Destination calculus

A linear λ−calculus for pure, functional memory updates

ARNAUD SPIWACK, Modus Create, France
THOMAS BAGREL, LORIA/Inria, France and Modus Create, 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, 20 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 $\to\mathsf{h}$, 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: ${}_{\{\mathsf{h}\}}\langle \mathsf{h} \mathbin{{}_\wedge} \to\mathsf{h}\rangle$. 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).

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

The destination-feeding primitive $\triangleleft\mathtt{Inl}$ 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 $\to\mathsf{h}'$ is returned to represented the (yet unspecified) payload for the `Inl` constructor.

This new destination $\to\mathsf{h}'$ is then passed to the destination-feeding primitive $\triangleleft()$, which writes a unit value to the hole pointed by $\to\mathsf{h}'$. 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:

$$
\begin{aligned}
t_1 :=_{\{\mathsf{h}_1\}}\ & \langle \mathsf{h}_1 \mathbin{{}_\wedge} \to\mathsf{h}_1\rangle \triangleright \mathtt{map}\ \mathsf{d}_1 \mapsto \\
& \mathsf{d}_1 \triangleleft (,) \triangleright \mathtt{case}_{1\nu}\ (\mathsf{d}_2 , \mathsf{d}_3) \mapsto \\
& \quad \mathsf{d}_2 \triangleleft (,) \triangleright \mathtt{case}_{1\nu}\ (\mathsf{d}_4 , \mathsf{d}_5) \mapsto \\
& \quad\quad \mathsf{d}_3 \triangleleft (,) \triangleright \mathtt{case}_{1\nu}\ (\mathsf{d}_6 , \mathsf{d}_7) \mapsto {}^{s}[\mathsf{d}_4, \mathsf{d}_5, \mathsf{d}_6, \mathsf{d}_7]
\end{aligned}
$$

evaluates to:

$${}_{\{\mathsf{h}_4,\mathsf{h}_5,\mathsf{h}_6,\mathsf{h}_7\}}\langle (((\mathsf{h}_4 , \mathsf{h}_5) , (\mathsf{h}_6 , \mathsf{h}_7)) \mathbin{{}_\wedge} {}^{s}[\to\mathsf{h}_4, \to\mathsf{h}_5, \to\mathsf{h}_6, \to\mathsf{h}_7]\rangle$$

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:

$$
\begin{aligned}
t_2 :=\ & t_1 \triangleright \mathtt{map}\ \mathsf{d} \mapsto \\
& \mathsf{d} \triangleright \mathtt{case}_{1\nu}\ {}^{s}[\mathsf{d}_4, \mathsf{d}_5, \mathsf{d}_6, \mathsf{d}_7] \mapsto \\
& \quad \mathsf{d}_5 \triangleleft \mathtt{Inr} \triangleleft () \,;\, \mathsf{d}_7 \triangleleft \mathtt{Inr} \triangleleft () \,;\, {}^{s}[\mathsf{d}_4, \mathsf{d}_6]
\end{aligned}
$$

reduces to

$${}_{\{\mathsf{h}_4,\mathsf{h}_6\}}\langle (((\mathsf{h}_4 , \mathtt{Inr}\ ()) , (\mathsf{h}_6 , \mathtt{Inr}\ ())) \mathbin{{}_\wedge} {}^{s}[\to\mathsf{h}_4, \to\mathsf{h}_6]\rangle$$

Now only left leaves are left unspecified, that's why the associated destinations $\to\mathsf{h}_4$ and $\to\mathsf{h}_6$ are the only left on the right side. We can finally feed them:

$$
\begin{aligned}
t_3 :=\ & t_2 \triangleright \mathtt{map}\ \mathsf{d} \mapsto \\
& \mathsf{d} \triangleright \mathtt{case}_{1\nu}\ {}^{s}[\mathsf{d}_4, \mathsf{d}_6] \mapsto \\
& \quad \mathsf{d}_4 \triangleleft \mathtt{Inl} \triangleleft () \,;\, \mathsf{d}_6 \triangleleft \mathtt{Inr} \triangleleft ()
\end{aligned}
$$

t_3 reduces to ${}_{\{\,\}}\langle (((\mathtt{Inl}\ () , \mathtt{Inr}\ ()) , (\mathtt{Inl}\ () , \mathtt{Inr}\ ())) \mathbin{{}_\wedge} ()\rangle$ 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.

