

# Destination calculus

A linear  $\lambda$ -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  $\lambda$ -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 [Bagrel 2024]. We illustrate this with a range of pseudo-Haskell examples.

## ACM Reference Format:

Arnaud Spiwack and Thomas Bagrel. 2024. Destination calculus: A linear  $\lambda$ -calculus for pure, functional memory updates. In . ACM, New York, NY, USA, 31 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

## 1 INTRODUCTION

**TODO: Redire plein de fois: On introduit de la mutation controlee dans les FP languages sans endommager la purete (comme la laziness peut être vu aussi) + ordre de building flexible + des systemes se sont deja interesses a ca, mais nous on subsume tout ca**

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 [Bernardy et al. 2018], 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  $\lambda$ -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

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

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

© 2024 Association for Computing Machinery.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...\$15.00

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

the structure in its canonical order (e.g. from the leaves up to the root of the structure when using data constructors).

## 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  $T \rightarrow U$ , in  $\lambda_d$  you would have  $T \rightarrow [U] \rightarrow 1$ , where  $[U]$  is read “destination of type  $U$ ”. For instance, here is a destination-passing version of the identity function:

$$\begin{aligned} \text{dId} &: T \rightarrow [T] \rightarrow 1 \\ \text{dId } x \ d &\triangleq d \triangleleft x \end{aligned}$$

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

The form  $d \triangleleft x$  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.

$$\begin{aligned} \text{fillWithInl} &: [T \oplus U] \rightarrow [T] \\ \text{fillWithInl } d &\triangleq d \triangleleft \text{Inl} \end{aligned}$$

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

Notice that we are constructing the term from the outermost constructor inward: we’ve built a value of the form  $\text{Inl } \square$  but we have yet to describe the constructor’s payload. We call such incomplete values “hollow constructors”, where  $\square$  represent the hole(s) they contain. 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  $\text{Inl}$  to make an  $\text{Inl } v$ . 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:

$$\begin{aligned} \text{fillWithAPair} &: [T \otimes U] \rightarrow [T] \otimes [U] \\ \text{fillWithAPair } d &\triangleq d \triangleleft (,) \end{aligned}$$

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.

*Structures 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  $[T]$  while it is used to build a structure of type  $T \oplus U$ . To represents this,  $\lambda_d$  uses a type  $S \ltimes [T]$  for a structure of type  $S$  missing a value of type  $T$  to be complete; we

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.  $\rightarrow$  Thomas : that’s the case for non-terminals in the grammar

Make sure that we clarify early on that when we say “structure” we mean “linked data structure”

then 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:  $S \ltimes ([T] \otimes [U])$ .

The form  $S \ltimes [T]$  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  $\lambda_d$  in Section 5.1. For now, it’s sufficient to observe that  $S \ltimes [T]$  is akin to a  $\text{S}\&\text{T}^\perp$  in linear logic, indeed you can think of  $S \ltimes [T]$  as a (linear) function from  $T$  to  $S$ . That structures with holes could be seen as linear functions was first observed in [Minamide 1998], 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 [Bagrel 2024].

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  $[T]$ , 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,  $\lambda_d$  provides a **map** construction, which lets us apply a function to the right-hand side of an ampar:

$$\begin{array}{ll} \text{fillWithInl}' : S \ltimes [T \oplus U] \rightarrow S \ltimes [T] & \text{fillWithAPair}' : S \ltimes [T \otimes U] \rightarrow S \ltimes ([T] \otimes [U]) \\ \text{fillWithInl}' x \triangleq \text{map } x \text{ with } d \mapsto d \triangleleft \text{Inl} & \text{fillWithAPair}' x \triangleq \text{map } x \text{ with } d \mapsto d \triangleleft (,) \end{array}$$

To tie this up, we need a way to introduce and to eliminate structures with holes. Structures with holes are introduced with **alloc** which creates a value of type  $T \ltimes [T]$ . **alloc** is a bit like the identity function: it is a hole of type  $T$  that needs a structure of type  $T$  to be a complete structure of type  $T$ . Structures with holes are eliminated with<sup>1</sup> **from<sub>x</sub>** :  $S \ltimes 1 \rightarrow S$ : if all the destinations have been consumed and only unit remains on the right side, then a structure with holes is really just a normal, complete structure.

Equipped with these, we can, for instance, derive traditional constructors from piecemeal filling. In fact,  $\lambda_d$  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.

$$\begin{array}{ll} \text{Inl} : T \rightarrow T \oplus U & \\ \text{Inl } x \triangleq \text{from}'_x (\text{map alloc with } d \mapsto d \triangleleft \text{Inl} \triangleleft x) & \\ (,) : T \rightarrow U \rightarrow T \otimes U & \\ (x, y) \triangleq \text{from}'_x (\text{map alloc with } d \mapsto \text{case } (d \triangleleft (,)) \text{ of } (d_1, d_2) \mapsto d_1 \triangleleft x \ ; \ d_2 \triangleleft y) & \end{array}$$

*Memory safety and purity.* At this point, the reader may be forgiven for feeling distressed at all the talk of using memory mutations as a semantic for destination-filling primitives. How is it consistent with our claim to be building a pure and memory-safe language?

In fact, we build our system on three core principles:

- (1) destinations are linear resources: they have to be used exactly once;
- (2) a structure that still have holes somewhere in its structure cannot be read;
- (3) destinations cannot escape to an older scope than the one they originate from (how we ensure this will be detailed in Section 6).

The first principle ensures that such programs are ruled out:

$$\begin{array}{ll} \text{forget} : T & \\ \text{forget} \triangleq \text{from}'_x (\text{map alloc with } d \mapsto ()) & \\ \text{forget}' : (T_1 \otimes T_2) \ltimes [T_2] & \\ \text{forget}' \triangleq \text{map alloc with } d \mapsto \text{case } (d \triangleleft (,)) \text{ of } (d_1, d_2) \mapsto \text{from}'_x (\text{map alloc with } dd \mapsto dd \triangleleft d_1 \ ; \ dd \triangleleft d_2) & \end{array}$$

<sup>1</sup>As the name suggest, there is a more general elimination **from<sub>x</sub>**. It will be discussed in Section 6.

Somewhere around this point it would be good to explain the difference between holes and destinations. → Thomas: see DIFF-HOLE-DEST

DIFF-HOLE-DEST

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

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.

Ensure that we do claim that

$$\begin{array}{l|l}
\text{List } T \stackrel{\text{rec}}{\triangleq} 1 \oplus (T \otimes (\text{List } T)) & \\
\hline
\triangleleft [] : [\text{List } T] \rightarrow 1 & \triangleleft (::) : [\text{List } T] \rightarrow [T] \otimes [\text{List } T] \\
d \triangleleft [] \triangleq d \triangleleft \text{Inl } \triangleleft () & d \triangleleft (::) \triangleq d \triangleleft \text{Inr } \triangleleft (,)
\end{array}$$

Fig. 1. List implementation in equirecursive destination calculus

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

In the more complex **forget'**, we fill  $dd$  twice to drop the first value that it receives into oblivion, even though that value is marked as being consumed. So the hole associated to  $d_1$  can't be filled ever, and that will eventually lead to reading uninitialized memory.

We will see in section 3 why scope escape of destination is also a concern for memory safety.

Purity of the system is ensured by the combination of these principles. The main way to break purity would be to reuse a destination to mutate a structure after that structure has already been read once. But a structure cannot be read while it still has holes, and once it has been completed, the destinations that were used to fill it are no longer accessible nor usable (because of the first and third principles). As a result, in that controlled setting, we can think of destinations as write-once memory locations, and the mutations behind the hood are totally invisible as no read can happen before the (only one allowed) writing is made.

## 2.2 Functional queues, with destinations

Now that we have an intuition of how destinations work, let's see how they can be used to build usual data structures. For that section, we suppose that  $\lambda_d$  is equipped with equirecursive types and a fixed-point operator, that isn't part of our formally proven fragment.

*Linked lists.* For starters, we can define lists as the fixpoint of the functor  $X \mapsto 1 \oplus (T \otimes X)$  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”  $\triangleleft []$  and “fillCons”  $\triangleleft (::)$  operators, as presented in Figure 1.

Just like we did in Section 2.1 for the primitive constructors, we can recover the “cons” constructor:

$$\begin{array}{l}
(::) : T \otimes (\text{List } T) \rightarrow \text{List } T \\
(::) x xs \triangleq \text{from}'_{\times} (\text{map alloc with } d \mapsto \text{case } (d \triangleleft (::)) \text{ of } (dx, dxs) \mapsto dx \triangleleft x \ ; \ dxs \triangleleft xs)
\end{array}$$

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  $x_1 : x_2 : \dots : x_k : \square$  is represented by the linear function  $\lambda xs \mapsto x_1 : x_2 : \dots : x_k : xs$ . 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 than a naive nested list concatenation.

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. The

<pre> 197  DList T <math>\triangleq</math> (List T) <math>\ltimes</math> [List T] 198  append : DList T <math>\rightarrow</math> T <math>\rightarrow</math> DList T 199  ys append y <math>\triangleq</math> 200    map ys with dys <math>\mapsto</math> case (dys <math>\triangleleft</math> (::)) of 201      (dy, dys') <math>\mapsto</math> dy <math>\triangleleft</math> y <math>\circ</math> dys' 202  concat : DList T <math>\rightarrow</math> DList T <math>\rightarrow</math> DList T 203  ys concat ys' <math>\triangleq</math> map ys with d <math>\mapsto</math> d <math>\triangleleft</math> • ys' 204  toList : DList T <math>\rightarrow</math> List T 205  toList ys <math>\triangleq</math> from' <math>\ltimes</math> (map ys with d <math>\mapsto</math> d <math>\triangleleft</math> []) </pre>	<pre> Queue T <math>\triangleq</math> (List T) <math>\otimes</math> (DList T) singleton : T <math>\rightarrow</math> Queue T singleton x <math>\triangleq</math> (Inr (x, Inl ()), alloc) enqueue : Queue T <math>\rightarrow</math> T <math>\rightarrow</math> Queue T q enqueue y <math>\triangleq</math>   case q of (xs, ys) <math>\mapsto</math> (xs, ys append y) dequeue : Queue T <math>\rightarrow</math> 1 <math>\oplus</math> (T <math>\otimes</math> (Queue T)) dequeue q <math>\triangleq</math>   case q of {     (Inr (x, xs), ys) <math>\mapsto</math> Inr (x, (xs, ys)),     (Inl (), ys) <math>\mapsto</math> case (toList ys) of {       Inl () <math>\mapsto</math> Inl (),       Inr (x, xs) <math>\mapsto</math> Inr (x, (xs, alloc))     }   } </pre>
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Fig. 2. Difference list and queue implementation in equirecursive destination calculus

incomplete list is represented by the left side of the ampar, and the destination is represented by its right side. No read on the unfinished list can happen until the destination on the right has been consumed.

Type and operators definitions for difference lists in  $\lambda_d$  are presented on the left of Figure 2.

An empty difference list can be created without the need for a new operator as it is exactly what the alloc primitive already does when specialized to type `List T`.

The **append** operator uses “fillCons” to link a new hollow “cons” cell at the end of the list (represented by *dys*), and then handles the two associated destinations *dy* and *dys'*. 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 at the end of the first one. This is done using the “fillComp” ( $\triangleleft$ ) primitive, that here has type `[List T]  $\rightarrow$  DList T  $\mapsto$  [List T]`. It takes a destination as a first argument and an ampar as its second argument. The left side of the ampar is fed to the destination to be merged with another incomplete structure (the first difference list), while right side of the ampar (the destination for the end of the second difference list) is returned.

