementation is presented in Figure 3.

```
Queue T ≜ (List T)⊗(DList T)

singleton : T ⊸ Queue T
singleton x ≜ (Inr (x, Inl ()), alloc)

enqueue : Queue T ⊸ T ⊸ Queue T
q enqueue y ≜ q ▷ case (xs, ys) ↦ (xs, ys append y)

dequeue : Queue T ⊸ 1⊕(T⊗(Queue T))
dequeue q ≜ q ▷ case {
                  (Inr (x, xs), ys) ↦ Inr (x, (xs, ys)),
                  (Inl (), ys) ↦ (to_List ys) ▷ case {
                      Inl () ↦ Inl (),
                      Inr (x, xs) ↦ Inr (x, (xs, alloc))
                  }
              }
```

Fig. 3. Queue implementation in equirecursive destination calculus

Replace by the empty queue

The **singleton** operator creates a pair of a list with a single element, and a fresh difference list (obtained via **alloc**).

The **enqueue** operator appends an element to the difference list, while letting the front list unchanged.

The **dequeue** operator is more complex though. It first checks if there is at least one element available in the front list. If there is, it extracts the element x by removing the first "cons" cell of the front list, and returns it alongside the rest of the queue (xs, ys). If there isn't, it converts the difference list ys to a normal list, and pattern-matches on it to look for an available element. If none is found again, it returns Inl () to signal that the queue is definitely empty. If an element x is found, then it returns it alongside the updated queue, made of the tail xs of the difference list turned into a list, and a fresh difference list given by **alloc**.

3 LIMITATIONS OF THE PREVIOUS APPROACH

Everything described above is in fact already possible in destination-passing style for Haskell as presented in [1]. However, there is one fundamental limitation in [1]: the inability to store destinations in destination-based data structures.

Indeed, that first approach of destination-passing style for Haskell can only be used to build non-linear data structures. More precisely, the "fillLeaf" operator ($\triangleleft$) can only take arguments with multiplicity ω. This is in fact a much stronger restriction than necessary ; the core idea is *just* to prevent any destination (which is always a linear resource) to appear somewhere in the right-hand side of "fillLeaf".

Don't start with the solution, start with the problem. Then we can explain how previous attempts addresses it. In JFLA: fill's right-hand operand wants omega, in

3.1 The problem with stored destinations

One core assumption of destination-passing style programming is that once a destination has been linearly consumed, the associated hole has been written to.

However, in a realm where destinations $\lfloor T \rfloor$ can be of arbitrary inner type T, they can in particular be used to store a destination itself when $T = \lfloor T' \rfloor$!

We have to mark the value being fed in a destination as linearly consumed, so that it cannot be both stored away (to be used later) and pattern-matched on/used in the current context. But that means we have to mark the destination $d : \lfloor T' \rfloor$ as linearly consumed too when it is fed to $dd : \lfloor \lfloor T' \rfloor \rfloor$ in $dd \blacktriangleleft d$.

As a result, there are in fact two ways to consume a destination: feed it now with a value, or store it away and feed it later. The latter is a much weaker form of consumption, as it doesn't guarantee that the hole associated to the destination has been written to *now*, only that it will be written to later. So our assumption above doesn't hold in general case.

The issue is particularly visible when trying to give semantics to the **alloc′** operator with signature $\textbf{alloc}' : (\lfloor T \rfloor \to 1) \to T$. It reads: "given a way of consuming a destination of type T, I'll return an object of type T". This is an operator we very much want in our system!

The morally correct semantics (in destination calculus pseudo-syntax) would be:

$$\textbf{alloc}'\ (\lambda d \mapsto t) \longrightarrow \textbf{withTmpStore}\ \{h := \square\}\ \textbf{do}\ \{t[d := \to h]\ ;\ \textbf{deref}\ \to h\}$$

I'm [Arnaud] really unsure about introducing this one-off notation that we're never using again. Even though I like the use of the underwave to highlight the problem.

It works as expected when the function supplied to **alloc′** will indeed use the destination to store a value:

$$\begin{aligned}
&\textbf{alloc}'\ (\lambda d \mapsto d \triangleleft \mathsf{Inl} \triangleleft ()) \\
\longrightarrow\ &\textbf{withTmpStore}\ \{h := \square\}\ \textbf{do}\ \{\to h \triangleleft \mathsf{Inl} \triangleleft ()\ ;\ \textbf{deref}\ \to h\} \\
\longrightarrow\ &\textbf{withTmpStore}\ \{h := \mathsf{Inl}\ ()\}\ \textbf{do}\ \{\textbf{deref}\ \to h\} \\
\longrightarrow\ &\mathsf{Inl}\ ()
\end{aligned}$$

However this falls short when calls to **alloc′** are nested in the following way (where $dd : \lfloor \lfloor 1 \rfloor \rfloor$ and $d : \lfloor 1 \rfloor$):

$$\begin{aligned}
&\textbf{alloc}'\ (\lambda dd \mapsto \textbf{alloc}'\ (\lambda d \mapsto dd \blacktriangleleft d)) \\
\longrightarrow\ &\textbf{withTmpStore}\ \{hd := \square\}\ \textbf{do}\ \{\textbf{alloc}'\ (\lambda d \mapsto \to hd \blacktriangleleft d)\ ;\ \textbf{deref}\ \to hd\} \\
\longrightarrow\ &\textbf{withTmpStore}\ \{hd := \square\}\ \textbf{do}\ \{\textbf{withTmpStore}\ \{h := \square\}\ \textbf{do}\ \{\to hd \blacktriangleleft \to h\ ;\ \textbf{deref}\ \to h\}\ ;\ \textbf{deref}\ \to hd\} \\
\longrightarrow\ &\textbf{withTmpStore}\ \{hd := \to h\}\ \textbf{do}\ \{\textbf{withTmpStore}\ \{h := \square\}\ \textbf{do}\ \{\textbf{deref}\ \to h\}\ ;\ \textbf{deref}\ \to hd\}
\end{aligned}$$

The original term $\textbf{alloc}'\ (\lambda dd \mapsto \textbf{alloc}'\ (\lambda d \mapsto dd \blacktriangleleft d))$ is well typed, as the inner call to **alloc′** returns a value of type 1 (as d is of type $\lfloor 1 \rfloor$) and consumes d linearly. However, we see that because $\to h$ escaped to the parent scope by being stored in a destination of destination coming from the parent scope, the hole h has not been written to, and thus the inner expression $\textbf{withTmpStore}\ \{h := \square\}\ \textbf{do}\ \{\textbf{deref}\ \to h\}$ cannot reduce in a meaningful way.

One could argue that the issue comes from the destination-filling primitive $\triangleleft$ returning unit instead of a special value of a distinct *effect* type. However, the same issue arise if we introduce a distinct type $\|$ for the effect of filling a destination ; there is always a way to cheat the system and make a destination escape to a parent scope. This distinct type for effects has in fact existed during the early prototypes of destination calculus, but we removed it as it doesn't solve the scope escape for destination and is indistinguishable in practice from the unit type.

I remember that we did wonder about that at some point, so it's worth preempting the question. Though I'd rather we found a slightly different example that doesn't exploit the return type of fill than a paragraph recounting our life story.

3.2 Age control to prevent scope escape of destinations

The solution we chose is to instead track the age of destinations (as De-Brujin-like scope indices), and prevent a destination to escape into the parent scope when stored through age-control restriction on the typing rule of destination-filling primitives.

Age is represented by a commutative semiring, where ν indicates that a destination originates from the current scope, and $\uparrow$ indicates that it originates from the scope just before. We also extend ages to variables (a variable of age a stands for a value of age a). Finally, age ∞ is introduced for

I don't think this is the right place for this section. It should appear in the formal system I think.

variables standing in place of a non-age-controlled value. In particular, destinations can never have age ∞ in practice.

Semiring addition $+$ is used to find the age of a variable or destination that is used in two different branches of a program. Semiring multiplication $\cdot$ corresponds to age composition, and is in fact an integer sum on scope indices. ∞ is absorbing for both addition and multiplication.

Tables for the operations $+$ and $\cdot$ on ages are presented in Figure 4.

We pose $\uparrow^0 = \nu$ and $\uparrow^n = \uparrow\cdot\uparrow^{n-1}$

$+$	$\uparrow^n$	∞
$\uparrow^m$	if $n = m$ then $\uparrow^n$ else ∞	∞
∞	∞	∞

$\cdot$	$\uparrow^n$	∞
$\uparrow^m$	$\uparrow^{n+m}$	∞
∞	∞	∞

Fig. 4. Tables for age operations

Age commutative semiring is then combined with the multiplicity commutative semiring from [2] to form a canonical product commutative semiring that is used to represent the mode of each typing context binding in our final type system.

