Destination calculus

A linear λ−calculus for pure, functional memory updates

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

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

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

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

1 INTRODUCTION

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

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

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

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

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

2 SYSTEM IN ACTION ON SIMPLE EXAMPLES

The idea of destination calculus is to provide a building canvas for data structures, represented as a special pair-like structure, called an *ampar*, whose left side is the structure being built, and whose right side carries destinations pointing to holes present in the left side.

A *hole*, denoted h in the language, is a memory cell that has not been written to yet. In a practical application, a *hole* would contain garbage data left by the previous use of the memory location, and thus should not be read in any circumstances (or it could lead to a segmentation fault!).

A *destination*, denoted $\rightarrow h$, is the address of a hole. It is meant to be one-use only; unlike mutable references which can often be reused.

The simplest form of ampar is a single empty cell on the left with a destination pointing to it on the right: ${}_{\{h\}}\langle h \,{}_{\wedge}\!\rightarrow h\rangle$. This ampar is highly analogous to the identity function: the final structure on the left side will correspond to what is fed in the right side.

The main operation to operate on an ampar is **map**, which binds the right side of the ampar to a variable, while temporarily forgetting about the left side (the incomplete structure that is being mutated behind the scenes).

${}_{\{h\}}\langle h \,{}_{\wedge}\!\rightarrow h\rangle \triangleright \textbf{map}\ d \mapsto d \triangleleft \mathsf{Inl} \triangleleft () \text{ reduces to } {}_{\{h'\}}\langle \mathsf{Inl}\ h' \,{}_{\wedge}\!\rightarrow h'\rangle \text{ then to } {}_{\{\}}\langle \mathsf{Inl}\ () \,{}_{\wedge}\, ()\rangle$

The destination-feeding primitive $\triangleleft\mathsf{Inl}$ will in fact follow the destination on the left-hand side and write an Inl data constructor to the hole pointed by the destination. A new destination $\rightarrow h'$ is returned to represented the (yet unspecified) payload for the Inl constructor.

This new destination $\rightarrow h'$ is then passed to the destination-feeding primitive $\triangleleft()$, which writes a unit value to the hole pointed by $\rightarrow h'$. The unit data constructor () doesn't hold any payload, so there is no new destination to return here (so () is returned by the primitive instead).

Arbitrary building order. As mentioned earlier, one key point of destination calculus is being able to build a structure in any desired order. For example, one can build a balanced binary tree of depth two by first building the skeleton, then giving values to the right leaves, then to the left leaves.

Let's build the skeleton first:

$$
\begin{aligned}
&x_1 \coloneqq {}_{\{h_1\}}\langle h_1 \,{}_{\wedge}\!\rightarrow h_1\rangle \triangleright \textbf{map}\ d_1 \mapsto \\
&\quad (d_1 \triangleleft (,)) \triangleright \textbf{case}\ (d_2\,,\, d_3) \mapsto \\
&\qquad (d_2 \triangleleft (,)) \triangleright \textbf{case}\ (d_4\,,\, d_5) \mapsto \\
&\qquad\quad (d_3 \triangleleft (,)) \triangleright \textbf{case}\ (d_6\,,\, d_7) \mapsto [d_4, d_5, d_6, d_7]
\end{aligned}
$$

evaluates to:

$${}_{\{h_4,h_5,h_6,h_7\}}\langle((h_4\,,\, h_5)\,,\, (h_6\,,\, h_7)) \,{}_{\wedge}\, [\rightarrow h_4, \rightarrow h_5, \rightarrow h_6, \rightarrow h_7]\rangle$$

We can see that the skeleton of the tree is here on the left-hand side, but leaves are unspecified yet (so are represented by holes). Let's now fill the right leaves:

$$
\begin{aligned}
&x_2 \coloneqq x_1 \triangleright \textbf{map}\ d \mapsto \\
&\quad d \triangleright \textbf{case}\ [d_4, d_5, d_6, d_7] \mapsto \\
&\qquad d_5 \triangleleft \mathsf{Inr} \triangleleft () \fatsemi d_7 \triangleleft \mathsf{Inr} \triangleleft () \fatsemi [d_4, d_6]
\end{aligned}
$$

reduces to

$${}_{\{h_4,h_6\}}\langle((h_4\,,\, \mathsf{Inr}\ ())\,,\, (h_6\,,\, \mathsf{Inr}\ ())) \,{}_{\wedge}\, [\rightarrow h_4, \rightarrow h_6]\rangle$$

Now only left leaves are left unspecified, that's why the associated destinations $\rightarrow h_4$ and $\rightarrow h_6$ are the only left on the right side. We can finally feed them:

$$
\begin{aligned}
&x_3 \coloneqq x_2 \triangleright \textbf{map}\ d \mapsto \\
&\quad d \triangleright \textbf{case}\ [d_4, d_6] \mapsto \\
&\qquad d_4 \triangleleft \mathsf{Inl} \triangleleft () \fatsemi d_6 \triangleleft \mathsf{Inr} \triangleleft ()
\end{aligned}
$$

x_3 reduces to ${}_{\{\}}\langle((\mathsf{Inl}\ ()\,,\, \mathsf{Inr}\ ())\,,\, (\mathsf{Inl}\ ()\,,\, \mathsf{Inr}\ ())) \,{}_{\wedge}\, ()\rangle$ which is now a complete structure. There is no destination remaining on the right-hand side.

We also see that structure can in fact be built in several stages with ease, with other operations taking place in-between.

2.1 Progressing towards an efficient queue implementation

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

Linked lists. For starters, we can define lists as the fixpoint of the functor $X \mapsto 1 \oplus (T \otimes X)$ where T is the type of list items. Instead of defining the usual Nil [] and Cons (::) constructors, we define the more general FillNil $\triangleleft[]$ and FillCons $\triangleleft(::)$ operators, as presented in Figure 1.

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

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

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

Fig. 1. List implementation in equirecursive destination calculus

The FillNil operator consumes a destination of list and fills its associated hole with the value Inl () representing the Nil constructor in our encoding. It returns unit () as there is no new hole/destination created in the process.

The FillCons operator consumes a destination of list as an input, and writes a hollow Cons constructor (represented by Inr (h_1, h_2) in our encoding) to the hole pointed by the destination. It then returns a pair of destinations $\to h_1$ and $\to h_2$ to represent the two holes in the hollow Cons constructor ($\to h_1$ is a destination for the list item, and $\to h_2$ is for the list tail). This behavior is hidden behind the primitives $\triangleleft\mathsf{Inr} : \lfloor_n 1 \oplus (T \otimes (\text{List } T)) \rfloor \to \lfloor_n T \otimes (\text{List } T) \rfloor$ (choosing the right branch of a sum type by writing a hollow Inr constructor) and $\triangleleft(,) : \lfloor_n T \otimes (\text{List } T) \rfloor \to \lfloor_n T \rfloor \otimes \lfloor_n \text{List } T \rfloor$ (transforming a destination of pair to a pair of destination by writing a hollow pair constructor).

