Destination calculus

A linear λ−calculus for purely functional memory writes

ANONYMOUS AUTHOR(S)

Destination passing —aka. out parameters— is taking a parameter to fill rather than returning a result from a function. Due to its apparent imperative nature, destination passing has struggled to find its way to pure functional programming. In this paper, we present a pure functional calculus with destinations at its core. Our calculus subsumes all the similar systems, and can be used to reason about their correctness or extension. In addition, our calculus can express programs that were previously not known to be expressible in a pure language. This is guaranteed by a modal type system where modes are used to manage both linearity and scopes. Type safety of our core calculus was proved formally with the Coq proof assistant.

CCS Concepts: • **Theory of computation → Type structures**; • **Software and its engineering → Formal language definitions**; **Functional languages**; **Data types and structures**.

Additional Key Words and Phrases: destination passing, functional programming, linear types, pure language

ACM Reference Format:
Anonymous Author(s). 2025. Destination calculus: A linear λ−calculus for purely functional memory writes. *Proc. ACM Program. Lang.* XX, XX, Article XXX (January 2025), 30 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

In destination-passing style, a function doesn't return a value: it takes as an argument a location where the output of the function ought to be written. A function of type $\mathsf{T} \to \mathsf{U}$ would, in destination-passing style, have type $\mathsf{T} \to \lfloor \mathsf{U} \rfloor \to 1$ instead, where $\lfloor \mathsf{U} \rfloor$ denotes a destination for value of type U. This style is common in system programming, where destinations $\lfloor \mathsf{U} \rfloor$ are more commonly known as "out parameters" (in C, $\lfloor \mathsf{U} \rfloor$ would typically be a pointer of type $\mathsf{U}*$).

The reason why system programs rely on destinations so much is that using destinations can save calls to the memory allocator. If a function returns a U, it has to allocate the space for a U. But with destinations, the caller is responsible for finding space for a U. The caller may simply ask for the space to the memory allocator, in which case we've saved nothing; but it can also reuse the space of an existing U which it doesn't need anymore, or space in an array, or even space in a region of memory that the allocator doesn't have access to, like a memory-mapped file.

This does all sound quite imperative, but we argue that the same considerations are relevant for functional programming, albeit to a lesser extent. In fact Shaikhha et al. [2017] has demonstrated that using destination passing in the intermediate language of a functional array-programming language allowed for significant optimizations. Where destinations truly shine in functional programming, however, is that they increase the expressiveness of the language; destinations as first-class values allow for meaningfully new programs to be written, as first explored in [Bagrel 2024].

The trouble, of course, is that destinations are imperative; we wouldn't want to sacrifice the immutability of our linked data structures (later on abbreviated *structures*) for the sake of the more situational destinations. The goal here is to extend functional programming just enough to be able

2475-1421/2025/1-ARTXXX $15.00
https://doi.org/XXXXXXX.XXXXXXX

to build immutable structures by destination passing without endangering purity and memory safety. This is already what Bagrel [2024] does, using a linear type system to restrict mutation. Destinations become write-once-only references into a structure with holes. In that we follow their leads, but we refine the type system further to allow for even more programs (see Section 3).

There are two key elements to the expressiveness of destination passing:

- structures can be built in any order. Not only from the leaves to the root, like in ordinary functional programming, but also from the root to the leaves, or any combination thereof. This can be done in ordinary functional programming using function composition in a form of continuation-passing; and destinations act as an optimization. This line of work was pioneered by Minamide [1998]. While this only increases expressiveness when combined with the next point, the optimization is significant enough that destination passing has been implemented in the Ocaml optimizer to support tail modulo constructor [Bour et al. 2021];
- when destinations are first-class values, they can be passed and stored like ordinary values. This is the innovation of [Bagrel 2024] upon which we build. The consequence is that not only the order in which a structure is built is arbitrary, this order can be determined dynamically during the runtime of the program.

To support this programming style, we introduce λ_d. We intend λ_d to serve as a foundational, theoretical calculus to reason about safe destinations in a functional setting. Indeed λ_d subsumes all the systems that we've discussed in this section. As such we expect that these systems or their extensions can be justified simply by giving them a translation into λ_d, in order to get all the safety results and metatheory of λ_d for free. Even though λ_d is not really meant to be implemented as a real programming language, we still draft an implementation strategy for it based on efficient mutations, in the line of [Bagrel 2024; Bour et al. 2021].

Our contributions are as follows:

- λ_d, a modal, linear, simply typed λ-calculus with destinations (Sections 5 and 6). λ_d is expressive enough to serve as an encoding for previous calculi with destinations (see Section 9);
- a demonstration that λ_d is more expressive than previous calculi with destinations (Sections 3 and 4), namely that destinations can be stored in structures with holes. We show how we can improve, in particular, on the breadth-first traversal example of [Bagrel 2024];
- an implementation strategy for λ_d which uses mutation without compromising the purity of λ_d (Section 8);
- formally-verified proofs, with the Coq proof assistant, of type safety (Section 7).

2 WORKING WITH DESTINATIONS

Let's introduce and get familiar with λ_d, our simply typed λ-calculus with destination. The syntax is standard, except that we use linear logic's $\mathsf{T} \oplus \mathsf{U}$ and $\mathsf{T} \otimes \mathsf{U}$ for sums and products, and linear function arrow $\multimap$, since λ_d is linearly typed, even though it isn't a focus in this section.

2.1 Building up a vocabulary

In its simplest form, destination passing, much like continuation passing, is using a location, received as an argument, to return a value. Instead of a linear function with signature $\mathsf{T} \multimap \mathsf{U}$, in λ_d you would have $\mathsf{T} \multimap \lfloor \mathsf{U} \rfloor \multimap 1$, where $\lfloor \mathsf{U} \rfloor$ is read "destination for type U". For instance, here is a destination-passing version of the identity function:

$$\mathbf{dId} : \mathsf{T} \multimap \lfloor \mathsf{T} \rfloor \multimap 1$$
$$\mathbf{dId}\ x\ d \triangleq d \blacktriangleleft x$$

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

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

$$\textbf{fillWithInl} : \lfloor T \oplus U \rfloor \multimap \lfloor T \rfloor$$
$$\textbf{fillWithInl}\ d \triangleq d \triangleleft \mathsf{Inl}$$

In this example, we're filling a destination for type $T \oplus U$ by setting the outermost constructor to left variant Inl. We think of $d \triangleleft \mathsf{Inl}$ (read "fill d with Inl") as allocating memory to store a block of the form Inl $\square$, write the address of that block to the location that d points to, and return a new destination of type $\lfloor T \rfloor$ pointing to the uninitialized argument of Inl. Uninitialized memory, when part of a structure or value, like $\square$ in Inl $\square$, is called a *hole*.

Notice that with **fillWithInl** we are constructing the structure from the outermost constructor inward: we've written a value of the form Inl $\square$ into a hole, but we have yet to describe what goes in the new hole $\square$. Such data constructors with uninitialized arguments are called *hollow constructors*[1]. This is opposite to how functional programming usually works, where values are built from the innermost constructors outward: first we make a value v and only then can we use Inl to make 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 destinations different from continuations? Part of the answer lies in our intention to effectively implement destinations as pointers to uninitialized memory (see Section 8). But where destinations really differ from continuations is when one has several destinations at hand. Then they have to fill *all* the destinations; whereas when one has multiple continuations, they can only return to one of them. Multiple destination arises when a destination for a pair gets filled with a hollow pair constructor:

$$\textbf{fillWithPair} : \lfloor T \otimes U \rfloor \multimap \lfloor T \rfloor \otimes \lfloor U \rfloor$$
$$\textbf{fillWithPair}\ d \triangleq d \triangleleft (,)$$

After using **fillWithPair**, both the first field *and* the second field must be filled, using the destinations of type $\lfloor T \rfloor$ and $\lfloor U \rfloor$ respectively. The key remark here is that **fillWithPair** couldn't exist if we replaced destinations by continuations, as we couldn't use both returned continuations easily.

Structures with holes. It is crucial to note 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 that is being filled. A destination of type $\lfloor T \rfloor$ is a pointer to a yet-undefined part of a bigger structure. We say that such a structure has a hole of type T; but the type of the structure itself isn't specified (and never appears in the signature of destination-filling functions). For instance, using **fillWithPair** only indicates that the structure being operated on has a hole of type $T \otimes U$ that is being written to.

Thus, we still need a type to tie the structure under construction — left implicit by destination-filling primitives — with the destinations representing its holes. To represent this, λ_d introduces a type $S \ltimes \lfloor T \rfloor$ for a structure of type S missing a value of type T to be complete. There can be several holes in S — resulting in several destinations on the right hand side — and as long as there remains holes in S, it cannot be read. For instance, $S \ltimes (\lfloor T \rfloor \otimes \lfloor U \rfloor)$ represents a S that misses both a T and a U to be complete (thus to be readable).

The general form $S \ltimes T$ is read "S ampar T". The name "ampar" stands for "asymmetric memory par"; we will explain how we came up with this type and name in Section 9.3. A similar connective is called Incomplete in [Bagrel 2024]. For now, it's sufficient to observe that $S \ltimes \lfloor T \rfloor$ is akin to a "par" type $S \parr T^{\perp}$ in linear logic; you can think of $S \ltimes \lfloor T \rfloor$ 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

[1]The full triangle $\blacktriangleleft$ is used to fill a destination with a fully-formed value, while the *hollow* triangle $\triangleleft$ is used to fill a destination with a *hollow constructor*.

elaborate on the value of having a "par" type with access to first-class destinations, rather than just linear functions to represent structures with holes, in Section 4.

Destinations always exist within the context of a structure with holes. A destination is both a witness of a hole present in the structure, and a handle to write to it. Crucially, destinations are otherwise ordinary values. To access the destinations of an ampar, λ_d provides a $\mathbf{upd}_{\ltimes}$ construction, which lets us apply a function to the right-hand side of the ampar. It is in the body of $\mathbf{upd}_{\ltimes}$ that functions operating on destinations can be called to update the structure:

$$\mathbf{fillWithInl'} : S \ltimes \lfloor T \oplus U \rfloor \multimap S \ltimes \lfloor T \rfloor$$
$$\mathbf{fillWithInl'}\ x \triangleq \mathbf{upd}_{\ltimes}\ x\ \mathbf{with}\ d \mapsto \mathbf{fillWithInl}\ d$$
$$\mathbf{fillWithPair'} : S \ltimes \lfloor T \otimes U \rfloor \multimap S \ltimes (\lfloor T \rfloor \otimes \lfloor U \rfloor)$$
$$\mathbf{fillWithPair'}\ x \triangleq \mathbf{upd}_{\ltimes}\ x\ \mathbf{with}\ d \mapsto \mathbf{fillWithPair}\ d$$

To tie this up, we need a way to introduce and to eliminate structures with holes. Structures with holes are introduced with $\mathbf{new}_{\ltimes}$ which creates a value of type $T \ltimes \lfloor T \rfloor$. $\mathbf{new}_{\ltimes}$ is a bit like the identity function: it is a hole (of type T) that needs a value of type T to be a complete value of type T. Memory-wise, it is an uninitialized block large enough to host a value of type T, and a destination pointing to it. Conversely, structures with holes are eliminated with[2] $\mathbf{from'}_{\ltimes} : S \ltimes 1 \multimap S$: if all the destinations have been consumed and only unit remains on the right side, then S no longer has holes and thus is just a normal, complete structure.

Equipped with these, we can, for instance, derive traditional constructors from piecemeal filling. In fact, λ_d doesn't have primitive constructor forms, constructors in λ_d are syntactic sugar. We show here the definition of Inl and (,), but the other constructors are derived similarly. Operator $\fatsemi$, present in second example, is used to chain operations returning unit type 1.

$$\mathsf{Inl} : T \multimap T \oplus U$$
$$\mathsf{Inl}\ x \triangleq \mathbf{from'}_{\ltimes}(\mathbf{upd}_{\ltimes}\ (\mathbf{new}_{\ltimes} : (T \oplus U) \ltimes \lfloor T \oplus U \rfloor)\ \mathbf{with}\ d \mapsto d \triangleleft \mathsf{Inl} \blacktriangleleft x)$$
$$(,) : T \multimap U \multimap T \otimes U$$
$$(x, y) \triangleq \mathbf{from'}_{\ltimes}(\mathbf{upd}_{\ltimes}\ (\mathbf{new}_{\ltimes} : (T \otimes U) \ltimes \lfloor T \otimes U \rfloor)\ \mathbf{with}\ d \mapsto \mathbf{case}\ (d \triangleleft (,))\ \mathbf{of}\ (d_1, d_2) \mapsto d_1 \blacktriangleleft x \fatsemi d_2 \blacktriangleleft y)$$

Memory safety and purity. At this point, the reader may be forgiven for feeling distressed at all the talk of mutations and uninitialized memory. How is it consistent with our claim to be building a pure and memory-safe language? The answer is that it wouldn't be if we'd allow unrestricted use of destinations. Instead λ_d uses a linear type system to ensure that:

- destinations are written at least once, preventing examples like:

  $$\mathbf{forget} : T$$
  $$\mathbf{forget} \triangleq \mathbf{from'}_{\ltimes}(\mathbf{upd}_{\ltimes}\ (\mathbf{new}_{\ltimes} : T \ltimes \lfloor T \rfloor)\ \mathbf{with}\ d \mapsto ())$$

  where reading the result of **forget** would result in reading the location pointed to by a destination that we never used, in other words, reading uninitialized memory;
- destinations are written at most once, preventing examples like:

  $$\mathbf{ambiguous1} : \mathsf{Bool}$$
  $$\mathbf{ambiguous1} \triangleq \mathbf{from'}_{\ltimes}(\mathbf{upd}_{\ltimes}\ (\mathbf{new}_{\ltimes} : \mathsf{Bool} \ltimes \lfloor \mathsf{Bool} \rfloor)\ \mathbf{with}\ d \mapsto d \blacktriangleleft \mathsf{true} \fatsemi d \blacktriangleleft \mathsf{false})$$
  $$\mathbf{ambiguous2} : \mathsf{Bool}$$
  $$\mathbf{ambiguous2} \triangleq \mathbf{from'}_{\ltimes}(\mathbf{upd}_{\ltimes}\ (\mathbf{new}_{\ltimes} : \mathsf{Bool} \ltimes \lfloor \mathsf{Bool} \rfloor)\ \mathbf{with}\ d \mapsto \mathbf{let}\ x := (d \blacktriangleleft \mathsf{false})\ \mathbf{in}\ d \blacktriangleleft \mathsf{true} \fatsemi x)$$

  where **ambiguous1** would return false and **ambiguous2** would return true due to evaluation order, even though let-expansion should be valid in a pure language.

[2] As the name suggest, there is a more general elimination $\mathbf{from}_{\ltimes}$. It will be discussed in Section 5.

```
List T ≜ 1⊕(T⊗(List T))
map′ : (T ⊸ U) ω∞⊸ List T ⊸ ⌊List U⌋ ⊸ 1
map′ f l dl ≜
  case l of {
    [] ↦ dl ◃ [],
    x :: xs ↦ case (dl ◃ (::)) of
      (dx, dxs) ↦ dx ◀ f x ⨟ map′ f xs dxs}
map : (T ⊸ U) ω∞⊸ List T ⊸ List U
map f l ≜ from′ᵏ (updᵏ (newᵏ : (List U) ⋉ ⌊List U⌋) with dl ↦ map′ f l dl)
```

Fig. 1. Tail-recursive map function on lists

2.2 Tail-recursive map

Now that we have an intuition of how destinations work, let's see how they can be used to build usual data structures. For this section, we suppose that λ_d has equirecursive types and a fixed-point operator. These aren't part of the formal system of Section 5 but don't add any complication.