The main restriction to prevent parent scope escape is materialized the simplified typing rules of Figure 5.

$$\textsc{Ty-val-Dest}^\star \quad \frac{}{\to h :_\nu \lfloor \mathsf{T} \rfloor \vdash \to h : \lfloor \mathsf{T} \rfloor}$$

$$\textsc{Ty-term-FillLeaf}^\star \quad \frac{\Theta_1 \vdash \mathsf{t} : \lfloor \mathsf{T} \rfloor \qquad \Theta_2 \vdash \mathsf{t}' : \mathsf{T}}{\Theta_1 + \uparrow\cdot\Theta_2 \vdash \mathsf{t} \triangleleft \mathsf{t}' : 1}$$

Fig. 5. Simplified typing rules for age control of destinations

Typing a destination $\to h$ alone requires $\to h$ to have age ν in the context. And when storing a value through a destination, the ages of the value's dependencies in the context must be one higher than the corresponding ages required to type the value alone (this is the meaning of $\uparrow\cdot\Theta_2$).

Such a rule system prevents in particular the previous faulty expression $\to hd \blacktriangleleft \to h$ where $\to hd$ originates from the context parent to the one of $\to h$.

4 BREADTH-FIRST TREE TRAVERSAL

The core example that showcases the power of destination-passing style programming with first-class destination is breadth-first tree traversal:

> Given a tree, create a new one of the same shape, but with the values at the nodes replaced by the numbers $1 \ldots |T|$ in breadth-first order.

Indeed, breadth-first traversal implies that the order in which the structure must be populated (left-to-right, top-to-bottom) is not the same as the structural order of a functional binary tree i.e., building the leaves first and going up to the root.

In [1], the author presents a breadth-first traversal implementation that relies on first-class destinations so as to build the final tree in a single pass over the input tree. Their implementation, much like ours, uses a queue to store pairs of an input subtree and a destination to the corresponding output subtree. This queue is what materialize the breadth-first processing order: the leading pair $(\langle \textit{input subtree} \rangle, \langle \textit{dest to output subtree} \rangle)$ of the queue is processed, and its children pairs are added back at the end of the queue to be processed later.

Rework the next couple of paragraph to flow a little bit better.

However, as evoked earlier in Section 3.1, The API presented in [1] is not able to store linear data, and in particular destinations, in destination-based data structures. It is thus reliant on regular constructor-based Haskell data structures for destination storage.

This is quite impractical as we would like to use the efficient, destination-based queue implementation from Section 2.2 to power up the breadth-first tree traversal implementation[3]. In our present work fortunately, thanks to the finer age-control mechanism, we can store linear resources in destination-based structures without any issue. Our system is in fact self-contained, as any structure, whatever the use for it be, can be built using a small core of destination-based primitives (and regular data constructors can be retrieved from destination-based primitives, see Section 5.4).

Figure 6 introduces a few extra tools needed for implementation of breadth-first tree traversal, while Figure 7 presents the actual implementation of the traversal. This implementation is as similar as possible to the one from [1], as to make it easier to spot the few differences between the two systems.

$$\mathsf{Tree}\ T \triangleq 1 \oplus (T \otimes ((\mathsf{Tree}\ T) \otimes (\mathsf{Tree}\ T)))$$

$$\triangleleft\mathsf{Nil} : \lfloor_n \mathsf{Tree}\ T \rfloor \to 1$$
$$d \triangleleft \mathsf{Nil} \triangleq d \triangleleft \mathsf{Inl} \triangleleft ()$$

$$\triangleleft\mathsf{Node} : \lfloor_n \mathsf{Tree}\ T \rfloor \to \lfloor_n T \rfloor \otimes (\lfloor_n \mathsf{Tree}\ T \rfloor \otimes \lfloor_n \mathsf{Tree}\ T \rfloor)$$
$$d \triangleleft \mathsf{Node} \triangleq (d \triangleleft \mathsf{Inr} \triangleleft (,)) \triangleright \mathbf{case}\ (dv, dtlr) \mapsto (dv, dtlr \triangleleft (,))$$

$$\mathsf{Nat} \overset{\mathrm{rec}}{\triangleq} 1 \oplus \mathsf{Nat}$$

$$\mathbf{zero} : \mathsf{Nat}$$
$$\mathbf{zero} \triangleq \mathsf{Inl}\ ()$$

$$\mathbf{succ} : \mathsf{Nat} \to \mathsf{Nat}$$
$$\mathbf{succ}\ x \triangleq \mathsf{Inr}\ x$$

There's way too many mode annotations here. If we are to do that we have to dedicate some space above to explaining the modes and what all the constructions around them mean.

Fig. 6. Boilerplate for breadth-first tree traversal

The first important difference is that in the destination calculus implementation, the input tree of type $\mathsf{Tree}\ T_1$ is consumed linearly. The stateful transformer that is applied to each input node to get the output node value is also linear in its two arguments. The state has to be wrapped in an exponential $!_{1\infty}$ so that it can be extracted from the right side of the ampar at the end of the processing (with $\mathbf{from}_{\ltimes}$). We could imagine a more general version of the traversal, having no constraint on the state type, but necessitating a finalization function $S \to !_{1\infty} S'$ so that the final state can be returned.

The **go** function is in charge of consuming the queue containing the pairs of input subtrees and destinations to the corresponding output subtrees. It dequeues the first pair, and processes it. If the input subtree is Nil, it feeds Nil to the destination for the output tree and continues the processing of next elements with unchanged state. If the input subtree is a node, it writes a hollow Node constructor to the hole pointed to by the destination, processes the value of the node with the stateful transformer f, and continues the processing of the updated queue where children subtrees and their accompanying destinations have been enqueued.

mapAccumBFS is in charge of spawning the initial memory slot for the output tree together with the associated destination, and preparing the initial queue containing a single pair, made of the whole input tree and the aforementioned destination.

relabelDPS is a special case of **mapAccumBFS** that takes the skeleton of a tree (where node values are all unit) and returns a tree of integers, with the same skeleton, but with node values replaced by naturals $1 \ldots |T|$ in breadth-first order. The higher-order function passed to **mapAccumBFS** is quite verbose: it must consume the previous node value (unit), using ⨟, then extract the state (representing the next natural number to attribute to a node) from its two nested exponential

[3]This efficient queue implementation can be, and is in fact, implemented in [1]: see archive.softwareheritage.org/swh:1:cnt:29e9d1fd48d94fa8503023bee0d607d281f512f8. But it cannot store linear data

```
go : ((!1∞S) → T1 → (!1∞S)⊗T2) ων⟿ (!1∞S) → Queue (Tree T1⊗⌊Tree T2⌋) → (!1∞S)
go f st q ≜rec (dequeue q) ▷ case {
                  Inl () ↦ st,
                  Inr ((tree, dtree), q′) ↦ tree ▷ case {
                      Inl () ↦ dtree ◃ Nil ⨟ go f st q′,
                      Inr (x, (tl, tr)) ↦ (dtree ◃ Node) ▷ case
                          (dy, (dtl, dtr)) ↦ (f st x) ▷ case
                              (st′, y)↦ dy ◀ y ⨟ go f st′ (
                                  q′ enqueue (tl, dtl) enqueue (tr, dtr)
                              )
                  }
              }

mapAccumBFS : ((!1∞S) → T1 → (!1∞S)⊗T2) ων⟿ (!1∞S) → Tree T1 → Tree T2⊗(!1∞S)
mapAccumBFS f st tree ≜ from⊸ (alloc ▷ map dtree ↦ go f st (singleton (tree, dtree)))

relabelDPS : Tree 1 → (Tree Nat)⊗(!1∞(!ωνNat))
relabelDPS tree ≜ mapAccumBFS
                      (λex ↦ λun ↦ un ⨟ ex ▷ case
                          E1∞ ex′ ↦ ex′ ▷ case1∞
                              Eων st ↦ (E1∞ (Eων (succ st)), st))
                      (E1∞ (Eων (succ zero)))
                       tree
```

Fig. 7. Breadth-first tree traversal in destination-passing style

constructors, and finally return a pair, whose left side is the incremented natural wrapped back into its two exponential layers (new state), and whose right side is the plain natural representing the node value for the output subtree.

Two exponentials are needed here. The first one $!_{1\infty}$ is part of $\mathbf{from}_{\otimes}$ contract, and ensures that the state cannot capture destinations, so that it can be returned at the end of the processing (as **go** runs under a **map** over an ampar). The second one $!_{\omega\nu}$ allows the natural number to be used in a non-linear fashion: it is used once in **succ** *st* to serve as the new state, and another time as the value for the output node. With a more general $\mathbf{from}_{\otimes}$ operator, we would be able to use just one exponential layer $!_{\omega\infty}$ over the natural number to achieve the same result.

To Thomas: didn't you tell me you had plan of making something more general for this very case? I don't remember clearly. If so, don't forget to improve here

5 LANGUAGE SYNTAX

5.1 Introducing the *ampar*

Minamide's work[5] is the earliest record we could find of a functional calculus integrating the idea of incomplete data structures (structures with holes) that exist as first class values and can be interacted with by the user.

In that paper, a structure with a hole is named *hole abstraction*. In the body of a hole abstraction, the bound *hole variable* should be used linearly (exactly once), and must only be used as a parameter of a data constructor. In other terms, the bound *hole variable* cannot be pattern-matched on or used as a parameter of a function call. A hole abstraction is thus a weak form of linear lambda abstraction, which just moves a piece of data into a bigger data structure.