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 $\mathsf{d} : \lfloor \mathsf{T}' \rfloor$ as linearly consumed too when it is fed to $\mathsf{dd} : \lfloor\lfloor \mathsf{T}' \rfloor\rfloor$ in $\mathsf{dd} \triangleleft \mathsf{d}$.

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 $\mathtt{alloc'} : (\lfloor \mathsf{T} \rfloor \,{}_1\!\!\longrightarrow 1) \,{}_1\!\!\longrightarrow \mathsf{T}$. 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:

$\mathtt{alloc'}\ (\lambda\, \mathsf{d} \,{}_1\!\!\mapsto\ t) \longrightarrow \mathtt{withTmpStore}\ \{\mathsf{h} := \_\}\ \mathtt{do}\ \{t[\mathsf{d} := \rightarrow\!\mathsf{h}]\ ;\ \mathtt{deref}\ \rightarrow\!\mathsf{h}\}$

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

$\mathtt{alloc'}\ (\lambda\, \mathsf{d} \,{}_1\!\!\mapsto\ \mathsf{d} \triangleleft \mathtt{Inl} \triangleleft ())$
$\longrightarrow\ \mathtt{withTmpStore}\ \{\mathsf{h} := \_\}\ \mathtt{do}\ \{\rightarrow\!\mathsf{h} \triangleleft \mathtt{Inl} \triangleleft ()\ ;\ \mathtt{deref}\ \rightarrow\!\mathsf{h}\}$
$\longrightarrow\ \mathtt{withTmpStore}\ \{\mathsf{h} := \mathtt{Inl}\ ()\}\ \mathtt{do}\ \{\mathtt{deref}\ \rightarrow\!\mathsf{h}\}$
$\longrightarrow\ \mathtt{Inl}\ ()$

However this falls short when calls to `alloc'` are nested in the following way (where $\mathsf{dd} : \lfloor\lfloor 1 \rfloor\rfloor$ and $\mathsf{d} : \lfloor 1 \rfloor$):

$\mathtt{alloc'}\ (\lambda\, \mathsf{dd} \,{}_1\!\!\mapsto\ \mathtt{alloc'}\ (\lambda\, \mathsf{d} \,{}_1\!\!\mapsto\ \mathsf{dd} \triangleleft \mathsf{d}))$
$\longrightarrow\ \mathtt{withTmpStore}\ \{\mathsf{hd} := \_\}\ \mathtt{do}\ \{\mathtt{alloc'}\ (\lambda\, \mathsf{d} \,{}_1\!\!\mapsto\ \rightarrow\!\mathsf{hd} \triangleleft \mathsf{d})\ ;\ \mathtt{deref}\ \rightarrow\!\mathsf{hd}\}$
$\longrightarrow\ \mathtt{withTmpStore}\ \{\mathsf{hd} := \_\}\ \mathtt{do}\ \{\mathtt{withTmpStore}\ \{\mathsf{h} := \_\}\ \mathtt{do}\ \{\rightarrow\!\mathsf{hd} \triangleleft \rightarrow\!\mathsf{h}\ ;\ \mathtt{deref}\ \rightarrow\!\mathsf{h}\}\ ;\ \mathtt{deref}\ \rightarrow\!\mathsf{hd}\}$
$\longrightarrow\ \mathtt{withTmpStore}\ \{\mathsf{hd} := \rightarrow\!\mathsf{h}\}\ \mathtt{do}\ \{\mathtt{withTmpStore}\ \{\mathsf{h} := \_\}\ \mathtt{do}\ \{\mathtt{deref}\ \rightarrow\!\mathsf{h}\}\ ;\ \mathtt{deref}\ \rightarrow\!\mathsf{hd}\}$

The original term $\mathtt{alloc'}\ (\lambda\, \mathsf{dd} \,{}_1\!\!\mapsto\ \mathtt{alloc'}\ (\lambda\, \mathsf{d} \,{}_1\!\!\mapsto\ \mathsf{dd} \triangleleft \mathsf{d}))$ 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 $\rightarrow\!\mathsf{h}$ 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 `withTmpStore {h := _} do {deref →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 $\mathbb{I}$ for the effect of feeding a destination[1], as long as destinations of type $\lfloor\,\mathbb{I}\,\rfloor$ are allowed. And forbidding destinations of type $\lfloor\,\mathbb{I}\,\rfloor$ gets annoying very fast: typing rules are no longer as simple, parametric polymorphism is not really possible for most functions without a system of constraints to ensure the absence of $\mathbb{I}$ in anything we want to store, etc.