There is a duality between constructors and destinations-feeding operators (we will dig into it more in Section 5.1), and that stays true for our newly user-defined operators. If we reverse the arrow direction and remove the destination symbols from FillCons signature, we get $T \otimes (\text{List } T) \to \text{List } T$ which is the type of the usual Cons constructor! We can in fact recover the Cons constructor:

$$(::) \;:\; T \otimes (\text{List } T) \to \text{List } T$$
$$(::)\ x\ xs \triangleq \mathbf{alloc} \rhd \mathbf{map}\ d \mapsto (d \triangleleft (::)) \rhd \mathbf{case}\ (dx, dxs) \mapsto dx \triangleleft x \fatsemi dxs \triangleleft xs$$

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

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

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

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

$$\begin{aligned}
&\mathsf{DList}\ T \triangleq (\mathsf{List}\ T) \ltimes (\lfloor \mathsf{List}\ T \rfloor)\\
&\textbf{append} : \mathsf{DList}\ T \to T \to \mathsf{DList}\ T\\
&ys\ \textbf{append}\ y \triangleq ys \triangleright \textbf{map}\ dys \mapsto (dys \triangleleft (::)) \triangleright \textbf{case}\\
&\qquad\qquad (dy, dys') \mapsto dy \triangleleft y \fatsemi dys'\\
&\textbf{concat} : \mathsf{DList}\ T \to \mathsf{DList}\ T \to \mathsf{DList}\ T\\
&ys\ \textbf{concat}\ ys' \triangleq ys \triangleright \textbf{map}\ d \mapsto d \mathrel{\triangleleft\!\bullet} ys'\\
&\textbf{to}_{\mathsf{List}} : \mathsf{DList}\ T \to \mathsf{List}\ T\\
&\textbf{to}_{\mathsf{List}}\ ys \triangleq \textbf{from}'_{\ltimes}\,(ys \triangleright \textbf{map}\ d \mapsto d \triangleleft [])
\end{aligned}$$

Fig. 2. Difference list implementation in equirecursive destination calculus

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

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

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

Finally, the $\textbf{to}_{\mathsf{List}}$ operator converts a difference list to a regular list by filling the hole left in the incomplete list using FillNil.

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

[1] In this particular context, $U_1 = \mathsf{List}\ T$ and $U_2 = \lfloor \mathsf{List}\ T \rfloor$, so FillComp has signature $\mathrel{\triangleleft\!\bullet} : \lfloor \mathsf{List}\ T \rfloor \to \mathsf{DList}\ T \xrightarrow{1\uparrow} \lfloor \mathsf{List}\ T \rfloor$

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

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

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

```
Queue T ≜ (List T)⊗(DList T)

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

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

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

Fig. 3. Queue implementation in equirecursive destination calculus

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

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

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

3 LIMITATIONS OF THE PREVIOUS APPROACH

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

Indeed, that first approach of destination-passing style for Haskell can only be used to build non-linear data structures. More precisely, the FillLeaf operator ($\triangleleft$) can only take arguments with

multiplicity ω. This is in fact a much stronger restriction than necessary ; the core idea is *just* to prevent any destination (which is always a linear resource) to appear somewhere in the right-hand side of FillLeaf.

3.1 Why stored destinations are problematic

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

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

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

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

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

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

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

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

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

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

$$\begin{aligned}
&\mathbf{alloc'}\ (\lambda dd \mapsto \mathbf{alloc'}\ (\lambda d \mapsto dd \triangleleft d)) \\
&\quad\longrightarrow \mathbf{withTmpStore}\ \{hd := _\}\ \mathbf{do}\ \{\mathbf{alloc'}\ (\lambda d \mapsto \to hd \triangleleft d)\ ;\!\!;\ \mathbf{deref}\ \to hd\} \\
&\quad\longrightarrow \mathbf{withTmpStore}\ \{hd := _\}\ \mathbf{do}\ \{\mathbf{withTmpStore}\ \{h := _\}\ \mathbf{do}\ \{\to hd \triangleleft \to h\ ;\!\!;\ \mathbf{deref}\ \to h\}\ ;\!\!;\ \mathbf{deref}\ \to hd\} \\
&\quad\longrightarrow \mathbf{withTmpStore}\ \{hd := \to h\}\ \mathbf{do}\ \{\mathbf{withTmpStore}\ \{h := _\}\ \mathbf{do}\ \{\mathbf{deref}\ \to h\}\ ;\!\!;\ \mathbf{deref}\ \to hd\}
\end{aligned}$$

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

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

3.2 Age control to prevent scope escape of destinations

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

Age is represented by a commutative semiring, where ν indicates that a destination originates from the current scope, and $\uparrow$ indicates that it originates from the scope just before. We also extend ages to variables (a variable of age a stands for a value of age a). Finally, age ∞ is introduced for variables standing in place of a non-age-controlled value. In particular, destinations can never have age ∞ in practice.

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

Given $\uparrow^0 = \nu$ and $\uparrow^n = \uparrow \cdot \uparrow^{n-1}$, we have the following age operation tables:

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

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

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

The main restriction to prevent parent scope escape is materialized in these simplified typing rules:

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

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

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

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

4 (UPDATED) BREADTH-FIRST TREE TRAVERSAL

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

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

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

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

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

This is quite impractical as we would like to use the efficient, destination-based queue implementation from Section 2.1 to power up the breadth-first tree traversal implementation[2]. In our present work fortunately, thanks to the finer age-control mechanism, we can store linear resources in destination-based structures without any issue. Our system is in fact self-contained, as any structure, whatever the use for it be, can be built using a small core of destination-based primitives (and regular data constructors can be retrieved from destination-based primitives, see Section 5.4).

The implementation for the efficient queue as well as other is presented as part of Figure 4.

Figure 4 introduces a few extra tools needed for implementation of breadth-first tree traversal, while Figure 5 presents the actual implementation of the traversal. This implementation is as similar as possible to the one from [1], as to make it easier to spot the few differences between the two systems.

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

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

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

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

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

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

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

Fig. 4. Boilerplate for breadth-first tree traversal

5 LANGUAGE SYNTAX

5.1 Introducing the *ampar*

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

In that paper, a structure with a hole is named *hole abstraction*. In the body of a hole abstraction, the bound *hole variable* should be used linearly (exactly once), and must only be used as a parameter of a data constructor. In other terms, the bound *hole variable* cannot be pattern-matched on or

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

```
go : ((!_{1∞}S) → T_1 → (!_{1∞}S)⊗T_2) _{ων}→ (!_{1∞}S) → Queue (Tree T_1⊗⌊Tree T_2⌋) → (!_{1∞}S)
go f st q ≜ (dequeue q) ▷ case {
              Inl () ↦ st,
              Inr ((tree, dtree), q') ↦ tree ▷ case {
                  Inl () ↦ dtree ◁ Nil ⨟ go f st q',
                  Inr (x, (tl, tr)) ↦ (dtree ◁ Node) ▷ case
                      (dy, (dtl, dtr)) ↦ (f st x) ▷ case
                          (st', y) ↦ dy ◁ y ⨟ go f st' (
                              q' enqueue (tl, dtl) enqueue (tr, dtr)
                          )
                      }
          }
mapAccumBFS : ((!_{1∞}S) → T_1 → (!_{1∞}S)⊗T_2) _{ων}→ (!_{1∞}S) → Tree T_1 → Tree T_2⊗(!_{1∞}S)
mapAccumBFS f st tree ≜ from'_⋉ (alloc ▷ map dtree↦ go f st (singleton (tree, dtree)))