I'm [Arnaud] honestly uncomfortable at the quantity of mathematics written in this section without macros and completely uncorrelated from OTT. If we change any notation, we'll have to [illegible]

In fact, the type of hole abstraction (T_1, T_2)hfun in Minamine's work shares a lot of similarity with the separating implication or *magic wand* $T_1 \multimap T_2$ from separation logic: given a piece of

memory matching description T_1, we obtain a (complete) piece of memory matching description T_2.

Now, in classical linear logic, we know we can transform linear implication $\mathsf{T}_1 \multimap \mathsf{T}_2$ into $\mathsf{T}_1{}^{\perp} \parr \mathsf{T}_2$. Doing so for the *wand* type $(\mathsf{T}_1, \mathsf{T}_2)\mathsf{hfun}$ or $\mathsf{T}_1 \mathrel{-\!\!*} \mathsf{T}_2$ gives $\lfloor \mathsf{T}_1 \rfloor \mathbin{\widehat{\parr}} \mathsf{T}_2$, where $\lfloor \bullet \rfloor$ is memory negation, and $\widehat{\parr}$ is a memory *par* (weaker than the CLL *par* that allows more "interaction" of its two sides).

Transforming the hole abstraction from its original implication form to a *par* form let us consider the type $\lfloor \mathsf{T}_1 \rfloor$ of *sink* or *destination* of T_1 as a first class component of our calculus. We also get to see the hole abstraction aka memory par as a pair-like structure, where the two sides might be coupled together in a way that prevent using both of them simultaneously.

From memory par $\widehat{\parr}$ to ampar $\ltimes$. In CLL, the cut rule states that given $\mathsf{T}_1 \parr \mathsf{T}_2$, we can free up T_1 by providing an eliminator of T_2, or free up T_2 by providing an eliminator of T_1. The eliminator of T can be $\mathsf{T}^{\perp}$, or $\mathsf{T}^{\perp^{-1}} = \mathsf{T}'$ if T is already of the form $\mathsf{T}'^{\perp}$. In a classical setting, thanks to the involutive nature of negation $\bullet^{\perp}$, the two potential forms of the eliminator of T are equal.

In destination calculus though, we don't have an involutive memory negation $\lfloor \bullet \rfloor$. If we are provided with a destination of destination $\rightarrow h' : \lfloor \lfloor \mathsf{T} \rfloor \rfloor$, we know that some structure is expecting to store a destination of type $\lfloor \mathsf{T} \rfloor$. If ever that structure is consumed, then the destination stored inside will have to be fed with a value (remember we are in a linear calculus). So if we allocate a new memory slot of type $\boxed{h} : \mathsf{T}$ and its linked destination $\rightarrow h : \lfloor \mathsf{T} \rfloor$, and write $\rightarrow h$ to the memory slot pointed to by $\rightarrow h'$, then we can get back a value of type T at $\boxed{h}$ if ever the structure pointed to by $\rightarrow h'$ is consumed. Thus, a destination of destination is only equivalent to the promise of an eventual value, not an immediate usable one.

As a result, in destination calculus, we cannot have the same kind of cut rule as in CLL. This is, in fact, the part of destination calculus that was the hardest to design, and the source of a lot of early errors. For a destination of type $\lfloor \mathsf{T} \rfloor$, both storing it through a destination of destination $\lfloor \lfloor \mathsf{T} \rfloor \rfloor$ or using it to store a value of type T constitute a linear use of the destination. But only the latter is a genuine consumption in the sense that it guarantees that the hole associated to the destination has been written to! Storing away the destination of type $\lfloor \mathsf{T} \rfloor$ originating from $\mathsf{T} \mathbin{\widehat{\parr}} \lfloor \mathsf{T} \rfloor$ (through a destination of destination of type $\lfloor \lfloor \mathsf{T} \rfloor \rfloor$) should not allow to free up the T, as it would in a CLL-like setting.

However, we can recover a memory abstraction that is usable in practice if we know the nature of an memory par side:

- if the memory par side is a value made only of inert elements and destinations (negative polarity), then we can pattern-match/**map** on it, but we cannot store it away to free up the other side;
- if the memory par side is a value made only of inert elements and holes (positive polarity), then we can store it away in a bigger struct and free up the associated destinations (this is not an issue as the bigger struct will be locked by an memory par too), but we cannot pattern-match/**map** on it as it (may) contains holes;
- if one memory par side is only made of inert elements, we can in fact convert the memory par to a pair, as the memory par doesn't have any form of interaction between its sides.

It is important to note that the type of an memory par side is not really enough to determine the nature of the side, as a hole of type T and and inert value of type T are indistinguishable at the type level.

So we introduced a more restricted form of memory par, named *ampar* ($\ltimes$), for *asymmetrical memory par*, in which:

- the left side is made of inert elements (normal values or destinations from previous scopes) and/or holes if and only if those holes are compensated by destinations on the right side;
- the right side is made of inert elements and/or destinations.

As the right side cannot contain any holes, it is always safe to pattern-match or **map** on it. Because the left side cannot contain destinations from the current scope, it is always safe to store it away in a bigger struct and release the right side.

Finally, it is enough to check for the absence of destinations in the right side (which we can do easily just by looking at its type) to convert an *ampar* to a pair, as any remaining hole on the left side would be compensated by a destination on the right side.

Destinations from previous scopes are inert. In destination calculus, scopes are delimited by the **map** operation over ampars. Anytime a **map** happens, we enter a new scope, and any preexisting destination or variable see its age increased by one ($\uparrow$). As soon as a destination or variable is no longer of age 0 (ν), it cannot be used actively but only passively (e.g. it cannot be applied if it is a function, or used to store a value if it is a destination, but it can be stored away in a dest, or pattern-matched on).

This is a core feature of the language that ensures part of its safety.

5.2 Names and variables

The destination calculus uses two classes of names: regular variable names x, y, and *hnames* h, h_1, h_2 which are identifiers for a memory cell that hasn't been written to yet, as illustrated in Figure 8.

Hole names are represented by natural numbers under the hood, so they can act both as relative offsets or absolute positions in memory. Typically, when a structure is effectively allocated, its hole (and destination) names are shifted by the maximum hname encountered so far in the program ; this corresponds to finding the next unused memory cell in which to write new data.

We sometimes need to keep track of hnames bound by a particular runtime value or evaluation context, hence we also define sets of hnames H, H_1, H_2 . . ..

Shifting all hnames in a set by a given offset h' is denoted $H \pm h'$. We also define a conditional shift operation $[H \pm h']$ which shifts each hname appearing in the operand to the left of the brackets by h' if this hname is also member of H. This conditional shift can be used on a single hname, a value, or a typing context.

var, *x*, *y*, *d*, *dd*, *un*, *xs*, *ys*, *ex*, *st*, *tree*, *tl*, *tr*, *dtree*, *f*, *dh*, *dt*, *dx*, *dy*, *dxs*, *dys*, *dv*, *dtlr*, *dtl*, *dtr*, *q*				Variable names
$hname$, h, hd	::=			Hole (or destination) name, represented by a natural number
	|	$h+h'$	M	
	|	$h[H \pm h']$	M	Shift by h' if $h \in H$
	|	$\mathbf{max}(H)$	M	Maximum of a set of hole names
$hnames$, H	::=			Set of hole names
	|	$\{h_1, .., h_k\}$		
	|	$H_1 \cup H_2$	M	Union of sets
	|	$H \pm h'$	M	Shift all names from H by h'.
	|	$\mathbf{hnames}(\Gamma)$	M	Hole names bound by the typing context Γ
	|	$\mathbf{hnames}(\mathrm{C})$	M	Hole names bound by the evaluation context C

Fig. 8. Grammar for variable, hole and destination names

5.3 Term and value core syntax

Destination calculus is based on linear simply-typed λ-calculus, with built-in support for sums, pairs, and exponentials. The syntax of terms, which is presented in Figure 9 is quite unusual, as we need to introduce all the tooling required to manipulate destinations, which constitute the primitive way of building a data structures for the user.