Linked lists. We define lists as a fixpoint, as usual: List T ≜ 1⊕(T⊗(List T)). For convenience, we also define filling operators ◃[] and ◃(::):

```
◃[] : ⌊List T⌋ ⊸ 1
d ◃ [] ≜ d ◃ Inl ◃ ()
```

```
◃(::) : ⌊List T⌋ ⊸ ⌊T⌋⊗⌊List T⌋
d ◃ (::) ≜ d ◃ Inr ◃ (,)
```

Just like we did in Section 2.1 we can recover traditional constructors from filling operators, e.g.:

```
(::) : T⊗(List T) ⊸ List T
x :: xs ≜ from′ᵏ (updᵏ (newᵏ : (List T) ⋉ ⌊List T⌋) with d ↦
                  case (d ◃ (::)) of (dx, dxs) ↦ dx ◀ x ⨟ dxs ◀ xs)
```

A tail-recursive map function. List being ubiquitous in functional programming, the fact that the most natural way to write a map function on lists isn't tail recursive (hence consumes unbounded stack space), is unpleasant. Map can be made tail-recursive in two passes: first build the result list in reverse, then reverse it. But destinations let us avoid this two-pass process altogether, as they let us extend the tail of the result list directly. We give a complete implementation in Figure 1.

The main function is **map′**, it has type (T ⊸ U) ω∞⊸ List T ⊸ ⌊List U⌋ ⊸ 1. That is, instead of returning a resulting list, it takes a destination as an input and fills it with the result. At each recursive call, **map′** creates a new hollow cons cell to fill the destination. A destination pointing to the tail of the new cons cell is also created, on which **map′** is called (tail) recursively. This is really the same algorithm that you could write to implement map on a mutable list in an imperative language. Nevertheless λ_d is a pure language with only immutable types.

To obtain the regular **map** function, all is left to do is to call **new**ᵏ to create an initial destination, and **from′**ᵏ, much like when we make constructors out of filling operators, like (::) above.

2.3 Functional queues, with destinations

Implementations for a tail-recursive map are present in most previous work, from [Minamide 1998], to recent work [Bagrel 2024; Bour et al. 2021; Leijen and Lorenzen 2023]. Tail-recursive map doesn't need the full power of λ_d's first-class destinations: it just needs a notion of structures with a (single) hole. In Section 4, we will build an example which fully uses first-class destinations, but first, we will need some more material.

```
DList T ≜ (List T) ⋉ ⌊List T⌋
append : DList T ⊸ T ⊸ DList T
ys append y ≜
    upd⋉ ys with dys ↦ case (dys ◃ (::)) of
        (dy, dys′) ↦ dy ◀ y ⅋ dys′
concat : DList T ⊸ DList T ⊸ DList T
ys concat ys′ ≜ upd⋉ ys with d ↦ d ◃◃ ys′
toList : DList T ⊸ List T
toList ys ≜ from′⋉ (upd⋉ ys with d ↦ d ◃ [])
```

```
Queue T ≜ (List T)⊗(DList T)
singleton : T ⊸ Queue T
singleton x ≜ (Inr (x :: []), (new⋉ : DList T))
enqueue : Queue T ⊸ T ⊸ Queue T
q enqueue y ≜
    case q of (xs, ys) ↦ (xs, ys append y)
dequeue : Queue T ⊸ 1⊕(T⊗(Queue T))
dequeue q ≜
    case q of {
        ((x :: xs), ys) ↦ Inr (x, (xs, ys)),
        ([], ys) ↦ case (toList ys) of {
            [] ↦ Inl (),
            x :: xs ↦ Inr (x, (xs, (new⋉ : DList T)))}}
```

Fig. 2. Difference list and queue implementation in equirecursive λ_d

Difference lists. Just like in any language, iterated concatenation of lists $((xs_1 +\!\!+ xs_2) +\!\!+ \ldots) +\!\!+ xs_n$ is quadratic in λ_d. The usual solution to this is difference lists. The name difference lists covers many related implementations, but in pure functional languages, a difference list is usually represented as a function [Hughes 1986]. A singleton difference list is $\boldsymbol{\lambda} ys \mapsto x :: ys$, and concatenation of difference lists is function composition. A difference list is turned into a list by applying it to the empty list. The consequence is that no matter how many compositions we have, each cons cell will be allocated a single time, making the iterated concatenation linear indeed.

However, each concatenation allocates a closure. If we're building a difference list from singletons and composition, there's roughly one composition per cons cell, so iterated composition effectively performs two traversals of the list. In λ_d, we can do better by representing a difference list as a list with a hole. A singleton difference list is $x::\square$. Concatenation is filling the hole with another difference list, using operator $\triangleleft\!\triangleleft$. The details are on the left of Figure 2. The λ_d encoding for difference lists makes no superfluous traversal: concatenation is just an $O(1)$ in-place update.

Efficient queue using difference lists. In an immutable functional language, a queue can be implemented as a pair of lists (*front*, *back*) [Hood and Melville 1981]. *back* stores new elements in reverse order ($O(1)$ prepend). We pop elements from *front*, except when it is empty, in which case we set the queue to (**reverse** *back*, []), and pop from the new front.

For their simple implementation, Hood-Melville queues are surprisingly efficient: the cost of the reverse operation is $O(1)$ amortized for a single-threaded use of the queue. Still, it would be better to get rid of this full traversal of the back list. Taking a step back, this *back* list that has to be reversed before it is accessed is really merely a representation of a list that can be extended from the back. And we already know an efficient implementation for this: difference lists.

So we can give an improved version of the simple functional queue using destinations. This implementation is presented on the right-hand side of Figure 2. Note that contrary to an imperative programming language, we can't implement the queue as a single difference list: as mentioned earlier, our type system prevents us from reading the front elements of difference lists. Just like for the simple functional queue, we need a pair of one list that we can read from, and one that we can extend. Nevertheless this implementation of queues is both pure, as guaranteed by the λ_d type system, and nearly as efficient as what an imperative programming language would afford.

3 SCOPE ESCAPE OF DESTINATIONS

In Section 2, we've been making an implicit assumption: establishing a linear discipline on destinations ensures that all destinations will eventually find their way to the left of a fill operator ◀ or ◁, so that the associated holes get written to. This turns out to be slightly incomplete.

To see why, let's consider the type $\lfloor\lfloor \mathsf{T} \rfloor\rfloor$: the type of a destination pointing to a hole where a destination is expected. Think of it as an equivalent of the pointer type $\mathsf{T}**$ in the C language. Destinations are indeed ordinary values, so they can be stored in data structures, and before they get stored, holes stand in their place in the structure. For instance, if we have $d : \lfloor \mathsf{T} \rfloor$ and $dd : \lfloor\lfloor \mathsf{T} \rfloor\rfloor$, we can form $dd \blacktriangleleft d$: d will be stored in the structure pointed to by dd.

Should we count d as linearly used here? The alternatives don't seem promising:

- If we count this as a non-linear use of d, then $dd \blacktriangleleft d$ is rejected since destinations (represented here by d) can only be used linearly. This choice is fairly limiting, as it would prevent us from storing destinations in structures with holes, as we do, crucially, in Section 4. Nonetheless, that's the option chosen in [Bagrel 2024].
- If we do not count this use of d at all, we can write $dd \blacktriangleleft d \fatsemi d \blacktriangleleft \mathsf{v}$ so that d is both stored for later use *and* filled immediately (resulting in the corresponding hole being potentially written to twice), which is unsound, as discussed in Section 2.1.

So linear use it is. But it creates a problem: there's no way, within our linear type system, to distinguish between "a destination has been used on the left of a triangle so its corresponding hole has been filled" and "a destination has been stored and its hole still exists at the moment". This oversight may allow us to read uninitialized memory!

Let's compare two examples. We assume a simple store semantics for now where structures with holes stay in the store until they are completed. We'll need the $\mathbf{alloc} : (\lfloor \mathsf{T} \rfloor \multimap 1) \multimap \mathsf{T}$ operator. The semantics of **alloc** is: allocate a structure with a single root hole in the store, call the supplied function with the destination to the root hole as an argument; when the function has consumed all destinations (so only unit remains), pop the structure from the store to obtain a complete T.

In this snippet, structures with holes are given names v and vd in the store; holes are given names too and denoted by $\boxed{h}$ and $\boxed{hd}$, and concrete destinations are denoted by $\rightarrow h$ and $\rightarrow hd$.

When the building scope of $\mathsf{v} : \mathsf{Bool}$ is parent to the one of $\mathsf{vd} : \lfloor \mathsf{Bool} \rfloor$, everything works well because vd, that contains destination pointing to $\boxed{h}$, has to be consumed before v can be read:

$$
\begin{array}{rrl}
 & \{\} & \mid \mathbf{alloc}\ (\lambda d \mapsto (\mathbf{alloc}\ (\lambda dd \mapsto dd \blacktriangleleft d) : \lfloor \mathsf{Bool} \rfloor) \blacktriangleleft \mathsf{true}) \\
\longrightarrow & \{\mathsf{v} := \boxed{h}\} & \mid (\mathbf{alloc}\ (\lambda dd \mapsto dd \blacktriangleleft \rightarrow h) : \lfloor \mathsf{Bool} \rfloor) \blacktriangleleft \mathsf{true} \fatsemi \mathbf{deref}\ \mathsf{v} \\
\longrightarrow & \{\mathsf{v} := \boxed{h}, \mathsf{vd} := \boxed{hd}\} & \mid (\rightarrow hd \blacktriangleleft \rightarrow h \fatsemi \mathbf{deref}\ \mathsf{vd}) \blacktriangleleft \mathsf{true} \fatsemi \mathbf{deref}\ \mathsf{v} \\
\longrightarrow & \{\mathsf{v} := \boxed{h}, \mathsf{vd} := \rightarrow h\} & \mid \mathbf{deref}\ \mathsf{vd} \blacktriangleleft \mathsf{true} \fatsemi \mathbf{deref}\ \mathsf{v} \\
\longrightarrow & \{\mathsf{v} := \boxed{h}\} & \mid \rightarrow h \blacktriangleleft \mathsf{true} \fatsemi \mathbf{deref}\ \mathsf{v} \\
\longrightarrow & \{\mathsf{v} := \mathsf{true}\} & \mid \mathbf{deref}\ \mathsf{v} \\
\longrightarrow & \{\} & \mid \mathsf{true}
\end{array}
$$

However, when vd's scope is parent to v's, we can write a linearly typed yet unsound program:

$$
\begin{array}{rrl}
 & \{\} & \mid \mathbf{alloc}\ (\lambda dd \mapsto \mathbf{case}\ (\mathbf{alloc}\ (\lambda d \mapsto dd \blacktriangleleft d) : \mathsf{Bool})\ \mathbf{of}\ \{\mathsf{true} \mapsto (), \mathsf{false} \mapsto ()\}) \\
\longrightarrow & \{\mathsf{vd} := \boxed{hd}\} & \mid \mathbf{case}\ (\mathbf{alloc}\ (\lambda d \mapsto \rightarrow hd \blacktriangleleft d) : \mathsf{Bool})\ \mathbf{of}\ \{\mathsf{true} \mapsto (), \mathsf{false} \mapsto ()\} \fatsemi \mathbf{deref}\ \mathsf{vd} \\
\longrightarrow & \{\mathsf{vd} := \boxed{hd}, \mathsf{v} := \boxed{h}\} & \mid \mathbf{case}\ (\rightarrow hd \blacktriangleleft \rightarrow h \fatsemi \mathbf{deref}\ \mathsf{v})\ \mathbf{of}\ \{\mathsf{true} \mapsto (), \mathsf{false} \mapsto ()\} \fatsemi \mathbf{deref}\ \mathsf{vd} \\
\longrightarrow & \{\mathsf{vd} := \rightarrow h, \mathsf{v} := \boxed{h}\} & \mid \mathbf{case}\ (\mathbf{deref}\ \mathsf{v})\ \mathbf{of}\ \{\mathsf{true} \mapsto (), \mathsf{false} \mapsto ()\} \fatsemi \mathbf{deref}\ \mathsf{vd} \\
\longrightarrow & \{\mathsf{vd} := \rightarrow h\} & \mid \mathbf{case}\ \boxed{h}\ \mathbf{of}\ \{\mathsf{true} \mapsto (), \mathsf{false} \mapsto ()\} \fatsemi \mathbf{deref}\ \mathsf{vd}
\end{array}
$$

Here the expression $dd \blacktriangleleft d$ results in d escaping its scope for the parent one, so v is just uninitialized memory (the hole $\boxed{h}$) when we dereference it. This example must be rejected by our type system.

Again, using purely a linear type system, we can only reject this example if we also reject the first, sound example, as in [Bagrel 2024]. In this case, the type $\lfloor\lfloor \mathsf{T} \rfloor\rfloor$ becomes practically useless: such destinations can never be filled.

This isn't the direction we want to take: we really want to be able to store destinations in data structures with holes. So we want t in $d \blacktriangleleft t$ to be allowed to be linear. Without further restrictions, it wouldn't be sound, so to address this, λ_d uses a system of ages to represent scopes. Ages are described in Section 5.

4 BREADTH-FIRST TREE TRAVERSAL

As a full-fledged example, which uses the full expressive power of λ_d, we borrow and improve on an example from Bagrel [2024], breadth-first tree relabeling: *"Given a tree, create a new one of the same shape, but with the values at the nodes replaced by the numbers* $1 \ldots |T|$ *in breadth-first order."*

This isn't a very natural problem in functional programming, as breadth-first traversal implies that the order in which the structure must be built (left-to-right, top-to-bottom) is not the same as the structural order of a functional tree — building the leaves first and going up to the root. So it usually requires fancy functional workarounds [Gibbons 1993; Gibbons et al. 2023; Okasaki 2000].

It's very tempting to implement this example in an efficient imperative-like fashion, where a queue drives the processing order, thanks to the power of destinations. For that, Minamide [1998]'s system where structures with holes are represented as linear functions is not enough. Destinations as first-class values are very much required.

Figure 3 presents the λ_d implementation of the breadth-first tree traversal. The core idea is that we hold a queue of pairs, storing each input subtree with (a destination to) its corresponding output subtree. When the element (*tree*, *dtree*) at the front of the queue has been processed, the children nodes of *tree* and children destinations of *dtree* are enqueued to be processed later. There, Tree T is defined unsurprisingly as Tree T $\triangleq$ 1⊕(T⊗((Tree T)⊗(Tree T))); we refer to the constructors of Tree T as Nil and Node, defined in the obvious way. We also assume some encoding of the type Nat of natural number. Queue T is the efficient queue type from Section 2.3.

We implement the actual breadth-first relabeling **relabelDPS** as an instance of a more general breadth-first traversal function **mapAccumBFS**, which applies any state-passing style transformation of labels in breadth-first order.

In **mapAccumBFS**, we create a new destination *dtree* into which we will write the result of the traversal, then call **go**. The **go** function is in destination-passing style, but what's remarkable is that **go** takes an unbounded number of destinations as arguments, since there are as many destinations as items in the queue. This is where we use the fact that destinations are ordinary values.

The implementation of Figure 3 is very close to the one found in [Bagrel 2024]. The difference is that, because they can't store destinations in structures with holes (see the discussion in Section 3), their implementation can't use the efficient queue implementation from Section 2.3. So they have to revert to using a Hood-Melville queue for breadth-first traversal.

However this improvement comes at a cost: we a introduce *mode* system that combines linearity and age to make the system sound, hence the new fuchsia annotations in the code. We'll describe modes in detail in Section 5. In the meantime, 1 and ω control linearity: we use ω to mean that the state and function f can be used many times. On the other hand, ∞ is an *age* annotation; in particular, the associated argument cannot carry destinations. Arguments with no modes are otherwise linear and can capture destinations. We introduce the exponential modality $!_m$T to reify mode m in a type; this is useful to return several values having different modes from a function, like in f. An exponential is rarely needed in an argument position, as we have $(!_m T) \multimap U \simeq T \,{}_m\!\multimap U$.