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:

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

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

Typing a destination $\rightarrow\mathsf{h}$ requires $\rightarrow\mathsf{h}$ 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 $\rightarrow\mathsf{hd} \triangleleft \rightarrow\mathsf{h}$ where $\rightarrow\mathsf{hd}$ originates from the context parent to the one of $\rightarrow\mathsf{h}$.

```
type rec Int ≜ 1⊕Int
operator zero : Int
         zero ≜ Inl ()
operator succ : Int ₁ν⟶ Int
         succ x ≜ ˢInr x
type rec List T ≜ 1⊕(T⊗(List T))
operator ◁[] : ⌊ₙ List T⌋ ₁ν⟶ 1
         d ◁ [] ≜ d ◁ Inl ◁ ()
operator ◁(:) : ⌊ₙ List T⌋ ₁ν⟶ ⌊ₙ T⌋⊗⌊ₙ List T⌋
         d ◁ (:) ≜ d ◁ Inr ◁ (,)
type     DList T ≜ (List T) ⋉ (⌊₁ν List T⌋)
operator append : DList T ₁ν⟶ T ₁ν⟶ DList T
         x append y ≜ x ▷ map d⟼ d ◁ (:) ▷ case₁ν
                        (dh, dt)⟼ dh ◁ y ; dt
operator concat : DList T ₁ν⟶ DList T ₁ν⟶ DList T
         x concat x′ ≜ x ▷ map d⟼ d ◁• x′
operator toₗᵢₛₜ : DList T ₁ν⟶ List T
         toₗᵢₛₜ x ≜ from′⋉ (x ▷ map d⟼ d ◁ [])
type     Queue T ≜ (List T)⊗(DList T)
operator singleton : T ₁ν⟶ Queue T
         singleton x ≜ ˢ(ˢInr ˢ(x, Inl ()), alloc)
operator enqueue : Queue T ₁ν⟶ T ₁ν⟶ Queue T
         x enqueue y ≜ x ▷ case₁ν (x₁, x₂)⟼ ˢ(x₁, x₂ append y)
operator dequeue : Queue T ₁ν⟶ 1⊕(T⊗Queue T)
         dequeue x ≜ x ▷ case₁ν
                       (x, y)⟼ x ▷ case₁ν{
                         Inl un ⟼ un ; toₗᵢₛₜ y ▷ case₁ν{
                           Inl un ⟼ ˢInl un,
                           Inr y′ ⟼ y′ ▷ case₁ν
                             (y′₁, y′₂)⟼ ˢInr ˢ(y′₁, ˢ(y′₂, alloc))
                         },
                         Inr x′ ⟼ x′ ▷ case₁ν
                           (x′₁, y′₁)⟼ ˢInr ˢ(x′₁, ˢ(y′₁, y))
                       }
type     Tree T ≜ 1⊕(T⊗((Tree T)⊗(Tree T)))
operator ◁Nil : ⌊ₙ Tree T⌋ ₁ν⟶ 1
         d ◁ Nil ≜ d ◁ Inl ◁ ()
operator ◁Node : ⌊ₙ Tree T⌋ ₁ν⟶ ⌊ₙ T⌋⊗(⌊ₙ Tree T⌋⊗⌊ₙ Tree T⌋)
         d ◁ Node ≜ d ◁ Inr ◁ (,) ▷ case₁ν (dn, dlr)⟼ ˢ(dn, dlr ◁ (,))
```