term, t, u	::=		Term
	| v		Value
	| x		Variable
	| $\mathrm{t}'\ \mathrm{t}$		Application
	| $\mathrm{t} \fatsemi \mathrm{u}$		Pattern-match on unit
	| $\mathrm{t} \triangleright \mathbf{case}_{\mathrm{m}}\ \{\mathsf{Inl}\ x_1 \mapsto \mathrm{u}_1\ ,\ \mathsf{Inr}\ x_2 \mapsto \mathrm{u}_2\}$		Pattern-match on sum
	| $\mathrm{t} \triangleright \mathbf{case}_{\mathrm{m}}\ (x_1\ ,\ x_2) \mapsto \mathrm{u}$		Pattern-match on product
	| $\mathrm{t} \triangleright \mathbf{case}_{\mathrm{m}}\ \mathsf{E}_{\mathrm{n}}\ x \mapsto \mathrm{u}$		Pattern-match on exponential
	| $\mathrm{t} \triangleright \mathbf{map}\ x \mapsto \mathrm{t}'$		Map over the right side of ampar
	| $\mathbf{to}_{\bowtie}\ \mathrm{u}$		Wrap into a trivial ampar
	| $\mathbf{from}_{\bowtie}\ \mathrm{t}$		Convert ampar to a pair
	| $\mathrm{t} \triangleleft ()$		Fill destination with unit
	| $\mathrm{t} \triangleleft \mathsf{Inl}$		Fill destination with left variant
	| $\mathrm{t} \triangleleft \mathsf{Inr}$		Fill destination with right variant
	| $\mathrm{t} \triangleleft \mathsf{E}_{\mathrm{m}}$		Fill destination with exponential constructor
	| $\mathrm{t} \triangleleft (,)$		Fill destination with product constructor
	| $\mathrm{t} \triangleleft (\lambda x\ {}_{\mathrm{m}}\!\!\mapsto \mathrm{u})$		Fill destination with function
	| $\mathrm{t} \triangleleft\!\bullet\ \mathrm{t}'$		Fill destination with root of other ampar
	| $\mathrm{t}[x \coloneqq \mathrm{v}]$	M	
val, v	::=		Value
	| $\boxed{h}$		Hole
	| $\rightarrow h$		Destination
	| $()$		Unit
	| $^{\mathrm{v}}\lambda x\ {}_{\mathrm{m}}\!\!\mapsto \mathrm{u}$		Function with no free variable
	| $\mathsf{Inl}\ \mathrm{v}$		Left variant for sum
	| $\mathsf{Inr}\ \mathrm{v}$		Right variant for sum
	| $\mathsf{E}_{\mathrm{m}}\ \mathrm{v}$		Exponential
	| $(\mathrm{v}_1\ ,\ \mathrm{v}_2)$		Product
	| $H\langle \mathrm{v}_2\ {}_{\wedge}\mathrm{v}_1 \rangle$		Ampar
	| $\mathrm{v}[H \doteq h']$	M	Shift hole names inside v by h' if they belong to H

Fig. 9. Grammar for terms and values

In fact, the grammatical class of values v, presented as a subset of terms t, could almost be removed completely from the user syntax, and just used as a denotation for runtime data structures. We only need to keep the *ampar* value ${}_{\{h\}}\langle \boxed{h}\ {}_{\wedge} \rightarrow h \rangle$ as part of the user syntax as a way to spawn a fresh memory cell to be later filled using destination-filling primitives (see **alloc** in Section 5.4).

Pattern-matching on every type of structure (except unit) is parametrized by a mode m to which the scrutinee is consumed. The variables which bind the subcomponents of the scrutinee then inherit this mode. In particular, this choice crystalize the equivalence $!_{\omega a}(\mathrm{T}_1 \otimes \mathrm{T}_2) \simeq (!_{\omega a}\mathrm{T}_1) \otimes (!_{\omega a}\mathrm{T}_2)$, which is not part of intuitionistic linear logic, but valid in Linear Haskell[2]. We omit the mode annotation

on **case** statements and lambda abstractions when the mode in question is the multiplicative neutral element 1_ν of the mode semiring.

map is the main primitive to operate on an *ampar*, which represents an incomplete data structure whose building is in progress. **map** binds the right-hand side of the *ampar* — the one containing destinations of that *ampar* — to a variable, allowing those destinations to be operated on by destination-filling primitives. The left-hand side of the *ampar* is inaccessible as it is being mutated behind the scenes by the destination-filling primitives.

$\mathbf{to}_{\ltimes}$ embeds an already completed structure in an *ampar* whose left side is the structure, and right side is unit. We have an operator "fillComp" ($\triangleleft\bullet$) allowing to compose two *ampar*s by writing the root of the second one to a destination of the first one, so by throwing $\mathbf{to}_{\ltimes}$ to the mix, we can compose an *ampar* with a normal (completed) structure (see the sugar operator "fillLeaf" ($\triangleleft$) in Section 5.4).

$\mathbf{from}_{\ltimes}$ is used to convert an *ampar* to a pair, when the right side of the *ampar* is an exponential of the form $\mathsf{E}_{1\omega}\ \mathrm{v}$. Indeed, when the right side has such form, it cannot contains destinations (as destinations always have a finite age), thus it cannot contain holes in its left side either (as holes on the left side are always compensated 1:1 by a destination on the right side). As a result, it is valid to convert an *ampar* to a pair in these circumstances. $\mathbf{from}_{\ltimes}$ is in particular used to extract a structure from its *ampar* building shell when it is complete (see the sugar operator $\mathbf{from'}_{\ltimes}$ in Section 5.4).

The remaining term operators $\triangleleft(), \triangleleft\mathsf{Inl}, \triangleleft\mathsf{Inr}, \triangleleft\,\mathsf{E}_{\mathrm{m}}\,, \triangleleft(,), \triangleleft(\lambda x_{\,\mathrm{m}} \mapsto \mathrm{u})$ are all destination-filling primitives. They write a layer of value/constructor to the hole pointed by the destination operand, and return the potential new destinations that are created in the process (or unit if there is none).

Values. There are two important things to note on the value class.

First, a variable cannot contains any free variable. This will be better visible in the typing rule for function value form Ty-val-Fun, but a value only admits hole and destination type bindings in its typing context, no variable binding. This is quite useful for substitution lemmas, as no undesired capture can happen.

Secondly, values are allowed to have holes inside (represented by $\boxed{h}$, $\boxed{h_1}$, $\boxed{h_2}$…), but a value used as a term isn't allowed to have any free hole (i.e. a hole that is not compensated by an associated destination inside an ampar). This is enforced by the typing context Δ meaning "destination-only" in the rule Ty-term-Val.

5.4 Syntactic sugar for constructors and commonly used operations

As we said in section 5.3, the grammatical class of values is mostly used for runtime only; in particular, data constructors in the value class can only take other values as arguments, not terms (this help us ensure that no free variable can appear in a value). Thus we introduce syntactic for data constructors taking arbitrary terms as parameters (as we often find in functional programming languages) using destination-filling primitives in Figure 10.

$\mathbf{from}_{\ltimes'}$ is a simpler variant of $\mathbf{from}_{\ltimes}$ that allows to extract the right side of an ampar when the right side has been fully consumed. We implement it in terms of $\mathbf{from}_{\ltimes}$ to keep the core calculus tidier (and limit the number of typing rules, evaluation contexts, etc), but it can be implemented much more efficiently in a real-world implementation.

All the desugarings are presented in Figure 11.

6 TYPE SYSTEM

6.1 Types, modes, and typing contexts

The type system of λ_d is highly inspired from Linear Haskell [2]. In particular, it uses the same additive/multiplicative approach on mode as Linear Haskell for linearity enforcement, except here

sterm	::=			Syntactic sugar for terms
	|	**alloc**	M	Evaluate to a fresh new ampar
	|	$t \blacktriangleleft t'$	M	Fill destination with supplied term
	|	$\mathbf{from}'_{\ltimes}\ t$	M	Extract left side of ampar when right side is unit
	|	$\lambda x\ {}_{m}\!\!\mapsto u$	M	Allocate function
	|	$\mathsf{Inl}\ t$	M	Allocate left variant
	|	$\mathsf{Inr}\ t$	M	Allocate right variant
	|	$\mathsf{E}_m\ t$	M	Allocate exponential
	|	(t_1, t_2)	M	Allocate product

Fig. 10. Syntactic sugar forms for terms

$$\mathbf{alloc} \triangleq {}_{\{1\}}\langle \boxed{1}_{\wedge} \to 1 \rangle$$

$$\mathbf{from}'_{\ltimes}\ t \triangleq (\mathbf{from}_{\ltimes}\ (t \triangleright \mathbf{map}\ un \mapsto un\ \fatsemi\ \mathsf{E}_{1\infty}\ ())) \triangleright \mathbf{case}\ (st, ex) \mapsto ex \triangleright \mathbf{case}\ \mathsf{E}_{1\infty}\ un \mapsto un\ \fatsemi\ st$$

$$\lambda x\ {}_{m}\!\!\mapsto u \triangleq \mathbf{from}'_{\ltimes}\ (\ \mathbf{alloc} \triangleright \mathbf{map}\ d \mapsto d \triangleleft (\lambda x\ {}_{m}\!\!\mapsto u)\ )$$

$$(t_1, t_2) \triangleq \mathbf{from}'_{\ltimes}\ (\ \mathbf{alloc} \triangleright \mathbf{map}\ d \mapsto (d \triangleleft (,)) \triangleright \mathbf{case}\ (d_1, d_2) \mapsto d_1 \blacktriangleleft t_1\ \fatsemi\ d_2 \blacktriangleleft t_2\ )$$

$$t \blacktriangleleft t' \triangleq t \triangleleft\!\bullet (\mathbf{to}_{\ltimes}\ t')$$

$$\mathsf{Inl}\ t \triangleq \mathbf{from}'_{\ltimes}\ (\ \mathbf{alloc} \triangleright \mathbf{map}\ d \mapsto d \triangleleft \mathsf{Inl} \blacktriangleleft t\ )$$

$$\mathsf{Inr}\ t \triangleq \mathbf{from}'_{\ltimes}\ (\ \mathbf{alloc} \triangleright \mathbf{map}\ d \mapsto d \triangleleft \mathsf{Inr} \blacktriangleleft t\ )$$

$$\mathsf{E}_m\ t \triangleq \mathbf{from}'_{\ltimes}\ (\ \mathbf{alloc} \triangleright \mathbf{map}\ d \mapsto d \triangleleft \mathsf{E}_m \blacktriangleleft t\ )$$

Fig. 11. Desugaring of syntactic sugar forms for terms

we use a product semiring for mode that keeps track of both age and multiplicity of variables (whereas only multiplicity is tracked in the former).