5 TYPE SYSTEM

λ_d is a simply typed λ-calculus with unit (1), product (⊗) and sum (⊕) types. Its most salient features are the destination $\lfloor_m T\rfloor$ and ampar $S \ltimes T$ types which we've introduced in Sections 2 to 4.

```
go : (S ω∞⊸ T1 ⊸ (!ω∞S)⊗T2) ω∞⊸ S ω∞⊸ Queue (Tree T1⊗⌊Tree T2⌋) ⊸ 1
go f st q ≜ case (dequeue q) of {
                Inl () ↦ (),
                Inr ((tree, dtree), q′) ↦ case tree of {
                    Nil ↦ dtree ◃ Nil ⨟ go f st q′,
                    Node x tl tr ↦ case (dtree ◃ Node) of
                        (dy, (dtl, dtr)) ↦ case (f st x) of
                            (Mod_ω∞ st′, y) ↦
                                dy ◀ y ⨟
                                go f st′ (q′ enqueue (tl, dtl) enqueue (tr, dtr))}}
mapAccumBFS : (S ω∞⊸ T1 ⊸ (!ω∞S)⊗T2) ω∞⊸ S ω∞⊸ Tree T1 1∞⊸ Tree T2
mapAccumBFS f st tree ≜ from′_⋉ (upd_⋉ (new_⋉ : (Tree T2) ⋉ ⌊Tree T2⌋) with dtree ↦
                                    go f st (singleton (tree, dtree)))
relabelDPS : Tree 1 1∞⊸ Tree Nat
relabelDPS tree ≜ mapAccumBFS (λst ω∞↦ λun ↦ un ⨟ (Mod_ω∞ (succ st), st)) 1 tree
```

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

To ensure that destinations are used soundly, we need both to enforce the linearity of destination but also to prevent destinations from escaping their scope, as discussed in Section 3. To that effect, λ_d tracks the *age* of destinations, that is how many nested scope have been open between the current expression and the scope from which a destination originates. We'll see in Section 5.2 that scopes are introduced by $\mathbf{upd}_{\ltimes}\ t\ \mathbf{with}\ x \mapsto t'$. For instance, we have a term $\mathbf{upd}_{\ltimes}\ t_1\ \mathbf{with}\ x_1 \mapsto \mathbf{upd}_{\ltimes}\ t_2\ \mathbf{with}\ x_2 \mapsto \mathbf{upd}_{\ltimes}\ t_3\ \mathbf{with}\ x_3 \mapsto x_1$, then wil will say that the innermost occurrence of x_1 has age $\uparrow^2$ because two nested $\mathbf{upd}_{\ltimes}$ separate the definition and use site of x_1.

A natural idea, to track ages, is to introduce a modality $\uparrow T$ to mean "a T in the previous scope". Let's explore why this isn't quite going to work for us, hence why we need something more general.

A typical presentation of modal type theories is with a pair of context $\Gamma_\uparrow\ ;\ \Gamma$ of bindings from the previous scope and the current scope respectively [Pfenning and Wong 1995], and rules such as

$$\frac{}{\Gamma_\uparrow\ ;\ \Gamma, x : T \vdash x : T} \quad (1) \qquad \frac{\bullet\ ;\ \Gamma_\uparrow \vdash t : T}{\Gamma_\uparrow\ ;\ \Gamma \vdash t : \uparrow T} \quad (2)$$

The idea is that only variables from the current scope can be used to make a term for the current scope (1), and to make a term at the previous scope, you need to make it only with variables from $\Gamma_\uparrow$ (2). But this can't be the whole story here. Indeed, there's no way to refer to variables from two scopes ago, and it would be unsound for λ_d, in the manner described in Section 3, to mash all the older scope together in $\Gamma_\uparrow$. So, following this route, we'd need infinitely many contexts (and as many modalities, or more realistically a single graded modality), sequents would look like $\ldots\ ;\ \Gamma_2\ ;\ \Gamma_1\ ;\ \Gamma_0 \vdash t : T$. Finicky, but manageable perhaps. But it's not all! We need yet another context: one for bindings which are ageless because they don't contain destinations, so that we can fill destinations with harmless values like $(1, 2)$ that have been bound two or more scopes ago. More even: linear logic, famously, is also a modal logic (the modality is the exponential $!T$), so we'd need to double each of these contexts. This would be quite messy.

Fortunately, there's a way to simplify all this. First observe that having several contexts is the same as having a single context but with annotation on the bindings: instead of $x : S\ ;\ y : T\ ;\ z : U \vdash \ldots$, we can have $x :_2 S, y :_1 T, z :_0 U \vdash \ldots$ without adding or losing any information (note how the semicolons separating contexts are replaced by commas separating bindings). We'll call these annotations *modes*. We build on the key insight, which seem to originate with [Ghica and Smith

Core grammar of terms:

$$\begin{aligned} t, u \;::=\;& x \mid t'\ t \mid t \fatsemi t' \\ & \mid \mathbf{case}_m\ t\ \mathbf{of}\ \{\mathsf{Inl}\ x_1 \mapsto u_1\,,\ \mathsf{Inr}\ x_2 \mapsto u_2\} \mid \mathbf{case}_m\ t\ \mathbf{of}\ (x_1\,, x_2) \mapsto u \mid \mathbf{case}_m\ t\ \mathbf{of}\ \mathsf{Mod}_n\ x \mapsto u \\ & \mid \mathbf{upd}_{\ltimes}\ t\ \mathbf{with}\ x \mapsto t' \mid \mathbf{to}_{\ltimes}\ t \mid \mathbf{from}_{\ltimes}\ t \mid \mathbf{new}_{\ltimes} \\ & \mid t \triangleleft () \mid t \triangleleft \mathsf{Inl} \mid t \triangleleft \mathsf{Inr} \mid t \triangleleft (,) \mid t \triangleleft \mathsf{Mod}_m \mid t \triangleleft (\boldsymbol{\lambda} x_m \mapsto u) \mid t \lessdot t' \mid t \blacktriangleleft t' \end{aligned}$$

Syntactic sugar for terms:

$$\begin{aligned} \mathsf{Inl}\ t &\triangleq \mathbf{from}'_{\ltimes}\,(\mathbf{upd}_{\ltimes}\ \mathbf{new}_{\ltimes}\ \mathbf{with}\ d \mapsto d \triangleleft \mathsf{Inl} \blacktriangleleft t) \\ \mathsf{Inr}\ t &\triangleq \mathbf{from}'_{\ltimes}\,(\mathbf{upd}_{\ltimes}\ \mathbf{new}_{\ltimes}\ \mathbf{with}\ d \mapsto d \triangleleft \mathsf{Inr} \blacktriangleleft t) \\ \mathsf{Mod}_m\ t &\triangleq \mathbf{from}'_{\ltimes}\,(\mathbf{upd}_{\ltimes}\ \mathbf{new}_{\ltimes}\ \mathbf{with}\ d \mapsto d \triangleleft \mathsf{Mod}_m \blacktriangleleft t) \\ \boldsymbol{\lambda} x_m \mapsto u &\triangleq \mathbf{from}'_{\ltimes}\,(\mathbf{upd}_{\ltimes}\ \mathbf{new}_{\ltimes}\ \mathbf{with}\ d \mapsto d \triangleleft (\boldsymbol{\lambda} x_m \mapsto u)) \end{aligned}$$

$$\begin{aligned} \mathbf{from}'_{\ltimes}\ t \triangleq\ & \mathbf{case}\ (\mathbf{from}_{\ltimes}\ (\mathbf{upd}_{\ltimes}\ t\ \mathbf{with}\ un \mapsto un \fatsemi \mathsf{Mod}_{1\infty}\ ())))\ \mathbf{of} \\ & (st\,, ex) \mapsto \mathbf{case}\ ex\ \mathbf{of}\ \mathsf{Mod}_{1\infty}\ un \mapsto un \fatsemi st \\ () &\triangleq \mathbf{from}'_{\ltimes}\,(\mathbf{upd}_{\ltimes}\ \mathbf{new}_{\ltimes}\ \mathbf{with}\ d \mapsto d \triangleleft ()) \\ (t_1\,, t_2) &\triangleq \mathbf{from}'_{\ltimes}\,(\mathbf{upd}_{\ltimes}\ \mathbf{new}_{\ltimes}\ \mathbf{with}\ d \mapsto \mathbf{case}\ (d \triangleleft (,))\ \mathbf{of}\ (d_1\,, d_2) \mapsto d_1 \blacktriangleleft t_1 \fatsemi d_2 \blacktriangleleft t_2) \end{aligned}$$

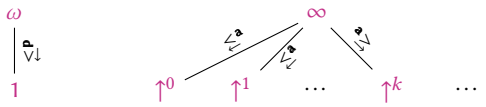
Grammar of types, modes and contexts:

$$\begin{aligned} T, U, S \;::=\;& \lfloor_n T \rfloor \quad \textit{(destination)} \\ & \mid S \ltimes T \quad \textit{(ampar)} \\ & \mid 1 \mid T_1 \oplus T_2 \mid T_1 \otimes T_2 \mid !_m T \mid T \multimap_m U \\ \Gamma \;::=\;& \cdot \mid x :_m T \mid \Gamma_1, \Gamma_2 \end{aligned}$$

$$\begin{aligned} m, n \;::=\;& \mathsf{pa} \quad \textit{(pair of multiplicity and age)} \\ p \;::=\;& 1 \mid \omega \\ a \;::=\;& \uparrow^k \mid \infty \\ \nu \triangleq \uparrow^0 \quad \uparrow \triangleq \uparrow^1 & \end{aligned}$$

Ordering on modes:

$$\mathsf{pa} \preceq \mathsf{p'a'} \iff \mathsf{p} \preceq^{\mathsf{p}} \mathsf{p'} \wedge \mathsf{a} \preceq^{\mathsf{a}} \mathsf{a'}$$



Operations on modes:

+	1	ω
1	ω	ω
ω	ω	ω

·	1	ω
1	1	ω
ω	ω	ω

$$(\mathsf{pa}) \cdot (\mathsf{p'a'}) \triangleq (\mathsf{p} \cdot \mathsf{p'})(\mathsf{a} \cdot \mathsf{a'})$$

+	$\uparrow^k$	∞
$\uparrow^j$	if $k = j$ then $\uparrow^k$ else ∞	∞
∞	∞	∞

·	$\uparrow^k$	∞
$\uparrow^j$	$\uparrow^{k+j}$	∞
∞	∞	∞

$$(\mathsf{pa}) + (\mathsf{p'a'}) \triangleq (\mathsf{p} + \mathsf{p'})(\mathsf{a} + \mathsf{a'})$$

Operations on typing contexts:

$$\begin{aligned} \mathsf{n} \cdot \cdot &\triangleq \cdot \\ \mathsf{n} \cdot (x :_m T, \Gamma) &\triangleq (x :_{\mathsf{n} \cdot m} T), \mathsf{n} \cdot \Gamma \end{aligned}$$

$$\begin{aligned} \cdot + \Gamma &\triangleq \Gamma \\ (x :_m T, \Gamma_1) + \Gamma_2 &\triangleq x :_m T, (\Gamma_1 + \Gamma_2) \quad \text{if } x \notin \Gamma_2 \\ (x :_m T, \Gamma_1) + (x :_{m'} T, \Gamma_2) &\triangleq x :_{m+m'} T, (\Gamma_1 + \Gamma_2) \end{aligned}$$

Fig. 4. Terms, types and modes of λ_d

2014], that equipping the set of modes with a particular algebraic structure is sufficient to express, algebraically, all the context manipulation that we need. They use a rig structure; for λ_d we don't need a 0 element, but we'll need a partial order on modes. The motivation for this structure in [Ghica and Smith 2014] is generating electronic circuits, the same approach has since been used, for instance, for Linear Haskell [Bernardy et al. 2018] where it is used to support mode

polymorphism. In both of those cases, this algebraic mode style is used in the context of linear types, but Abel and Bernardy [2020] show that it can generalize to a variety of modalities.

For λ_d, we're following this approach: λ_d has a single modality $!_{\mathsf{m}}$ indexed by a mode. This will greatly simplify our context management (especially in the computer-mechanized proofs), but, just as interestingly, we can define the set of modes as being the Cartesian product of the set of multiplicities (which keep track of linearity) and of the set of ages. The algebraic structure carries over by a generic theorem on products. This means that we can define multiplicities and ages independently, and the combination is taken care of for free. The syntax of λ_d terms is presented in Figure 4, including the syntactic sugar that we've been using in Sections 2 to 4.

5.1 Modes and the age semiring

The precise algebraic structure that we'll be needing on modes is both a commutative additive semigroup $+$ and a multiplicative monoid $(\cdot, 1)$, with the usual distributivity law $\mathsf{n}\cdot(\mathsf{m}_1 + \mathsf{m}_2) = \mathsf{n}\cdot\mathsf{m}_1 + \mathsf{n}\cdot\mathsf{m}_2$. In addition we require a partial order $\leq$, such that $+$ and $\cdot$ are order-preserving. In other words, we'll be dealing with ordered semirings[3]. In the rest of the article, we'll just say "semiring". In practice, all our semirings will be commutative, and we won't be paying attention to the order of factors in mode multiplication.

Our mode semiring is, as promised, the product of a multiplicity semiring, to track linearity, and an age semiring, to prevent scope escape. The multiplicity semiring has elements 1 (linear) and ω (unrestricted), it's the same semiring as in [Atkey 2018] or [Bernardy et al. 2018]. It's mostly unsurprising, the key is that that $1 + 1 = \omega$ will enforce that a linear variable can only be used once, the full description of the multiplicity semiring is given in Figure 4.

Ages are more interesting. We write ages as $\uparrow^k$ (with k a natural number), for "defined k scopes ago". We also have an age ∞ for variables that don't originate from a $\mathbf{upd}_{\ltimes}\ \mathsf{t}\ \mathbf{with}\ x \mapsto \mathsf{t}'$ i.e. that aren't destinations, and can be freely used in and returned by any scope. The main role of age ∞ is thus to act as a guarantee that a value doesn't contain destinations. Finally, we will write $\nu \triangleq \uparrow^0$ ("now") for the age of destinations that originate from the current scope; and $\uparrow \triangleq \uparrow^1$.

The operations or order aren't the usual ones on natural numbers though. It is crucial that λ_d tracks the precise age of variables. Variables from 2 scopes ago cannot be used as if they were from 1 scope ago, or vice-versa. The ordering reflects this with finite ages being arranged in a flat order, with ∞ being bigger than all of them. Multiplication of ages will reflect nesting of scope, as such, (finite) ages are multiplied by adding their numerical exponents $\uparrow^k.\uparrow^j = \uparrow^{k+j}$. In the typing rules, the most common form of scope nesting is opening a scope, which is represented by multiplying by $\uparrow$ (that is, adding 1 to the ages seen as a natural numbers). Finally $+$ is used to share a variable between two subterms, it's given by the least upper bound (for the age order above). The intuition here, is still precise age tracking: a variable must be at the same age in both subterms, or it can be ∞, and assume whichever age it needs, including different ones in different subterms.

Bindings in the context are annotated by a mode. The insight of [Ghica and Smith 2014], is that mode addition and multiplication by a mode (aka *scaling*) lift to contexts pointwise, we have all the tools we need to define a modal type system, including a sub-structural one like linear logic.

The operations and preorders on mode, contexts, etc. are presented in Figure 4. We will usually omit mode annotations when the mode is the multiplicative unit 1_ν of the semiring.

[3]There terminology dispute where some prefer to use the term "semiring" when the additive semigroup has a zero. This terminology is arguably more popular, but leaves no term for the version without a zero. We'll follow the convention, in this article, that semirings with a zero are called "rigs".