Finally, the **toList** 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 example 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 [Bour et al. 2021] and [Leijen and Lorenzen 2023]. 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 to 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 reverse order (e.g the latest inserted element will be at the front of the second list).

With such a queue implementation, dequeuing the front element is efficient until the first list is depleted. In that case, one now has to transfer elements from the second list to the first one, and so has to reverse the second list beforehand, which is a  $O(n)$  operation (although it is amortized).

With access to efficient difference lists, as shown in the previous paragraph, we can replace the second list by a difference list, to remove the need for a reverse operation when the first list get depleted. The corresponding implementation is presented on the left of Figure 2.

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 first checks if there is one element available in the front list. If there is, it extracts the element  $x$  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 remaining elements  $xs$  paired with a fresh difference list).

### 3 SCOPE ESCAPE OF DESTINATIONS

Everything described in Section 2 is in fact already possible in destination-passing style for Haskell as presented in [Bagrel 2024]. However, in the aforementioned paper, destinations cannot be stored in destination-based data structures. This restriction is a rather blunt approach to prevent scope escape of destinations that itself pose a great threat to the memory safety of the system. Let's see why.

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  $[T]$  can be of arbitrary inner type  $T$ , they can in particular be used to store a destination itself when  $T = [T']!$

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 used in the current context. But that means we have to mark the destination  $d : [T']$  as linearly consumed too when it is fed to  $dd : [[T']]$  in  $dd \triangleleft d$ .

As a result, there are in fact two ways to consume a destination: fill it now with a value, or store it away and fill 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 initial assumption above doesn't hold in general case.

The issue is particularly visible when trying to give semantics to the **alloc'** operator:

$\text{alloc}' : ([T] \rightarrow 1) \rightarrow T$   
 $\text{alloc}' f \triangleq \text{from}'_{\times} (\text{map } \text{alloc } \text{with } d \mapsto f d)$

With linear store semantics, this is how **alloc'** would behave:

$S \mid \text{alloc}' (\lambda d \mapsto t) \rightarrow S \sqcup \{h := \square\} \mid (t[d := \rightarrow h] \text{ ; } \text{deref} \rightarrow h)$

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

$\begin{array}{ll} \{\} & \mid \text{alloc}' (\lambda d \mapsto d \triangleleft \text{Inl } \triangleleft ()) \\ \rightarrow & \{h := \square\} \mid \rightarrow h \triangleleft \text{Inl } \triangleleft () \text{ ; } \text{deref} \rightarrow h \\ \rightarrow & \{h := \text{Inl } ()\} \mid \text{deref} \rightarrow h \\ \rightarrow & \{\} \mid \text{Inl } () \end{array}$

However this falls short when calls to **alloc'** are nested in the following way (where  $dd : [[1]]$  and  $d : [1]$ ):

the mode on the right-most arrow should be  $\text{!}\infty$ , but do we want to show that now?

285 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.



$$\begin{array}{lcl}
\{\} & | & \mathbf{alloc}' (\lambda dd \mapsto \mathbf{alloc}' (\lambda d \mapsto dd \blacktriangleleft d)) \\
\rightarrow \quad \{hd := \square\} & | & \mathbf{alloc}' (\lambda d \mapsto \rightarrow hd \blacktriangleleft d) \ ; \ \mathbf{deref} \rightarrow hd \\
\rightarrow \quad \{hd := \square, h := \square\} & | & \rightarrow hd \blacktriangleleft \rightarrow h \ ; \ \mathbf{deref} \rightarrow h \ ; \ \mathbf{deref} \rightarrow hd \\
\rightarrow \quad \{hd := \rightarrow h, h := \square\} & | & \mathbf{deref} \rightarrow h \ ; \ \mathbf{deref} \rightarrow hd
\end{array}$$

The original term  $\mathbf{alloc}' (\lambda dd \mapsto \mathbf{alloc}' (\lambda d \mapsto dd \blacktriangleleft d))$  is well typed, as the inner call to  $\mathbf{alloc}'$  returns a value of type  $\mathbf{1}$  (as  $d$  is of type  $\mathbf{1}$ ) and consumes  $d$  linearly. However, the variable  $d$  that stands for destination  $\rightarrow h$  isn't filled with a value but instead escapes its scope by being fed to a destination of destination  $dd$  coming from the outer scope (this still counts as a linear consumption). Hence the associated hole  $h$  doesn't receive a value and the reduction get stuck when trying to dereference  $\rightarrow h$ .

One could argue that the issue comes from the primitive  $\blacktriangleleft$  returning a value of type  $\mathbf{1}$  instead of a dedicated effect type. However, the same issue arise in following program, which is well-typed even when  $\blacktriangleleft$  and the function accepted by  $\mathbf{alloc}'$  have an arbitrary return type  $\mathbf{E}$ , so we decided not to introduce yet another type in the system:

$$\begin{array}{l}
\mathbf{alloc}' (\lambda dd \mapsto \mathbf{case} (dd \blacktriangleleft (,)) \mathbf{of} (dd_1, d_2) \mapsto \\
\quad \mathbf{case} (\mathbf{alloc}' (\lambda d \mapsto dd_1 \blacktriangleleft d)) \mathbf{of} \{\mathbf{true} \mapsto d_2 \blacktriangleleft \mathbf{true}, \mathbf{false} \mapsto d_2 \blacktriangleleft \mathbf{false}\})
\end{array}$$

where  $dd : [\mathbf{Bool}] \otimes \mathbf{Bool}$ ,  $dd_1 : [\mathbf{Bool}]$ ,  $d_2 : [\mathbf{Bool}]$ ,  $d : [\mathbf{Bool}]$ .

In the next section, we motivate why being able to store destinations in destinations of destinations is a desirable property of our system that we don't want to give up. In Section 6, we present a finer type system that prevents scope escape while still allowing to store destinations in destination-based data structures.

#### 4 BREADTH-FIRST TREE TRAVERSAL

The core example that showcases the power of destination-passing style programming with first-class destination — that we borrow from [Bagrel 2024] — 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  $1 \dots |T|$  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, that is, building the leaves first and going up to the root.

In the aforementioned paper, 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 (*input subtree*, *dest to output subtree*) of the queue is processed, and pairs of the same shape for children nodes are appended at the end of the queue.

However, as evoked earlier, the API presented in [Bagrel 2024] is not able to store linear data, and in particular destinations, in destination-based data structures. So they cannot use the efficient, destination-based queue implementation from Section 2.2 to power up the breadth-first tree traversal implementation<sup>2</sup>. With  $\lambda_d$ , this is now possible. In fact, our system is self-contained, in the sense that every possible structure can be built using destination-based primitives (and regular data constructors can be retrieved from destination-based primitives, as detailed in Section 5.4).

Figure 3 presents the  $\lambda_d$  implementation of the breadth-first tree traversal. We assume that we have a binary tree type alias  $\mathbf{Tree\ T}$  and natural number type alias  $\mathbf{Nat}$  encoded using standard

I remember that we did wonder about that at some point, so it's worth pre-empting 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. Thomas: the second example is too long/complex to show the whole semantics, so I kept the first one in a detailed fashion, and just mentioned the second one.

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

<sup>2</sup>This efficient queue implementation can be, and is in fact, implemented in [Bagrel 2024]: see [archive.softwareheritage.org/swh:1:cnt:29e9d1fd48d94fa8503023bee0d607d281f512f8](https://archive.softwareheritage.org/swh:1:cnt:29e9d1fd48d94fa8503023bee0d607d281f512f8). But it cannot store linear data

```

344 go : ((!100recS) → T1 → (!100S) ⊗ T2) ωV → (!100S) → Queue (Tree T1 ⊗ [Tree T2]) → (!100S)
345 go f st q  $\triangleq$  case (dequeue q) of {
346   Inl ()  $\mapsto$  st,
347   Inr ((tree, dtree), q')  $\mapsto$  case tree of {
348     Inl ()  $\mapsto$  dtree  $\triangleleft$  Nil ; go f st q',
349     Inr (x, (tl, tr))  $\mapsto$ 
350       let z := case (dtree  $\triangleleft$  Node) of (dy, (dtl, dtr))  $\mapsto$  (f st x) in
351       case z of (st', y)  $\mapsto$ 
352         dy  $\triangleleft$  y ;
353         go f st' (q' enqueue (tl, dtl) enqueue (tr, dtr))
354   }
355 }
356 mapAccumBFS : ((!100S) → T1 → (!100S) ⊗ T2) ωV → (!100S) → Tree T1 → Tree T2 ⊗ (!100S)
357 mapAccumBFS f st tree  $\triangleq$  from× (map alloc with dtree  $\mapsto$  go f st (singleton (tree, dtree)))
358 relabelDPS : Tree 1 → (Tree Nat) ⊗ (!100(!ωVNat))
359 relabelDPS tree  $\triangleq$  mapAccumBFS
360   (λex  $\mapsto$  λun  $\mapsto$  case100 case un ; ex of
361     E100 ex'  $\mapsto$  ex' of
362       EωV st  $\mapsto$  (E100 (EωV (succ st)), st))
363     (E100 (EωV (succ zero))))
364   tree
365
366
367
368
369
370
371
372
373

```

Fig. 3. Breadth-first tree traversal in destination-passing style

sum and product types. **Tree T** is equipped with operators  $\triangleleft$ Nil and  $\triangleleft$ Node, that are implemented in terms of our core destination-filling primitives.

The implementation is as similar as possible to the one from [Bagrel 2024], as to make it easier to spot the few differences between the two systems.

**TODO:** There is an issue here: because **go** is called after the call to **map**, then every argument is now older, especially **tree** is no longer of the same age as **dtree** (need an exponential probably)

Thomas: Stopped there in the refactor at 17:30 on 02/07/2024.

The first important difference is that in the destination calculus implementation, the input tree of type **Tree T<sub>1</sub>** 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 !<sub>100</sub> so that it can be extracted from the right side of the ampar at the end of the processing (with **from**<sub>×</sub>). We could imagine a more general version of the traversal, having no constraint on the state type, but necessitating a finalization function  $S \rightarrow !_{100}S'$  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  $1 \dots |T|$  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 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  $!_{\infty}$  is part of **from<sub>x</sub>** 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  $!_{\omega}$  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 **from<sub>x</sub>** operator, we would be able to use just one exponential layer  $!_{\omega\infty}$  over the natural number to achieve the same result.

## 5 LANGUAGE SYNTAX

### 5.1 Introducing the ampar

Minamide's work [Minamide 1998] 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.

In fact, the type of hole abstraction  $(T_1, T_2)\text{hfun}$  in Minamide'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  $T_1 \multimap T_2$  into  $T_1^\perp \wp T_2$ . Doing so for the *wand* type  $(T_1, T_2)\text{hfun}$  or  $T_1 \multimap T_2$  gives  $[T_1] \wp T_2$ , where  $[\cdot]$  is memory negation, and  $\wp$  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  $[T_1]$  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*  $\wp$  to *ampar*  $\ltimes$ . In CLL, the cut rule states that given  $T_1 \wp T_2$ , 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  $T^\perp$ , or  $T^{\perp-1} = T'$  if  $T$  is already of the form  $T'^\perp$ . In a classical setting, thanks to the involutive nature of negation  $^\perp$ , the two potential forms of the eliminator of  $T$  are equal.

In destination calculus though, we don't have an involutive memory negation  $[\cdot]$ . If we are provided with a destination of destination  $\rightarrow h' : [T]$ , we know that some structure is expecting to store a destination of type  $[T]$ . If ever that structure is consumed, then the destination stored

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

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 trawl through a lot of text to make sure they are properly propagated.

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  $\boxed{h} : T$  and its linked destination  $\rightarrow h : [T]$ , 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  $[T]$ , both storing it through a destination of destination  $[[T]]$  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  $[T]$  originating from  $T \hat{\bowtie} [T]$  (through a destination of destination of type  $[[T]]$ ) 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 an 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 ( $\nu$ ), 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.