The grammar of types, modes, and typing context for λ_d is presented in Figure 12. It is based on intuitionistic linear logic, and thus provides multiplicative conjunction $\otimes$ (product type) and additive disjunction (sum type) $\oplus$. It also provides a function arrow ${}_{m}\!\!\to$ and exponential connective $!_m$ that are both parametrized by a mode m. We omit the mode annotation on the function arrow, as well as on the destination type, when the mode in question is the multiplicative neutral element 1_ν of the semiring (in particular, a function arrow without annotation is linear by default). A function arrow with multiplicity 1 is equivalent to the linear arrow $\multimap$ from [**?**].

A mode is either a pair of a multiplicity and an age, or the special symbol ☠ representing an unsatisfiable requirement for a variable in a typing context (e.g. a variable that is used with different types in two branches of a **case** statement on a sum type). This special mode "invalid" ☠ is only used in the formal proof of type safety of λ_d , when summing typing contexts. However, every mode m present in the typing rules of this paper is assumed to be valid (i.e. made of a pair of a multiplicity and an age), and every typing context in a rule premises is assumed to contain only valid modes too.

type, T, U, S	::=		Type
	|	1	Unit
	|	$T_1 \oplus T_2$	Sum
	|	$T_1 \otimes T_2$	Product
	|	$!_m T$	Exponential
	|	$U \ltimes T$	Ampar
	|	$T \multimap_m U$	Function
	|	$\lfloor_m T \rfloor$	Destination
mode, m, n	::=		Mode (Semiring)
	|	pa	Pair of a multiplicity and age
	|	💀	Error case (incompatible types, multiplicities, or ages)
mul, p	::=		Multiplicity (Semiring, first component of mode)
	|	1	Linear use
	|	ω	Non-linear use
age, a	::=		Age (Semiring, second component of mode)
	|	ν	Born now
	|	$\uparrow$	One scope older
	|	∞	Infinitely old / static
ctx, Ω, Γ, Δ, Θ	::=		Typing context
	|	$x :_m T$	Variable typing binding
	|	$\boxed{h} :_n T$	Hole typing binding
	|	$\rightarrow h :_m \lfloor_n T \rfloor$	Destination typing binding
	|	$m \cdot \Omega$	M Multiply the leftmost mode of each binding by m
	|	$\Omega_1 + \Omega_2$	M Sum (incompatible bindings get tagged with 💀)
	|	Ω_1, Ω_2	M Disjoint sum
	|	$\rightarrow^{-1}\Delta$	M Transforms dest bindings into a hole bindings
	|	$\Omega[H \hat{=} h']$	M Shift hole/dest names by h' if they belong to H

Fig. 12. Types, modes, and typing contexts

6.2 Typing of terms and values

Figure 13 presents the typing rules for values and terms.

In every figure,

- Ω denotes an arbitrary typing context, with no particular constraints;
- Γ denotes a typing context made only of holes and destination bindings;
- Θ denotes a typing context made only of destination and variable bindings;
- Δ denotes a typing context made only of destination bindings.

Destinations and holes are two faces of the same coin, as seen in Section 2.1, and must always be in 1:1 correspondance. Thus, the core idea of the type system is to have hole bindings $\boxed{h} :_n T$ (hname in positive polarity) and destination bindings $\rightarrow h :_m \lfloor_n T \rfloor$ (hname in negative polarity), in addition to the variable bindings $x :_m T$ that usually populates typing contexts. Hole bindings and destination bindings of the same hname are meant to compensate each other in the typing context, a bit like how matter (positive polarity, hole) and antimatter (negative polarity, destination)

$\boxed{\Gamma \overset{\nu}{\vdash} v : T}$ *(Typing judgment for values)*

Ty-val-Hole
$$\boxed{h} :_{1\nu} T \overset{\nu}{\vdash} \boxed{h} : T$$

Ty-val-Dest
$$\to h :_{1\nu} \lfloor_n T \rfloor \overset{\nu}{\vdash} \to h : \lfloor_n T \rfloor$$

Ty-val-Unit
$$\overset{\nu}{\vdash} () : 1$$

Ty-val-Fun
$$\frac{\Delta,\ x :_m T \vdash u : U}{\Delta \overset{\nu}{\vdash} \lambda x \,{}_m\!\mapsto u : T \,{}_m\!\to U}$$

Ty-val-Left
$$\frac{\Gamma \overset{\nu}{\vdash} v_1 : T_1}{\Gamma \overset{\nu}{\vdash} \mathsf{Inl}\, v_1 : T_1 \oplus T_2}$$

Ty-val-Right
$$\frac{\Gamma \overset{\nu}{\vdash} v_2 : T_2}{\Gamma \overset{\nu}{\vdash} \mathsf{Inr}\, v_2 : T_1 \oplus T_2}$$

Ty-val-Prod
$$\frac{\Gamma_1 \overset{\nu}{\vdash} v_1 : T_1 \qquad \Gamma_2 \overset{\nu}{\vdash} v_2 : T_2}{\Gamma_1 + \Gamma_2 \overset{\nu}{\vdash} (v_1, v_2) : T_1 \otimes T_2}$$

Ty-val-Exp
$$\frac{\Gamma \overset{\nu}{\vdash} v' : T}{n \cdot \Gamma \overset{\nu}{\vdash} \mathsf{E}_n\, v' : !_n T}$$

Ty-val-Ampar
$$\frac{\begin{array}{c}\mathtt{LinOnly}\ \Delta_3 \\ \mathtt{FinAgeOnly}\ \Delta_3 \\ 1\uparrow \cdot \Delta_1,\ \Delta_3 \overset{\nu}{\vdash} v_1 : T \\ \Delta_2,\ (\to^{-1} \Delta_3) \overset{\nu}{\vdash} v_2 : U\end{array}}{\Delta_1,\ \Delta_2 \overset{\nu}{\vdash} \mathsf{hnames}(\to^{-1}\Delta_3)\langle v_2 \wedge v_1 \rangle : U \ltimes T}$$

$\boxed{\Theta \vdash t : T}$ *(Typing judgment for terms)*

Ty-term-Val
$$\frac{\mathtt{DisposableOnly}\ \Theta \qquad \Delta \overset{\nu}{\vdash} v : T}{\Theta,\ \Delta \vdash v : T}$$

Ty-term-Var
$$\frac{\mathtt{DisposableOnly}\ \Theta \qquad 1\nu <: m}{\Theta,\ x :_m T \vdash x : T}$$

Ty-term-App
$$\frac{\Theta_1 \vdash t : T \qquad \Theta_2 \vdash t' : T \,{}_m\!\to U}{m \cdot \Theta_1 + \Theta_2 \vdash t'\ t : U}$$

Ty-term-PatU
$$\frac{\Theta_1 \vdash t : 1 \qquad \Theta_2 \vdash u : U}{\Theta_1 + \Theta_2 \vdash t \fatsemi u : U}$$

Ty-term-PatS
$$\frac{\begin{array}{c}\Theta_1 \vdash t : T_1 \oplus T_2 \\ \Theta_2,\ x_1 :_m T_1 \vdash u_1 : U \\ \Theta_2,\ x_2 :_m T_2 \vdash u_2 : U\end{array}}{m \cdot \Theta_1 + \Theta_2 \vdash t \rhd \mathbf{case}_m \{\mathsf{Inl}\, x_1 \mapsto u_1,\ \mathsf{Inr}\, x_2 \mapsto u_2\} : U}$$

Ty-term-PatP
$$\frac{\begin{array}{c}\Theta_1 \vdash t : T_1 \otimes T_2 \\ \Theta_2,\ x_1 :_m T_1,\ x_2 :_m T_2 \vdash u : U\end{array}}{m \cdot \Theta_1 + \Theta_2 \vdash t \rhd \mathbf{case}_m\, (x_1, x_2) \mapsto u : U}$$

Ty-term-PatE
$$\frac{\begin{array}{c}\Theta_1 \vdash t : !_n T \\ \Theta_2,\ x :_{m \cdot n} T \vdash u : U\end{array}}{m \cdot \Theta_1 + \Theta_2 \vdash t \rhd \mathbf{case}_m\, \mathsf{E}_n\, x \mapsto u : U}$$

Ty-term-Map
$$\frac{\begin{array}{c}\Theta_1 \vdash t : U \ltimes T \\ 1\uparrow \cdot \Theta_2,\ x :_{1\nu} T \vdash t' : T'\end{array}}{\Theta_1 + \Theta_2 \vdash t \rhd \mathbf{map}\ x \mapsto t' : U \ltimes T'}$$

Ty-term-ToA
$$\frac{\Theta \vdash u : U}{\Theta \vdash \mathbf{to}_{\ltimes}\, u : U \ltimes 1}$$