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

$$\textsc{Ty-term-Var}\ \frac{\texttt{DisposableOnly}\ \Gamma \qquad 1\nu \leq m}{\Gamma,\ x :_m T \vdash x : T} \qquad \textsc{Ty-term-App}\ \frac{\Gamma_1 \vdash t : T \qquad \Gamma_2 \vdash t' : T \multimap_m U}{m\cdot\Gamma_1 + \Gamma_2 \vdash t'\ t : U} \qquad \textsc{Ty-term-PatU}\ \frac{\Gamma_1 \vdash t : 1 \qquad \Gamma_2 \vdash u : U}{\Gamma_1 + \Gamma_2 \vdash t \fatsemi u : U}$$

$$\textsc{Ty-term-PatS}\ \frac{\Gamma_1 \vdash t : T_1 \oplus T_2 \qquad \Gamma_2,\ x_1 :_m T_1 \vdash u_1 : U \qquad \Gamma_2,\ x_2 :_m T_2 \vdash u_2 : U}{m\cdot\Gamma_1 + \Gamma_2 \vdash \mathbf{case}_m\ t\ \mathbf{of}\ \{\mathsf{Inl}\ x_1 \mapsto u_1\ ,\ \mathsf{Inr}\ x_2 \mapsto u_2\} : U}$$

$$\textsc{Ty-term-PatP}\ \frac{\Gamma_1 \vdash t : T_1 \otimes T_2 \qquad \Gamma_2,\ x_1 :_m T_1,\ x_2 :_m T_2 \vdash u : U}{m\cdot\Gamma_1 + \Gamma_2 \vdash \mathbf{case}_m\ t\ \mathbf{of}\ (x_1\ ,\ x_2) \mapsto u : U}$$

$$\textsc{Ty-term-PatE}\ \frac{\Gamma_1 \vdash t : !_n T \qquad \Gamma_2,\ x :_{m\cdot n} T \vdash u : U}{m\cdot\Gamma_1 + \Gamma_2 \vdash \mathbf{case}_m\ t\ \mathbf{of}\ \mathsf{Mod}_n\ x \mapsto u : U} \qquad \textsc{Ty-term-UpdA}\ \frac{\Gamma_1 \vdash t : U \ltimes T \qquad 1\uparrow\cdot\Gamma_2,\ x :_{1\nu} T \vdash t' : T'}{\Gamma_1 + \Gamma_2 \vdash \mathbf{upd}_{\ltimes}\ t\ \mathbf{with}\ x \mapsto t' : U \ltimes T'}$$

$$\textsc{Ty-term-ToA}\ \frac{\Gamma \vdash u : U}{\Gamma \vdash \mathbf{to}_{\ltimes}\ u : U \ltimes 1} \qquad \textsc{Ty-term-FromA}\ \frac{\Gamma \vdash t : U \ltimes (!_{1\infty} T)}{\Gamma \vdash \mathbf{from}_{\ltimes}\ t : U \otimes (!_{1\infty} T)} \qquad \textsc{Ty-term-NewA}\ \frac{\texttt{DisposableOnly}\ \Gamma}{\Gamma \vdash \mathbf{new}_{\ltimes} : T \ltimes \lfloor T \rfloor} \qquad \textsc{Ty-term-FillU}\ \frac{\Gamma \vdash t : \lfloor_n 1 \rfloor}{\Gamma \vdash t \triangleleft () : 1}$$

$$\textsc{Ty-term-FillL}\ \frac{\Gamma \vdash t : \lfloor_n T_1 \oplus T_2 \rfloor}{\Gamma \vdash t \triangleleft \mathsf{Inl} : \lfloor_n T_1 \rfloor} \qquad \textsc{Ty-term-FillR}\ \frac{\Gamma \vdash t : \lfloor_n T_1 \oplus T_2 \rfloor}{\Gamma \vdash t \triangleleft \mathsf{Inr} : \lfloor_n T_2 \rfloor} \qquad \textsc{Ty-term-FillP}\ \frac{\Gamma \vdash t : \lfloor_n T_1 \otimes T_2 \rfloor}{\Gamma \vdash t \triangleleft (,) : \lfloor_n T_1 \rfloor \otimes \lfloor_n T_2 \rfloor} \qquad \textsc{Ty-term-FillE}\ \frac{\Gamma \vdash t : \lfloor_n !_{n'} T \rfloor}{\Gamma \vdash t \triangleleft \mathsf{Mod}_{n'} : \lfloor_{n'\cdot n} T \rfloor}$$

$$\textsc{Ty-term-FillF}\ \frac{\Gamma_1 \vdash t : \lfloor_n T \multimap_m U \rfloor \qquad \Gamma_2,\ x :_m T \vdash u : U}{\Gamma_1 + (1\uparrow\cdot n)\cdot\Gamma_2 \vdash t \triangleleft (\boldsymbol{\lambda} x\ {}_m\!\!\mapsto u) : 1} \qquad \textsc{Ty-term-FillComp}\ \frac{\Gamma_1 \vdash t : \lfloor U \rfloor \qquad \Gamma_2 \vdash t' : U \ltimes T}{\Gamma_1 + 1\uparrow\cdot\Gamma_2 \vdash t \triangleleft\!\!\circ\, t' : T} \qquad \textsc{Ty-term-FillLeaf}\ \frac{\Gamma_1 \vdash t : \lfloor_n T \rfloor \qquad \Gamma_2 \vdash t' : T}{\Gamma_1 + (1\uparrow\cdot n)\cdot\Gamma_2 \vdash t \blacktriangleleft t' : 1}$$

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

$$\textsc{Ty-sterm-FromA'}\ \frac{\Gamma \vdash t : T \ltimes 1}{\Gamma \vdash \mathbf{from}'_{\ltimes}\ t : T} \qquad \textsc{Ty-sterm-Unit}\ \frac{\texttt{DisposableOnly}\ \Gamma}{\Gamma \vdash () : 1} \qquad \textsc{Ty-sterm-Fun}\ \frac{\Gamma_2,\ x :_m T \vdash u : U}{\Gamma_2 \vdash \boldsymbol{\lambda} x\ {}_m\!\!\mapsto u : T \multimap_m U} \qquad \textsc{Ty-sterm-Left}\ \frac{\Gamma_2 \vdash t : T_1}{\Gamma_2 \vdash \mathsf{Inl}\ t : T_1 \oplus T_2}$$

$$\textsc{Ty-sterm-Right}\ \frac{\Gamma_2 \vdash t : T_2}{\Gamma_2 \vdash \mathsf{Inr}\ t : T_1 \oplus T_2} \qquad \textsc{Ty-sterm-Exp}\ \frac{\Gamma_2 \vdash t : T}{m\cdot\Gamma_2 \vdash \mathsf{Mod}_m\ t : !_m T} \qquad \textsc{Ty-sterm-Prod}\ \frac{\Gamma_{21} \vdash t_1 : T_1 \qquad \Gamma_{22} \vdash t_2 : T_2}{\Gamma_{21} + \Gamma_{22} \vdash (t_1\ ,\ t_2) : T_1 \otimes T_2}$$

Fig. 5. Typing rules of λ_d

5.2 Typing rules

The typing rules for λ_d are highly inspired from[Abel and Bernardy 2020] and Linear Haskell [Bernardy et al. 2018], and are detailed in Figure 5. In particular, we use the same algebraic approach on contexts for mode tracking. Per Section 5.1, a mode is a pair of a multiplicity and an age.

Figure 5 presents the typing rules, including rules for syntactic sugar forms. We'll now walk through the few peculiarities of the type system for terms.

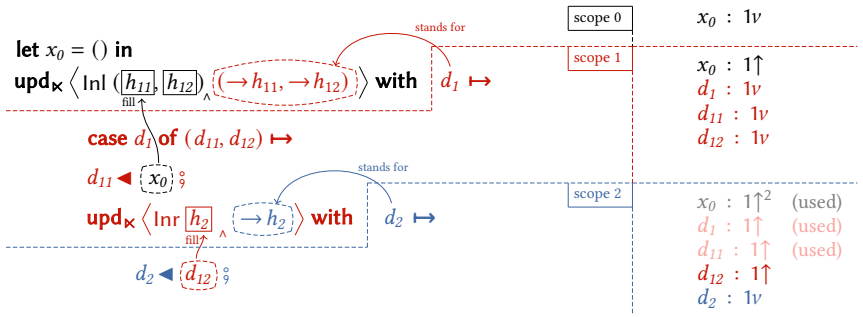


Fig. 6. Scope rules for $\mathbf{upd}_{\ltimes}$ in λ_d

Predicate `DisposableOnly` Γ in rules Ty-term-Var, Ty-term-NewA and Ty-sterm-Unit says that Γ can only contain bindings with multiplicity ω, for which weakening is allowed in linear logic. We only need weakening in these three rules, as they are the only possible leaves of the typing tree.

Rule Ty-term-Var, in addition to weakening, allows for coercion of the mode m of the variable used, with ordering constraint $1\nu \leq \mathsf{m}$ as defined in Figure 4. Notably, mode coercion still doesn't allow for a finite age to be changed to another, as $\uparrow^j$ and $\uparrow^k$ are not comparable w.r.t. $\lesssim^{\mathbf{a}}$ when $j \neq k$.

Rule Ty-term-PatU is the elimination for unit, and is also used to chain fill operations.

Pattern-matching with rules Ty-term-App, Ty-term-PatS, Ty-term-PatP and Ty-term-PatE is parametrized by a mode m by which the typing context Γ_1 of the scrutinee is multiplied. The variables which bind the subcomponents of the scrutinee then inherit this mode. In particular, this allows distributing the $!_{\mathsf{m}}$ modality over $\otimes$, which is not part of Girard's intuitionistic linear logic, but is included in [Bernardy et al. 2018] and referred to as *deep* modes in [Lorenzen et al. 2024b].

Rules for scoping. As destinations always exist in the context of a structure with holes, and must stay in that context, we need a formal notion of *scope*. Scopes are created by Ty-term-UpdA, as destinations are only ever accessed through $\mathbf{upd}_{\ltimes}$. More precisely, $\mathbf{upd}_{\ltimes}$ t **with** $x \mapsto \mathsf{t}'$ creates a new scope which spans over t'. In that scope, x has age ν (now), and the ages of the existing bindings in Γ_2 are multiplied by $\uparrow$ (i.e. we add 1 to ages seen as a numbers). That is represented by t' typing in $1\uparrow \cdot \Gamma_2$, $x :_{1\nu} \mathsf{T}$ while the parent term $\mathbf{upd}_{\ltimes}$ t **with** $x \mapsto \mathsf{t}'$ types in unscaled contexts $\Gamma_1 + \Gamma_2$. This difference of age between x — introduced by $\mathbf{upd}_{\ltimes}$, containing destinations — and Γ_2 lets us see what originates from older scopes. Specifically, distinguishing the age of destinations is crucial when typing filling primitives to avoid the pitfalls of Section 3. Figure 6 illustrates scopes introduced by $\mathbf{upd}_{\ltimes}$, and how the typing rules for $\mathbf{upd}_{\ltimes}$ and ◀ interact.

Anticipating Section 6.1, ampar values are pairs with a structure with holes on the left, and destinations on the right. With $\mathbf{upd}_{\ltimes}$ we enter a new scope where the destinations are accessible, but the structure with holes remains in the outer scope. As a result, when filling a destination with Ty-term-FillLeaf, for instance $d_{11} \blacktriangleleft x_0$ in Figure 6, we type d_{11} in the new scope, while we type x_0 in the outer scope, as it's being moved to the structure with holes on the left of the ampar, which lives in the outer scope too. This is the opposite of the scaling that $\mathbf{upd}_{\ltimes}$ does: while $\mathbf{upd}_{\ltimes}$ creates a new scope for its body, operator ◀, and similarly, ◃◦ and $\triangleleft(\boldsymbol{\lambda} x\ {}_{\mathsf{m}}\!\!\mapsto \mathsf{u})$[4], transfer their right operand to the outer scope. In other words, the right-hand side of ◀ or ◃ is an enclave for the parent scope.

When using $\mathbf{from}'_{\ltimes}$ (rule Ty-sterm-FromA'), the left of an ampar is extracted to the current scope (as seen at the bottom of Figure 6 with x_{22}): this is the fundamental reason why the left of an ampar has to "take place" in the current scope. We know the structure is complete and can be

[4]We chose the form $\triangleleft(\boldsymbol{\lambda} x\ {}_{\mathsf{m}}\!\!\mapsto \mathsf{u})$ for function creation so that any data can be built through piecemeal destination filling

extracted because the right-hand side is of type unit (1), and thus no destination on the right-hand side means no hole can remain on the left. $\mathbf{from}'_{\ltimes}$ is implemented in terms of $\mathbf{from}_{\ltimes}$ in Figure 4 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.

When an ampar is eliminated with the more general $\mathbf{from}_{\ltimes}$ in rule Ty-term-FromA however, we extract both sides of the ampar to the current scope, even though the right-hand side is normally in a different scope. This is only safe to do because the right-hand side is required to have type $!_{1\infty}\mathsf{T}$, which means it is scope-insensitive: it can't contain any scope-controlled resource. This also ensures that the right-hand side cannot contain destinations, so the structure is ready to be read.

In Ty-term-ToA, on the other hand, there is no need to bother with scopes: the operator $\mathbf{to}_{\ltimes}$ embeds an already completed structure in an ampar whose left side is the structure (that continues to type in the current scope), and right-hand side is unit.

The remaining operators $\triangleleft(), \triangleleft\mathsf{Inl}, \triangleleft\mathsf{Inr}, \triangleleft\,\mathsf{Mod}_{\mathsf{m}}, \triangleleft(,)$ from rules Ty-term-Fill$*$ are the other destination-filling primitives. They write a hollow constructor to the hole pointed by the destination operand, and return the potential new destinations that are created for new holes in the hollow constructor (or unit if there is none).

6 OPERATIONAL SEMANTICS

Before we define the operational semantics of λ_d we need to introduce a few more concepts. We'll need commands $\mathsf{E}[\mathsf{t}]$, they're described in Section 6.2; and we'll need runtime values (we'll often just say *values*), described in Figure 7. Indeed, the terms of λ_d lack any way to represent destinations or holes, or really any kind of value (for instance $\mathsf{Inl}\,()$ has been, so far, just syntactic sugar for a term $\mathbf{from}'_{\ltimes}(\mathbf{upd}_{\ltimes}\ \mathbf{new}_{\ltimes}\ \mathbf{with}\ d \mapsto \ldots)$). It's a peculiarity of λ_d that values (in particular, data constructors) only exist during the reduction; usually they are part of the term syntax of functional languages. We also extend the type system to cover commands and values, so as to be able to state and prove type safety theorems.

6.1 Runtime values and new typing context forms

The syntax of runtime values is given in Figure 7. It features constructors for all of our basic types, as well as functions (note that in $\lambda x\,{}_{\mathsf{m}}\!\mapsto \mathsf{u}$, u is a term, not a value). The more interesting values are holes $\boxed{h}$, destinations $\rightarrow h$, and ampars ${}_{H}\langle \mathsf{v}_2 \,{}_{\wedge}\, \mathsf{v}_1 \rangle$, which we'll describe in the rest of the section. In order for the operational semantics to use substitution, which requires substituting variables with values, we also extend the syntax of terms to include values through rule Ty-term-Val.

Destinations and holes are two faces of the same coin, as seen in Section 2.1, and we must ensure that throughout the reduction, a destination always points to a hole, and a hole is always the target of exactly one destination. Thus, the new idea of our system is to feature *hole bindings* $\boxed{h} :_{\mathsf{n}} \mathsf{T}$ and *destination bindings* $\rightarrow h :_{\mathsf{m}} \lfloor_{\mathsf{n}} \mathsf{T} \rfloor$ in typing contexts in addition to the usual variable bindings $x :_{\mathsf{m}} \mathsf{T}$. In both cases, we call h a *hole name*. By definition, a context Θ can contain both destination bindings and hole bindings, but *not a destination binding and a hole binding for the same hole name.*

We extend our previous context operations + and · to act on the new binding forms, as described in Figure 7. Context addition is still very partial; for instance, $(\boxed{h} :_{\mathsf{n}} \mathsf{T}) + (\rightarrow h :_{\mathsf{m}} \lfloor_{\mathsf{n}'} \mathsf{T} \rfloor)$ is not defined, as h is present on both sides but with different binding forms.

One of the main goals of λ_d is to ensure that a hole value is never read. The type system (Figure 7) maintains this invariant by simply not allowing any hole bindings in the context when typing terms (see Figure 7 for the different type of contexts used in the typing judgment). In fact, the only place where holes are introduced, is the left-hand side v_2 in an ampar ${}_{H}\langle \mathsf{v}_2 \,{}_{\wedge}\, \mathsf{v}_1 \rangle$, in Ty-val-Ampar.