491	$var, x, y, d, dd, un, xs, ys, ex, st, tree, tl, tr, dtree, f, dh, dt, dx, dy, dxs, dys, dv, dtlr, dtl, dtr, q, tok, z$	Variable names
492	$hname, h, hd ::=$	Hole (or destination) name, represented by a natural number
493	$  h+h'$	M
494	$  h[H\pm h']$	M Shift by $h'$ if $h \in H$
495	$  \max(H)$	M Maximum of a set of hole names
496		
497	$hnames, H ::=$	Set of hole names
498	$  \{h_1, \dots, h_k\}$	
499	$  H_1 \cup H_2$	M Union of sets
500	$  H\pm h'$	M Shift all names from $H$ by $h'$ .
501	$  hnames(\Gamma)$	M Hole names bound by the typing context $\Gamma$
502	$  hnames(C)$	M Hole names bound by the evaluation context $C$
503		
504		
505		
506		
507		

Fig. 4. Grammar for variable, hole and destination names

## 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 4.

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  $hnames$   $H, H_1, H_2 \dots$

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.

## 5.3 Term and value core syntax

Destination calculus is based on linear simply-typed  $\lambda$ -calculus, with built-in support for sums, pairs, and exponentials. The syntax of terms, which is presented in Figure 5 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.

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  $\{h\} \langle \boxed{h} \rangle \rightarrow h$  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  $!_{\text{coa}}(T_1 \otimes T_2) \simeq (!_{\text{coa}} T_1) \otimes (!_{\text{coa}} T_2)$ , which is not part of intuitionistic linear logic, but valid in Linear Haskell[Bernardy et al. 2018]. We omit the mode annotation on **case** statements and lambda abstractions when the mode in question is the multiplicative neutral element  $1v$  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

term, $t, u$	Term
$v$	Value
$x$	Variable
$t' t$	Application
$t \S u$	Pattern-match on unit
$\text{case}_m t \text{ of } \{\text{Inl } x_1 \mapsto u_1, \text{Inr } x_2 \mapsto u_2\}$	Pattern-match on sum
$\text{case}_m t \text{ of } (x_1, x_2) \mapsto u$	Pattern-match on product
$\text{case}_m t \text{ of } E_n x \mapsto u$	Pattern-match on exponential
$\text{map } t \text{ with } x \mapsto t'$	Map over the right side of ampar
$\text{to}_\times u$	Wrap into a trivial ampar
$\text{from}_\times t$	Convert ampar to a pair
$t \triangleleft ()$	Fill destination with unit
$t \triangleleft \text{Inl}$	Fill destination with left variant
$t \triangleleft \text{Inr}$	Fill destination with right variant
$t \triangleleft E_m$	Fill destination with exponential constructor
$t \triangleleft ()$	Fill destination with product constructor
$t \triangleleft (\lambda x_m \mapsto u)$	Fill destination with function
$t \triangleleft \bullet t'$	Fill destination with root of other ampar
$t[x := v] \quad M$	
val, $v$	Value
$\boxed{h}$	Hole
$\rightarrow h$	Destination
$()$	Unit
$\forall \lambda x_m \mapsto u$	Function with no free variable
$\text{Inl } v$	Left variant for sum
$\text{Inr } v$	Right variant for sum
$E_m v$	Exponential
$(v_1, v_2)$	Product
$H \langle v_2 \wedge v_1 \rangle$	Ampar
$v[H \triangleq h'] \quad M$	Shift hole names inside $v$ by $h'$ if they belong to $H$ .

Fig. 5. Grammar for terms and values

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.

$\text{to}_\times$  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 ampars by writing the root of the second one to a destination of the first one, so by throwing  $\text{to}_\times$  to the mix, we can compose an ampar with a normal (completed) structure (see the sugar operator “fillLeaf” ( $\triangleleft$ ) in Section 5.4).

$\text{from}_\times$  is used to convert an ampar to a pair, when the right side of the ampar is an exponential of the form  $E_{\text{too}} v$ . Indeed, when the right side has such form, it cannot contain 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.  $\text{from}_\times$  is in particular used to extract a structure from its ampar building shell when it is complete (see the sugar operator  $\text{from}_{\times'}$  in Section 5.4).

589	term, $\tilde{t}$	::=		Syntactic sugar for terms
590		alloc	M	Evaluate to a fresh new ampar
591		$t \blacktriangleleft t'$	M	Fill destination with supplied term
592		<b>from</b> <sub><math>\times</math></sub> ' $t$	M	Extract left side of ampar when right side is unit
593		$\lambda x \text{ }_m \mapsto u$	M	Allocate function
594		lnl $t$	M	Allocate left variant
595		lnr $t$	M	Allocate right variant
596		$E_m t$	M	Allocate exponential
597		$(t_1, t_2)$	M	Allocate product

Fig. 6. Syntactic sugar forms for terms

The remaining term operators  $\blacktriangleleft()$ ,  $\blacktriangleleft\text{lnl}$ ,  $\blacktriangleleft\text{lnr}$ ,  $\blacktriangleleft E_m$ ,  $\blacktriangleleft()$ ,  $\blacktriangleleft(\lambda x \text{ }_m \mapsto u)$  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  $\Delta$  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 6.

**from** <sub>$\times$</sub> ' is a simpler variant of **from** <sub>$\times$</sub>  that allows to extract the right side of an ampar when the right side has been fully consumed. We implement it in terms of **from** <sub>$\times$</sub>  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 7.

### 6 TYPE SYSTEM

$\lambda_d$  is equipped with a first-order type system, featuring modal function types and modal boxing, in addition to unit (1), product ( $\otimes$ ) and sum ( $\oplus$ ) types. Modes have two axes — multiplicity (i.e. linear/non-linear), and age control — and they take place on variable bindings in typing contexts, and on function arrows, but are not part of the type itself. The type system is highly inspired from [Abel and Bernardy 2020] and Linear Haskell [Bernardy et al. 2018]. In particular, it uses the same additive/multiplicative approach on contexts for linearity and age enforcement.

Let's now focus on age control, which is a novel feature of this calculus.

```

638 alloc  $\triangleq \{1\} \langle \boxed{1} \rightarrow 1 \rangle$ 
639
640  $\text{from}'_{\times} t \triangleq \text{case case } (\text{from}_{\times} (\text{map } t \text{ with } un \mapsto un \circ E_{\text{loc}})) \text{ of}$ 
641    $(st, ex) \mapsto ex \text{ of}$ 
642    $E_{\text{loc}} un \mapsto un \circ st$ 
643
644  $\lambda x_{\text{m}} t \mapsto u \triangleq \text{from}'_{\times} (\text{map}$ 
645    $\text{alloc with } d \mapsto$ 
646    $d \triangleleft (\lambda x_{\text{m}} t \mapsto u)$ 
647    $)$ 
648
649  $(t_1, t_2) \triangleq \text{from}'_{\times} (\text{map}$ 
650    $\text{alloc with } d \mapsto$ 
651    $\text{case } (d \triangleleft (,)) \text{ of}$ 
652    $(d_1, d_2) \mapsto d_1 \triangleleft t_1 \circ d_2 \triangleleft t_2$ 
653    $)$ 
654
655  $t \triangleleft t' \triangleq t \triangleleft \bullet (to_{\times} t')$ 
656
657  $\text{ofInl } t \triangleq \text{from}'_{\times} (\text{map}$ 
658    $\text{alloc with } d \mapsto$ 
659    $d \triangleleft \text{Inl} \triangleleft t$ 
660    $)$ 
661
662  $\text{Inr } t \triangleq \text{from}'_{\times} (\text{map}$ 
663    $\text{alloc with } d \mapsto$ 
664    $d \triangleleft \text{Inr} \triangleleft t$ 
665    $)$ 
666
667  $E_{\text{m}} t \triangleq \text{from}'_{\times} (\text{map}$ 
668    $\text{alloc with } d \mapsto$ 
669    $d \triangleleft E_{\text{m}} \triangleleft t$ 
670    $)$ 

```

Fig. 7. Desugaring of syntactic sugar forms for terms

### 6.1 Keep track of age of bindings to prevent scope escape of destinations

The solution we chose to alleviate the issue detailed in Section 3 is 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  $\nu$  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  $\infty$  is introduced for variables standing in place of a non-age-controlled value. In particular, destinations can never have age  $\infty$  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.  $\infty$  is absorbing for both addition and multiplication.

Tables for the operations  $+$  and  $\cdot$  on ages are presented in Figure 8.

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

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

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

Fig. 8. Tables for age operations

Age commutative semiring is then combined with the multiplicity commutative semiring from [Bernardy et al. 2018] to form a canonical product commutative semiring that is used to represent the mode of each typing context binding in our final type system.

### 6.2 Types, modes, and typing contexts

The whole grammar of types, modes, and typing context for  $\lambda_d$  is presented in Figure 9. As said earlier, the function arrow  $\rightarrow_{\text{m}}$  and exponential connective  $!_{\text{m}}$  are both parametrized by a mode  $\text{m}$ ; we omit that mode annotation when the mode in question is the multiplicative neutral element  $1_{\nu}$  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  $\rightarrow$  from [Girard 1995].



687	type, $T, U, S$	::=	Type
688		$1$	Unit
689		$T_1 \oplus T_2$	Sum
690		$T_1 \otimes T_2$	Product
691		$!_m T$	Exponential
692		$U \ltimes T$	Ampar
693		$T_m \rightarrow U$	Function
694		$[_m T]$	Destination
695	mode, $m, n$	::=	Mode (Semiring)
696		$pa$	Pair of a multiplicity and age
697		$\omega$	Error case (incompatible types, multiplicities, or ages)
698			
699	mul, $p$	::=	Multiplicity (Semiring, first component of mode)
700		$1$	Linear use
701		$\omega$	Non-linear use
702			
703	age, $a$	::=	Age (Semiring, second component of mode)
704		$\nu$	Born now
705		$\uparrow$	One scope older
706		$\infty$	Infinitely old / static
707			
708	ctx, $\Omega, \Gamma, \Delta, \Theta$	::=	Typing context
709		$x :_m T$	Variable typing binding
710		$[h] :_n T$	Hole typing binding
711		$\rightarrow h :_m [{}_n T]$	Destination typing binding
712		$m \cdot \Omega$	$M$ Multiply the leftmost mode of each binding by $m$
713		$\Omega_1 + \Omega_2$	$M$ Sum (incompatible bindings get tagged with $\omega$ )
714		$\Omega_1, \Omega_2$	$M$ Disjoint sum
715		$\rightarrow^{-1} \Delta$	$M$ Transforms dest bindings into a hole bindings
716		$\Omega[H \sharp h']$	$M$ Shift hole/dest names by $h'$ if they belong to $H$

Fig. 9. Types, modes, and typing contexts

### 6.3 Typing of terms and values

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 new idea of our type system is to feature *hole bindings*  $[h] :_n T$  and *destination bindings*  $\rightarrow h :_m [{}_n T]$  in addition to the variable bindings  $x :_m T$  that usually populates typing contexts.

The mode  $n$  of a hole binding  $[h] :_n T$  (also present in the corresponding destination type  $[{}_n T]$ ) 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).<sup>3</sup> 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  $[h]$  in  $E_{\omega\nu}[h]$ . In particular, we should not store a destination into this hole, otherwise it could later be extracted and used in a non-linear fashion.

<sup>3</sup>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.