relabelDPS : Tree 1 → (Tree Int)⊗(!_{1∞}(!_{ων}Int))
relabelDPS tree ≜ mapAccumBFS
                    (λex ↦ λun ↦ un ⨟ ex ▷ case
                        E_{1∞} ex' ↦ ex' ▷ case_{1∞}
                            E_{ων} st ↦ (E_{1∞} (E_{ων} (succ st)), st))
                    (E_{1∞} (E_{ων} (succ zero)))
                    tree
```

Fig. 5. Breadth-first tree traversal in destination-passing style

used as a parameter of a function call. A hole abstraction is thus a weak form of linear lambda abstraction, which just moves a piece of data into a bigger data structure.

In fact, the type of hole abstraction (T_1, T_2)hfun in Minamine's work shares a lot of similarity with the separating implication or *magic wand* $T_1 \mathrel{-\!\!*} T_2$ from separation logic: given a piece of memory matching description T_1, we obtain a (complete) piece of memory matching description T_2.

Now, in classical linear logic, we know we can transform linear implication $T_1 \multimap T_2$ into $T_1{}^{\perp} \parr T_2$. Doing so for the *wand* type (T_1, T_2)hfun or $T_1 \mathrel{-\!\!*} T_2$ gives $\lfloor T_1 \rfloor \mathbin{\widehat{\parr}} T_2$, where $\lfloor \bullet \rfloor$ is memory negation, and $\widehat{\parr}$ is a memory *par* (weaker than the CLL *par* that allows more "interaction" of its two sides).

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

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

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

to by $\to h'$ is consumed. Thus, a destination of destination is only equivalent to the promise of an eventual value, not an immediate usable one.

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

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

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

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

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

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

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

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

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

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

5.2 Names and variables

The destination calculus uses two classes of names: regular (meta) variable names x, y, and hole names, h, h_1, h_2 which represents the identifier or address of a memory cell that hasn't been written to yet.

var, *x*, *y*, *d*, *dd*, *un*, *xs*, *ys*, *ex*, *st*, *tree*, *tl*, *tr*, *dtree*, *f*, *dh*, *dt*, *dx*, *dy*, *dxs*, *dys*, *dv*, *dtlr*, *dtl*, *dtr*, *q*			Variable
$hvar$, h, hd	::=		Hole (or destination) name, represented by a natural number
	|	$h{+}h'$	M
	|	$h[H{\pm}h']$	M Shift by h' if $h \in H$
	|	$\mathbf{max}(H)$	M Maximum of a set of hole names

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

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

$hvars$, H	::=		Set of hole names
	|	$\{h_1, .., h_k\}$	
	|	$H_1 \cup H_2$	M Union of sets
	|	$H{\pm}h'$	M Shift all names from H by h'.
	|	$\mathbf{hvars}(\Gamma)$	M Hole names bound by the typing context Γ
	|	$\mathbf{hvars}(C)$	M Hole names bound by the evaluation context C

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

5.3 Term and value core syntax

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

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

$term$, t, u	::=		Term
	|	v	Value
	|	x	Variable
	|	$t'\ t$	Application
	|	$t \fatsemi u$	Pattern-match on unit
	|	$t \triangleright \mathbf{case}_{\mathsf{m}} \{\mathsf{Inl}\, x_1 \mapsto u_1\,,\ \mathsf{Inr}\, x_2 \mapsto u_2\}$	Pattern-match on sum
	|	$t \triangleright \mathbf{case}_{\mathsf{m}} (x_1\,, x_2) \mapsto u$	Pattern-match on product
	|	$t \triangleright \mathbf{case}_{\mathsf{m}}\ \mathsf{E}_{\mathsf{n}}\ x \mapsto u$	Pattern-match on exponential
	|	$t \triangleright \mathbf{map}\ x \mapsto t'$	Map over the right side of ampar
	|	$\mathbf{to}_{\ltimes}\ u$	Wrap into a trivial ampar
	|	$\mathbf{from}_{\ltimes}\ t$	Convert ampar to a pair
	|	$t \triangleleft ()$	Fill destination with unit

	$t \triangleleft \mathsf{Inl}$		Fill destination with left variant
|	$t \triangleleft \mathsf{Inr}$		Fill destination with right variant
|	$t \triangleleft \mathsf{E}_{\mathsf{m}}$		Fill destination with exponential constructor
|	$t \triangleleft (,)$		Fill destination with product constructor
|	$t \triangleleft (\lambda x \,{}_{\mathsf{m}}\!\mapsto u)$		Fill destination with function
|	$t \triangleleft\!\bullet\, t'$		Fill destination with root of other ampar
|	$t[x := v]$	M	

val, v	::=			Value
	|	h		Hole
	|	$\rightarrow h$		Destination
	|	$()$		Unit
	|	$\mathsf{v}\lambda x \,{}_{\mathsf{m}}\!\mapsto u$		Function with no free variable
	|	$\mathsf{Inl}\ v$		Left variant for sum
	|	$\mathsf{Inr}\ v$		Right variant for sum
	|	$\mathsf{E}_{\mathsf{m}}\ v$		Exponential
	|	(v_1 , v_2)		Product
	|	$_H\langle v_2 \wedge v_1 \rangle$		Ampar
	|	$v[H \doteq h']$	M	Shift hole names inside v by h' if they belong to H.

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

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

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

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

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

5.4 Syntactic sugar for constructors and commonly used operations