Grammar extended with values:

$$t, u ::= \ldots \mid v$$

$$\begin{array}{lll} v ::= & \boxed{h} & \textit{(hole)} \\ \mid & \to h & \textit{(destination)} \\ \mid & H\langle v_2 \mathbin{\&} v_1\rangle & \textit{(ampar value)} \\ \mid & () \mid \lambda x :_m \mapsto u \mid \mathsf{Inl}\, v & \\ \mid & \mathsf{Inr}\, v \mid \mathsf{Mod}_m\, v \mid (v_1, v_2) & \end{array}$$

Typing values as terms:

$$\textsc{Ty-term-Val}\quad \frac{\mathtt{DisposableOnly}\ \Gamma \qquad \Delta \vdash^{\vee} v : T}{\Gamma, \Delta \vdash v : T}$$

Extended grammar of typing contexts:

$$\begin{array}{lllll} \Delta & ::= & \cdot \mid \to h :_m \lfloor_n T \rfloor & \mid & \Delta_1, \Delta_2 \\ \Gamma & ::= & \cdot \mid \to h :_m \lfloor_n T \rfloor \mid x :_m T & \mid & \Gamma_1, \Gamma_2 \\ \Theta & ::= & \cdot \mid \to h :_m \lfloor_n T \rfloor \mid \boxed{h} :_n T & \mid & \Theta_1, \Theta_2 \end{array}$$

Operations extended to new typing context forms:

$$n' \cdot (\boxed{h} :_n T, \Theta) \triangleq (\boxed{h} :_{n'\cdot n} T), n' \cdot \Theta$$
$$n' \cdot (\to h :_m \lfloor_n T \rfloor, \Gamma) \triangleq (\to h :_{n'\cdot m} \lfloor_n T \rfloor), n' \cdot \Gamma \quad \dagger$$

$$(\boxed{h} :_n T, \Theta_1) + \Theta_2 \triangleq \boxed{h} :_n T, (\Theta_1 + \Theta_2) \quad \text{if } h \notin \Theta_2$$
$$(\boxed{h} :_n T, \Theta_1) + (\boxed{h} :_{n'} T, \Theta_2) \triangleq \boxed{h} :_{n+n'} T, (\Theta_1 + \Theta_2)$$

$$(\to h :_m \lfloor_n T \rfloor, \Gamma_1) + \Gamma_2 \triangleq \to h :_m \lfloor_n T \rfloor, (\Gamma_1 + \Gamma_2) \quad \text{if } h \notin \Gamma_2 \quad \dagger$$
$$(\to h :_m \lfloor_n T \rfloor, \Gamma_1) + (\to h :_{m'} \lfloor_n T \rfloor, \Gamma_2) \triangleq \to h :_{m+m'} \lfloor_n T \rfloor, (\Gamma_1 + \Gamma_2) \quad \dagger$$

$$\to^{-1}(\cdot) \triangleq \cdot$$
$$\to^{-1}(\to h :_{1\nu} \lfloor_n T \rfloor, \Delta) \triangleq (\boxed{h} :_n T), \to^{-1}(\Delta)$$

$\dagger$: *same rule is also true for* Θ *or* Δ *replacing* Γ

$\boxed{\Theta \vdash^{\vee} v : T}$ *(Typing judgment for values)*

$$\textsc{Ty-val-Hole}\quad \frac{}{\boxed{h} :_{1\nu} T \vdash^{\vee} \boxed{h} : T} \qquad \textsc{Ty-val-Dest}\quad \frac{1\nu \preceq m}{\to h :_m \lfloor_n T \rfloor \vdash^{\vee} \to h : \lfloor_n T \rfloor} \qquad \textsc{Ty-val-Unit}\quad \frac{}{\cdot \vdash^{\vee} () : 1}$$

$$\textsc{Ty-val-Fun}\quad \frac{\Delta, x :_m T \vdash u : U}{\Delta \vdash^{\vee} \lambda x :_m \mapsto u : T \multimap_m U}$$

$$\textsc{Ty-val-Left}\quad \frac{\Theta \vdash^{\vee} v_1 : T_1}{\Theta \vdash^{\vee} \mathsf{Inl}\, v_1 : T_1 \oplus T_2} \qquad \textsc{Ty-val-Right}\quad \frac{\Theta \vdash^{\vee} v_2 : T_2}{\Theta \vdash^{\vee} \mathsf{Inr}\, v_2 : T_1 \oplus T_2}$$

$$\textsc{Ty-val-Prod}\quad \frac{\Theta_1 \vdash^{\vee} v_1 : T_1 \qquad \Theta_2 \vdash^{\vee} v_2 : T_2}{\Theta_1 + \Theta_2 \vdash^{\vee} (v_1, v_2) : T_1 \otimes T_2}$$

$$\textsc{Ty-val-Exp}\quad \frac{\Theta \vdash^{\vee} v' : T}{n \cdot \Theta \vdash^{\vee} \mathsf{Mod}_n\, v' : !_n T} \qquad \textsc{Ty-val-Ampar}\quad \frac{1\uparrow \cdot \Delta_1, \Delta_3 \vdash^{\vee} v_1 : T \qquad \Delta_2, \to^{-1}\Delta_3 \vdash^{\vee} v_2 : U}{\Delta_1, \Delta_2 \vdash^{\vee} \mathsf{hnames}(\Delta_3)\langle v_2 \mathbin{\&} v_1\rangle : U \ltimes T}$$

Fig. 7. Runtime values and new typing context forms

Specifically, holes come from the operator $\to^{-1}$, which represents the matching hole bindings for a set of destination bindings. It's a partial, pointwise operation on typing contexts Δ, as defined in Figure 7. Note that $\to^{-1}\Delta$ is undefined if any destination binding in Δ has a mode other than 1ν.

Furthermore, in Ty-val-Ampar, the holes $\to^{-1}\Delta_3$ and the corresponding destinations Δ_3 are bound together and consequently removed from the ampar's typing context: this is how we ensure that, indeed, there's one destination per hole and one hole per destination. That being said, both sides of the ampar may also contain stored destinations from other scopes, represented by $1\uparrow \cdot \Delta_1$ and Δ_2 in the respective typing contexts of v_1 and v_2.

Rule Ty-val-Hole indicates that a hole must have mode 1ν in typing context to be well-typed; in particular mode coercion is not allowed here, and neither is weakening. Only when a hole is behind an exponential, that mode can change to some arbitrary mode n. The mode of a hole constrains which values can be written to it, e.g. in $\boxed{h} :_n T \vdash^{\vee} \mathsf{Mod}_n\, \boxed{h} : !_n T$, only a value with mode n (more precisely, a value typed in a context of the form $n \cdot \Theta$) can be written to $\boxed{h}$.

Surprisingly, in Ty-val-Dest, we see that a destination can be typed with any mode m coercible to 1ν. We did this to mimic the rule Ty-term-Var and make the general modal substitution lemma expressible for λ_d[5]. We formally proved however that throughout a well-typed closed program, m will never be of multiplicity ω or age ∞ — a destination is always linear and of finite age — so mode coercion is never actually used; and we used this result during the formal proof of the substitution lemma to make it quite easier. The other mode n, appearing in Ty-val-Dest, is not the mode of the destination binding; instead it is part of the type $\lfloor_{\mathsf{n}}\mathsf{T}\rfloor$ and corresponds to the mode of values that we can write to the corresponding h; so for it no coercion can take place.

Other salient points. We don't distinguish values with holes from fully-defined values at the syntactic level: instead types prevent holes from being read. In particular, while values are typed in contexts Θ allowing both destination and hole bindings, when using a value as a term in Ty-term-Val, it's only allowed to have free destinations, but no free holes.

Notice, also, that values can't have free variables, since contexts Θ only contain hole and destination bindings, no variable binding. That values are closed is a standard feature of denotational semantics or abstract machine semantics. This is true even for function values (Ty-val-Fun), which, also is prevented from containing free holes. Since a function's body is unevaluated, it's unclear what it'd mean for a function to contain holes; at the very least it'd complicate our system a lot, and we are unaware of any benefit supporting free holes in functions could bring.

One might wonder how we can represent a curried function $\boldsymbol{\lambda} x \mapsto \boldsymbol{\lambda} y \mapsto x \ \mathbf{concat}\ y$ at 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 $\overset{\nu}{\boldsymbol{\lambda}} x \mapsto \mathbf{from}'_{\ltimes}\,(\mathbf{upd}_{\ltimes}\ \mathbf{new}_{\ltimes}\ \mathbf{with}\ d \mapsto d \triangleleft (\boldsymbol{\lambda} y \mapsto x \ \mathbf{concat}\ y))$, where the inner closure is not yet in value form. As the form $d \triangleleft (\boldsymbol{\lambda} y \mapsto x \ \mathbf{concat}\ y)$ is part of term syntax, it's allowed to have free variable x.

6.2 Evaluation contexts and commands

The semantics of λ_d is given using small-step reductions on a pair $\mathsf{E}\lceil \mathsf{t} \rceil$ of an evaluation context E, and an (extended) term t under focus. We call such a pair $\mathsf{E}\lceil \mathsf{t} \rceil$ a *command*, borrowing the terminology from Curien and Herbelin [2000].

The grammar of evaluation contexts is given in Figure 8. An evaluation context E is the composition of an arbitrary number of focusing components e. We chose to represent evaluation contexts syntactically, taking inspiration from Felleisen [1987] and subsequent [Biernacka and Danvy 2007; Danvy and Nielsen 2004]. The intuition here is that destination filling only require a very tame notion of state. So tame, in fact, that we can simply represent writing to a hole by a substitution in the evaluation context, instead of using more heavy store semantics. With this choice, focusing and defocusing steps are made explicit in the semantics, resulting in a verbose but simpler proof. It is also easier to derive an abstract machine for the language, should one want to do that.

Consequently, $\mathsf{E}\lceil \mathsf{t} \rceil$ is formally a pair (although we use the notation usually reserved for one-hole contexts, to make rules look more familiar). It's important to keep in mind that won't always have a corresponding term (for instance, when E contains open ampar focusing components).

Focusing components are all directly derived from the term syntax, except for the *open ampar* component $\overset{\mathsf{op}}{H}\langle \mathsf{v}_2 \wedge [] \rangle$. This focusing component indicates that an ampar is currently being processed by $\mathbf{upd}_{\ltimes}$, with its left-hand side v_2 (the structure being built) being attached to the open ampar focusing component, while its right-hand side (containing destinations) is either in subsequent focusing components, or in the term under focus. Ampars being open during the evaluation of $\mathbf{upd}_{\ltimes}$'s body and closed back afterwards is counterpart to the notion of scopes in typing rules.

[5] Generally, in modal systems, if $x :_{\mathsf{m}} \mathsf{T}$, $\Gamma \vdash \mathsf{u} : \mathsf{U}$ and $\Delta \vdash \mathsf{v} : \mathsf{T}$ then $\mathsf{m} \cdot \Delta$, $\Gamma \vdash \mathsf{u}[x := \mathsf{v}] : \mathsf{U}$ [Abel and Bernardy 2020]. We have $x :_{\omega\infty} \lfloor_{\mathsf{n}}\mathsf{T}\rfloor \vdash () : 1$ and $\to h :_{1\nu} \lfloor_{\mathsf{n}}\mathsf{T}\rfloor \vdash \to h : \lfloor_{\mathsf{n}}\mathsf{T}\rfloor$ so $\omega\infty \cdot (\to h :_{1\nu} \lfloor_{\mathsf{n}}\mathsf{T}\rfloor) \vdash ()[x := \to h] : 1$ should be valid.

Grammar of evaluation contexts:

$$
\begin{array}{rcl}
\mathsf{e} & ::= & \mathsf{t}'\,[\,] \mid [\,]\,\mathsf{v} \mid [\,]\fatsemi \mathsf{u} \\
& \mid & \mathbf{case}_{\mathsf{m}}\,[\,]\,\mathbf{of}\,\{\mathsf{Inl}\,x_1 \mapsto \mathsf{u}_1\,,\ \mathsf{Inr}\,x_2 \mapsto \mathsf{u}_2\} \mid \mathbf{case}_{\mathsf{m}}\,[\,]\,\mathbf{of}\,(x_1\,,x_2) \mapsto \mathsf{u} \mid \mathbf{case}_{\mathsf{m}}\,[\,]\,\mathbf{of}\,\mathsf{Mod}_{\mathsf{n}}\,x \mapsto \mathsf{u} \\
& \mid & \mathbf{upd}_{\ltimes}\,[\,]\,\mathbf{with}\,x \mapsto \mathsf{t}' \mid \mathbf{to}_{\ltimes}\,[\,] \mid \mathbf{from}_{\ltimes}\,[\,] \mid [\,] \lhd\!\!\circ\, \mathsf{t}' \mid \mathsf{v} \lhd\!\!\circ [\,] \mid [\,] \blacktriangleleft \mathsf{t}' \mid \mathsf{v} \blacktriangleleft [\,] \\
& \mid & [\,] \lhd () \mid [\,] \lhd \mathsf{Inl} \mid [\,] \lhd \mathsf{Inr} \mid [\,] \lhd (,) \mid [\,] \lhd \mathsf{Mod}_{\mathsf{m}} \mid [\,] \lhd (\lambda x\,{}_{\mathsf{m}}\!\mapsto \mathsf{u}) \\
& \mid & \overset{\mathsf{op}}{H}\langle \mathsf{v}_2 \,{}_{\wedge}[\,]\rangle \qquad \textit{(open ampar focusing component)} \\
\mathsf{E} & ::= & [\,] \mid \mathsf{E} \circ \mathsf{e}
\end{array}
$$

$\boxed{\Delta \dashv \mathsf{E} : \mathsf{T} \rightarrowtail \mathsf{U}_0}$ *(Typing judgment for evaluation contexts)*

Ty-ectxs-Id

$$\frac{}{\cdot \dashv [\,] : \mathsf{U}_0 \rightarrowtail \mathsf{U}_0}$$

Ty-ectxs-App1

$$\frac{\mathsf{m}\cdot\Delta_1,\ \Delta_2 \dashv \mathsf{E} : \mathsf{U} \rightarrowtail \mathsf{U}_0 \qquad \Delta_2 \vdash \mathsf{t}' : \mathsf{T}\ {}_{\mathsf{m}}\!\multimap \mathsf{U}}{\Delta_1 \dashv \mathsf{E} \circ \mathsf{t}'\,[\,] : \mathsf{T} \rightarrowtail \mathsf{U}_0}$$

Ty-ectxs-App2

$$\frac{\mathsf{m}\cdot\Delta_1,\ \Delta_2 \dashv \mathsf{E} : \mathsf{U} \rightarrowtail \mathsf{U}_0 \qquad \Delta_1 \vdash \mathsf{v} : \mathsf{T}}{\Delta_2 \dashv \mathsf{E} \circ [\,]\,\mathsf{v} : (\mathsf{T}\ {}_{\mathsf{m}}\!\multimap \mathsf{U}) \rightarrowtail \mathsf{U}_0}$$

Ty-ectxs-PatU

$$\frac{\Delta_1,\ \Delta_2 \dashv \mathsf{E} : \mathsf{U} \rightarrowtail \mathsf{U}_0 \qquad \Delta_2 \vdash \mathsf{u} : \mathsf{U}}{\Delta_1 \dashv \mathsf{E} \circ [\,] \fatsemi \mathsf{u} : 1 \rightarrowtail \mathsf{U}_0}$$

Ty-ectxs-PatS

$$\frac{\begin{array}{c}\mathsf{m}\cdot\Delta_1,\ \Delta_2 \dashv \mathsf{E} : \mathsf{U} \rightarrowtail \mathsf{U}_0 \\ \Delta_2,\ x_1 :_{\mathsf{m}} \mathsf{T}_1 \vdash \mathsf{u}_1 : \mathsf{U} \\ \Delta_2,\ x_2 :_{\mathsf{m}} \mathsf{T}_2 \vdash \mathsf{u}_2 : \mathsf{U}\end{array}}{\Delta_1 \dashv \mathsf{E} \circ \mathbf{case}_{\mathsf{m}}\,[\,]\,\mathbf{of}\,\{\mathsf{Inl}\,x_1 \mapsto \mathsf{u}_1\,,\ \mathsf{Inr}\,x_2 \mapsto \mathsf{u}_2\} : (\mathsf{T}_1 \oplus \mathsf{T}_2) \rightarrowtail \mathsf{U}_0}$$