$\Theta \vdash t : T$			(Typing judgment for terms)		
$\frac{\text{TY-TERM-VAL} \quad \text{DisposableOnly } \Theta \quad \Delta \Vdash v : T}{\Theta, \Delta \vdash v : T}$	$\frac{\text{TY-TERM-VAR} \quad \text{DisposableOnly } \Theta \quad !v <: m}{\Theta, x : m T \vdash x : T}$	$\frac{\text{TY-TERM-APP} \quad \Theta_1 \vdash t : T \quad \Theta_2 \vdash t' : T \multimap U}{m \Theta_1 + \Theta_2 \vdash t' t : U}$			
$\frac{\text{TY-TERM-PATU} \quad \Theta_1 \vdash t : T \quad \Theta_2 \vdash u : U}{\Theta_1 + \Theta_2 \vdash t ; u : U}$	$\frac{\text{TY-TERM-PATS} \quad \Theta_1 \vdash t : T_1 \oplus T_2 \quad \Theta_2, x_1 : m T_1 \vdash u_1 : U \quad \Theta_2, x_2 : m T_2 \vdash u_2 : U}{m \Theta_1 + \Theta_2 \vdash \text{case}_m t \text{ of } \{ \text{Inl } x_1 \mapsto u_1, \text{Inr } x_2 \mapsto u_2 \} : U}$				
$\frac{\text{TY-TERM-PATP} \quad \Theta_1 \vdash t : T_1 \otimes T_2 \quad \Theta_2, x_1 : m T_1, x_2 : m T_2 \vdash u : U}{m \Theta_1 + \Theta_2 \vdash \text{case}_m t \text{ of } (x_1, x_2) \mapsto u : U}$	$\frac{\text{TY-TERM-PATE} \quad \Theta_1 \vdash t : !_n T \quad \Theta_2, x : m \cdot n T \vdash u : U}{m \Theta_1 + \Theta_2 \vdash \text{case}_m t \text{ of } E_n x \mapsto u : U}$				
$\frac{\text{TY-TERM-MAP} \quad \Theta_1 \vdash t : U \times T \quad !\uparrow \Theta_2, x : !v T \vdash t' : T'}{\Theta_1 + \Theta_2 \vdash \text{map } t \text{ with } x \mapsto t' : U \times T'}$	$\frac{\text{TY-TERM-TOA} \quad \Theta \vdash u : U}{\Theta \vdash \text{to}_\times u : U \times 1}$	$\frac{\text{TY-TERM-FROMA} \quad \Theta \vdash t : U \times (!_{100} T)}{\Theta \vdash \text{from}_\times t : U \otimes (!_{100} T)}$			
$\frac{\text{TY-TERM-FILLU} \quad \Theta \vdash t : [!_n T]}{\Theta \vdash t \triangleleft () : 1}$	$\frac{\text{TY-TERM-FILLL} \quad \Theta \vdash t : [!_n T_1 \oplus T_2]}{\Theta \vdash t \triangleleft \text{Inl} : [!_n T_1]}$	$\frac{\text{TY-TERM-FILLR} \quad \Theta \vdash t : [!_n T_1 \oplus T_2]}{\Theta \vdash t \triangleleft \text{Inr} : [!_n T_2]}$	$\frac{\text{TY-TERM-FILLP} \quad \Theta \vdash t : [!_n T_1 \otimes T_2]}{\Theta \vdash t \triangleleft () : [!_n T_1] \otimes [!_n T_2]}$		
$\frac{\text{TY-TERM-FILLE} \quad \Theta \vdash t : [!_n !_{n'} T]}{\Theta \vdash t \triangleleft E_{n'} : [!_{n' \cdot n} T]}$	$\frac{\text{TY-TERM-FILLF} \quad \Theta_1 \vdash t : [!_n T \multimap U] \quad \Theta_2, x : m T \vdash u : U}{\Theta_1 + (!\uparrow \cdot n) \Theta_2 \vdash t \triangleleft (\lambda x_m \mapsto u) : 1}$		$\frac{\text{TY-TERM-FILLCOMP} \quad \Theta_1 \vdash t : [!_n U] \quad \Theta_2 \vdash t' : U \times T}{\Theta_1 + (!\uparrow \cdot n) \Theta_2 \vdash t \triangleleft \bullet t' : T}$		
$\Theta \vdash \tilde{t} : T$			(Derived typing judgment for syntactic sugar forms)		
$\frac{\text{TY-STERM-ALLOC} \quad \text{DisposableOnly } \Theta}{\Theta \vdash \text{alloc} : T \times [T]}$	$\frac{\text{TY-STERM-FROMA}' \quad \Theta \vdash t : T \times 1}{\Theta \vdash \text{from}'_\times t : T}$	$\frac{\text{TY-STERM-FILLLEAF} \quad \Theta_1 \vdash t : [!_n T] \quad \Theta_2 \vdash t' : T}{\Theta_1 + (!\uparrow \cdot n) \Theta_2 \vdash t \triangleleft t' : 1}$			
$\frac{\text{TY-STERM-FUN} \quad \Theta_2, x : m T \vdash u : U}{\Theta_2 \vdash \lambda x_m \mapsto u : T \multimap U}$	$\frac{\text{TY-STERM-LEFT} \quad \Theta_2 \vdash t : T_1}{\Theta_2 \vdash \text{Inl } t : T_1 \oplus T_2}$	$\frac{\text{TY-STERM-RIGHT} \quad \Theta_2 \vdash t : T_2}{\Theta_2 \vdash \text{Inr } t : T_1 \oplus T_2}$	$\frac{\text{TY-STERM-EXP} \quad \Theta_2 \vdash t : T}{m \Theta_2 \vdash E_m t : !_m T}$		
$\frac{\text{TY-STERM-PROD} \quad \Theta_{21} \vdash t_1 : T_1 \quad \Theta_{22} \vdash t_2 : T_2}{\Theta_{21} + \Theta_{22} \vdash (t_1, t_2) : T_1 \otimes T_2}$					

Fig. 10. Typing rules for terms and syntactic sugar

A destination binding  $\rightarrow h : m [!_n T]$  mentions two modes,  $m$  and  $n$ ; but only the former (left one,  $m$ ) is the actual mode of the binding (in particular, it informs on the age of the destination itself). The latter,  $n$ , is part of the destination's type  $[!_n T]$  and corresponds to the mode a value has to have to be written to the corresponding hole. In a well-typed, closed program, the mode  $m$  of a

destination binding can never be of multiplicity  $\omega$  or age  $\infty$  in the typing tree; it is always linear and of finite age.

We also extend mode product to a point-wise action on typing contexts:

$$\left\{ \begin{array}{lcl} n' \cdot (x :_m T) & = & x :_{n' \cdot m} T \\ n' \cdot ([h] :_n T) & = & [h] :_{n' \cdot n} T \\ n' \cdot (\rightarrow h :_m [n] T) & = & \rightarrow h :_{n' \cdot m} [n] T \end{array} \right.$$

Figure 10 presents the typing rules for terms, and rules for syntactic sugar forms that have been derived from term rules and proven formally too. Figure 12 presents the typing rules for values of the language. In every figure,

- $\Omega$  denotes an arbitrary typing context, with no particular constraint;
- $\Gamma$  denotes a typing context made only of hole and destination bindings;
- $\Theta$  denotes a typing context made only of destination and variable bindings;
- $\Delta$  denotes a typing context made only of destination bindings.

**6.3.1 Typing of terms  $\vdash$ .** A term  $t$  always types in a context  $\Theta$  made only of destination and variable bindings. That being said, typing rules for terms and their syntactic sugar in Figure 10 never explicitly mention a destination binding; only variable bindings. Only at runtime some variables will be substituted by destinations having a matching type and mode (that is why terms cannot type in a context made of variable bindings alone). As a result, the type system the user has to deal with is only slightly more complex than a linear type system *à la* [Bernardy et al. 2018], because of the addition of the age control axis. So at the moment, we can forget about destination and hole bindings specificities.

Let's focus on a few particularities of the type system for terms.

The predicate DisposableOnly  $\Theta$  in rules TY-TERM-VAL and TY-TERM-VAR says that  $\Theta$  can only contain variable bindings with multiplicity  $\omega$ , for which weakening is allowed in linear logic. It is enough to allow weakening at the leaves of the typing tree, that is to say the two aforementioned rules (TY-TERM-VAL is indeed a leaf for judgments  $\vdash$ , that holds a subtree of judgments  $\mathbf{v} \vdash$ ).

Rule TY-TERM-VAR, in addition to weakening, allows for dereliction of the mode for the variable used, with subtyping constraint  $1v <: m$  defined as such:

$$\left\{ \begin{array}{l} pa <: p'a' \iff p \stackrel{p}{<} p' \wedge a \stackrel{a}{<} a' \\ m <: \omega \end{array} \right. \quad \left\{ \begin{array}{l} 1 \stackrel{p}{<} 1 \\ p \stackrel{p}{<} \omega \end{array} \right. \quad \left\{ \begin{array}{l} \uparrow^m \stackrel{a}{<} \uparrow^n \iff m = n \quad (\text{no finite age dereliction ; recall that } \uparrow^0 = v) \\ a \stackrel{a}{<} \infty \end{array} \right.$$

Rule TY-TERM-PATU is elimination (or pattern-matching) for unit, and is also used to chain destination-filling operations.

Rules TY-TERM-APP, TY-TERM-PATS, TY-TERM-PATP and TY-TERM-PATE are all parametrized by a mode  $m$  by which the typing context  $\Theta_1$  of the argument is multiplied. These rules otherwise follows closely from [Bernardy et al. 2018].

The Rule TY-TERM-MAP is where most of the safety of the system lies, and it is there where scope control takes place. It opens an ampar  $t$ , and binds its right side (containing destinations for holes on

revisit if we allow weakening for dests, as we could replace that by  $\omega v \cdot \Theta$  as in [Bernardy et al. 2018]

the other side, among other things) to variable  $x$  and then execute body  $t'$ . The core idea is that **map** creates a new scope for  $x$  and  $t'$ , so anything coming from the current ambient scope (represented by  $\Theta_2$  in the conclusion) appears older when we see it from  $t'$  point of view. Indeed, when entering a new scope, the age of every remaining binding from the previous scopes is incremented by  $\uparrow$ . That way we can distinguish  $x$  from anything else that was already bound using the age of bindings alone. That's why  $t'$  types in  $\uparrow\uparrow\Theta_2, x : \uparrow\uparrow T$  while the global term **map**  $t$  with  $x \mapsto t'$  types in  $\Theta_1, \Theta_2$  (notice the absence of shift on  $\Theta_2$ ). A schematic explanation of the scope rules is given in Figure 11.

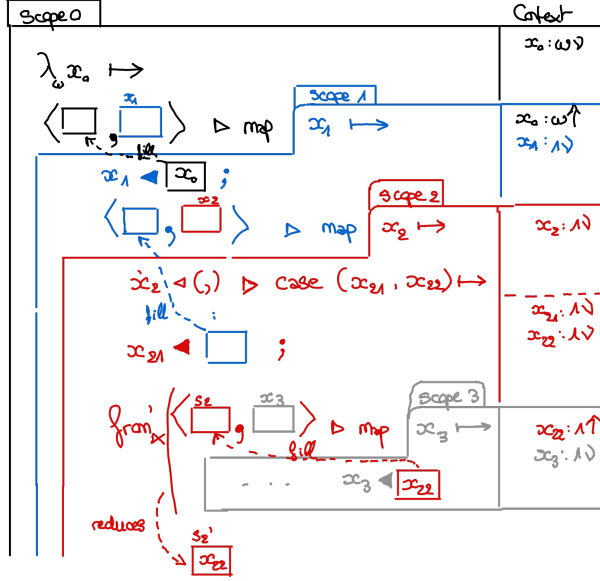


Fig. 11. Scope rules for **map** in  $\lambda_d$

We see in the schema that the left of an ampar (the structure being built) “takes place” in the ambient scope. The right side however, where destinations are, has its own new, inner scope that is opened when mapped over. When filling a destination (e.g.  $x_1 \blacktriangleleft x_0$  in the figure), the right operand must be from a scope one  $\uparrow$  older than the destination on the left of the operator, as this value will end up on the left of the ampar (which is thus in a scope  $\uparrow$  older than the destination originating from the right side).