Fig. 1. Boilerplate for breadth-first tree traversal

```
operator rec
  go : ((!1∞S) 1ν⟶ T1 1ν⟶ (!1∞S)⊗T2) ων⟶ (!1∞S) 1ν⟶ Queue (Tree T1⊗⌊1ν Tree T2⌋) 1ν⟶ (!1∞S)
  go f st q ≜ dequeue q ▷ case1ν{
                Inl un ↦ un ; st,
                Inr x ↦ x ▷ case1ν
                  (x′, q′)↦ x′ ▷ case1ν
                    (tr, dtr)↦ tr ▷ case1ν{
                      Inl un ↦ un ; dtr ◂ Nil ; go f st q′,
                      Inr y ↦ y ▷ case1ν
                        (y′, y″)↦ y″ ▷ case1ν
                          (trl, trr)↦ dtr ◂ Node ▷ case1ν
                            (dn, dlr)↦ dlr ▷ case1ν
                              (dl, dr)↦ f st y′ ▷ case1ν
                                (st′, y″)↦
                                  dn ◂ y″ ; go f st′ (q′ enqueue ˢ(trl, dl) enqueue ˢ(trr, dr))
                    }
              }
operator
  mapAccumBFS : ((!1∞S) 1ν⟶ T1 1ν⟶ (!1∞S)⊗T2) ων⟶ (!1∞S) 1ν⟶ Tree T1 1ν⟶ Tree T2⊗(!1∞S)
  mapAccumBFS f st tr ≜ from′ω (alloc ▷ map dtr↦ go f st (singleton ˢ(tr, dtr)))
operator
  relabelDPS : Tree 1 1ν⟶ (Tree Int)⊗Int
  relabelDPS tr ≜ mapAccumBFS
                    (ˢλ ex 1ν↦ ˢλ un 1ν↦ un ; ex ▷ case1ν
                      E1∞ ex′ ↦ ex′ ▷ case1∞
                        Eων st ↦ ˢ(ˢE1∞ (ˢEων (succ st)), st))
                    (ˢE1∞ (ˢEων (succ zero)))
                    tr
```

Fig. 2. Breadth-first tree traversal in destination-passing style

4 (UPDATED) BREADTH-FIRST TREE TRAVERSAL

5 LANGUAGE SYNTAX

5.1 Introducing the *ampar*

Minamide's work[3] is the earliest record we could find of a functional calculus integrating the idea of incomplete data structures (structures with holes) that exist as first class values and can be interacted with by the user.

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

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

[1]This type 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.

Now, in classical linear logic, we know we can transform linear implication $\mathtt{T}_1 \multimap \mathtt{T}_2$ into $\mathtt{T}_1{}^{\perp} \parr \mathtt{T}_2$. Doing so for the *wand* type $(\mathtt{T}_1, \mathtt{T}_2)\mathtt{hfun}$ or $\mathtt{T}_1 \mathrel{-\!\!*} \mathtt{T}_2$ gives $\lfloor \mathtt{T}_1 \rfloor \mathbin{\widehat{\parr}} \mathtt{T}_2$, 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 $\lfloor \mathtt{T}_1 \rfloor$ of *sink* or *destination* of $\mathtt{T}_1$ 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 $\mathtt{T}_1 \parr \mathtt{T}_2$, we can free up $\mathtt{T}_1$ by providing an eliminator of $\mathtt{T}_2$, or free up $\mathtt{T}_2$ by providing an eliminator of $\mathtt{T}_1$. The eliminator of $\mathtt{T}$ can be $\mathtt{T}^{\perp}$, or $\mathtt{T}^{\perp^{-1}} = \mathtt{T}'$ if $\mathtt{T}$ is already of the form $\mathtt{T}'^{\perp}$. In a classical setting, thanks to the involutive nature of negation $\bullet^{\perp}$, the two potential forms of the eliminator of $\mathtt{T}$ 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 $\rightarrow\mathtt{h}' : \lfloor\lfloor \mathtt{T} \rfloor\rfloor$, we know that some structure is expecting to store a destination of type $\lfloor \mathtt{T} \rfloor$. 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 $\mathtt{h} : \mathtt{T}$ and its linked destination $\rightarrow\mathtt{h} : \lfloor \mathtt{T} \rfloor$, and write $\rightarrow\mathtt{h}$ to the memory slot pointed to by $\rightarrow\mathtt{h}'$, then we can get back a value of type $\mathtt{T}$ at $\mathtt{h}$ if ever the structure pointed to by $\rightarrow\mathtt{h}'$ 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 $\lfloor \mathtt{T} \rfloor$, both storing it through a destination of destination $\lfloor\lfloor \mathtt{T} \rfloor\rfloor$ or using it to store a value of type $\mathtt{T}$ 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 $\lfloor \mathtt{T} \rfloor$ originating from $\mathtt{T} \mathbin{\widehat{\parr}} \lfloor \mathtt{T} \rfloor$ (through a destination of destination of type $\lfloor\lfloor \mathtt{T} \rfloor\rfloor$) should not allow to free up the $\mathtt{T}$, 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 $\mathtt{T}$ and and inert value of type $\mathtt{T}$ 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 $\mathtt{x}, \mathtt{y}$, and hole names, $\mathtt{h}, \mathtt{h}_1, \mathtt{h}_2$ which represents the identifier or address of a memory cell that hasn't been written to yet.