Ty-ectxs-PatP

$$\frac{\mathsf{m}\cdot\Delta_1,\ \Delta_2 \dashv \mathsf{E} : \mathsf{U} \rightarrowtail \mathsf{U}_0 \qquad \Delta_2,\ x_1 :_{\mathsf{m}} \mathsf{T}_1,\ x_2 :_{\mathsf{m}} \mathsf{T}_2 \vdash \mathsf{u} : \mathsf{U}}{\Delta_1 \dashv \mathsf{E} \circ \mathbf{case}_{\mathsf{m}}\,[\,]\,\mathbf{of}\,(x_1\,,x_2) \mapsto \mathsf{u} : (\mathsf{T}_1 \otimes \mathsf{T}_2) \rightarrowtail \mathsf{U}_0}$$

Ty-ectxs-PatE

$$\frac{\mathsf{m}\cdot\Delta_1,\ \Delta_2 \dashv \mathsf{E} : \mathsf{U} \rightarrowtail \mathsf{U}_0 \qquad \Delta_2,\ x :_{\mathsf{m}\cdot\mathsf{m}'} \mathsf{T} \vdash \mathsf{u} : \mathsf{U}}{\Delta_1 \dashv \mathsf{E} \circ \mathbf{case}_{\mathsf{m}}\,[\,]\,\mathbf{of}\,\mathsf{Mod}_{\mathsf{m}'}\,x \mapsto \mathsf{u} : !_{\mathsf{m}'}\mathsf{T} \rightarrowtail \mathsf{U}_0}$$

Ty-ectxs-UpdA

$$\frac{\Delta_1,\ \Delta_2 \dashv \mathsf{E} : \mathsf{U} \ltimes \mathsf{T}' \rightarrowtail \mathsf{U}_0 \qquad 1\!\uparrow\cdot\Delta_2,\ x :_{1\nu} \mathsf{T} \vdash \mathsf{t}' : \mathsf{T}'}{\Delta_1 \dashv \mathsf{E} \circ \mathbf{upd}_{\ltimes}\,[\,]\,\mathbf{with}\,x \mapsto \mathsf{t}' : (\mathsf{U} \ltimes \mathsf{T}) \rightarrowtail \mathsf{U}_0}$$

Ty-ectxs-ToA

$$\frac{\Delta \dashv \mathsf{E} : (\mathsf{U} \ltimes 1) \rightarrowtail \mathsf{U}_0}{\Delta \dashv \mathsf{E} \circ \mathbf{to}_{\ltimes}\,[\,] : \mathsf{U} \rightarrowtail \mathsf{U}_0}$$

Ty-ectxs-FromA

$$\frac{\Delta \dashv \mathsf{E} : (\mathsf{U} \otimes (!_{1\infty}\mathsf{T})) \rightarrowtail \mathsf{U}_0}{\Delta \dashv \mathsf{E} \circ \mathbf{from}_{\ltimes}\,[\,] : (\mathsf{U} \ltimes (!_{1\infty}\mathsf{T})) \rightarrowtail \mathsf{U}_0}$$

Ty-ectxs-FillU

$$\frac{\Delta \dashv \mathsf{E} : 1 \rightarrowtail \mathsf{U}_0}{\Delta \dashv \mathsf{E} \circ [\,] \lhd () : \lfloor_{\mathsf{n}} 1 \rfloor \rightarrowtail \mathsf{U}_0}$$

Ty-ectxs-FillL

$$\frac{\Delta \dashv \mathsf{E} : \lfloor_{\mathsf{n}}\mathsf{T}_1 \rfloor \rightarrowtail \mathsf{U}_0}{\Delta \dashv \mathsf{E} \circ [\,] \lhd \mathsf{Inl} : \lfloor_{\mathsf{n}}\mathsf{T}_1 \oplus \mathsf{T}_2 \rfloor \rightarrowtail \mathsf{U}_0}$$

Ty-ectxs-FillR

$$\frac{\Delta \dashv \mathsf{E} : \lfloor_{\mathsf{n}}\mathsf{T}_2 \rfloor \rightarrowtail \mathsf{U}_0}{\Delta \dashv \mathsf{E} \circ [\,] \lhd \mathsf{Inr} : \lfloor_{\mathsf{n}}\mathsf{T}_1 \oplus \mathsf{T}_2 \rfloor \rightarrowtail \mathsf{U}_0}$$

Ty-ectxs-FillP

$$\frac{\Delta \dashv \mathsf{E} : (\lfloor_{\mathsf{n}}\mathsf{T}_1 \rfloor \otimes \lfloor_{\mathsf{n}}\mathsf{T}_2 \rfloor) \rightarrowtail \mathsf{U}_0}{\Delta \dashv \mathsf{E} \circ [\,] \lhd (,) : \lfloor_{\mathsf{n}}\mathsf{T}_1 \otimes \mathsf{T}_2 \rfloor \rightarrowtail \mathsf{U}_0}$$

Ty-ectxs-FillE

$$\frac{\Delta \dashv \mathsf{E} : \lfloor_{\mathsf{m}\cdot\mathsf{n}}\mathsf{T} \rfloor \rightarrowtail \mathsf{U}_0}{\Delta \dashv \mathsf{E} \circ [\,] \lhd \mathsf{Mod}_{\mathsf{m}} : \lfloor_{\mathsf{n}} !_{\mathsf{m}}\mathsf{T} \rfloor \rightarrowtail \mathsf{U}_0}$$

Ty-ectxs-FillF

$$\frac{\Delta_1,\ (1\!\uparrow\cdot\mathsf{n})\cdot\Delta_2 \dashv \mathsf{E} : 1 \rightarrowtail \mathsf{U}_0 \qquad \Delta_2,\ x :_{\mathsf{m}} \mathsf{T} \vdash \mathsf{u} : \mathsf{U}}{\Delta_1 \dashv \mathsf{E} \circ [\,] \lhd (\lambda x\,{}_{\mathsf{m}}\!\mapsto \mathsf{u}) : \lfloor_{\mathsf{n}} \mathsf{T}\ {}_{\mathsf{m}}\!\multimap \mathsf{U} \rfloor \rightarrowtail \mathsf{U}_0}$$

Ty-ectxs-FillComp1

$$\frac{\Delta_1,\ 1\!\uparrow\cdot\Delta_2 \dashv \mathsf{E} : \mathsf{T} \rightarrowtail \mathsf{U}_0 \qquad \Delta_2 \vdash \mathsf{t}' : \mathsf{U} \ltimes \mathsf{T}}{\Delta_1 \dashv \mathsf{E} \circ [\,] \lhd\!\!\circ\, \mathsf{t}' : \lfloor \mathsf{U} \rfloor \rightarrowtail \mathsf{U}_0}$$

Ty-ectxs-FillComp2

$$\frac{\Delta_1,\ 1\!\uparrow\cdot\Delta_2 \dashv \mathsf{E} : \mathsf{T} \rightarrowtail \mathsf{U}_0 \qquad \Delta_1 \vdash \mathsf{v} : \lfloor \mathsf{U} \rfloor}{\Delta_2 \dashv \mathsf{E} \circ \mathsf{v} \lhd\!\!\circ [\,] : \mathsf{U} \ltimes \mathsf{T} \rightarrowtail \mathsf{U}_0}$$

Ty-ectxs-FillLeaf1

$$\frac{\Delta_1,\ (1\!\uparrow\cdot\mathsf{n})\cdot\Delta_2 \dashv \mathsf{E} : 1 \rightarrowtail \mathsf{U}_0 \qquad \Delta_2 \vdash \mathsf{t}' : \mathsf{T}}{\Delta_1 \dashv \mathsf{E} \circ [\,] \blacktriangleleft \mathsf{t}' : \lfloor_{\mathsf{n}}\mathsf{T} \rfloor \rightarrowtail \mathsf{U}_0}$$

Ty-ectxs-FillLeaf2

$$\frac{\Delta_1,\ (1\!\uparrow\cdot\mathsf{n})\cdot\Delta_2 \dashv \mathsf{E} : 1 \rightarrowtail \mathsf{U}_0 \qquad \Delta_1 \vdash \mathsf{v} : \lfloor_{\mathsf{n}}\mathsf{T} \rfloor}{\Delta_2 \dashv \mathsf{E} \circ \mathsf{v} \blacktriangleleft [\,] : \mathsf{T} \rightarrowtail \mathsf{U}_0}$$

Ty-ectxs-OpenAmpar

$$\frac{\begin{array}{c}\mathbf{hnames}(\mathsf{E})\ \#\#\ \mathbf{hnames}(\Delta_3) \\ \Delta_1,\ \Delta_2 \dashv \mathsf{E} : (\mathsf{U} \ltimes \mathsf{T}') \rightarrowtail \mathsf{U}_0 \\ \Delta_2,\ \rightarrow^{-1}\!\Delta_3 \ \overset{\nu}{\vdash}\ \mathsf{v}_2 : \mathsf{U}\end{array}}{1\!\uparrow\cdot\Delta_1,\ \Delta_3 \dashv \mathsf{E} \circ \overset{\mathsf{op}}{\mathbf{hnames}(\Delta_3)}\langle \mathsf{v}_2 \,{}_{\wedge}[\,]\rangle : \mathsf{T}' \rightarrowtail \mathsf{U}_0}$$

Fig. 8. Evaluation contexts and their typing rules

Evaluation contexts are typed in a context Δ that can only contains destination bindings. As we will later see in rule Ty-cmd of Figure 9, Δ is exactly the typing context that the term t has to use to form a valid $\mathsf{E}\left[\mathsf{t}\right]$. In other words, while $\Gamma \vdash \mathsf{t} : \mathsf{T}$ *requires* the bindings of Γ, judgment $\Delta \dashv \mathsf{E} : \mathsf{T} \rightarrowtail \mathsf{U}_0$ *provides* the bindings of Δ. Typing rules for evaluation contexts are given in Figure 8.

An evaluation context has a context type $\mathsf{T} \rightarrowtail \mathsf{U}_0$. The meaning of $\mathsf{E} : \mathsf{T} \rightarrowtail \mathsf{U}_0$ is that given $\mathsf{t} : \mathsf{T}$, $\mathsf{E}\left[\mathsf{t}\right]$ returns a value of type U_0. Composing an evaluation context $\mathsf{E} : \mathsf{T} \rightarrowtail \mathsf{U}_0$ with a new focusing component never affects the type U_0 of the future command; only the type T of the focus is altered.

All typing rules for evaluation contexts can be derived systematically from the ones for the corresponding term (except for the rule Ty-ectxs-OpenAmpar that is a truly new form). Let's take the rule Ty-ectxs-PatP as an example:

$$\textsc{Ty-term-PatP}\ \frac{\Gamma_1 \vdash \mathsf{t} : \mathsf{T}_1 \otimes \mathsf{T}_2 \qquad \Gamma_2,\ x_1 :_{\mathsf{m}} \mathsf{T}_1,\ x_2 :_{\mathsf{m}} \mathsf{T}_2 \vdash \mathsf{u} : \mathsf{U}}{\mathsf{m} \cdot \Gamma_1 + \Gamma_2 \vdash \mathbf{case}_{\mathsf{m}}\ \mathsf{t}\ \mathbf{of}\ (x_1, x_2) \mapsto \mathsf{u} : \mathsf{U}}$$

$$\textsc{Ty-ectxs-PatP}\ \frac{\mathsf{m} \cdot \Delta_1,\ \Delta_2 \dashv \mathsf{E} : \mathsf{U} \rightarrowtail \mathsf{U}_0 \qquad \Delta_2,\ x_1 :_{\mathsf{m}} \mathsf{T}_1,\ x_2 :_{\mathsf{m}} \mathsf{T}_2 \vdash \mathsf{u} : \mathsf{U}}{\Delta_1 \dashv \mathsf{E} \circ \mathbf{case}_{\mathsf{m}}\ []\ \mathbf{of}\ (x_1, x_2) \mapsto \mathsf{u} : (\mathsf{T}_1 \otimes \mathsf{T}_2) \rightarrowtail \mathsf{U}_0}$$

- the typing context $\mathsf{m} \cdot \Delta_1,\ \Delta_2$ in the premise for E corresponds to $\mathsf{m} \cdot \Gamma_1 + \Gamma_2$ in the conclusion of Ty-term-PatP;
- the typing context $\Delta_2,\ x_1 :_{\mathsf{m}} \mathsf{T}_1,\ x_2 :_{\mathsf{m}} \mathsf{T}_2$ in the premise for term u corresponds to the typing context $\Gamma_2,\ x_1 :_{\mathsf{m}} \mathsf{T}_1,\ x_2 :_{\mathsf{m}} \mathsf{T}_2$ for the same term in Ty-term-PatP;
- the typing context Δ_1 in the conclusion for $\mathsf{E} \circ \mathbf{case}_{\mathsf{m}}\ []\ \mathbf{of}\ (x_1, x_2) \mapsto \mathsf{u}$ corresponds to the typing context Γ_1 in the premise for t in Ty-term-PatP (the term t is located where the focus $[]$ is in Ty-ectxs-OpenAmpar).

We think of the typing rule for an evaluation context as a rotation of the typing rule for the associated term, where the typing contexts of one subterm and the conclusion are swapped, and the typing contexts of the other potential subterms are kept unchanged (with the difference that typing contexts for evaluation contexts are of shape Δ instead of Γ).

6.3 Small-step semantics

We equip λ_d with small-step semantics. There are three sorts of semantic rules:

- focus rules, where we focus on a subterm of a term, by pushing a corresponding focusing component on the stack E;
- unfocus rules, where the term under focus is in fact a value, and thus we pop a focusing component from the stack E and transform it back to the corresponding 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 the focus, unfocus, and reduction rules for PatP:

$$\mathsf{E}\left[\mathbf{case}_{\mathsf{m}}\ \mathsf{t}\ \mathbf{of}\ (x_1, x_2) \mapsto \mathsf{u}\right] \longrightarrow (\mathsf{E} \circ \mathbf{case}_{\mathsf{m}}\ []\ \mathbf{of}\ (x_1, x_2) \mapsto \mathsf{u})\left[\mathsf{t}\right] \quad \textit{when} \quad \texttt{NotVal}\ \mathsf{t}$$

$$(\mathsf{E} \circ \mathbf{case}_{\mathsf{m}}\ []\ \mathbf{of}\ (x_1, x_2) \mapsto \mathsf{u})\left[\mathsf{v}\right] \longrightarrow \mathsf{E}\left[\mathbf{case}_{\mathsf{m}}\ \mathsf{v}\ \mathbf{of}\ (x_1, x_2) \mapsto \mathsf{u}\right]$$

$$\mathsf{E}\left[\mathbf{case}_{\mathsf{m}}\ (\mathsf{v}_1, \mathsf{v}_2)\ \mathbf{of}\ (x_1, x_2) \mapsto \mathsf{u}\right] \longrightarrow \mathsf{E}\left[\mathsf{u}[x_1 := \mathsf{v}_1][x_2 := \mathsf{v}_2]\right]$$

Rules are triggered in a purely deterministic fashion; once a subterm is a value, it cannot be focused on again. As focusing and defocusing rules are entirely mechanical (they are just a matter of pushing and popping a focusing component on the stack), we only present the set of reduction rules for the system in Figure 9, but the whole system is included in the annex (Figures 10 and 11).

Reduction rules for function application, pattern-matching, $\mathbf{to}_{\ltimes}$ and $\mathbf{from}_{\ltimes}$ are straightforward.