Ty-term-FromA
$$\frac{\Theta \vdash t : U \ltimes (!_{1\infty} T)}{\Theta \vdash \mathbf{from}_{\ltimes}\, t : U \otimes (!_{1\infty} T)}$$

Ty-term-FillU
$$\frac{\Theta \vdash t : \lfloor_n 1 \rfloor}{\Theta \vdash t \triangleleft () : 1}$$

Ty-term-FillL
$$\frac{\Theta \vdash t : \lfloor_n T_1 \oplus T_2 \rfloor}{\Theta \vdash t \triangleleft \mathsf{Inl} : \lfloor_n T_1 \rfloor}$$

Ty-term-FillR
$$\frac{\Theta \vdash t : \lfloor_n T_1 \oplus T_2 \rfloor}{\Theta \vdash t \triangleleft \mathsf{Inr} : \lfloor_n T_2 \rfloor}$$

Ty-term-FillP
$$\frac{\Theta \vdash t : \lfloor_n T_1 \otimes T_2 \rfloor}{\Theta \vdash t \triangleleft (,) : \lfloor_n T_1 \rfloor \otimes \lfloor_n T_2 \rfloor}$$

Ty-term-FillE
$$\frac{\Theta \vdash t : \lfloor_n !_{n'} T \rfloor}{\Theta \vdash t \triangleleft \mathsf{E}_{n'} : \lfloor_{n' \cdot n} T \rfloor}$$

Ty-term-FillF
$$\frac{\begin{array}{c}\Theta_1 \vdash t : \lfloor_n T \,{}_m\!\to U \rfloor \\ \Theta_2,\ x :_m T \vdash u : U\end{array}}{\Theta_1 + (1\uparrow \cdot n) \cdot \Theta_2 \vdash t \triangleleft (\lambda x \,{}_m\!\mapsto u) : 1}$$

Ty-term-FillComp
$$\frac{\begin{array}{c}\Theta_1 \vdash t : \lfloor_n U \rfloor \\ \Theta_2 \vdash t' : U \ltimes T\end{array}}{\Theta_1 + (1\uparrow \cdot n) \cdot \Theta_2 \vdash t \triangleleft\!\bullet t' : T}$$

Fig. 13. Typing rules for values and terms

annihilate each other. That way, the typing context of a term can stay constant during reduction,

even when destination-filling primitives are evaluated to build up data structures, as those linearly consume a destination and write to a hole which makes it disappear.

However, that annihilation between a destination and a hole having the same hname is only allowed to happen around an ampar, as it is the ampar connective that bind the two polarities of a name together (the names bound are actually stored in a set H on the ampar value ${}_H\langle \mathrm{v}_2 \parallel \mathrm{v}_1 \rangle$). In fact, an ampar can be seen as a sort of lambda-abstraction, whose body (containing holes instead of variables) and sink/input site are split on two sides, and magically interconnected through the ampar connective.

Thus, the sum of typing contexts, used in almost all rules, results in an erroneous context when the input operands contains the same hname but in different polarities: $\{\boxed{h} :_{\mathrm{n}} \mathsf{T}\} + \{\rightarrow h :_{1\nu} \lfloor_{\mathrm{n}} \mathsf{T}\rfloor\} = \{\boxed{h} :_{\omega} 1\}$ (the fact that the result is a hole binding instead of a destination binding is purely arbitrary ; the only important bit here is to have an "invalid" mode on the result binding). An individual context is also not allowed to contain a same hname in two different bindings. Instead, in the only position where the annihilation is allowed to happen, that is to say the Ty-val-Ampar rule, we explicitly identify and remove the parts of input contexts that can interact and annihilate each other, in the form of Δ in the right side context and $\rightarrow^{-1}\Delta$ for the left one.

$\rightarrow^{-1}$ is a point-wise operation on typing bindings of a context where:

$$\begin{cases} \rightarrow^{-1}(\rightarrow h :_{1\nu} \lfloor_{\mathrm{n}} \mathsf{T}\rfloor) &= \boxed{h} :_{\mathrm{n}} \mathsf{T} \\ \rightarrow^{-1}(\mathrm{n} :_{\mathrm{m}} \mathsf{T}) &= \mathrm{n} :_{\omega} 1 \quad \text{otherwise} \end{cases}$$

Only an input context Δ made only of destination bindings, with leftmost mode being 1ν, results in a valid output context, which is then only composed of hole bindings.

It might be time to discuss what modes mean for hole and destination bindings.

A hole binding $\boxed{h} :_{\mathrm{n}} \mathsf{T}$ has only one mode, n, that indicates the mode a value must have to be written to it (that is to say, the mode of bindings that the value depends on to type correctly). To this day, the only way for a value to have a constraining mode is to capture a destination (otherwise the value has mode $\omega\infty$, meaning it can be used in any possible way), as destinations are the only intrinsically linear values in the calculus, but we will see in Section 9 that other forms of intrinsic linearity can be added to the langage for practical reasons. We see the mode of a hole coming into play when a hole is located behind an exponential constructor: we should only write a non-linear value to the hole $\boxed{h}$ in $\mathrm{E}_{\omega\nu}\,\boxed{h}$. In particular, we should not store a destination into this hole, otherwise it could later be extracted and used in a non-linear fashion.

On the other hand, a destination binding $\rightarrow h :_{\mathrm{m}} \lfloor_{\mathrm{n}} \mathsf{T}\rfloor$ has two modes. The left-most one, m, tells how the destination can be used as a passive value, e.g. if the destination get stored in a destination of destination or passed as a function argument. The right-most one, n, corresponds to the mode of its associated hole, and thus how it behaves as an active entity, when filled with a value. We'll see in an instant how a binding can *grow old*, but let's just remember for now that a destination can only be filled with a value when $\mathbf{age}(\mathrm{m}) = \nu$. As soon at it gets older (only the mode m is affected, the mode n of its associated hole never changes), it behaves like a passive normal value and can only be stored, passed around, or returned.

In practice, destinations can never have left-most multiplicity ω or age ∞. They can only be linear and have a finite age. Indeed, only Ty-val-Ampar (or more precisely, Ty-ectxs-OpenAmpar-Foc) adds new destination bindings to a typing context, and those initially have mode 1ν, and can only grow old later by increments of $\uparrow$ (but can never reach age ∞). We have formally proved that this property holds during execution. So any typing context of the form $\rightarrow h :_{\omega\mathrm{a}} \lfloor_{\mathrm{n}} \mathsf{T}\rfloor$, Ω or $\rightarrow h :_{\mathrm{p}\infty} \lfloor_{\mathrm{n}} \mathsf{T}\rfloor$, Ω is never satisfiable.

Value typing $\vdash^{v}$. Values type in a typing context Γ made only of hole and destinations bindings. The absence of free variables in values make it easier to prove substitution properties (in particular, we never have to go under an ampar to perform a substitution).

Rules Ty-val-Hole and Ty-val-Dest indicates that a hole or destination have left-most mode 1_ν when used alone.

revisit this if we allow weakening for dests

6.3 Derived typing rules for syntactic sugar forms

$\boxed{\Theta \vdash^{s} t : T}$ *(Derived typing judgment for syntactic sugar forms)*

Ty-sterm-Alloc
$$\frac{\texttt{DisposableOnly}\ \Theta}{\Theta \vdash^{s} \mathbf{alloc} : T \ltimes \lfloor T \rfloor}$$

Ty-sterm-FromA'
$$\frac{\Theta \vdash t : T \ltimes 1}{\Theta \vdash^{s} \mathbf{from}'_{\ltimes}\, t : T}$$

Ty-sterm-FillLeaf
$$\frac{\Theta_1 \vdash t : \lfloor {}_n T \rfloor \qquad \Theta_2 \vdash t' : T}{\Theta_1 + (1\uparrow\cdot n)\cdot\Theta_2 \vdash^{s} t \blacktriangleleft t' : 1}$$

Ty-sterm-Fun
$$\frac{\Theta_2,\ x :_m T \vdash u : U}{\Theta_2 \vdash^{s} \lambda x_{\,m}\!\mapsto u : T \,{}_m\!\!\rightarrow U}$$

Ty-sterm-Left
$$\frac{\Theta_2 \vdash t : T_1}{\Theta_2 \vdash^{s} \mathsf{Inl}\, t : T_1 \oplus T_2}$$

Ty-sterm-Right
$$\frac{\Theta_2 \vdash t : T_2}{\Theta_2 \vdash^{s} \mathsf{Inr}\, t : T_1 \oplus T_2}$$

Ty-sterm-Exp
$$\frac{\Theta_2 \vdash t : T}{m\cdot\Theta_2 \vdash^{s} \mathsf{E}_m\ t : {}!_m T}$$

Ty-sterm-Prod
$$\frac{\Theta_{21} \vdash t_1 : T_1 \qquad \Theta_{22} \vdash t_2 : T_2}{\Theta_{21} + \Theta_{22} \vdash^{s} (t_1, t_2) : T_1 \otimes T_2}$$