As we said in section 5.3, the grammatical class of values is mostly used for runtime only; in particular, data constructors can only take other values as arguments, not terms. Thus we introduce

syntactic for data constructors taking arbitrary terms as parameters (as we often find in functional programming languages) using destination-feeding primitives.

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

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

$\mathbf{alloc}$	$\triangleq \{_1\}\langle 1 \wedge \to 1\rangle$	$t \triangleleft t'$	$\triangleq t \triangleleft\!\bullet (\mathbf{to}_{\ltimes}\ t')$
$\mathbf{from}'_{\ltimes}\ t$	$\triangleq (\mathbf{from}_{\ltimes}\ (t \triangleright \mathbf{map}\ un \mapsto un \fatsemi \mathsf{E}_{!\infty}\ ()))\ \triangleright \mathbf{case}$ $(st, ex) \mapsto ex \triangleright \mathbf{case}$ $\mathsf{E}_{!\infty}\ un \mapsto un \fatsemi st$	$\lambda x\ {}_{\mathsf{m}}\!\mapsto u$	$\triangleq \mathbf{from}'_{\ltimes}\ ($ $\mathbf{alloc} \triangleright \mathbf{map}\ d \mapsto$ $d \triangleleft (\lambda x\ {}_{\mathsf{m}}\!\mapsto u)$ $)$
$\mathsf{Inl}\ t$	$\triangleq \mathbf{from}'_{\ltimes}\ ($ $\mathbf{alloc} \triangleright \mathbf{map}\ d \mapsto$ $d \triangleleft \mathsf{Inl} \triangleleft t$ $)$	$\mathsf{Inr}\ t$	$\triangleq \mathbf{from}'_{\ltimes}\ ($ $\mathbf{alloc} \triangleright \mathbf{map}\ d \mapsto$ $d \triangleleft \mathsf{Inr} \triangleleft t$ $)$
$\mathsf{E}_{\mathsf{m}}\ t$	$\triangleq \mathbf{from}'_{\ltimes}\ ($ $\mathbf{alloc} \triangleright \mathbf{map}\ d \mapsto$ $d \triangleleft \mathsf{E}_{\mathsf{m}} \triangleleft t$ $)$	(t_1, t_2)	$\triangleq \mathbf{from}'_{\ltimes}\ ($ $\mathbf{alloc} \triangleright \mathbf{map}\ d \mapsto$ $(d \triangleleft (,)) \triangleright \mathbf{case}$ $(d_1, d_2) \mapsto d_1 \triangleleft t_1 \fatsemi d_2 \triangleleft t_2$ $)$

Table 1. Desugaring of syntactic sugar forms for terms

6 TYPE SYSTEM

6.1 Syntax for types, modes, and typing contexts

type, T, U, S	::=		Type
	|	1	Unit
	|	$T_1 \oplus T_2$	Sum
	|	$T_1 \otimes T_2$	Product
	|	$!_m T$	Exponential
	|	$U \ltimes T$	Ampar
	|	$T \xrightarrow{m} U$	Function
	|	$\lfloor_m T \rfloor$	Destination
mode, m, n	::=		Mode (Semiring)
	|	pa	Pair of a multiplicity and age
	|	$☠$	Error case (incompatible types, multiplicities, or ages)
mul, p	::=		Multiplicity (Semiring, first component of modality)
	|	1	Linear use
	|	ω	Non-linear use
age, a	::=		Age (Semiring, second component of modality)
	|	ν	Born now
	|	$\uparrow$	One scope older
	|	∞	Infinitely old / static
ctx, Γ, Δ, Θ	::=		Typing context
	|	$x :_m T$	Variable typing binding
	|	$h :_n T$	Hole typing binding
	|	$\rightarrow h :_m \lfloor_n T \rfloor$	Destination typing binding
	|	$m \cdot \Gamma$	M Multiply the leftmost mode of each binding by m
	|	$\Gamma_1 + \Gamma_2$	M Sum (incompatible bindings get tagged with ☠)
	|	Γ_1, Γ_2	M Disjoint sum
	|	$\rightarrow^{-1} \Gamma$	M Transforms dest bindings into a hole bindings
	|	$\rightarrow \Gamma$	M Transforms hole bindings into dest bindings
	|	$\Gamma[H \doteq h']$	M Shift hole/dest names by h' if they belong to H

6.2 Typing of terms and values

$\boxed{\Gamma \Vdash v : \mathsf{T}}$ *(Typing judgment for values)*

$$\textsc{Ty-val-Hole}\ \frac{}{h :_{1\nu} \mathsf{T} \Vdash h : \mathsf{T}} \qquad \textsc{Ty-val-Dest}\ \frac{}{\to h :_{1\nu} \lfloor_{\mathsf{n}} \mathsf{T} \rfloor \Vdash \to h : \lfloor_{\mathsf{n}} \mathsf{T} \rfloor} \qquad \textsc{Ty-val-Unit}\ \frac{}{\Vdash () : 1}$$

$$\textsc{Ty-val-Fun}\ \frac{\Delta,\ x :_{\mathsf{m}} \mathsf{T} \vdash u : \mathsf{U}}{\Delta \Vdash \lambda x \,{}_{\mathsf{m}}\!\mapsto u : \mathsf{T} \,{}_{\mathsf{m}}\!\to \mathsf{U}} \qquad \textsc{Ty-val-Left}\ \frac{\Gamma \Vdash v_1 : \mathsf{T}_1}{\Gamma \Vdash \mathsf{Inl}\ v_1 : \mathsf{T}_1 \oplus \mathsf{T}_2} \qquad \textsc{Ty-val-Right}\ \frac{\Gamma \Vdash v_2 : \mathsf{T}_2}{\Gamma \Vdash \mathsf{Inr}\ v_2 : \mathsf{T}_1 \oplus \mathsf{T}_2}$$

$$\textsc{Ty-val-Prod}\ \frac{\Gamma_1 \Vdash v_1 : \mathsf{T}_1 \qquad \Gamma_2 \Vdash v_2 : \mathsf{T}_2}{\Gamma_1 + \Gamma_2 \Vdash (v_1, v_2) : \mathsf{T}_1 \otimes \mathsf{T}_2} \qquad \textsc{Ty-val-Exp}\ \frac{\Gamma \Vdash v' : \mathsf{T}}{\mathsf{n}\cdot\Gamma \Vdash \mathsf{E}_{\mathsf{n}}\ v' : !_{\mathsf{n}} \mathsf{T}}$$

$$\textsc{Ty-val-Ampar}\ \frac{\begin{array}{c}\mathtt{LinOnly}\ \Delta_3 \qquad \mathtt{FinAgeOnly}\ \Delta_3 \\ 1\!\uparrow\!\cdot\Delta_1,\ \Delta_3 \Vdash v_1 : \mathsf{T} \qquad \Delta_2,\ (\to^{-1}\Delta_3) \Vdash v_2 : \mathsf{U}\end{array}}{\Delta_1,\ \Delta_2 \Vdash \mathbf{hvars}(\to^{-1}\Delta_3)\langle v_2 \wedge v_1 \rangle : \mathsf{U} \ltimes \mathsf{T}}$$

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

$$\textsc{Ty-term-Val}\ \frac{\mathtt{DisposableOnly}\ \Theta \qquad \Delta \Vdash v : \mathsf{T}}{\Theta,\ \Delta \vdash v : \mathsf{T}} \qquad \textsc{Ty-term-Var}\ \frac{\mathtt{DisposableOnly}\ \Theta \qquad 1\nu <: \mathsf{m}}{\Theta,\ x :_{\mathsf{m}} \mathsf{T} \vdash x : \mathsf{T}} \qquad \textsc{Ty-term-App}\ \frac{\Theta_1 \vdash t : \mathsf{T} \qquad \Theta_2 \vdash t' : \mathsf{T} \,{}_{\mathsf{m}}\!\to \mathsf{U}}{\mathsf{m}\cdot\Theta_1 + \Theta_2 \vdash t'\ t : \mathsf{U}}$$