The rule **TY-TERM-FILLCOMP** from Figure 10, or its simpler variant, **TY-TERM-FILLLEAF** confirm this intuition. The left operand of these operators must be a destination that types in the ambient context (both  $\Theta_1$  unchanged in the premise and conclusion of the rules). The right operand, however, is a value that types in a context  $\Theta_2$  in the premise, but requires  $\uparrow\uparrow\Theta_2$  in the conclusion. This is the opposite of the shift that **map** does: while **map** opens a child scope for its body, “fillComp” ( $\blacktriangleleft$ )/“fillLeaf” ( $\blacktriangleleft$ ) opens a portal to the parent scope for their right operand, as seen in the schema. The same phenomenon happens for the resources captured by the body of a lambda abstraction in **TY-TERM-FILLF**.

When using **from<sub>x</sub>**, the left of an ampar is extracted to the ambient scope (as seen at the bottom of the figure with  $x_{22}$ ). This is the fundamental reason why the left of an ampar has to “take place” in that scope. When an ampar is complete and disposed of with the more general **from<sub>x</sub>** however, we extract both sides of the ampar to the ambient scope, even though the right side is not part of

883	$\boxed{\Gamma \vdash v : T}$				(Typing judgment for values)
884	TY-VAL-HOLE	TY-VAL-DEST	TY-VAL-UNIT	TY-VAL-FUN	
885				$\Delta, x : {}_m T \vdash u : U$	
886	$\boxed{h} : {}_l v \vdash \boxed{h} : T$	$\rightarrow h : {}_l v \vdash \rightarrow h : {}_n T$	$\cdot \vdash () : 1$	$\Delta \vdash \lambda x. {}_m \mapsto u : T \mapsto U$	
887					
888	TY-VAL-LEFT	TY-VAL-RIGHT	TY-VAL-PROD	TY-VAL-EXP	
889	$\Gamma \vdash v_1 : T_1$	$\Gamma \vdash v_2 : T_2$	$\Gamma_1 \vdash v_1 : T_1 \quad \Gamma_2 \vdash v_2 : T_2$	$\Gamma \vdash v' : T$	
890	$\Gamma \vdash \text{Inl } v_1 : T_1 \oplus T_2$	$\Gamma \vdash \text{Inr } v_2 : T_1 \oplus T_2$	$\Gamma_1 + \Gamma_2 \vdash (v_1, v_2) : T_1 \otimes T_2$	${}^n \Gamma \vdash E_n v' : {}^n T$	
891					
892					
893					
894					
895					
896					
897					
898					
899					
900					
901					
902					
903					
904					
905					
906					
907					
908					
909					
910					
911					
912					
913					
914					
915					
916					
917					
918					
919					
920					
921					
922					
923					
924					
925					
926					
927					
928					
929					
930					
931					

Fig. 12. Typing rules for values

that scope. This is only safe to do because the right side is required to have type  $!_{\infty} T$ , which means it is scope-insensitive (it cannot contain any scope-controlled resource).

**6.3.2 Typing of values  $\vdash$ .** The typing of runtime values is where hole and destination bindings appears. In fact, only these kinds of bindings can appear: the absence of free variables in values make it easier to prove substitution properties (as we will see in Section 7.3, we perform substitutions not only in terms, but also in evaluation contexts sometimes).

Hole bindings and destination bindings of the same hole name  $h$  are meant to annihilate each other in the typing context given they have a matching base type and mode. That way, the typing context of a term can stay constant during reduction:

- when destination-filling primitives are evaluated to build up data structures, they linearly consume a destination and write to a hole at the same time which makes both disappear, thus the typing context stays balanced;
- when a new hole is created, a matching destination is returned too, so the typing context stays balanced too.

However, the annihilation between a destination and a hole binding having the same name in the typing tree is only allowed to happen around an ampar, as it is the ampar connective that bind the name across the two sides (the names bound are actually stored in a set  $H$  on the ampar value  $H \langle v_2 \wedge v_1 \rangle$ ). In fact, an ampar can be seen as a sort of lambda-abstraction, whose body (containing holes instead of variables) and sink are split across two sides, and magically interconnected through the ampar connective.

In most rules, we use a sum  $\Gamma_1 + \Gamma_2$  for typing contexts (or the disjoint variant  $\Gamma_1, \Gamma_2$ ). This sum doesn't not allow for annihilation of bindings with the same name; the operation is partial, and in particular it isn't defined if a same hole name is present in the operands in the two different forms (hole binding and destination binding). In particular, a pair  $(\boxed{h}, \rightarrow h)$  is not well-typed. A single typing context  $\Gamma$  is not allowed either to contain both a hole binding and a destination binding for the same hole name.

**Typing of ampars.** As stated above, the core idea of TY-VAL-AMPAR is to act as a binding connective for hole and destinations.

We define a new operator  $\rightarrow^{-i}$  to represent the matching hole bindings for a set of destination bindings. It is a partial, point-wise operation on typing bindings of a context where:

$$\rightarrow^{-i}(\rightarrow h :_{iV} \lfloor_n T \rfloor) = \boxed{h} :_n T$$

Only an input context  $\Delta$  made only of destination bindings, with mode  $1v$ , results in a valid output context (which is then only composed of hole bindings).

Equipped with this operator, we introduce the annihilation of bindings using  $\Delta_3$  to represent destinations on the right side  $v_1$ , and  $\rightarrow^{-i}\Delta_3$  to represent the matching hole bindings on the left side  $v_2$  (the structure under construction). Those bindings are only present in the premises of the rule but get removed in the conclusion. Those hole names are only local, bound to that ampar and don't affect the outside world nor can be referenced from the outside.

Both sides of the ampar may also contain stored destinations from other scopes, represented by  $\uparrow\Delta_1$  and  $\Delta_2$  in the respective typing contexts of  $v_1$  and  $v_2$ . All holes introduced by this ampar have to be annihilated by matching destinations; following our naming convention, no hole binding can appear in  $\Delta_1$ ,  $\Delta_2$  in the conclusion.

The properties  $\text{LinOnly } \Delta_3$  and  $\text{FinAgeOnly } \Delta_3$  are true given that  $\rightarrow^{-i}\Delta_3$  is a valid typing context, so are not really a new restriction on  $\Delta_3$ . They are mostly used to ease the mechanical proof of type safety for the system.

#### Other notable typing rules for value.

Rules TY-VAL-HOLE and TY-VAL-DEST indicates that a hole or destination must have mode  $1v$  in the typing context to be used (except when a destination is stored away, as we will see later).

Rules for unit, left and right variants, and product are straightforward.

Rule TY-VAL-EXP is rather classic too: we multiply the dependencies  $\Gamma$  of the value by the mode  $n$  of the exponential. The intuition is that if  $v$  uses a resource  $v'$  twice, then  $E_2 v$ , that corresponds to two uses of  $v$  (in a system with such a mode), will use  $v'$  four times.

Rule TY-VAL-FUN indicates that (value level) lambda abstractions cannot have holes inside. In other terms, a function value cannot be built piecemeal like other data structures, its whole body must be a complete term right from the beginning. It cannot contain free variables either, as the body of the function must type in context  $\Delta$ ,  $x :_m T$  where  $\Delta$  is made only of destination bindings. One might wonder, how can we represent a curried function  $\lambda x \mapsto \lambda y \mapsto x \text{ concat } y$  as the value level, as the inner abstraction captures the free variable  $x$ ? The answer is that such a function, at value level, is encoded as  $\lambda x \mapsto \text{from}'_x (\text{map alloc with } d \mapsto d \triangleleft (\lambda y \mapsto x \text{ concat } y))$ , where the inner closure is not yet in value form, but pending to be built into a value. As the form  $d \triangleleft (\lambda y \mapsto t)$  is part of term syntax, and not value syntax, we allow free variable captures in it.

## 7 EVALUATION CONTEXTS AND SEMANTICS

The semantics of  $\lambda_d$  are given using small-step reductions on a pair  $C[t]$  of an evaluation context  $C$  (represented by a stack) determining a focusing path, and a term  $t$  under focus. Such a pair  $C[t]$  is called a *command*, and represents a running program.

### 7.1 Evaluation contexts forms

The grammar of evaluation contexts is given in Figure 13. An evaluation context  $C$  is the composition of an arbitrary number of focusing components  $c_1, c_2 \dots$ . We chose to represent this composition explicitly using a stack, instead of a meta-operation that would only let us access its final result. As a result, focusing and defocusing operations are made explicit in the semantics, resulting in a more verbose but simpler proof. It is also easier to imagine how to build a stack-based interpreter for such a language.

Not sure whether we should explain why  $\Delta_1$  is offset by  $1i$  in the premise but not in conclusion. The reason is "because it is needed" more than having a good intuition narrative behind it.

I want to mark a break in the text here, but I'm not sure how to do it.

revisit this if we allow weakening for dests



999	ectxs, C	::=		Evaluation context stack
1000			$\square$	Represent the empty stack / "identity" evaluation context
1001			$C \circ c$	Push $c$ on top of $C$
1002			$C[h :=_H v]$	Fill $h$ in $C$ with value $v$ (that may contain holes)
			$M$	

Fig. 13. Grammar for evaluation contexts

Focusing components are all directly derived from the term syntax, except for the “open ampar” focus  $\overset{\text{op}}{\underset{H}{\text{F}}} \langle v_2 \wedge \square \rangle$ . This focus component indicates that an ampar is currently being **m**apped on, with its left-hand side  $v_2$  (the structure being built) being attached to the “open ampar” focus component, while its right-hand side (containing destinations) is either in subsequent focus components, or in the term under focus.

We introduce a special substitution  $\mathbb{C}[h :=_H v]$  that is used to update structures under construction that are attached to open ampar focus components in the stack. Such a substitution is triggered when a destination  $\rightarrow h$  is filled in the term under focus, and results in the value  $v$  (that may contain holes itself, e.g. if it is a hollow constructor  $(\boxed{h_1}, \boxed{h_2})$ ) being written to the hole  $\boxed{h}$  (that must appear somewhere on an open ampar payload). The set  $H$  tracks the potential hole names introduced by value  $v$ .

## 7.2 Typing of evaluation contexts and commands

Evaluation contexts are typed in a context  $\Delta$  that can only contains destination bindings.  $\Delta$  represents the typing context available for the term that will be put in the box  $\square$  of the evaluation context. In other terms, while the typing context of a term is a list of requirements so that it can be typed, the typing context of an evaluation context is the set of bindings that it makes available to the term under focus. As a result, while the typing of a term  $\Theta \vdash t : T$  is additive (the typing requirements for a function application is the sum of the requirements for the function itself and for its argument), the typing of an evaluation context  $\Delta \dashv C : T \multimap U_0$  is subtractive : adding the focus component  $t' \square$  to the stack  $C$  will remove whatever is needed to type  $t'$  from the typing

context provided by  $C$ . The whole typing rules for evaluation contexts  $C$  as well as commands  $C[t]$  are presented in Figure 14.

An evaluation context has a pseudo-type  $T \rightarrow U_0$ , where  $T$  denotes the type of the focus (i.e. the type of the term that can be put in the box of the evaluation context) while  $U_0$  denotes the type of the resulting command (when the box of the evaluation context is filled with a term).