`var, x, y, d, dd, un, ex, st, tr, trl, trr, dtr, f, dh, dt, dn, dlr, dl, dr, q` Variable names

hvar, h, hd	::=			Hole (or destination) name, represented by a natural number
	|	h+h′	M	
	|	h[H≟h′]	M	Shift by h′ if h ∈ H
	|	*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 $\mathtt{H}, \mathtt{H}_1, \mathtt{H}_2 \ldots$.

hvars, H	::=			Set of hole names
	|	$\{\mathtt{h}_1, .., \mathtt{h}_k\}$		
	|	$\mathtt{H}_1 \cup \mathtt{H}_2$	M	Union of sets
	|	H≟h′	M	Shift all names from H by h′.
	|	*hvars*(Γ)	M	Hole names bound by the typing context Γ
	|	*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≟h′. We also define a conditional shift operation [H≟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 $_{\{\mathsf{h}\}}\langle \mathsf{h}\ {}_{\wedge}{\to}\mathsf{h}\rangle$ 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
	|	$t\,;u$		Pattern-match on unit
	|	$t \triangleright \mathsf{case}_{\mathsf{m}}\,\{\mathtt{Inl}\ \mathsf{x}_1 \mapsto u_1\,,\ \mathtt{Inr}\ \mathsf{x}_2 \mapsto u_2\}$		Pattern-match on sum
	|	$t \triangleright \mathsf{case}_{\mathsf{m}}\,(\mathsf{x}_1\,,\mathsf{x}_2) \mapsto u$		Pattern-match on product
	|	$t \triangleright \mathsf{case}_{\mathsf{m}}\,\mathsf{E}_{\mathsf{n}}\,\mathsf{x} \mapsto u$		Pattern-match on exponential
	|	$t \triangleright \mathtt{map}\ \mathsf{x} \mapsto t'$		Map over the right side of ampar
	|	$\mathtt{to}_{\ltimes}\ u$		Wrap into a trivial ampar
	|	$\mathtt{from}_{\ltimes}\ t$		Convert ampar to a pair
	|	$t \triangleleft ()$		Fill destination with unit
	|	$t \triangleleft \mathtt{Inl}$		Fill destination with left variant
	|	$t \triangleleft \mathtt{Inr}$		Fill destination with right variant
	|	$t \triangleleft \mathsf{E}_{\mathsf{m}}$		Fill destination with exponential const
	|	$t \triangleleft (,)$		Fill destination with product construct
	|	$t \triangleleft (\lambda\,\mathsf{x}\ {}_{\mathsf{m}}{\mapsto}\ u)$		Fill destination with function
	|	$t \triangleleft\!\bullet\ t'$		Fill destination with root of other amp
	|	$t[\mathsf{x} \coloneqq v]$	M	

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

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 $!_{\omega\mathsf{a}}(\mathsf{T}_1 \otimes \mathsf{T}_2) \simeq (!_{\omega\mathsf{a}}\mathsf{T}_1) \otimes (!_{\omega\mathsf{a}}\mathsf{T}_2)$, 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.

$\texttt{to}_{\ltimes}$ 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 $\texttt{to}_{\ltimes}$ to the mix, we can compose an *ampar* with a normal (completed) structure (see the sugar operator FillLeaf ($\triangleleft$) in Section 5.4).

$\texttt{from}_{\ltimes}$ is used to convert an *ampar* to a pair, when the right side of the *ampar* is an exponential of the form $\texttt{E}_{1\infty}\ v$. 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. $\texttt{from}_{\ltimes}$ is in particular used to extract a structure from its *ampar* building shell when it is complete (see the sugar operator $\texttt{from}'_{\ltimes}$ in Section 5.4).

The remaining term operators $\triangleleft()$, $\triangleleft\texttt{Inl}$, $\triangleleft\texttt{Inr}$, $\triangleleft\,\texttt{E}_{\texttt{m}}$, $\triangleleft(,)$, $\triangleleft(\lambda\,\texttt{x}_{\texttt{m}}\!\mapsto\ u)$ 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.