$$\textsc{Ty-term-PatU}\ \frac{\Theta_1 \vdash t : 1 \qquad \Theta_2 \vdash u : \mathsf{U}}{\Theta_1 + \Theta_2 \vdash t \fatsemi u : \mathsf{U}} \qquad \textsc{Ty-term-PatS}\ \frac{\begin{array}{c}\Theta_1 \vdash t : \mathsf{T}_1 \oplus \mathsf{T}_2 \\ \Theta_2,\ x_1 :_{\mathsf{m}} \mathsf{T}_1 \vdash u_1 : \mathsf{U} \\ \Theta_2,\ x_2 :_{\mathsf{m}} \mathsf{T}_2 \vdash u_2 : \mathsf{U}\end{array}}{\mathsf{m}\cdot\Theta_1 + \Theta_2 \vdash t \triangleright \mathbf{case}_{\mathsf{m}}\ \{\mathsf{Inl}\ x_1 \mapsto u_1,\ \mathsf{Inr}\ x_2 \mapsto u_2\} : \mathsf{U}}$$

$$\textsc{Ty-term-PatP}\ \frac{\begin{array}{c}\Theta_1 \vdash t : \mathsf{T}_1 \otimes \mathsf{T}_2 \\ \Theta_2,\ x_1 :_{\mathsf{m}} \mathsf{T}_1,\ x_2 :_{\mathsf{m}} \mathsf{T}_2 \vdash u : \mathsf{U}\end{array}}{\mathsf{m}\cdot\Theta_1 + \Theta_2 \vdash t \triangleright \mathbf{case}_{\mathsf{m}}\ (x_1, x_2) \mapsto u : \mathsf{U}} \qquad \textsc{Ty-term-PatE}\ \frac{\begin{array}{c}\Theta_1 \vdash t : !_{\mathsf{n}} \mathsf{T} \\ \Theta_2,\ x :_{\mathsf{m}\cdot\mathsf{n}} \mathsf{T} \vdash u : \mathsf{U}\end{array}}{\mathsf{m}\cdot\Theta_1 + \Theta_2 \vdash t \triangleright \mathbf{case}_{\mathsf{m}}\ \mathsf{E}_{\mathsf{n}}\ x \mapsto u : \mathsf{U}}$$

$$\textsc{Ty-term-Map}\ \frac{\Theta_1 \vdash t : \mathsf{U} \ltimes \mathsf{T} \qquad 1\!\uparrow\!\cdot\Theta_2,\ x :_{1\nu} \mathsf{T} \vdash t' : \mathsf{T}'}{\Theta_1 + \Theta_2 \vdash t \triangleright \mathbf{map}\ x \mapsto t' : \mathsf{U} \ltimes \mathsf{T}'} \qquad \textsc{Ty-term-ToA}\ \frac{\Theta \vdash u : \mathsf{U}}{\Theta \vdash \mathbf{to}_{\ltimes}\ u : \mathsf{U} \ltimes 1} \qquad \textsc{Ty-term-FromA}\ \frac{\Theta \vdash t : \mathsf{U} \ltimes (!_{1\infty} \mathsf{T})}{\Theta \vdash \mathbf{from}_{\ltimes}\ t : \mathsf{U} \otimes (!_{1\infty} \mathsf{T})}$$

$$\textsc{Ty-term-FillU}\ \frac{\Theta \vdash t : \lfloor_{\mathsf{n}} 1 \rfloor}{\Theta \vdash t \triangleleft () : 1} \qquad \textsc{Ty-term-FillL}\ \frac{\Theta \vdash t : \lfloor_{\mathsf{n}} \mathsf{T}_1 \oplus \mathsf{T}_2 \rfloor}{\Theta \vdash t \triangleleft \mathsf{Inl} : \lfloor_{\mathsf{n}} \mathsf{T}_1 \rfloor} \qquad \textsc{Ty-term-FillR}\ \frac{\Theta \vdash t : \lfloor_{\mathsf{n}} \mathsf{T}_1 \oplus \mathsf{T}_2 \rfloor}{\Theta \vdash t \triangleleft \mathsf{Inr} : \lfloor_{\mathsf{n}} \mathsf{T}_2 \rfloor} \qquad \textsc{Ty-term-FillP}\ \frac{\Theta \vdash t : \lfloor_{\mathsf{n}} \mathsf{T}_1 \otimes \mathsf{T}_2 \rfloor}{\Theta \vdash t \triangleleft (,) : \lfloor_{\mathsf{n}} \mathsf{T}_1 \rfloor \otimes \lfloor_{\mathsf{n}} \mathsf{T}_2 \rfloor}$$

$$\textsc{Ty-term-FillE}\ \frac{\Theta \vdash t : \lfloor_{\mathsf{n}} !_{\mathsf{n}'} \mathsf{T} \rfloor}{\Theta \vdash t \triangleleft \mathsf{E}_{\mathsf{n}'} : \lfloor_{\mathsf{n}'\cdot\mathsf{n}} \mathsf{T} \rfloor} \qquad \textsc{Ty-term-FillF}\ \frac{\Theta_1 \vdash t : \lfloor_{\mathsf{n}} \mathsf{T} \,{}_{\mathsf{m}}\!\to \mathsf{U} \rfloor \qquad \Theta_2,\ x :_{\mathsf{m}} \mathsf{T} \vdash u : \mathsf{U}}{\Theta_1 + (1\!\uparrow\!\cdot\mathsf{n})\cdot\Theta_2 \vdash t \triangleleft (\lambda x \,{}_{\mathsf{m}}\!\mapsto u) : 1} \qquad \textsc{Ty-term-FillComp}\ \frac{\Theta_1 \vdash t : \lfloor_{\mathsf{n}} \mathsf{U} \rfloor \qquad \Theta_2 \vdash t' : \mathsf{U} \ltimes \mathsf{T}}{\Theta_1 + (1\!\uparrow\!\cdot\mathsf{n})\cdot\Theta_2 \vdash t \triangleleft\!\bullet\, t' : \mathsf{T}}$$

6.3 Derived typing rules for syntactic sugar forms

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

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

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

Ty-sterm-FillLeaf
$$\frac{\Theta_1 \vdash t : \lfloor {}_n\mathsf{T} \rfloor \quad \Theta_2 \vdash t' : \mathsf{T}}{\Theta_1 + (1 \uparrow \cdot n) \cdot \Theta_2 \overset{s}{\vdash} t \triangleleft t' : 1}$$

Ty-sterm-Fun
$$\frac{\Theta_2,\ x :_m \mathsf{T} \vdash u : \mathsf{U}}{\Theta_2 \overset{s}{\vdash} \lambda x\ {}_m\!\mapsto u : \mathsf{T} \ {}_m\!\to \mathsf{U}}$$

Ty-sterm-Left
$$\frac{\Theta_2 \vdash t : \mathsf{T}_1}{\Theta_2 \overset{s}{\vdash} \mathsf{Inl}\, t : \mathsf{T}_1 \oplus \mathsf{T}_2}$$

Ty-sterm-Right
$$\frac{\Theta_2 \vdash t : \mathsf{T}_2}{\Theta_2 \overset{s}{\vdash} \mathsf{Inr}\, t : \mathsf{T}_1 \oplus \mathsf{T}_2}$$

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

Ty-sterm-Prod
$$\frac{\Theta_{21} \vdash t_1 : \mathsf{T}_1 \quad \Theta_{22} \vdash t_2 : \mathsf{T}_2}{\Theta_{21} + \Theta_{22} \overset{s}{\vdash} (t_1, t_2) : \mathsf{T}_1 \otimes \mathsf{T}_2}$$

7 EVALUATION CONTEXTS AND SEMANTICS

7.1 Evaluation contexts forms

ectx, c	::=		Evaluation context component
	|	$t'\ \square$	
	|	$\square\ v$	
	|	$\square \overset{\circ}{9}\ u$	
	|	$\square \rhd \mathbf{case}_m\ \{\mathsf{Inl}\ x_1 \mapsto u_1,\ \mathsf{Inr}\ x_2 \mapsto u_2\}$	
	|	$\square \rhd \mathbf{case}_m\ (x_1, x_2) \mapsto u$	
	|	$\square \rhd \mathbf{case}_m\ \mathsf{E}_n\ x \mapsto u$	
	|	$\square \rhd \mathbf{map}\ x \mapsto t'$	
	|	$\mathbf{to}_{\ltimes}\ \square$	
	|	$\mathbf{from}_{\ltimes}\ \square$	
	|	$\square \triangleleft ()$	
	|	$\square \triangleleft \mathsf{Inl}$	
	|	$\square \triangleleft \mathsf{Inr}$	
	|	$\square \triangleleft \mathsf{E}_m$	
	|	$\square \triangleleft (,)$	
	|	$\square \triangleleft (\lambda x\ {}_m\!\mapsto u)$	
	|	$\square \triangleleft\!\bullet\ t'$	
	|	$v \triangleleft\!\bullet\ \square$	
	|	${}^{\mathrm{op}}_{H}\langle v_2\ {}_{\wedge}\square \rangle$	Open ampar, binding hole names in the r