Composing an evaluation context of pseudo-type  $T \rightarrow U_0$  with a new focus component never affects the type  $U_0$  of the future command ; only the type  $T$  of what can be put in the box is altered.

All typing rules for evaluation contexts can be derived from the ones for the corresponding term (except for the rule `TY-ECTXS-OPENAMPAR-FOC` that is the truly new form). Let's take the rule `TY-ECTXS-PATP-FOC` as an example:

- the typing context  $m \cdot \Delta_1 + \Delta_2$  in the premise for  $C$  corresponds to  $m \cdot \Theta_1 + \Theta_2$  in the conclusion of `TY-TERM-PATP` in Figure 10;
- the typing context  $\Delta_2, x_1 : m T_1, x_2 : m T_2$  in the premise for term  $u$  corresponds to the typing context  $\Theta_2, x_1 : m T_1, x_2 : m T_2$  for the same term in `TY-TERM-PATP`;
- the typing context  $\Delta_1$  in the conclusion for  $C \circ (\text{case}_m \square \text{ of } (x_1, x_2) \mapsto u)$  corresponds to the typing context  $\Theta_1$  in the premise for  $t$  in `TY-TERM-PATP` (the term  $t$  is located where the box  $\square$  is in `TY-ECTXS-OPENAMPAR-FOC`).

In a way, the typing rule for an evaluation context is a “rotation” of the typing rule for the associated term, where the typing contexts of one premise and the conclusion are swapped, and the typing context of the other potential premise is kept unchanged (with the added difference that free variables cannot appear in typing contexts of evaluation contexts, so any  $\Theta$  becomes a  $\Delta$ ).

As we see at the bottom of the figure, a command  $C[t]$  (i.e. a pair of an evaluation context and a term) is well typed when the evaluation context  $C$  provides a typing context  $\Delta$  that is exactly one in which  $t$  is well typed. We can always embed a well-typed, closed term  $\bullet \vdash t : T$  as a well-typed command using the identity evaluation context:  $t \simeq \square[t]$  and we thus have  $\vdash \square[t] : T$  where  $\Delta = \bullet$  (the empty context).

### 7.3 Small-step semantics

**TODO: Mention why we don't use store semantics as showcased in Section 3**

We equip  $\lambda_d$  with small-step semantics. There are three types of semantic rules:

- focus rules, where we remove a layer from term  $t$  (which cannot be a value) and push a corresponding focus component on the stack  $C$ ;
- unfocus rules, where  $t$  is a value and thus we pop a focus component from the stack  $C$  and transform it back to a term, so that a redex appears (or so that another focus/unfocus rule can be triggered);
- reduction rules, where the actual computation logic takes place.

Here is the whole set of rules for PATP:

$$\begin{array}{l}
 \text{SEM-PATP-FOC} \quad \frac{\text{NotVal } t}{C[\text{case}_m t \text{ of } (x_1, x_2) \mapsto u] \longrightarrow (C \circ (\text{case}_m \square \text{ of } (x_1, x_2) \mapsto u))[t]} \\
 \text{SEM-PATP-UNFOC} \quad \frac{}{(C \circ (\text{case}_m \square \text{ of } (x_1, x_2) \mapsto u))[v] \longrightarrow C[\text{case}_m v \text{ of } (x_1, x_2) \mapsto u]} \\
 \text{SEM-PATP-RED} \quad \frac{}{C[\text{case}_m (v_1, v_2) \text{ of } (x_1, x_2) \mapsto u] \longrightarrow C[u[x_1 := v_1][x_2 := v_2]]}
 \end{array}$$

Rules are triggered in a purely deterministic fashion; once a subterm is a value, it cannot be focused again. As focusing and defocusing rules are entirely mechanical (they are just a matter of

1079	$\boxed{\Delta \dashv C : T \rightarrow U_0}$		(Typing judgment for evaluation contexts)
1080		TY-ECTXS-APP-FOC1	TY-ECTXS-APP-FOC2
1081		$\mathfrak{m} \Delta_1, \Delta_2 \dashv C : U \rightarrow U_0$	$\mathfrak{m} \Delta_1, \Delta_2 \dashv C : U \rightarrow U_0$
1082	TY-ECTXS-ID	$\Delta_2 \vdash t' : T \xrightarrow{\mathfrak{m}} U$	$\Delta_1 \vdash v : T$
1083	$\bullet \dashv \square : U_0 \rightarrow U_0$	$\Delta_1 \dashv C \circ (t' \square) : T \rightarrow U_0$	$\Delta_2 \dashv C \circ (\square v) : (T \xrightarrow{\mathfrak{m}} U) \rightarrow U_0$
1084		TY-ECTXS-PATS-FOC	
1085			$\mathfrak{m} \Delta_1, \Delta_2 \dashv C : U \rightarrow U_0$
1086	TY-ECTXS-PATU-FOC	$\Delta_2, x_1 : \mathfrak{m} T_1 \vdash u_1 : U$	$\Delta_2, x_2 : \mathfrak{m} T_2 \vdash u_2 : U$
1087	$\Delta_2 \vdash u : U$		
1088	$\Delta_1 \dashv C \circ (\square \circ \circ u) : 1 \rightarrow U_0$	$\Delta_1 \dashv C \circ (\text{case}_{\mathfrak{m}} \square \text{ of } \{\text{Inl } x_1 \mapsto u_1, \text{Inr } x_2 \mapsto u_2\}) : (T_1 \oplus T_2) \rightarrow U_0$	
1089			
1090	TY-ECTXS-PATP-FOC		TY-ECTXS-PATE-FOC
1091	$\mathfrak{m} \Delta_1, \Delta_2 \dashv C : U \rightarrow U_0$		$\mathfrak{m} \Delta_1, \Delta_2 \dashv C : U \rightarrow U_0$
1092	$\Delta_2, x_1 : \mathfrak{m} T_1, x_2 : \mathfrak{m} T_2 \vdash u : U$		$\Delta_2, x : \mathfrak{m} \mathfrak{m}' T \vdash u : U$
1093	$\Delta_1 \dashv C \circ (\text{case}_{\mathfrak{m}} \square \text{ of } (x_1, x_2) \mapsto u) : (T_1 \otimes T_2) \rightarrow U_0$		$\Delta_1 \dashv C \circ (\text{case}_{\mathfrak{m}} \square \text{ of } E_{\mathfrak{m}'} x \mapsto u) : !\mathfrak{m}' T \rightarrow U_0$
1094			
1095	TY-ECTXS-MAP-FOC		TY-ECTXS-TOA-FOC
1096	$\Delta_1, \Delta_2 \dashv C : U \times T' \rightarrow U_0$		$\Delta \dashv C : (U \times 1) \rightarrow U_0$
1097	$\uparrow \Delta_2, x : \mathfrak{!} T \vdash t' : T'$		$\Delta \dashv C \circ (\text{to}_{\times} \square) : U \rightarrow U_0$
1098	$\Delta_1 \dashv C \circ (\text{map} \square \text{ of } x \mapsto t') : (U \times T) \rightarrow U_0$		
1099			
1100	TY-ECTXS-FROMA-FOC		TY-ECTXS-FILLU-FOC
1101	$\Delta \dashv C : (U \otimes (!_{\infty} T)) \rightarrow U_0$		$\Delta \dashv C : 1 \rightarrow U_0$
1102	$\Delta \dashv C \circ (\text{from}_{\times} \square) : (U \times (!_{\infty} T)) \rightarrow U_0$		$\Delta \dashv C \circ (\square \triangleleft ()) : [n] 1 \rightarrow U_0$
1103			
1104	TY-ECTXS-FILLL-FOC		TY-ECTXS-FILLR-FOC
1105	$\Delta \dashv C : [n] T_1 \rightarrow U_0$		$\Delta \dashv C : [n] T_2 \rightarrow U_0$
1106	$\Delta \dashv C \circ (\square \triangleleft \text{Inl}) : [n] T_1 \oplus T_2 \rightarrow U_0$		$\Delta \dashv C \circ (\square \triangleleft \text{Inr}) : [n] T_1 \oplus T_2 \rightarrow U_0$
1107			
1108	TY-ECTXS-FILLP-FOC		TY-ECTXS-FILLE-FOC
1109	$\Delta \dashv C : ([n] T_1 \otimes [n] T_2) \rightarrow U_0$		$\Delta \dashv C : [m, n] T \rightarrow U_0$
1110	$\Delta \dashv C \circ (\square \triangleleft (,)) : [n] T_1 \otimes T_2 \rightarrow U_0$		$\Delta \dashv C \circ (\square \triangleleft E_m) : [n, m] T \rightarrow U_0$
1111			
1112	TY-ECTXS-FILLF-FOC		TY-ECTXS-FILLCOMP-FOC1
1113	$\Delta_1, (\uparrow \mathfrak{n}) \Delta_2 \dashv C : 1 \rightarrow U_0$		$\Delta_1, (\uparrow \mathfrak{n}) \Delta_2 \dashv C : T \rightarrow U_0$
1114	$\Delta_2, x : \mathfrak{m} T \vdash u : U$		$\Delta_2 \vdash t' : U \times T$
1115	$\Delta_1 \dashv C \circ (\square \triangleleft (\lambda x_{\mathfrak{m}} \mapsto u)) : [n] T \xrightarrow{\mathfrak{m}} U \rightarrow U_0$		$\Delta_1 \dashv C \circ (\square \triangleleft \bullet t') : [n] U \rightarrow U_0$
1116			
1117	TY-ECTXS-FILLCOMP-FOC2		TY-ECTXS-OPENAMPAR-FOC
1118	$\Delta_1, (\uparrow \mathfrak{n}) \Delta_2 \dashv C : T \rightarrow U_0$		$\text{LinOnly } \Delta_3 \quad \text{FinAgeOnly } \Delta_3$
1119	$\Delta_1 \vdash v : [n] U$		$\text{hnames}(C) \quad \# \# \quad \text{hnames}(\Delta_3)$
1120	$\Delta_2 \dashv C \circ (v \triangleleft \bullet \square) : U \times T \rightarrow U_0$		$\Delta_1, \Delta_2 \dashv C : (U \times T') \rightarrow U_0$
1121			$\Delta_2, \rightarrow \cdot \Delta_3 \quad \forall \vdash \quad v_2 : U$
1122			$\uparrow \Delta_1, \Delta_3 \dashv C \circ (\text{hnames}(\Delta_3) \langle v_2 \wedge \square \rangle) : T' \rightarrow U_0$
1123			
1124	$\boxed{\vdash C[t] : T}$		(Typing judgment for commands)
1125		TY-CMD	
1126		$\Delta \dashv C : T \rightarrow U_0 \quad \Delta \vdash t : T$	
1127		$\vdash C[t] : U_0$	

Fig. 14. Typing rules for evaluation contexts and commands

pushing and popping a focus component on the stack), we only present the set of reduction rules for the system in Figure 15.

The reduction rules for function application, pattern-matching, **to**<sub>×</sub> and **from**<sub>×</sub> are straightforward.

All reduction rules for destination-filling primitives trigger a substitution  $C[h :=_H v]$  on the evaluation context  $C$  that corresponds to a memory update of a hole  $\boxed{h}$ . SEM-FILLU-RED and SEM-FILLF-RED do not create any new hole; they only write a value to an existing one. On the other hand, rules SEM-FILL-RED, SEM-FILLR-RED, SEM-FILLE-RED and SEM-FILLP-RED all write a hollow constructor to the hole  $h$ , that is to say a value containing holes itself. Thus, we need to generate fresh names for these new holes, and also return a destination for each new hole with a matching name.