We introduce a special substitution $\mathsf{E}\langle\!|h :=_H \mathsf{v}|\!\rangle$ that is used to update structures under construction, that are attached to open ampar focusing components in the stack. Such a substitution is

Special substitution for open ampars:

$$\begin{array}{rcl}(\mathsf{E} \circ {}^{\mathsf{op}}_{\{h\}\sqcup H}\langle \mathsf{v}_2 \wedge [\,]\rangle)\,(\!|h{:=}_{H'}\ \mathsf{v}'|\!) & = & \mathsf{E} \circ {}^{\mathsf{op}}_{H\sqcup H'}\langle \mathsf{v}_2\,(\!|h{:=}_{H'}\ \mathsf{v}'|\!) \wedge [\,]\rangle \\ (\mathsf{E} \circ \mathsf{e})\,(\!|h{:=}_{H'}\ \mathsf{v}'|\!) & = & \mathsf{E}\,(\!|h{:=}_{H'}\ \mathsf{v}'|\!) \circ \mathsf{e} \qquad \text{if } h \notin \mathsf{e}\end{array}$$

Name set shift and conditional name shift:

$$\begin{array}{rcl} H \dot{+} h' & \triangleq & \{h \dot{+} h' \mid h \in H\} \\ h[H \dot{+} h'] & \triangleq & \begin{cases} h \dot{+} h' & \text{if } h \in H \\ h & \text{otherwise} \end{cases}\end{array}$$

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

$$\textsc{Ty-cmd}\ \frac{\Delta \dashv \mathsf{E} : \mathsf{T} \rightarrowtail \mathsf{U}_0 \qquad \Delta \vdash \mathsf{t} : \mathsf{T}}{\vdash \mathsf{E}[\,\mathsf{t}\,] : \mathsf{U}_0}$$

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

$$\begin{array}{lll}
\mathsf{E}[\,(\boldsymbol{\lambda} x\ {}_{\mathsf{m}}\!\!\mapsto \mathsf{u})\ \mathsf{v}\,] \longrightarrow \mathsf{E}[\,\mathsf{u}[x := \mathsf{v}]\,] & & \textsc{App-Red} \\
\mathsf{E}[\,() \fatsemi \mathsf{u}\,] \longrightarrow \mathsf{E}[\,\mathsf{u}\,] & & \textsc{PatU-Red} \\
\mathsf{E}[\,\mathbf{case}_{\mathsf{m}}\ (\mathsf{Inl}\ \mathsf{v}_1)\ \mathbf{of}\ \{\mathsf{Inl}\ x_1 \mapsto \mathsf{u}_1\,,\ \mathsf{Inr}\ x_2 \mapsto \mathsf{u}_2\}\,] \longrightarrow \mathsf{E}[\,\mathsf{u}_1[x_1 := \mathsf{v}_1]\,] & & \textsc{PatL-Red} \\
\mathsf{E}[\,\mathbf{case}_{\mathsf{m}}\ (\mathsf{Inr}\ \mathsf{v}_2)\ \mathbf{of}\ \{\mathsf{Inl}\ x_1 \mapsto \mathsf{u}_1\,,\ \mathsf{Inr}\ x_2 \mapsto \mathsf{u}_2\}\,] \longrightarrow \mathsf{E}[\,\mathsf{u}_2[x_2 := \mathsf{v}_2]\,] & & \textsc{PatR-Red} \\
\mathsf{E}[\,\mathbf{case}_{\mathsf{m}}\ (\mathsf{v}_1\,,\ \mathsf{v}_2)\ \mathbf{of}\ (x_1\,,\ x_2) \mapsto \mathsf{u}\,] \longrightarrow \mathsf{E}[\,\mathsf{u}[x_1 := \mathsf{v}_1][x_2 := \mathsf{v}_2]\,] & & \textsc{PatP-Red} \\
\mathsf{E}[\,\mathbf{case}_{\mathsf{m}}\ \mathsf{Mod}_{\mathsf{n}}\ \mathsf{v}'\ \mathbf{of}\ \mathsf{Mod}_{\mathsf{n}}\ x \mapsto \mathsf{u}\,] \longrightarrow \mathsf{E}[\,\mathsf{u}[x := \mathsf{v}']\,] & & \textsc{PatE-Red} \\
\mathsf{E}[\,\mathbf{to}_{\bowtie}\ \mathsf{v}_2\,] \longrightarrow \mathsf{E}[\,{}_{\{\,\}}\langle \mathsf{v}_2 \wedge ()\rangle\,] & & \textsc{ToA-Red} \\
\mathsf{E}[\,\mathbf{from}_{\bowtie}\ {}_{\{\,\}}\langle \mathsf{v}_2 \wedge \mathsf{Mod}_{1\infty}\ \mathsf{v}_1\rangle\,] \longrightarrow \mathsf{E}[\,(\mathsf{v}_2\,,\ \mathsf{Mod}_{1\infty}\ \mathsf{v}_1)\,] & & \textsc{FromA-Red} \\
\mathsf{E}[\,\mathbf{new}_{\bowtie}\,] \longrightarrow \mathsf{E}[\,{}_{\{1\}}\langle \boxed{1}\, \wedge \rightarrow 1\rangle\,] & & \textsc{NewA-Red} \\
\mathsf{E}[\,\rightarrow h \triangleleft ()\,] \longrightarrow \mathsf{E}\,(\!|h{:=}_{\{\,\}}\ ()|\!)\,[\,()\,] & & \textsc{FillU-Red} \\
\mathsf{E}[\,\rightarrow h \triangleleft (\boldsymbol{\lambda} x\ {}_{\mathsf{m}}\!\!\mapsto \mathsf{u})\,] \longrightarrow \mathsf{E}\,(\!|h{:=}_{\{\,\}}\ \boldsymbol{\lambda} x\ {}_{\mathsf{m}}\!\!\mapsto \mathsf{u}|\!)\,[\,()\,] & & \textsc{FillF-Red} \\
\mathsf{E}[\,\rightarrow h \triangleleft \mathsf{Inl}\,] \longrightarrow \mathsf{E}\,(\!|h{:=}_{\{h'+1\}}\ \mathsf{Inl}\ \boxed{h'+1}|\!)\,[\,\rightarrow h'+1\,] & & \textsc{FillL-Red} \\
\mathsf{E}[\,\rightarrow h \triangleleft \mathsf{Inr}\,] \longrightarrow \mathsf{E}\,(\!|h{:=}_{\{h'+1\}}\ \mathsf{Inr}\ \boxed{h'+1}|\!)\,[\,\rightarrow h'+1\,] & & \textsc{FillR-Red} \\
\mathsf{E}[\,\rightarrow h \triangleleft \mathsf{Mod}_{\mathsf{m}}\,] \longrightarrow \mathsf{E}\,(\!|h{:=}_{\{h'+1\}}\ \mathsf{Mod}_{\mathsf{m}}\ \boxed{h'+1}|\!)\,[\,\rightarrow h'+1\,] & & \textsc{FillE-Red} \\
\mathsf{E}[\,\rightarrow h \triangleleft (,)\,] \longrightarrow \mathsf{E}\,(\!|h{:=}_{\{h'+1,h'+2\}}\ (\boxed{h'+1}\,,\ \boxed{h'+2})|\!)\,[\,(\rightarrow h'+1\,,\ \rightarrow h'+2)\,] & & \textsc{FillP-Red} \\
\mathsf{E}[\,\rightarrow h \mathrel{\text{-}\!\!\circ}_H \langle \mathsf{v}_2 \wedge \mathsf{v}_1\rangle\,] \longrightarrow \mathsf{E}\,(\!|h{:=}_{(H \dot{+} h'')}\ \mathsf{v}_2[H \dot{+} h'']|\!)\,[\,\mathsf{v}_1[H \dot{+} h'']\,] & & \textsc{FillComp-Red} \\
\mathsf{E}[\,\rightarrow h \blacktriangleleft \mathsf{v}\,] \longrightarrow \mathsf{E}\,(\!|h{:=}_{\{\,\}}\ \mathsf{v}|\!)\,[\,()\,] & & \textsc{FillLeaf-Red} \\
\mathsf{E}[\,\mathbf{upd}_{\bowtie}\ H\langle \mathsf{v}_2 \wedge \mathsf{v}_1\rangle\ \mathbf{with}\ x \mapsto \mathsf{t}'\,] \longrightarrow (\mathsf{E} \circ {}^{\mathsf{op}}_{H \dot{+} h'''}\langle \mathsf{v}_2[H \dot{+} h'''] \wedge [\,]\rangle)\,[\,\mathsf{t}'[x := \mathsf{v}_1[H \dot{+} h''']]\,] & & \textsc{Ampar-Open} \\
(\mathsf{E} \circ {}^{\mathsf{op}}_{H}\langle \mathsf{v}_2 \wedge [\,]\rangle)\,[\,\mathsf{v}_1\,] \longrightarrow \mathsf{E}[\,H\langle \mathsf{v}_2 \wedge \mathsf{v}_1\rangle\,] & & \textsc{Ampar-Close}
\end{array}$$

$$\textit{where}\ \begin{cases} h' & = & \mathbf{max}(\mathbf{hnames}(\mathsf{E}) \cup \{h\}){+}1 \\ h'' & = & \mathbf{max}(H \cup (\mathbf{hnames}(\mathsf{E}) \cup \{h\})){+}1 \\ h''' & = & \mathbf{max}(H \cup \mathbf{hnames}(\mathsf{E})){+}1 \end{cases}$$

Fig. 9. Small-step semantics

triggered when a destination $\rightarrow h$ is filled in the term under focus, typically in destination-filling primitives reductions, and results in the value v being written to hole $\boxed{h}$. The value v may contain holes itself (e.g. when the hollow constructor $\mathsf{Inl}\ \boxed{h'+1}$ is being written to the hole $\boxed{h}$ in FillL-Red),

hence the set H tracks the potential hole names introduced by value v, and is used to update the hole name set of the corresponding (open) ampar. Proper definition of $\mathsf{E}(\!|h :=_H \mathsf{v}|\!)$ is given in Figure 9.

FillU-Red and FillF-Red do not create any new hole; they only write a value to an existing one. On the other hand, rules FillL-Red, FillR-Red, FillE-Red and FillP-Red all write a hollow constructor to the hole $\boxed{h}$ that contains new holes. Thus, we need to generate fresh names for these new holes, and also return a destination for each new hole with a matching name.

The substitution $\mathsf{E}(\!|h :=_H \mathsf{v}|\!)$ should only be performed if h is a globally unique name; otherwise we break the promise of a write-once memory model. To this effect, we allow name shadowing while an ampar is closed, but as soon as an ampar is open, it should have globally unique hole names. This restriction is enforced in rule Ty-ectxs-OpenAmpar by premise $\mathbf{hnames}(\mathsf{E}) \mathbin{\#\#} \mathbf{hnames}(\Delta_3)$, requiring hole name sets from E and Δ_3 to be disjoint when an open ampar focusing component is created during reduction of $\mathbf{upd}_{\ltimes}$. Likewise, any hollow constructor written to a hole should have globally unique hole names. We assume that hole names are natural numbers for simplicity's sake.

To obtain globally fresh names, in the premises of the corresponding rules, we first set $h' = \mathbf{max}(\mathbf{hnames}(\mathsf{E}) \cup \{h\}) + 1$ or similar definitions for h'' and h''' (see in Figure 9) to find the next unused name. Then we use either the *shifted set* $H \doteq h'$ or the *conditional shift operator* $h[H \doteq h']$ as defined in Figure 9 to replace all names or just specific one with fresh unused names. We extend *conditional shift* $\bullet[H \doteq h']$ to arbitrary values, terms, and typing contexts in the obvious way (keeping in mind that ${}_{H'}\langle \mathsf{v}_2 \mathbin{,} \mathsf{v}_1 \rangle$ binds the names in H').

Rules Ampar-Open and Ampar-Close dictate how and when a closed ampar (a value) is converted to an open ampar (a focusing component) and vice-versa, and they make use of the shifting strategy we've just introduced. With Ampar-Open, the hole names bound by the ampar gets renamed to fresh ones, and the left-hand side gets attached to the focusing component ${}^{\mathsf{op}}_{H \doteq h'}\langle \mathsf{v}_2[H \doteq h'''] \mathbin{,} [] \rangle$ while the right-hand side (containing destinations) is substituted in the body of the $\mathbf{upd}_{\ltimes}$ statement (which becomes the new term under focus). The rule Ampar-Close triggers when the body of a $\mathbf{upd}_{\ltimes}$ statement has reduced to a value. In that case, we can close the ampar, by popping the focusing component from the stack E and merging back with v_2 to form a closed ampar again.

In rule FillComp-Red, we write the left-hand side v_2 of a closed ampar ${}_H\langle \mathsf{v}_2 \mathbin{,} \mathsf{v}_1 \rangle$ to a hole $\boxed{h}$ that is part of a structure with holes somewhere inside E. This results in the composition of two structures with holes. Because we dissociate v_2 and v_1 that were previously bound together by the ampar connective (v_2 is merged with another structure, while v_1 becomes the new focus), their hole names are no longer bound, so we need to make them globally unique, as we do when an ampar is opened with $\mathbf{upd}_{\ltimes}$. This renaming is carried out by the conditional shift $\mathsf{v}_2[H \doteq h'']$ and $\mathsf{v}_1[H \doteq h'']$.

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

Theorem 6.1 (Type preservation). *If* $\vdash \mathsf{E}[\mathsf{t}] : \mathsf{T}$ *and* $\mathsf{E}[\mathsf{t}] \longrightarrow \mathsf{E}'[\mathsf{t}']$ *then* $\vdash \mathsf{E}'[\mathsf{t}'] : \mathsf{T}$.

Theorem 6.2 (Progress). *If* $\vdash \mathsf{E}[\mathsf{t}] : \mathsf{T}$ *and* $\forall \mathsf{v}, \mathsf{E}[\mathsf{t}] \neq [][\mathsf{v}]$ *then* $\exists \mathsf{E}', \mathsf{t}'.\ \mathsf{E}[\mathsf{t}] \longrightarrow \mathsf{E}'[\mathsf{t}']$.

A command of the form $[][\mathsf{v}]$ cannot be reduced further, as it only contains a fully determined value, and no pending computation. This it is the stopping point of the reduction, and any well-typed command eventually reaches this form.

7 FORMAL PROOF OF TYPE SAFETY

We've proved type preservation and progress theorems with the Coq proof assistant. Turning to a proof assistant was a pragmatic choice: typing context handling in λ_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 λ_d that we've proved is written in Ott [Sewell et al. 2007], 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; though 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 7000 lines long, and contains nearly 500 lemmas. Many of the cases of the type preservation and progress lemmas are similar. To handle such repetitive cases, the use of a large-language-model based autocompletion system has proven quite effective.

The proofs aren't particularly elegant. For instance, we don't have any abstract formalization of semirings: 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.

First, we represent contexts as finite-domain functions, rather than as syntactic lists. This works much better when defining sums of context. There are a handful 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 in exchange equality gets more complex than with a bespoke data type.

Secondly, Addition of context is partial since we can only add two binding of the same name if they also have the same type. Instead of representing addition as a binary function to an optional context, we represent addition as a total function to contexts, but we change contexts to allow faulty bindings on some names. This works well better for our Ott-written rules, at the cost of needing well-formedness preconditions in the premises of typing rules as well as some lemmas.

Finally, to simplify equalities mostly, we assumed a few axioms: functional extensionality, classical logic, and indefinite description:

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

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.

8 IMPLEMENTATION OF λ_d USING IN-PLACE MEMORY MUTATIONS

The formal language presented in Sections 5 and 6 is not meant to be implemented as-is.

First, λ_d doesn't have recursion, this would have obscured the presentation of the system. However, adding a standard form of recursion doesn't create any complication.