ectxs, C	::=		Evaluation context stack
	|	$\square$	Represent the empty stack / "identity" ev
	|	$C \circ c$	Push c on top of C
	|	$C[h :=_H v]$ M	Fill h in C with value v (that may contain

7.2 Typing of evaluation contexts and commands

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

$$\textsc{Ty-ectxs-Id}\ \frac{}{\dashv \Box : \mathsf{U}_0 \rightarrowtail \mathsf{U}_0} \qquad \textsc{Ty-ectxs-App-Foc1}\ \frac{\mathsf{m}\cdot\Delta_1,\ \Delta_2 \dashv C : \mathsf{U} \rightarrowtail \mathsf{U}_0 \quad \Delta_2 \vdash t' : \mathsf{T} \multimap_{\mathsf{m}} \mathsf{U}}{\Delta_1 \dashv C \circ (t'\ \Box) : \mathsf{T} \rightarrowtail \mathsf{U}_0}$$

$$\textsc{Ty-ectxs-App-Foc2}\ \frac{\mathsf{m}\cdot\Delta_1,\ \Delta_2 \dashv C : \mathsf{U} \rightarrowtail \mathsf{U}_0 \quad \Delta_1 \vdash v : \mathsf{T}}{\Delta_2 \dashv C \circ (\Box\ v) : (\mathsf{T} \multimap_{\mathsf{m}} \mathsf{U}) \rightarrowtail \mathsf{U}_0}$$

$$\textsc{Ty-ectxs-PatU-Foc}\ \frac{\Delta_1,\ \Delta_2 \dashv C : \mathsf{U} \rightarrowtail \mathsf{U}_0 \quad \Delta_2 \vdash u : \mathsf{U}}{\Delta_1 \dashv C \circ (\Box \fatsemi u) : 1 \rightarrowtail \mathsf{U}_0}$$

$$\textsc{Ty-ectxs-PatS-Foc}\ \frac{\mathsf{m}\cdot\Delta_1,\ \Delta_2 \dashv C : \mathsf{U} \rightarrowtail \mathsf{U}_0 \quad \Delta_2,\ x_1 :_{\mathsf{m}} \mathsf{T}_1 \vdash u_1 : \mathsf{U} \quad \Delta_2,\ x_2 :_{\mathsf{m}} \mathsf{T}_2 \vdash u_2 : \mathsf{U}}{\Delta_1 \dashv C \circ (\Box \rhd \mathbf{case}_{\mathsf{m}}\ \{\mathsf{Inl}\ x_1 \mapsto u_1\,,\ \mathsf{Inr}\ x_2 \mapsto u_2\}) : (\mathsf{T}_1 \oplus \mathsf{T}_2) \rightarrowtail \mathsf{U}_0}$$

$$\textsc{Ty-ectxs-PatP-Foc}\ \frac{\mathsf{m}\cdot\Delta_1,\ \Delta_2 \dashv C : \mathsf{U} \rightarrowtail \mathsf{U}_0 \quad \Delta_2,\ x_1 :_{\mathsf{m}} \mathsf{T}_1,\ x_2 :_{\mathsf{m}} \mathsf{T}_2 \vdash u : \mathsf{U}}{\Delta_1 \dashv C \circ (\Box \rhd \mathbf{case}_{\mathsf{m}}\ (x_1\,, x_2) \mapsto u) : (\mathsf{T}_1 \otimes \mathsf{T}_2) \rightarrowtail \mathsf{U}_0}$$

$$\textsc{Ty-ectxs-PatE-Foc}\ \frac{\mathsf{m}\cdot\Delta_1,\ \Delta_2 \dashv C : \mathsf{U} \rightarrowtail \mathsf{U}_0 \quad \Delta_2,\ x :_{\mathsf{m}\cdot\mathsf{m}'} \mathsf{T} \vdash u : \mathsf{U}}{\Delta_1 \dashv C \circ (\Box \rhd \mathbf{case}_{\mathsf{m}}\ \mathsf{E}_{\mathsf{m}'}\ x \mapsto u) : !_{\mathsf{m}'}\mathsf{T} \rightarrowtail \mathsf{U}_0} \qquad \textsc{Ty-ectxs-Map-Foc}\ \frac{\Delta_1,\ \Delta_2 \dashv C : \mathsf{U} \ltimes \mathsf{T}' \rightarrowtail \mathsf{U}_0 \quad 1\!\uparrow\cdot\Delta_2,\ x :_{1\nu} \mathsf{T} \vdash t' : \mathsf{T}'}{\Delta_1 \dashv C \circ (\Box \rhd \mathbf{map}\ x \mapsto t') : (\mathsf{U} \ltimes \mathsf{T}) \rightarrowtail \mathsf{U}_0}$$

$$\textsc{Ty-ectxs-ToA-Foc}\ \frac{\Delta \dashv C : (\mathsf{U} \ltimes 1) \rightarrowtail \mathsf{U}_0}{\Delta \dashv C \circ (\mathbf{to}_{\ltimes}\ \Box) : \mathsf{U} \rightarrowtail \mathsf{U}_0} \qquad \textsc{Ty-ectxs-FromA-Foc}\ \frac{\Delta \dashv C : (\mathsf{U} \otimes (!_{1\infty}\mathsf{T})) \rightarrowtail \mathsf{U}_0}{\Delta \dashv C \circ (\mathbf{from}_{\ltimes}\ \Box) : (\mathsf{U} \ltimes (!_{1\infty}\mathsf{T})) \rightarrowtail \mathsf{U}_0}$$

$$\textsc{Ty-ectxs-FillU-Foc}\ \frac{\Delta \dashv C : 1 \rightarrowtail \mathsf{U}_0}{\Delta \dashv C \circ (\Box \lhd ()) : \lfloor_{\mathsf{n}} 1 \rfloor \rightarrowtail \mathsf{U}_0} \qquad \textsc{Ty-ectxs-FillL-Foc}\ \frac{\Delta \dashv C : \lfloor_{\mathsf{n}} \mathsf{T}_1 \rfloor \rightarrowtail \mathsf{U}_0}{\Delta \dashv C \circ (\Box \lhd \mathsf{Inl}) : \lfloor_{\mathsf{n}} \mathsf{T}_1 \oplus \mathsf{T}_2 \rfloor \rightarrowtail \mathsf{U}_0}$$

$$\textsc{Ty-ectxs-FillR-Foc}\ \frac{\Delta \dashv C : \lfloor_{\mathsf{n}} \mathsf{T}_2 \rfloor \rightarrowtail \mathsf{U}_0}{\Delta \dashv C \circ (\Box \lhd \mathsf{Inr}) : \lfloor_{\mathsf{n}} \mathsf{T}_1 \oplus \mathsf{T}_2 \rfloor \rightarrowtail \mathsf{U}_0} \qquad \textsc{Ty-ectxs-FillP-Foc}\ \frac{\Delta \dashv C : (\lfloor_{\mathsf{n}} \mathsf{T}_1 \rfloor \otimes \lfloor_{\mathsf{n}} \mathsf{T}_2 \rfloor) \rightarrowtail \mathsf{U}_0}{\Delta \dashv C \circ (\Box \lhd (,)) : \lfloor_{\mathsf{n}} \mathsf{T}_1 \otimes \mathsf{T}_2 \rfloor \rightarrowtail \mathsf{U}_0}$$

$$\textsc{Ty-ectxs-FillE-Foc}\ \frac{\Delta \dashv C : \lfloor_{\mathsf{m}\cdot\mathsf{n}} \mathsf{T} \rfloor \rightarrowtail \mathsf{U}_0}{\Delta \dashv C \circ (\Box \lhd \mathsf{E}_{\mathsf{m}}) : \lfloor_{\mathsf{n}} !_{\mathsf{m}} \mathsf{T} \rfloor \rightarrowtail \mathsf{U}_0} \qquad \textsc{Ty-ectxs-FillF-Foc}\ \frac{(1\!\uparrow\cdot\mathsf{n})\cdot\Delta_1 \dashv C : 1 \rightarrowtail \mathsf{U}_0 \quad \Delta_2,\ x :_{\mathsf{m}} \mathsf{T} \vdash u : \mathsf{U}}{\Delta_1 \dashv C \circ (\Box \lhd (\lambda x\ {}_{\mathsf{m}}\!\mapsto u)) : \lfloor_{\mathsf{n}} \mathsf{T} \multimap_{\mathsf{m}} \mathsf{U} \rfloor \rightarrowtail \mathsf{U}_0}$$

$$\textsc{Ty-ectxs-FillComp-Foc1}\ \frac{\Delta_1,\ (1\uparrow\cdot n)\cdot\Delta_2 \dashv C : \mathsf{T} \rightarrowtail \mathsf{U}_0 \qquad \Delta_2 \vdash t' : \mathsf{U} \ltimes \mathsf{T}}{\Delta_1 \dashv C \circ (\square \triangleleft\!\bullet\, t') : \lfloor_n \mathsf{U} \rfloor \rightarrowtail \mathsf{U}_0}$$