Obtaining a fresh name is represented by the statement  $h' = \max(hnames(C) \cup \{h\}) + 1$  in the premises of these rules. The implicit invariant of the system is that we always rename hole names inside an ampar to fresh names when that ampar is **mapped** on. Names bound by a closed ampar are only local, and thus can shadow already existing names (that's why **alloc** is allowed to reuse the same hole name 1 at each invocation: it doesn't matter as it desugars to a closed ampar). Thus we only need to take into account the names of already opened ampars — all part of the evaluation context — when we want fresh names (either for new holes, or for ampar renaming).

In rule SEM-FILLCOMP-RED, we write the left-hand side  $v_2$  of a closed ampar  $H\langle v_2 \wedge v_1 \rangle$  to a hole  $\boxed{h}$  that is part of some focus fragment  $^{op}_{H'}\langle v'_2 \wedge \square \rangle$  in the evaluation context  $C$ . That fragment is not mentioned explicitly in the rule, as the destination  $\rightarrow h$  is enough to target it. This results in the composition of two structures with holes  $v'_2$  and  $v_2$  through filling of  $\rightarrow h$ . Because we split open the ampar  $H\langle v_2 \wedge v_1 \rangle$  (its left-hand side gets written to a hole, while its right hand side is returned), we need to rename any hole name that it contains to a fresh one, as we do when an ampar is opened in the **map** rule. The renaming is carried out by the conditional shift  $[H \pm h']$  on  $v_2$  and  $v_1$  (only hole names local to the ampar, represented by the set  $H$ , gets renamed).

Last but not least, rules SEM-MAP-RED-OPENAMPAR-FOC and SEM-OPENAMPAR-UNFOC dictates how and when a closed ampar (a term) is converted to an open ampar (a focusing fragment) and vice-versa.

With SEM-MAP-RED-OPENAMPAR-FOC, the local hole names of the ampar gets renamed to fresh ones, and the left-hand side gets attached to the focusing fragment  $^{op}_{H \pm h'}\langle v_2 [H \pm h'] \wedge \square \rangle$  while the right-hand side (containing destinations) is substituted in the body of the **map** statement (which becomes the new term under focus). This effectively allows the right-hand side of an ampar to be a term instead of a value for a limited time.

The rule SEM-OPENAMPAR-UNFOC triggers when the body of a **map** statement has reduced to a value. In that case, we can close the ampar, by popping the focus fragment from the stack  $C$  and merging back with  $v_1$  to reform a closed ampar.

*Type safety.* With the semantics now defined, we can state the usual type safety theorems:

**THEOREM 7.1 (TYPE PRESERVATION).** *If  $\vdash C[t] : T$  and  $C[t] \rightarrow C'[t']$  then  $\vdash C'[t'] : T$ .*

**THEOREM 7.2 (PROGRESS).** *If  $\vdash C[t] : T$  and  $\forall v, C[t] \neq \square[v]$  then  $\exists C', t'. C[t] \rightarrow C'[t']$ .*

A command of the form  $\square[v]$  cannot be reduced further, as it only contains a fully determined value, and no pending computation. This is the expected stopping point of the reduction, and any well-typed command is supposed to reach such a form at some point.

1177	$\boxed{C[t] \longrightarrow C'[t']}$	(Small-step evaluation of commands)
1178	SEM-APP-RED	SEM-PATU-RED
1179	$\frac{}{C[(\lambda x_{\mathfrak{m}} \mapsto u) v] \longrightarrow C[u[x := v]]}$	$\frac{}{C[(\circledast u)] \longrightarrow C[u]}$
1180		
1181	SEM-PATL-RED	
1182	$\frac{}{C[\text{case}_{\mathfrak{m}} (\text{Inl } v_1) \text{ of } \{\text{Inl } x_1 \mapsto u_1, \text{Inr } x_2 \mapsto u_2\}] \longrightarrow C[u_1[x_1 := v_1]]}$	
1183		
1184	SEM-PATR-RED	
1185	$\frac{}{C[\text{case}_{\mathfrak{m}} (\text{Inr } v_2) \text{ of } \{\text{Inl } x_1 \mapsto u_1, \text{Inr } x_2 \mapsto u_2\}] \longrightarrow C[u_2[x_2 := v_2]]}$	
1186		
1187	SEM-PATP-RED	
1188	$\frac{}{C[\text{case}_{\mathfrak{m}} (v_1, v_2) \text{ of } (x_1, x_2) \mapsto u] \longrightarrow C[u[x_1 := v_1][x_2 := v_2]]}$	
1189		
1190	SEM-PATE-RED	SEM-TOA-RED
1191	$\frac{}{C[\text{case}_{\mathfrak{m}} E_{\mathfrak{n}} v' \text{ of } E_{\mathfrak{n}} x \mapsto u] \longrightarrow C[u[x := v']]} \quad \frac{}{C[\text{to}_{\ltimes} v_2] \longrightarrow C[\{\langle v_2 \wedge () \rangle]}$	
1192		
1193	SEM-FROMA-RED	SEM-FILLU-RED
1194	$\frac{}{C[\text{from}_{\ltimes} \{\langle v_2 \wedge E_{\mathfrak{m}} v_1 \rangle\}] \longrightarrow C[(v_2, E_{\mathfrak{m}} v_1)]} \quad \frac{}{C[\rightarrow h \triangleleft ()] \longrightarrow C[h := \{\} ()][()]} \quad \frac{}{C[\rightarrow h \triangleleft (\lambda x_{\mathfrak{m}} \mapsto u)] \longrightarrow C[h := \{\} \lambda x_{\mathfrak{m}} \mapsto u][()]}$	
1195		
1196	SEM-FILLF-RED	
1197	$\frac{}{C[\rightarrow h \triangleleft (\lambda x_{\mathfrak{m}} \mapsto u)] \longrightarrow C[h := \{\} \lambda x_{\mathfrak{m}} \mapsto u][()]}$	
1198		
1199	SEM-FILLL-RED	SEM-FILLR-RED
1200	$\frac{h' = \max(\text{hnames}(C) \cup \{h\}) + 1}{C[\rightarrow h \triangleleft \text{Inl}] \longrightarrow C[h := \{h'+1\} \text{Inl } \boxed{h'+1}][\rightarrow h'+1]} \quad \frac{h' = \max(\text{hnames}(C) \cup \{h\}) + 1}{C[\rightarrow h \triangleleft \text{Inr}] \longrightarrow C[h := \{h'+1\} \text{Inr } \boxed{h'+1}][\rightarrow h'+1]}$	
1201		
1202	SEM-FILLE-RED	
1203	$\frac{h' = \max(\text{hnames}(C) \cup \{h\}) + 1}{C[\rightarrow h \triangleleft E_{\mathfrak{m}}] \longrightarrow C[h := \{h'+1\} E_{\mathfrak{m}} \boxed{h'+1}][\rightarrow h'+1]}$	
1204		
1205	SEM-FILLP-RED	
1206	$\frac{h' = \max(\text{hnames}(C) \cup \{h\}) + 1}{C[\rightarrow h \triangleleft (.)] \longrightarrow C[h := \{h'+1, h'+2\} (\boxed{h'+1}, \boxed{h'+2})][(\rightarrow h'+1, \rightarrow h'+2)]}$	
1207		
1208	SEM-FILLCOMP-RED	
1209	$\frac{h' = \max(\text{hnames}(C) \cup \{h\}) + 1}{C[\rightarrow h \triangleleft \bullet_H \langle v_2 \wedge v_1 \rangle] \longrightarrow C[h := (H \pm h') v_2 [H \pm h']][v_1 [H \pm h']]} \quad \frac{}{C[\text{map } H \langle v_2 \wedge v_1 \rangle \text{ with } x \mapsto t'] \longrightarrow (C \circ \text{op}_{H \pm h'}^{\text{op}} \langle v_2 [H \pm h'] \wedge \square \rangle)[t' [x := v_1 [H \pm h']]]}$	
1210		
1211	SEM-OPENAMPAR-UNFOC	
1212	$\frac{}{(C \circ \text{op}_H^{\text{op}} \langle v_2 \wedge \square \rangle)[v_1] \longrightarrow C[H \langle v_2 \wedge v_1 \rangle]}$	
1213		
1214		
1215		
1216		
1217		
1218		
1219		
1220		
1221		
1222		
1223		
1224		
1225		

Fig. 15. Small-step semantics

## 8 REMARKS ON THE FORMAL PROOFS

We've proved type preservation and progress theorems with the Coq proof assistant. At time of writing, we have assumed, rather than proved, the substitution lemmas. The choice of turning to a proof assistant was a pragmatic choice: the context handling in  $\lambda_d$  can be quite finicky, and it was hard, without computer assistance, to make sure that we hadn't made mistakes in our proofs. The version of  $\lambda_d$  that we've proved is written in Ott, the same Ott file is used as a source for this article, making sure that we've proved the same system as we're presenting; some visual simplification is applied by a script to produce the version in the article.

Most of the proof was done by an author with little prior experience with Coq. This goes to show that Coq is reasonably approachable even for non-trivial development. The proof is about 6000 lines long, and contains nearly 350 lemmas. Many of the cases of the type preservation and progress lemmas are similar, to handle such repetitive cases using of a large-language-model based autocompletion system has been quite effective.

Binders are the biggest problem. We've largely manage to make the proof to be only about closed terms, to avoid any complication with binders. This worked up until the substitution lemmas, which is the reason why we haven't proved them in Coq yet (that and the fact that it's much easier to be confident in our pen-and-paper proofs for those). There are backends to generate locally nameless representations from Ott definitions; we haven't tried them yet, but the unusual binding nature of ampars may be too much for them to handle.

The proofs aren't very elegant. For instance, we don't have any abstract formalization of semirings: since our semirings are finite it was more expedient to brute-force the properties we needed by hand. We've observed up to 232 simultaneous goals, but a computer makes short work of this: it was solved by a single call to the congruence tactic. Nevertheless there are a few points of interest.

- We represent context as finite-domain functions, rather than as syntactic lists. This works much better when defining sums of context. There are a bunch of finite-function libraries in the ecosystem, but we needed finite dependent functions (because the type of binders depend on whether we're binding a variable name or a hole name). This didn't exist, but for our limited purpose, it ended up not being too costly rolling our own. About 1000 lines of proofs. The underlying data type is actual functions, this was simpler to develop, but equality is more complex than with a bespoke data type.
- We make the mode semiring total by adding an invalid mode. This prevents us from having to deal with partiality at all. The cost is that contexts can contain binders with invalid mode. Proofs are written so as to rule out this case.

The inference rules produced by Ott aren't conducive to using setoid equality. This turned out to be a problem with our type for finite function:

```
Record T A B := {
  underlying :> forall x:A, option (B x);
  supported : exists l : list A, Support l underlying;
}.
```

where `Support l f` means that `l` contains the domain of `f`. To make the equality of finite function be strict equality `eq`, we assumed functional extensionality and proof irrelevance. In some circumstances, we've also needed to list the finite functions' domains. But in the definition, the domain is sealed behind a proposition, so we also assumed classical logic as well as indefinite description

```
Axiom constructive_indefinite_description :
  forall (A : Type) (P : A->Prop), (exists x, P x) -> { x : A | P x }.
```



together, they let us extract the domain from the proposition. Again this isn't particularly elegant, we could have avoided some of these axioms at the price of more complex development. But for the sake of this article, we decided to favor expediency over elegance.

## 9 IMPLEMENTATION OF DESTINATION CALCULUS USING IN-PLACE MEMORY MUTATIONS

The formal language presented in Sections 5 to 7 is not meant to be implemented as-is.

First,  $\lambda_d$  misses a form of recursion, but we believe that adding equirecursive types and a fix-point operator wouldn't compromise the safety of the system.