Secondly, ampars are not managed linearly in λ_d; only destinations are. That is to say that an ampar can be wrapped in an exponential, e.g. $\mathsf{Mod}_{\omega\nu\ \{h\}}\langle 0 :: \boxed{h}_{\wedge} \to h\rangle$ (representing a difference list $0::\square$ that can be used non-linearly), and then used twice, each time in a different way:

$$\begin{array}{l}\textbf{case}\ \mathsf{Mod}_{\omega\nu\ \{h\}}\langle 0 :: \boxed{h}_{\wedge} \to h\rangle\ \textbf{of}\ \mathsf{Mod}_{\omega\nu}\ x \mapsto \\ \quad \textbf{let}\ x_1 \coloneqq x\ \textbf{append}\ 1\ \textbf{in} \\ \quad \textbf{let}\ x_2 \coloneqq x\ \textbf{append}\ 2\ \textbf{in} \\ \quad\quad \textbf{toList}\ (x_1\ \textbf{concat}\ x_2)\end{array} \longrightarrow^* \quad 0 :: 1 :: 0 :: 2 :: [\,]$$

It may seem counter-intuitive at first, but this program is valid and safe in λ_d. Thanks to the renaming discipline we detailed in Section 6.3, every time an ampar is operated over with $\mathbf{upd}_{\ltimes}$, its hole names are renamed to fresh ones. One way we can support this is to allocate a fresh copy of x every time we call **append** (which is implemented in terms of $\mathbf{upd}_{\ltimes}$), in a copy-on-write fashion. This way filling destinations is still implemented as mutation.

However, this is a long way from the efficient implementation promised in Section 2. Copy-on-write can be optimized using fully-in-place functional programming [Lorenzen et al. 2023], where, thanks to reference counting, we don't need to perform a copy when the difference list isn't aliased.

An alternative is to refine the linear type system further in order to guarantee that ampars are unique and avoid copy-on-write altogether. We held back from doing that in the formalization of λ_d as it obfuscates the presentation of the system without adding much in return.

To make ampars linear, we follow a recipe proposed by Spiwack et al. [2022] and introduce a new type Token, together with primitives **dup** and **drop**. We also switch $\mathbf{new}_{\ltimes}$ for $\mathbf{new}_{\ltimes\mathbf{IP}}$:

$$\mathbf{dup} \ : \ \mathsf{Token} \multimap \mathsf{Token} \otimes \mathsf{Token}$$
$$\mathbf{drop} \ : \ \mathsf{Token} \multimap 1$$
$$\mathbf{new}_{\ltimes\mathbf{IP}} \ : \ \mathsf{Token} \multimap \mathsf{T} \ltimes \lfloor \mathsf{T} \rfloor$$

For the in-place system to work, we consider that a linear root token variable, tok_0, is available to a program. "Closed" programs can now typecheck in the non-empty context $\{tok_0 :_{1\infty} \mathsf{Token}\}$. tok_0 can be used to create new tokens tok_k via **dup**, but each of these tokens still has to be used linearly.

Ampar produced by $\mathbf{new}_{\ltimes\mathbf{IP}}$ have a linear dependency on a variable tok_k. If an ampar produced by $\mathbf{new}_{\ltimes\mathbf{IP}}$ tok_k were to be used twice in a block t, then t would require a typing context $\{tok_k :_{\omega\nu} \mathsf{Token}\}$, that itself would require tok_0 to have multiplicity ω too. Thus the program would be rejected.

An alternative to having a linear root token variable is to add a primitive function $\mathbf{withToken} \ : \ (\mathsf{Token} \ {}_{1\infty}\!\!\multimap\! !_{\omega\infty}\mathsf{T}) \multimap !_{\omega\infty}\mathsf{T}$ that fulfill the same goal.

Now that ampars are managed linearly, we can change the allocation and renaming mechanisms:

- the hole name for a new ampar is chosen fresh right from the start (this corresponds to a new heap allocation);
- adding a new hollow constructor still require freshness for its hole names (this corresponds to a new heap allocation too);
- Using $\mathbf{upd}_{\ltimes}$ over an ampar and filling destinations or composing two ampars using $\lhd\!\circ$ no longer require any renaming: we have the guarantee that the all the names involved are globally fresh, and can only be used once, so we can do in-place memory updates.

λ_d extended with Tokens and $\mathbf{new}_{\ltimes\mathbf{IP}}$ is in fact very close to the implementation described in [Bagrel 2024]. Our claim of efficiency is thus based on the results published in the latter and also [Bour et al. 2021; Lorenzen et al. 2024a], as an hypothetical implementation of λ_d would mostly resort to the same memory operations – that is, in-place updates in functional settings.

From purely linked structures to more efficient memory forms. In λ_d we only have binary product in sum types. However, it's very straightforward to extend the language and implement destination-based building for n-ary sums of n-ary products, with constructors for each variant having multiple fields directly, instead of each field needing an extra indirection as in the binary sum of products $1 \oplus (S \otimes (T \otimes U))$. This is, in fact, already implemented in [Bagrel 2024] without any issues. However, it's probably better for field's values to still be represented by pointers.

Indeed, composition of incomplete structures relies on the idea that destinations pointing to holes of a structure v will still be valid if v get assigned to a field f of a bigger structure v′. That's true indeed if just the address of v is written to $v'.f$. However, if v is moved into v′ completly (i.e. if f is an in-place/unpacked field), then the pointers representing destinations of v are now invalid.

Our early experiments around DPS support for unpacked fields seem to indicate that we would need two classes of destinations, one supporting composition (for indirected fields) and one disallowing it (for unpacked fields).

9 RELATED WORK

9.1 Destination-passing style for efficient memory management

Shaikhha et al. [2017] present a destination-based intermediate language for a functional array programming language, with destination-specific optimizations, that boasts near-C performance.

This is the most comprehensive evidence to date of the benefits of destination-passing style for performance in functional languages, although their work is on array programming, while this article focuses on linked data structures. They can therefore benefit from 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 see these two aspects as complementing each other: good compiler optimizations are important to alleviate the burden from the programmer and allow high-level abstraction; having the possibility to use destinations in code affords the programmer more control, should they need it.

9.2 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], it 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 λ_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 an automatic transformation. On the flip-side, tail modulo constructor is too weak to handle our difference lists or breadth-first traversal examples.

9.3 A functional representation of data structures with a hole

The idea of using linear types as a foundation of a functional calculus in which incomplete data structures can exist and be composed as first class values dates back to [Minamide 1998]. Our system is strongly inspired by theirs. In [Minamide 1998], a first-class structure with a hole is called a *hole abstraction*. Hole abstractions are represented by a special kind of linear functions with bespoke restrictions. As with any function, we can't pattern-match on their output (or pass it to another function) until they have been applied; but they also have the restriction that we cannot pattern-match on their argument —the *hole variable*— as that one can only be used directly as argument of data constructors, or of other hole abstractions. The type of hole abstractions, $(\mathsf{T}, \mathsf{S})\mathsf{hfun}$ is thus a weak form of linear function type $\mathsf{T} \multimap \mathsf{S}$.

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" type. In classical linear logic, linear implication $\mathsf{T} \multimap \mathsf{S}$ is reinterpreted as $\mathsf{S} \parr \mathsf{T}^{\perp}$. We, likewise, reinterpret $(\mathsf{T}, \mathsf{S})\mathsf{hfun}$ as $\mathsf{S} \ltimes \lfloor \mathsf{T} \rfloor$ (a sort of weak "par").

A key consequence is that destinations —as first-class representations of holes— appear naturally in λ_d, while [Minamide 1998] doesn't have them. This means that using [Minamide 1998], or the more recent but similarly expressive system from [Lorenzen et al. 2024a], one can implement the examples with difference lists and queues from Section 2.3, but couldn't do our breadth-first traversal example from Section 4, since it requires to be able to store destinations in a structure.

Nevertheless, we still retain the main restrictions that Minamide [1998] places on hole abstractions. For instance, we can't pattern-match on S in (unapplied) $(\mathsf{T}, \mathsf{S})\mathsf{hfun}$; so in λ_d, we can't act directly on the left-hand side S of $\mathsf{S} \ltimes \mathsf{T}$, only on the right-hand side T. Similarly, hole variables can only be used as arguments of constructors or hole abstractions; it's reflected in λ_d by the fact that the only way to act on destinations is via fill operations, with either hollow constructors or another ampar.

The ability to manipulate destinations, and in particular, store them, does come at a cost though: the system needs this additional notion of ages to ensure that destinations are used soundly. On the other hand, our system is strictly more general, in Minamide [1998]'s system can be embedded in λ_d, and if one stays in this fragment, we're never confronted with ages.

9.4 Destination-passing style programming: a Haskell implementation

Bagrel [2024] proposes a system much like ours: it has a destination type, and a *par*-like construct (that they call Incomplete), where only the right-hand side can be modified; together these elements give extra expressiveness to the language compared to [Minamide 1998].

In their system, $d \blacktriangleleft \mathsf{t}$ requires t to be unrestricted, while in λ_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; so in a breadth-first search algorithm like in Section 4, they have to build the queue using normal constructors, and cannot use destination-filling primitives. Therefore both normal constructors and DPS primitives must coexist in their work, while in λ_d, only DPS primitives are required to bootstrap the system, as we later derive normal constructors from them. In exchange, they don't need a system of ages to make their system safe; just linearity is enough.

A more profound difference between their work and ours is that they describe a practical implementation of destination-passing style for an existing functional language, while we present a slightly more general theoretical framework that is meant to justify safety of DPS implementations (such as [Bagrel 2024] itself), so goals are quite different.

9.5 Semi-axiomatic sequent calculus

In [DeYoung et al. 2020] 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 $\lfloor \mathsf{T} \rfloor$ 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 semantics of holes is that of a future or a promise. In turns this requires the semantics of their calculus to be fully concurrent, which is a very different point in the design space.

9.6 Rust lifetimes

Rust uses a system of lifetimes (see e.g. [Pearce 2021]) to ensure that borrows don't live longer than what they reference. It plays a similar role as our system of ages.

Rust lifetimes are symbolic. Borrows and moves generate constraints (inequalities of the form $\alpha \leqslant \beta$) on the symbolic lifetimes. For instance, that a the lifetime of a reference is larger than the lifetime of any structure the reference is stored in. Without such constraints, Rust would have similar problems to those of Section 3. The borrow checker then checks that the constraints are solvable. This contrasts with λ_d where ages are set explicitly, with no analysis needed.

Another difference between the two systems is that λ_d's ages (and modes in general) are relative. An explicit modality $!_{1\uparrow^k}$ must be used when a part has an age different than its parent, and means that the part is k scope older than the parent. On the other hand, Rust's lifetimes are absolute, the lifetime of a part is tracked independently of the lifetime of its parent.

9.7 Oxidizing OCaml

Lorenzen et al. [2024b] present an extension of the OCaml type system to support modes. Their modes are split along three different "axes", among which affinity and locality are comparable to our multiplicities and ages. Like our multiplicities, there are two modes for affinity `once` and `many`, though in [Lorenzen et al. 2024b], `once` supports weakening, whereas λ_d's 1 multiplicity is properly linear (proper linearity matters for destination lest we end up reading uninitialized memory).

Locality tracks scope. There are two locality modes, `local` (doesn't escape the current scope) and `global` (can escape the current scope). The authors present their locality mode as a drastic simplification of Rust's lifetime system, which nevertheless fits their need.

However, such a simplified system would be a bit too weak to track the scope of destinations. The observation is that if destinations from two nested scopes are given the same mode, then we can't safely do anything with them, as it would be enough to reproduce the counterexamples of Section 3. So in order to type the breadth-first traversal example of Section 4, where destinations are stored in a structure, we need at least ν (for the current scope), $\uparrow$ (for the previous scope exactly), plus at least one extra mode for the rest of the scopes (destinations of this generic age cannot be safely used). It turns out that such systems with finitely many ages are incredibly easy to get wrong, and it was in fact much simpler to design a system with infinitely many ages.

10 CONCLUSION AND FUTURE WORK

Using a system of ages in addition to linearity, λ_d is a purely functional calculus which supports destinations 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 λ_d's flexibility.

We don't anticipate that a system of ages like λ_d will actually be used in a programming language: it's unlikely that destinations 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 λ_d as a typed intermediate representation. But, more realistically, our expectation is that λ_d can be used as a theoretical framework to analyze destination-passing systems: if an API can be defined in λ_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.

DATA-AVAILABILITY STATEMENT

We submitted the machine-verified proof described in Section 7 for artifact evaluation. It's a formalization of λ_d as described in Sections 5 and 6, using the Coq proof assistant with some classical axioms. The preliminary, anonymous version is available at https://doi.org/10.5281/zenodo.13933661, together with the build instructions.

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

Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom) *(LICS '18)*. Association for Computing Machinery, New York, NY, USA, 56–65. https://doi.org/10.1145/3209108.3209189

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].

Małgorzata Biernacka and Olivier Danvy. 2007. A syntactic correspondence between context-sensitive calculi and abstract machines. *Theoretical Computer Science* 375, 1 (May 2007), 76–108. https://doi.org/10.1016/j.tcs.2006.12.028

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

Pierre-Louis Curien and Hugo Herbelin. 2000. The duality of computation. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 233–243. https://doi.org/10.1145/351240.351262

Olivier Danvy and Lasse R. Nielsen. 2004. Refocusing in Reduction Semantics. *BRICS Report Series* 11, 26 (Nov. 2004). https://doi.org/10.7146/brics.v11i26.21851

Henry DeYoung, Frank Pfenning, and Klaas Pruiksma. 2020. Semi-Axiomatic Sequent Calculus. In *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 167)*, Zena M. Ariola (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 29:1–29:22. https://doi.org/10.4230/LIPIcs.FSCD.2020.29

Matthias Felleisen. 1987. *The calculi of lambda-nu-cs conversion: a syntactic theory of control and state in imperative higher-order programming languages.* phd. Indiana University, USA. https://www2.ccs.neu.edu/racket/pubs/dissertation-felleisen.pdf AAI8727494.

Dan R. Ghica and Alex I. Smith. 2014. Bounded Linear Types in a Resource Semiring. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer, Berlin, Heidelberg, 331–350. https://doi.org/10.1007/978-3-642-54833-8_18

Jeremy Gibbons. 1993. Linear-time Breadth-first Tree Algorithms: An Exercise in the Arithmetic of Folds and Zips. No. 71 (1993). https://www.cs.ox.ac.uk/publications/publication2363-abstract.html Number: No. 71.

Jeremy Gibbons, Donnacha Oisín Kidney, Tom Schrijvers, and Nicolas Wu. 2023. Phases in Software Architecture. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Functional Software Architecture*. ACM, Seattle WA USA, 29–33. https://doi.org/10.1145/3609025.3609479

Robert Hood and Robert Melville. 1981. Real-time queue operations in pure LISP. *Inform. Process. Lett.* 13, 2 (1981), 50–54. https://doi.org/10.1016/0020-0190(81)90030-2

John Hughes. 1986. A Novel Representation of Lists and its Application to the Function "reverse". *Inf. Process. Lett.* 22 (01 1986), 141–144.

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

Anton Lorenzen, Daan Leijen, and Wouter Swierstra. 2023. FP^2: Fully in-Place Functional Programming. *Proceedings of the ACM on Programming Languages* 7, ICFP (Aug. 2023), 275–304. https://doi.org/10.1145/3607840

Anton Lorenzen, Daan Leijen, Wouter Swierstra, and Sam Lindley. 2024a. The Functional Essence of Imperative Binary Search Trees. *Proc. ACM Program. Lang.* 8, PLDI, Article 168 (jun 2024), 25 pages. https://doi.org/10.1145/3656398

Anton Lorenzen, Leo White, Stephen Dolan, Richard A. Eisenberg, and Sam Lindley. 2024b. Oxidizing OCaml with Modal Memory Management. *Proc. ACM Program. Lang.* 8, ICFP (Aug. 2024), 253:485–253:514. https://doi.org/10.1145/3674642

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

Chris Okasaki. 2000. Breadth-first numbering: lessons from a small exercise in algorithm design. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 131–136. https://doi.org/10.1145/351240.351253

David J. Pearce. 2021. A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust. *ACM Trans. Program. Lang. Syst.* 43, 1 (April 2021), 3:1–3:73. https://doi.org/10.1145/3443420

F. Pfenning and H. C. Wong. 1995. On a Modal λ-Calculus for S4. *Electronic Notes in Theoretical Computer Science* 1 (Jan. 1995), 515–534. https://doi.org/10.1016/S1571-0661(04)00028-3

Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. 2007. Ott: effective tool support for the working semanticist. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming* (Freiburg, Germany) *(ICFP '07)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/1291151.1291155

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