$$\textsc{Ty-ectxs-FillComp-Foc2}\ \frac{\Delta_1,\ (1\uparrow\cdot n)\cdot\Delta_2 \dashv C : \mathsf{T} \rightarrowtail \mathsf{U}_0 \qquad \Delta_1 \vdash v : \lfloor_n \mathsf{U} \rfloor}{\Delta_2 \dashv C \circ (v \triangleleft\!\bullet\, \square) : \mathsf{U} \ltimes \mathsf{T} \rightarrowtail \mathsf{U}_0}$$

$$\textsc{Ty-ectxs-OpenAmpar-Foc}\ \frac{\begin{array}{c}\mathbf{hvars}(C)\ \#\#\ \mathbf{hvars}(\rightarrow^{-1}\Delta_3) \\ \texttt{LinOnly}\ \Delta_3 \\ \texttt{FinAgeOnly}\ \Delta_3 \\ \Delta_1,\ \Delta_2 \dashv C : (\mathsf{U} \ltimes \mathsf{T}') \rightarrowtail \mathsf{U}_0 \\ \Delta_2,\ \rightarrow^{-1}\Delta_3 \Vdash v_2 : \mathsf{U}\end{array}}{1\uparrow\cdot\Delta_1,\ \Delta_3 \dashv C \circ ({}^{\mathrm{op}}_{\mathbf{hvars}(\rightarrow^{-1}\Delta_3)}\langle v_2 \wedge \square\rangle) : \mathsf{T}' \rightarrowtail \mathsf{U}_0}$$

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

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

7.3 Small-step semantics

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

$$\textsc{Sem-App-Foc1}\ \frac{\texttt{NotVal}\ t}{C[t'\ t] \longrightarrow (C \circ (t'\ \square))[t]} \qquad \textsc{Sem-App-Unfoc1}\ \frac{}{(C \circ (t'\ \square))[v] \longrightarrow C[t'\ v]}$$

$$\textsc{Sem-App-Foc2}\ \frac{\texttt{NotVal}\ t'}{C[t'\ v] \longrightarrow (C \circ (\square\ v))[t']} \qquad \textsc{Sem-App-Unfoc2}\ \frac{}{(C \circ (\square\ v))[v'] \longrightarrow C[v'\ v]}$$

$$\textsc{Sem-App-Red}\ \frac{}{C[(\overset{v}{\lambda} x_{\,m} \mapsto u)\ v] \longrightarrow C[u[x := v]]} \qquad \textsc{Sem-PatU-Foc}\ \frac{\texttt{NotVal}\ t}{C[t \fatsemi u] \longrightarrow (C \circ (\square \fatsemi u))[t]}$$

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

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

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

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

Sem-PatR-Red

$$C[(\mathsf{Inr}\ v_2) \triangleright \mathbf{case}_{\mathsf{m}}\ \{\mathsf{Inl}\ x_1 \mapsto u_1\,,\ \mathsf{Inr}\ x_2 \mapsto u_2\}] \longrightarrow C[u_2[x_2 \coloneqq v_2]]$$

Sem-PatP-Foc

$$\frac{\mathtt{NotVal}\ t}{C[t \triangleright \mathbf{case}_{\mathsf{m}}\ (x_1\,,\ x_2) \mapsto u] \longrightarrow (C \circ (\Box \triangleright \mathbf{case}_{\mathsf{m}}\ (x_1\,,\ x_2) \mapsto u))[t]}$$

Sem-PatP-Unfoc

$$(C \circ (\Box \triangleright \mathbf{case}_{\mathsf{m}}\ (x_1\,,\ x_2) \mapsto u))[v] \longrightarrow C[v \triangleright \mathbf{case}_{\mathsf{m}}\ (x_1\,,\ x_2) \mapsto u]$$

Sem-PatP-Red

$$C[(v_1\,,\ v_2) \triangleright \mathbf{case}_{\mathsf{m}}\ (x_1\,,\ x_2) \mapsto u] \longrightarrow C[u[x_1 \coloneqq v_1][x_2 \coloneqq v_2]]$$

Sem-PatE-Foc

$$\frac{\mathtt{NotVal}\ t}{C[t \triangleright \mathbf{case}_{\mathsf{m}}\ \mathsf{E}_{\mathsf{n}}\ x \mapsto u] \longrightarrow (C \circ (\Box \triangleright \mathbf{case}_{\mathsf{m}}\ \mathsf{E}_{\mathsf{n}}\ x \mapsto u))[t]}$$

Sem-PatE-Unfoc

$$(C \circ (\Box \triangleright \mathbf{case}_{\mathsf{m}}\ \mathsf{E}_{\mathsf{n}}\ x \mapsto u))[v] \longrightarrow C[v \triangleright \mathbf{case}_{\mathsf{m}}\ \mathsf{E}_{\mathsf{n}}\ x \mapsto u]$$

Sem-PatE-Red

$$C[\mathsf{E}_{\mathsf{n}}\ v' \triangleright \mathbf{case}_{\mathsf{m}}\ \mathsf{E}_{\mathsf{n}}\ x \mapsto u] \longrightarrow C[u[x \coloneqq v']]$$

Sem-Map-Foc

$$\frac{\mathtt{NotVal}\ t}{C[t \triangleright \mathbf{map}\ x \mapsto t'] \longrightarrow (C \circ (\Box \triangleright \mathbf{map}\ x \mapsto t'))[t]}$$

Sem-Map-Unfoc

$$(C \circ (\Box \triangleright \mathbf{map}\ x \mapsto t'))[v] \longrightarrow C[v \triangleright \mathbf{map}\ x \mapsto t']$$

Sem-Map-Red-OpenAmpar-Foc

$$\frac{h' \;=\; \mathbf{max}(\mathbf{hvars}(C))+1}{C[_H\langle v_2 \mathbin{_\wedge} v_1\rangle \triangleright \mathbf{map}\ x \mapsto t'] \longrightarrow (C \circ ({}^{\mathrm{op}}_{H \doteq h'}\langle v_2[H \doteq h'] \mathbin{_\wedge} \Box\rangle))[t'[x \coloneqq v_1[H \doteq h']]]}$$

Sem-OpenAmpar-Unfoc

$$(C \circ {}^{\mathrm{op}}_{H}\langle v_2 \mathbin{_\wedge} \Box\rangle)[v_1] \longrightarrow C[_H\langle v_2 \mathbin{_\wedge} v_1\rangle]$$

Sem-ToA-Foc

$$\frac{\mathtt{NotVal}\ u}{C[\mathbf{to}_{\ltimes}\ u] \longrightarrow (C \circ (\mathbf{to}_{\ltimes}\ \Box))[u]}$$

Sem-ToA-Unfoc

$$(C \circ (\mathbf{to}_{\ltimes}\ \Box))[v_2] \longrightarrow C[\mathbf{to}_{\ltimes}\ v_2]$$

Sem-ToA-Red

$$C[\mathbf{to}_{\ltimes}\ v_2] \longrightarrow C[\{\,\}\langle v_2 \mathbin{_\wedge} ()\rangle]$$

Sem-FromA-Foc

$$\frac{\mathtt{NotVal}\ t}{C[\mathbf{from}_{\ltimes}\ t] \longrightarrow (C \circ (\mathbf{from}_{\ltimes}\ \Box))[t]}$$

Sem-FromA-Unfoc

$$(C \circ (\mathbf{from}_{\ltimes}\ \Box))[v] \longrightarrow C[\mathbf{from}_{\ltimes}\ v]$$

Sem-FromA-Red

$$C[\mathbf{from}_{\ltimes\ \{\}}\langle v_2 \wedge \mathsf{E}_{1\infty}\ v_1\rangle] \longrightarrow C[(v_2\,,\ \mathsf{E}_{1\infty}\ v_1)]$$

Sem-FillU-Foc

$$\frac{\mathsf{NotVal}\ t}{C[t \triangleleft ()] \longrightarrow (C \circ (\square \triangleleft ()))[t]}$$

Sem-FillU-Unfoc

$$(C \circ (\square \triangleleft ()))[v] \longrightarrow C[v \triangleleft ()]$$

Sem-FillU-Red

$$C[{\rightarrow}h \triangleleft ()] \longrightarrow C[h :=_{\{\}}\ ()][()]$$

Sem-FillL-Foc

$$\frac{\mathsf{NotVal}\ t}{C[t \triangleleft \mathsf{Inl}] \longrightarrow (C \circ (\square \triangleleft \mathsf{Inl}))[t]}$$

Sem-FillL-Unfoc