Secondly, ampars are not managed linearly in  $\lambda_d$ ; only destinations are. That is to say that an ampar can be wrapped in an exponential, e.g.  $E_{\omega V} \{h\} \langle \text{Inr} (\text{Inl } (), \boxed{h}) \rangle_{\wedge} \rightarrow h$  (representing a non-linear difference list  $E_{\omega V} (0 :: \square)$ ), and then used twice, each time in a different way:

```

case  $E_{\omega V} \{h\} \langle \text{Inr} (\text{Inl } (), \boxed{h}) \rangle_{\wedge} \rightarrow h$  of  $E_{\omega V} x \mapsto$ 
  let  $x_1 := x$  append (succ zero) in
  let  $x_2 := x$  append (succ (succ zero)) in
  toList ( $x_1$  concat  $x_2$ )
   $\rightarrow * 0 :: 1 :: 0 :: 2 :: []$ 

```

It may seem counter-intuitive at first, but this program is valid and safe in  $\lambda_d$ . Thanks to the renaming discipline we detailed in Section 7.3, every time an ampar is **mapped** over, its hole names are renamed to fresh ones. So when we call **append** to build  $x_1$  (which is implemented in terms of **map**), we sort of allocate a new copy of the ampar before mutating it, effectively achieving a copy-on-write memory scheme. Thus it is safe to operate on  $x$  again to build  $x_2$ .

In the introduction of the article, we announced a safe framework for in-place memory mutation, so we will uphold this promise now. The key to go from a copy-on-write scheme to an in-place mutation scheme is to force ampars to be linearly managed too. For that we introduce a new type **Token**, together with primitives **dup** and **drop** (remember that unqualified arrows have mode **1v**, so are linear):

```

dup : Token  $\rightarrow$  Token $\otimes$ Token
drop : Token  $\rightarrow$  1
alloccow : T  $\times$  [ T ]
allocip : Token  $\rightarrow$  T  $\times$  [ T ]

```

We now have two possible versions of **alloc**: the new one with an in-place mutation memory model (**ip**), that has to be managed linearly, and the old one that doesn't have to be used linearly, and features a copy-on-write (**cow**) memory model.

We use the **Token** type as an intrinsic source of linearity that infects the ampar returned by **alloc**<sub>ip</sub>. Such a token can be duplicated using **dup**, but as soon as it is used to create an ampar, that ampar cannot be duplicated itself. In the system featuring the **Token** type and **alloc**<sub>ip</sub>, that we call  $\lambda_{dip}$ , "closed" programs now typecheck in the non-empty context  $\{tok_0 :_{100} \text{Token}\}$  containing a token variable that the user can **duplicate** and **drop** freely to give birth to an arbitrary number of ampars, that will then have to be managed linearly.

Having closed programs to typecheck in non-empty context  $\{tok_0 :_{100} \text{Token}\}$  is very similar to having a primitive function **withToken** : (**Token**  $_{100} \rightarrow !_{\omega 100} \text{T}$ )  $\rightarrow !_{\omega 100} \text{T}$  as it is done in [Bagrel 2024].

In  $\lambda_{dip}$ , as ampars are managed linearly, we can change the allocation and renaming mechanisms:

- the hole name for a new ampar can be chosen fresh right from the start (this corresponds to a new heap allocation );

I don't think we need to write this: 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.

- adding a new hollow constructor still require fresh freshness/renaming for its hole names (this corresponds to a new heap allocation too);
- **mapping** over an ampar and filling destinations or composing two ampars using “fillComp” ( $\Leftarrow$ ) no longer require any renaming: we have the guarantee that the names are globally fresh, and thus we can do in-place memory updates.

We decided to omit the linearity aspect of ampars in  $\lambda_d$  as it clearly obfuscate the presentation of the system without adding much to the understanding of the latter. We believe that the system is still sound with this linearity aspect, and articles such as [Spiwack et al. 2022] gives a pretty clear view on how to implement the linearity requirement for ampars in practice without too much noise for the user.

## 10 RELATED WORK

### 10.1 Destination-passing style for efficient memory management

In [Shaikhha et al. 2017], the authors present a destination-based intermediate language for a functional array programming language. They develop a system of destination-specific optimizations and boast near-C performance.

This is the most comprehensive evidence to date of the benefit of destination-passing style for performance in functional programming languages. Although their work is on array programming, while this article focuses on linked data structure. They can therefore benefit of optimizations that are perhaps less valuable for us, such as allocating one contiguous memory chunk for several arrays.

The main difference between their work and ours is that their language is solely an intermediate language: it would be unsound to program in it manually. We, on the other hand, are proposing a type system to make it sound for the programmer to program directly with destinations.

We consider that these two aspects complement each other: good compiler optimization are important to alleviate the burden from the programmer and allowing high-level abstraction; having the possibility to use destinations in code affords the programmer more control would they need it.

### 10.2 Tail modulo constructor

Another example of destinations in a compiler’s optimizer is [Bour et al. 2021]. It’s meant to address the perennial problem that the map function on linked lists isn’t tail-recursive, hence consumes stack space. The observation is that there’s a systematic transformation of functions where the only recursive call is under a constructor to a destination-passing tail-recursive implementation.

Here again, there’s no destination in user land, only in the intermediate representation. However, there is a programmatic interface: the programmer annotates a function like

```
let[@tail_mod_cons] rec map =
```

to ask the compiler to perform the translation. The compiler will then throw an error if it can’t. This way, contrary to the optimizations in [Shaikhha et al. 2017], this optimization is entirely predictable.

This has been available in OCaml since version 4.14. This is the one example we know of of destinations built in a production-grade compiler. Our  $\lambda_d$  makes it possible to express the result tail-modulo-constructor in a typed language. It can be used to write programs directly in that style, or it could serve as a typed target language for and automatic transformation. On the flip-side, tail modulo constructor is too weak to handle our difference lists or breadth-first traversal examples.

### 10.3 A functional representation of data structures with a hole

The idea of using linear types to safely represent structures with holes dates back to [Minamide 1998]. Our system is strongly inspired by theirs. In their system, we can only compose functions that represent data structures with holes, we can't pattern-match on the result; just like in our system we cannot act on the left-hand side of  $S \ltimes T$ , only the right hand part.

In [Minamide 1998], it's only ever possible to represent structures with a single hole. But this is a rather superficial restriction. The author doesn't comment on this, but we believe that this restriction only exists for convenience of the exposition: the language is lowered to a language without function abstraction and where composition is performed by combinators. While it's easy to write a combinator for single-argument-function composition, it's cumbersome to write combinators for functions with multiple arguments. But having multiple-hole data structures wouldn't have changed their system in any profound way.

The more important difference is that while their system is based on a type of linear functions, ours is based on the linear logic's par combinator. This, in turns, lets us define a type of destinations which are representations of holes in values, which [Minamide 1998] doesn't have. This means that [Minamide 1998] can implement our examples with difference lists and queues from Section 2.2, but it can't do our breadth-first traversal example from Section 4, since storing destinations in a data structure is the essential ingredient of this example.

This ability to store destination does come at a cost though: the system needs this additional notion of ages to ensure that destinations are use soundly. On the other hand, our system is strictly more general, in that the system from [Minamide 1998] can be embedded in  $\lambda_d$ , and if one stays in this fragment, we're never confronted with ages. Ages only show up when writing programs that go beyond Minamide's system.

### 10.4 Tail modulo context

### 10.5 Destination-passing style programming: a Haskell implementation

In [Bagrel 2024], the author proposes a system much like ours: it has a par-like construct (that they call *Incomplete*), where only the right-hand side can be modified, and a destination type. The main difference is that in their system,  $d \blacktriangleleft t$  requires  $t$  to be unrestricted, while in  $\lambda_d$ ,  $t$  can be linear.

The consequence is that in [Bagrel 2024], destinations can be stored in data structures but not in data structures with holes. In order to do a breadth-first search algorithm like in Section 4, they can't use improved queues like we do, they have to use regular functional queues.

However, unlike  $\lambda_d$ , [Bagrel 2024] is implemented in Haskell, which features linear types. Our  $\lambda_d$ , with the age modes, needs more than what Haskell provides. Our system subsumes theirs, however, ages will appear in the typing rules for that fragment.

### 10.6 Semi-axiomatic sequent calculus

In, the author develop a system where constructors return to a destination rather than allocating memory. It is very unlike the other systems described in this section in that it's completely founded in the Curry-Howard isomorphism. Specifically it gives an interpretation of a sequent calculus which mixes Gentzen-style deduction rules and Hilbert-style axioms. As a consequence, the par connective is completely symmetric, and, unlike our  $[T]$  type, their dualization connective is involutive.

The cost of this elegance is that computations may try to pattern-match on a hole, in which case they must wait for the hole to be filled. So the semantic of holes is that of a future or a promise. In

You'll [Thomas] have to do this section, because I'm not familiar enough with that work

More of these grey ts I don't know why that is.

Add citation

turns this requires the semantic of their calculus to be fully concurrent. Which is a very different point in the design space.

## 11 CONCLUSION AND FUTURE WORK

Using a system of ages in addition to linearity,  $\lambda_d$  is a purely functional calculus which supports destination in a very flexible way. It subsumes existing calculi from the literature for destination passing, allowing both composition of data structures with holes and storing destinations in data structures. Data structures are allowed to have multiple holes, and destinations can be stored in data structures that, themselves, have holes. The latter is the main reason to introduce ages and is key to  $\lambda_d$ 's flexibility.

We don't anticipate that a system of ages like  $\lambda_d$  will actually be used in a programming language: it's unlikely that destination are so central to the design of a programming language that it's worth baking them so deeply in the type system. Perhaps a compiler that makes heavy use of destinations in its optimizer could use  $\lambda_d$  as a typed intermediate representation. But, more realistically, our expectation is that  $\lambda_d$  can be used as a theoretical framework to analyze destination-passing systems: if an API can be defined in  $\lambda_d$  then it's sound.

In fact, we plan to use this very strategy to design an API for destination passing in Haskell, leveraging only the existing linear types, but retaining the possibility of storing destinations in data structures with holes.

REFERENCES

- Andreas Abel and Jean-Philippe Bernardy. 2020. A unified view of modalities in type systems. *Proc. ACM Program. Lang.* 4, ICFP, Article 90 (aug 2020), 28 pages. <https://doi.org/10.1145/3408972>
- Thomas Bagrel. 2024. Destination-passing style programming: a Haskell implementation. In *35es Journées Francophones des Langages Applicatifs (JFLA 2024)*. Saint-Jacut-de-la-Mer, France. <https://inria.hal.science/hal-04406360>
- 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].
- Frédéric Bour, Basile Clément, and Gabriel Scherer. 2021. Tail Modulo Cons. In *JFLA 2021 - Journées Francophones des Langages Applicatifs*. Saint Médard d’Excideuil, France. <https://inria.hal.science/hal-03146495>
- J.-Y. Girard. 1995. Linear Logic: its syntax and semantics. In *Advances in Linear Logic*, Jean-Yves Girard, Yves Lafont, and Laurent Regnier (Eds.). Cambridge University Press, Cambridge, 1–42. <https://doi.org/10.1017/CBO9780511629150.002>
- 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>
- 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>
- Amir Shaikhha, Andrew Fitzgibbon, Simon Peyton Jones, and Dimitrios Vytiniotis. 2017. Destination-passing style for efficient memory management. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*. ACM, Oxford UK, 12–23. <https://doi.org/10.1145/3122948.3122949>
- Arnaud Spiwack, Csongor Kiss, Jean-Philippe Bernardy, Nicolas Wu, and Richard A. Eisenberg. 2022. Linearly qualified types: generic inference for capabilities and uniqueness. *Proceedings of the ACM on Programming Languages* 6, ICFP (Aug. 2022), 95:137–95:164. <https://doi.org/10.1145/3547626>