Fig. 14. Derived typing rules for syntactic sugar forms

7 EVALUATION CONTEXTS AND SEMANTICS

7.1 Evaluation contexts forms

7.2 Typing of evaluation contexts and commands

7.3 Small-step semantics

8 PROOF OF TYPE SAFETY USING COQ PROOF ASSISTANT

- Not particularly elegant. Max number of goals observed 232 (solved by a single call to the `congruence` tactic). When you have a computer, brute force is a viable strategy. (in particular, no semiring formalisation, it was quicker to do directly)
- Rules generated by ott, same as in the article (up to some notational difference). Contexts are not generated purely by syntax, and are interpreted in a semantic domain (finite functions).
- Reasoning on closed terms avoids almost all complications on binder manipulation. Makes proofs tractable.
- Finite functions: making a custom library was less headache than using existing libraries (including `MMap`). Existing libraries don't provide some of the tools that we needed, but the most important factor ended up being the need for a modicum of dependency between key and value. There wasn't really that out there. Backed by actual functions for simplicity; cost: equality is complicated.
- Most of the proofs done by author with very little prior experience to Coq.
- Did proofs in Coq because context manipulations are tricky.
- Context sum made total by adding an extra invalid *mode* (rather than an extra context). It seems to be much simpler this way.
- It might be a good idea to provide statistics on the number of lemmas and size of Coq codebase.

ectx, c	::=		Evaluation context component
	|	$t' \ \square$	
	|	$\square \ v$	
	|	$\square \fatsemi u$	
	|	$\square \triangleright \mathbf{case}_m \ \{\mathsf{Inl}\ x_1 \mapsto u_1 \,,\ \mathsf{Inr}\ x_2 \mapsto u_2\}$	
	|	$\square \triangleright \mathbf{case}_m \ (x_1 \,, x_2) \mapsto u$	
	|	$\square \triangleright \mathbf{case}_m \ \mathsf{E}_n \ x \mapsto u$	
	|	$\square \triangleright \mathbf{map}\ x \mapsto t'$	
	|	$\mathbf{to}_{\ltimes} \ \square$	
	|	$\mathbf{from}_{\ltimes} \ \square$	
	|	$\square \triangleleft ()$	
	|	$\square \triangleleft \mathsf{Inl}$	
	|	$\square \triangleleft \mathsf{Inr}$	
	|	$\square \triangleleft \mathsf{E}_m$	
	|	$\square \triangleleft (,)$	
	|	$\square \triangleleft (\lambda x \ {}_m\!\mapsto u)$	
	|	$\square \triangleleft\!\bullet \ t'$	
	|	$v \triangleleft\!\bullet \ \square$	
	|	$\overset{\mathrm{op}}{H}\langle v_2 \ {}_\wedge\!\square \rangle$	Open ampar, binding hole names in the next compor

ectxs, C	::=		Evaluation context stack
	|	$\square$	Represent the empty stack / "identity" evaluation context
	|	$C \circ c$	Push c on top of C
	|	$C[h :=_H v]$ M	Fill h in C with value v (that may contain holes)

Fig. 15. Grammar for evaluation contexts

- (possibly) renaming as permutation, inspired by nominal sets, make more lemmas don't require a condition (but some lemmas that wouldn't in a straight renaming do in exchange).
- (possibly) methodology: assume a lot of lemmas, prove main theorem, prove assumptions, some wrong, fix. A number of wrong lemma initially assumed, but replacing them by correct variant was always easy to fix in proofs.
- Axioms that we use and why (in particular setoid equality not very natural with ott-generated typing rules).
- Talk about the use and benefits of Copilot.

9 IMPLEMENTATION OF DESTINATION CALCULUS USING IN-PLACE MEMORY MUTATIONS

What needs to be changed (e.g. linear alloc)

10 RELATED WORK

11 CONCLUSION AND FUTURE WORK

$\boxed{\Delta \dashv C : T \rightarrowtail U_0}$ *(Typing judgment for evaluation contexts)*

Ty-ectxs-Id
$$\frac{}{\dashv \Box : U_0 \rightarrowtail U_0}$$

Ty-ectxs-App-Foc1
$$\frac{m{\cdot}\Delta_1,\ \Delta_2 \dashv C : U \rightarrowtail U_0 \qquad \Delta_2 \vdash t' : T \multimap_m U}{\Delta_1 \dashv C \circ (t'\ \Box) : T \rightarrowtail U_0}$$

Ty-ectxs-App-Foc2
$$\frac{m{\cdot}\Delta_1,\ \Delta_2 \dashv C : U \rightarrowtail U_0 \qquad \Delta_1 \vdash v : T}{\Delta_2 \dashv C \circ (\Box\ v) : (T \multimap_m U) \rightarrowtail U_0}$$

Ty-ectxs-PatU-Foc
$$\frac{\Delta_1,\ \Delta_2 \dashv C : U \rightarrowtail U_0 \qquad \Delta_2 \vdash u : U}{\Delta_1 \dashv C \circ (\Box \fatsemi u) : 1 \rightarrowtail U_0}$$

Ty-ectxs-PatS-Foc
$$\frac{m{\cdot}\Delta_1,\ \Delta_2 \dashv C : U \rightarrowtail U_0 \qquad \Delta_2,\ x_1 :_m T_1 \vdash u_1 : U \qquad \Delta_2,\ x_2 :_m T_2 \vdash u_2 : U}{\Delta_1 \dashv C \circ (\Box \triangleright \mathbf{case}_m\ \{\mathsf{Inl}\ x_1 \mapsto u_1\ ,\ \mathsf{Inr}\ x_2 \mapsto u_2\}) : (T_1 \oplus T_2) \rightarrowtail U_0}$$

Ty-ectxs-PatP-Foc
$$\frac{m{\cdot}\Delta_1,\ \Delta_2 \dashv C : U \rightarrowtail U_0 \qquad \Delta_2,\ x_1 :_m T_1,\ x_2 :_m T_2 \vdash u : U}{\Delta_1 \dashv C \circ (\Box \triangleright \mathbf{case}_m\ (x_1\ , x_2) \mapsto u) : (T_1 \otimes T_2) \rightarrowtail U_0}$$

Ty-ectxs-PatE-Foc
$$\frac{m{\cdot}\Delta_1,\ \Delta_2 \dashv C : U \rightarrowtail U_0 \qquad \Delta_2,\ x :_{m{\cdot}m'} T \vdash u : U}{\Delta_1 \dashv C \circ (\Box \triangleright \mathbf{case}_m\ \mathsf{E}_{m'}\ x \mapsto u) : !_{m'} T \rightarrowtail U_0}$$

Ty-ectxs-Map-Foc
$$\frac{\Delta_1,\ \Delta_2 \dashv C : U \ltimes T' \rightarrowtail U_0 \qquad 1\!\uparrow{\cdot}\Delta_2,\ x :_{1_\mathcal{V}} T \vdash t' : T'}{\Delta_1 \dashv C \circ (\Box \triangleright \mathbf{map}\ x \mapsto t') : (U \ltimes T) \rightarrowtail U_0}$$

Ty-ectxs-ToA-Foc
$$\frac{\Delta \dashv C : (U \ltimes 1) \rightarrowtail U_0}{\Delta \dashv C \circ (\mathbf{to}_\ltimes\ \Box) : U \rightarrowtail U_0}$$

Ty-ectxs-FromA-Foc
$$\frac{\Delta \dashv C : (U \otimes (!_{1_\infty} T)) \rightarrowtail U_0}{\Delta \dashv C \circ (\mathbf{from}_\ltimes\ \Box) : (U \ltimes (!_{1_\infty} T)) \rightarrowtail U_0}$$

Ty-ectxs-FillU-Foc
$$\frac{\Delta \dashv C : 1 \rightarrowtail U_0}{\Delta \dashv C \circ (\Box \triangleleft ()) : \lfloor_n 1 \rfloor \rightarrowtail U_0}$$

Ty-ectxs-FillL-Foc
$$\frac{\Delta \dashv C : \lfloor_n T_1 \rfloor \rightarrowtail U_0}{\Delta \dashv C \circ (\Box \triangleleft \mathsf{Inl}) : \lfloor_n T_1 \oplus T_2 \rfloor \rightarrowtail U_0}$$

Ty-ectxs-FillR-Foc
$$\frac{\Delta \dashv C : \lfloor_n T_2 \rfloor \rightarrowtail U_0}{\Delta \dashv C \circ (\Box \triangleleft \mathsf{Inr}) : \lfloor_n T_1 \oplus T_2 \rfloor \rightarrowtail U_0}$$

Ty-ectxs-FillP-Foc
$$\frac{\Delta \dashv C : (\lfloor_n T_1 \rfloor \otimes \lfloor_n T_2 \rfloor) \rightarrowtail U_0}{\Delta \dashv C \circ (\Box \triangleleft (,)) : \lfloor_n T_1 \otimes T_2 \rfloor \rightarrowtail U_0}$$

Ty-ectxs-FillE-Foc
$$\frac{\Delta \dashv C : \lfloor_{m{\cdot}n} T \rfloor \rightarrowtail U_0}{\Delta \dashv C \circ (\Box \triangleleft \mathsf{E}_m) : \lfloor_n !_m T \rfloor \rightarrowtail U_0}$$