$$(C \circ (\square \triangleleft \mathsf{Inl}))[v] \longrightarrow C[v \triangleleft \mathsf{Inl}]$$

Sem-FillL-Red

$$\frac{h' = \mathbf{max}(\mathbf{hvars}(C) \cup \{h\}) + 1}{C[{\rightarrow}h \triangleleft \mathsf{Inl}] \longrightarrow C[h :=_{\{h'+1\}}\ \mathsf{Inl}\ (h'+1)][{\rightarrow}(h'+1)]}$$

Sem-FillR-Foc

$$\frac{\mathsf{NotVal}\ t}{C[t \triangleleft \mathsf{Inr}] \longrightarrow (C \circ (\square \triangleleft \mathsf{Inr}))[t]}$$

Sem-FillR-Unfoc

$$(C \circ (\square \triangleleft \mathsf{Inr}))[v] \longrightarrow C[v \triangleleft \mathsf{Inr}]$$

Sem-FillR-Red

$$\frac{h' = \mathbf{max}(\mathbf{hvars}(C) \cup \{h\}) + 1}{C[{\rightarrow}h \triangleleft \mathsf{Inr}] \longrightarrow C[h :=_{\{h'+1\}}\ \mathsf{Inr}\ (h'+1)][{\rightarrow}(h'+1)]}$$

Sem-FillE-Foc

$$\frac{\mathsf{NotVal}\ t}{C[t \triangleleft \mathsf{E}_m] \longrightarrow (C \circ (\square \triangleleft \mathsf{E}_m))[t]}$$

Sem-FillE-Unfoc

$$(C \circ (\square \triangleleft \mathsf{E}_m))[v] \longrightarrow C[v \triangleleft \mathsf{E}_m]$$

Sem-FillE-Red

$$\frac{h' = \mathbf{max}(\mathbf{hvars}(C) \cup \{h\}) + 1}{C[{\rightarrow}h \triangleleft \mathsf{E}_m] \longrightarrow C[h :=_{\{h'+1\}}\ \mathsf{E}_m\ (h'+1)][{\rightarrow}(h'+1)]}$$

Sem-FillP-Foc

$$\frac{\mathsf{NotVal}\ t}{C[t \triangleleft (,)] \longrightarrow (C \circ (\square \triangleleft (,)))[t]}$$

Sem-FillP-Unfoc

$$(C \circ (\square \triangleleft (,)))[v] \longrightarrow C[v \triangleleft (,)]$$

Sem-FillP-Red

$$\frac{h' = \mathbf{max}(\mathbf{hvars}(C) \cup \{h\}) + 1}{C[{\rightarrow}h \triangleleft (,)] \longrightarrow C[h :=_{\{h'+1,h'+2\}}\ ((h'+1)\,,\ (h'+2))][({\rightarrow}(h'+1)\,,\ {\rightarrow}(h'+2))]}$$

Sem-FillF-Foc

$$\frac{\mathsf{NotVal}\ t}{C[t \triangleleft (\lambda x_{\,m}{\mapsto}\ u)] \longrightarrow (C \circ (\square \triangleleft (\lambda x_{\,m}{\mapsto}\ u)))[t]}$$

Sem-FillF-Unfoc

$$(C \circ (\square \triangleleft (\lambda x_{\,m}{\mapsto}\ u)))[v] \longrightarrow C[v \triangleleft (\lambda x_{\,m}{\mapsto}\ u)]$$

Sem-FillF-Red

$$C[{\rightarrow}h \triangleleft (\lambda x_{\,m}{\mapsto}\ u)] \longrightarrow C[h :=_{\{\}}\ \lambda x_{\,m}{\mapsto}\ u][()]$$

Sem-FillComp-Foc1

$$\frac{\mathsf{NotVal}\ t}{C[t \triangleleft\!\bullet\ t'] \longrightarrow (C \circ (\square \triangleleft\!\bullet\ t'))[t]}$$

$$\textsc{Sem-FillComp-Unfoc1}\ \frac{}{(C \circ (\Box \blacktriangleleft\bullet t'))[v] \longrightarrow C[v \blacktriangleleft\bullet t']} \qquad \textsc{Sem-FillComp-Foc2}\ \frac{\texttt{NotVal}\ t'}{C[v \blacktriangleleft\bullet t'] \longrightarrow (C \circ (v \blacktriangleleft\bullet \Box))[t']}$$

$$\textsc{Sem-FillComp-Unfoc2}\ \frac{}{(C \circ (v \blacktriangleleft\bullet \Box))[v'] \longrightarrow C[v \blacktriangleleft\bullet v']} \qquad \textsc{Sem-FillComp-Red}\ \frac{h' = \mathbf{max}(\mathbf{hvars}(C) \cup \{h\}) + 1}{C[\to h \blacktriangleleft\bullet_H \langle v_2 \wedge v_1 \rangle] \longrightarrow C[h :=_{(H \doteq h')} v_2[H \doteq h']][v_1[H \doteq h']]}$$

8 PROOF OF TYPE SAFETY USING COQ PROOF ASSISTANT

- Not particularly elegant. Max number of goals observed 232 (solved by a single call to the `congruence` tactic). When you have a computer, brute force is a viable strategy. (in particular, no semiring formalisation, it was quicker to do directly)
- Rules generated by ott, same as in the article (up to some notational difference). Contexts are not generated purely by syntax, and are interpreted in a semantic domain (finite functions).
- Reasoning on closed terms avoids almost all complications on binder manipulation. Makes proofs tractable.
- Finite functions: making a custom library was less headache than using existing libraries (including `MMap`). Existing libraries don't provide some of the tools that we needed, but the most important factor ended up being the need for a modicum of dependency between key and value. There wasn't really that out there. Backed by actual functions for simplicity; cost: equality is complicated.
- Most of the proofs done by author with very little prior experience to Coq.
- Did proofs in Coq because context manipulations are tricky.
- Context sum made total by adding an extra invalid *mode* (rather than an extra context). It seems to be much simpler this way.
- It might be a good idea to provide statistics on the number of lemmas and size of Coq codebase.
- (possibly) renaming as permutation, inspired by nominal sets, make more lemmas don't require a condition (but some lemmas that wouldn't in a straight renaming do in exchange).
- (possibly) methodology: assume a lot of lemmas, prove main theorem, prove assumptions, some wrong, fix. A number of wrong lemma initially assumed, but replacing them by correct variant was always easy to fix in proofs.
- Axioms that we use and why (in particular setoid equality not very natural with ott-generated typing rules).
- Talk about the use and benefits of Copilot.

9 IMPLEMENTATION OF DESTINATION CALCULUS USING IN-PLACE MEMORY MUTATIONS

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

10 RELATED WORK

11 CONCLUSION AND FUTURE WORK

REFERENCES

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

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

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

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

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