Ty-ectxs-FillF-Foc
$$\frac{\Delta_1,\ (1\!\uparrow{\cdot}n){\cdot}\Delta_2 \dashv C : 1 \rightarrowtail U_0 \qquad \Delta_2,\ x :_m T \vdash u : U}{\Delta_1 \dashv C \circ (\Box \triangleleft (\lambda x_m \mapsto u)) : \lfloor_n T \multimap_m U \rfloor \rightarrowtail U_0}$$

Ty-ectxs-FillComp-Foc1
$$\frac{\Delta_1,\ (1\!\uparrow{\cdot}n){\cdot}\Delta_2 \dashv C : T \rightarrowtail U_0 \qquad \Delta_2 \vdash t' : U \ltimes T}{\Delta_1 \dashv C \circ (\Box \triangleleft\bullet t') : \lfloor_n U \rfloor \rightarrowtail U_0}$$

Ty-ectxs-FillComp-Foc2
$$\frac{\Delta_1,\ (1\!\uparrow{\cdot}n){\cdot}\Delta_2 \dashv C : T \rightarrowtail U_0 \qquad \Delta_1 \vdash v : \lfloor_n U \rfloor}{\Delta_2 \dashv C \circ (v \triangleleft\bullet \Box) : U \ltimes T \rightarrowtail U_0}$$

Ty-ectxs-OpenAmpar-Foc
$$\frac{\begin{array}{c}\mathbf{hnames}(C)\ \#\#\ \mathbf{hnames}(\rightarrow^{-1}\Delta_3) \\ \mathtt{LinOnly}\ \Delta_3 \\ \mathtt{FinAgeOnly}\ \Delta_3 \\ \Delta_1,\ \Delta_2 \dashv C : (U \ltimes T') \rightarrowtail U_0 \qquad \Delta_2,\ \rightarrow^{-1}\Delta_3\ \mathbin{\blacktriangledown}\vdash v_2 : U\end{array}}{1\!\uparrow{\cdot}\Delta_1,\ \Delta_3 \dashv C \circ ({}^{\mathrm{op}}_{\mathbf{hnames}(\rightarrow^{-1}\Delta_3)}\langle v_2 \wedge \Box\rangle) : T' \rightarrowtail U_0}$$

$\boxed{\vdash C[t] : T}$ *(Typing judgment for commands)*

Ty-cmd
$$\frac{\Delta \dashv C : T \rightarrowtail U_0 \qquad \Delta \vdash t : T}{\vdash C[t] : U_0}$$

Fig. 16. Typing rules for evaluation contexts and commands

$\boxed{C[t] \longrightarrow C'[t']}$ *(Small-step evaluation of commands)*

Sem-App-Foc1
$$\frac{\texttt{NotVal}\ t}{C[t'\ t] \longrightarrow (C \circ (t'\ \square))[t]}$$

Sem-App-Unfoc1
$$\frac{}{(C \circ (t'\ \square))[v] \longrightarrow C[t'\ v]}$$

Sem-App-Foc2
$$\frac{\texttt{NotVal}\ t'}{C[t'\ v] \longrightarrow (C \circ (\square\ v))[t']}$$

Sem-App-Unfoc2
$$\frac{}{(C \circ (\square\ v))[v'] \longrightarrow C[v'\ v]}$$

Sem-App-Red
$$\frac{}{C[(\lambda x_{\,m} \mapsto u)\ v] \longrightarrow C[u[x := v]]}$$

Sem-PatU-Foc
$$\frac{\texttt{NotVal}\ t}{C[t \fatsemi u] \longrightarrow (C \circ (\square \fatsemi u))[t]}$$

Sem-PatU-Unfoc
$$\frac{}{(C \circ (\square \fatsemi u))[v] \longrightarrow C[v \fatsemi u]}$$

Sem-PatU-Red
$$\frac{}{C[() \fatsemi u] \longrightarrow C[u]}$$

Sem-PatS-Foc
$$\frac{\texttt{NotVal}\ t}{C[t \triangleright \mathbf{case}_m\ \{\mathsf{Inl}\ x_1 \mapsto u_1,\ \mathsf{Inr}\ x_2 \mapsto u_2\}] \longrightarrow (C \circ (\square \triangleright \mathbf{case}_m\ \{\mathsf{Inl}\ x_1 \mapsto u_1,\ \mathsf{Inr}\ x_2 \mapsto u_2\}))[t]}$$

Sem-PatS-Unfoc
$$\frac{}{(C \circ (\square \triangleright \mathbf{case}_m\ \{\mathsf{Inl}\ x_1 \mapsto u_1,\ \mathsf{Inr}\ x_2 \mapsto u_2\}))[v] \longrightarrow C[v \triangleright \mathbf{case}_m\ \{\mathsf{Inl}\ x_1 \mapsto u_1,\ \mathsf{Inr}\ x_2 \mapsto u_2\}]}$$

Sem-PatL-Red
$$\frac{}{C[(\mathsf{Inl}\ v_1) \triangleright \mathbf{case}_m\ \{\mathsf{Inl}\ x_1 \mapsto u_1,\ \mathsf{Inr}\ x_2 \mapsto u_2\}] \longrightarrow C[u_1[x_1 := v_1]]}$$

Sem-PatR-Red
$$\frac{}{C[(\mathsf{Inr}\ v_2) \triangleright \mathbf{case}_m\ \{\mathsf{Inl}\ x_1 \mapsto u_1,\ \mathsf{Inr}\ x_2 \mapsto u_2\}] \longrightarrow C[u_2[x_2 := v_2]]}$$

Sem-PatP-Foc
$$\frac{\texttt{NotVal}\ t}{C[t \triangleright \mathbf{case}_m\ (x_1, x_2) \mapsto u] \longrightarrow (C \circ (\square \triangleright \mathbf{case}_m\ (x_1, x_2) \mapsto u))[t]}$$

Sem-PatP-Unfoc
$$\frac{}{(C \circ (\square \triangleright \mathbf{case}_m\ (x_1, x_2) \mapsto u))[v] \longrightarrow C[v \triangleright \mathbf{case}_m\ (x_1, x_2) \mapsto u]}$$

Sem-PatP-Red
$$\frac{}{C[(v_1, v_2) \triangleright \mathbf{case}_m\ (x_1, x_2) \mapsto u] \longrightarrow C[u[x_1 := v_1][x_2 := v_2]]}$$

Sem-PatE-Foc
$$\frac{\texttt{NotVal}\ t}{C[t \triangleright \mathbf{case}_m\ \mathsf{E}_n\ x \mapsto u] \longrightarrow (C \circ (\square \triangleright \mathbf{case}_m\ \mathsf{E}_n\ x \mapsto u))[t]}$$

Sem-PatE-Unfoc
$$\frac{}{(C \circ (\square \triangleright \mathbf{case}_m\ \mathsf{E}_n\ x \mapsto u))[v] \longrightarrow C[v \triangleright \mathbf{case}_m\ \mathsf{E}_n\ x \mapsto u]}$$

Sem-PatE-Red
$$\frac{}{C[\mathsf{E}_n\ v' \triangleright \mathbf{case}_m\ \mathsf{E}_n\ x \mapsto u] \longrightarrow C[u[x := v']]}$$

Sem-Map-Foc
$$\frac{\texttt{NotVal}\ t}{C[t \triangleright \mathbf{map}\ x \mapsto t'] \longrightarrow (C \circ (\square \triangleright \mathbf{map}\ x \mapsto t'))[t]}$$

Sem-Map-Unfoc
$$\frac{}{(C \circ (\square \triangleright \mathbf{map}\ x \mapsto t'))[v] \longrightarrow C[v \triangleright \mathbf{map}\ x \mapsto t']}$$

Sem-Map-Red-OpenAmpar-Foc
$$\frac{h = \mathbf{max}(\mathbf{fnames}(C)) + 1}{C[_H\langle v_2 \wedge v_1\rangle \triangleright \mathbf{map}\ x \mapsto t'] \longrightarrow (C \circ ({}^{\mathrm{op}}_{H \doteq h'}\langle v_2[H \doteq h'] \wedge \square\rangle))[t'[x := v_1[H \doteq h']]]}$$

Sem-ToA-Foc

REFERENCES

[1] Thomas Bagrel. 2024. Destination-passing style programming: a Haskell implementation. https://inria.hal.science/hal-04406360

[2] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2018. Linear Haskell: practical linearity in a higher-order polymorphic language. *Proceedings of the ACM on Programming Languages* 2, POPL (Jan. 2018), 1–29. https://doi.org/10.1145/3158093 arXiv:1710.09756 [cs].

[3] Frédéric Bour, Basile Clément, and Gabriel Scherer. 2021. Tail Modulo Cons. *arXiv:2102.09823 [cs]* (Feb. 2021). http://arxiv.org/abs/2102.09823 arXiv: 2102.09823.

[4] Daan Leijen and Anton Lorenzen. 2023. Tail Recursion Modulo Context: An Equational Approach. *Proceedings of the ACM on Programming Languages* 7, POPL (Jan. 2023), 1152–1181. https://doi.org/10.1145/3571233

[5] Yasuhiko Minamide. 1998. A functional representation of data structures with a hole. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '98)*. Association for Computing Machinery, New York, NY, USA, 75–84. https://doi.org/10.1145/268946.268953