$\texttt{from}_{\ltimes}{}'$ is a simpler variant of $\texttt{from}_{\ltimes}$ that allows to extract the right side of an ampar when the left side has been fully consumed. We implement it in terms of $\texttt{from}_{\ltimes}$ 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
	|	$\texttt{alloc}$	M	Evaluate to a fresh new ampar
	|	$t \triangleleft t'$	M	Fill destination with supplied term
	|	$\texttt{from}'_{\ltimes}\ t$	M	Extract left side of ampar when right side is unit
	|	$^{\texttt{s}}\lambda\,\texttt{x}_{\texttt{m}}\!\mapsto\ u$	M	Allocate function
	|	$^{\texttt{s}}\texttt{Inl}\ t$	M	Allocate left variant
	|	$^{\texttt{s}}\texttt{Inr}\ t$	M	Allocate right variant
	|	$^{\texttt{s}}\texttt{E}_{\texttt{m}}\ t$	M	Allocate exponential
	|	$^{\texttt{s}}(t_1\,,\,t_2)$	M	Allocate product

$\mathtt{alloc}$	$\triangleq$	$\{1\}\langle 1 \wedge \to 1\rangle$	$t \triangleleft t'$ $\triangleq$ $t \triangleleft\!\bullet (\mathtt{to}_{\ltimes}\ t')$
$\mathtt{from}'_{\ltimes}\ t$	$\triangleq$	$(\mathtt{from}_{\ltimes}\ (t \triangleright \mathtt{map}\ \mathtt{un} \mapsto \mathtt{un}\,;\mathtt{E}_{1\infty}\ ()))\ \triangleright \mathtt{case}_{1\nu}$ $(\mathtt{st}, \mathtt{ex}) \mapsto \mathtt{ex} \triangleright \mathtt{case}_{1\nu}$ $\mathtt{E}_{1\infty}\ \mathtt{un} \mapsto \mathtt{un}\,;\mathtt{st}$	$^{s}\lambda\, \mathtt{x}_{\mathtt{m}} \mapsto\ u$ $\triangleq$ $\mathtt{from}'_{\ltimes}\ ($ $\mathtt{alloc} \triangleright \mathtt{map}\ \mathtt{d} \mapsto$ $\mathtt{d} \triangleleft (\lambda\, \mathtt{x}_{\mathtt{m}} \mapsto\ u)$ $)$
$^{s}\mathtt{Inl}\ t$	$\triangleq$	$\mathtt{from}'_{\ltimes}\ ($ $\mathtt{alloc} \triangleright \mathtt{map}\ \mathtt{d} \mapsto$ $\mathtt{d} \triangleleft \mathtt{Inl} \triangleleft t$ $)$	$^{s}\mathtt{Inr}\ t$ $\triangleq$ $\mathtt{from}'_{\ltimes}\ ($ $\mathtt{alloc} \triangleright \mathtt{map}\ \mathtt{d} \mapsto$ $\mathtt{d} \triangleleft \mathtt{Inr} \triangleleft t$ $)$
$^{s}\mathtt{E}_{\mathtt{m}}\ t$	$\triangleq$	$\mathtt{from}'_{\ltimes}\ ($ $\mathtt{alloc} \triangleright \mathtt{map}\ \mathtt{d} \mapsto$ $\mathtt{d} \triangleleft \mathtt{E}_{\mathtt{m}} \triangleleft t$ $)$	$^{s}(t_1, t_2)$ $\triangleq$ $\mathtt{from}'_{\ltimes}\ ($ $\mathtt{alloc} \triangleright \mathtt{map}\ \mathtt{d} \mapsto$ $(\mathtt{d} \triangleleft (,)) \triangleright \mathtt{case}_{1\nu}$ $(\mathtt{d}_1, \mathtt{d}_2) \mapsto \mathtt{d}_1 \triangleleft t_1\,;\mathtt{d}_2 \triangleleft t_2$ $)$

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
	|	$U \invamp T$	Ampar
	|	$T \multimap_m U$	Function
	|	$\lfloor_m T \rfloor$	Destination
mode, m, n	::=		Mode (Semiring)
	|	pa	Pair of a multiplicity and age
	|	💀	Error case (incompatible types, multiplicities, or ages)
mul, p	::=		Multiplicity (Semiring, first component of 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
	|	$\Gamma[H \pm h']$ 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)*

Ty-val-Hole
$$\frac{}{\mathsf{h} :_{1\nu} \mathsf{T} \Vdash \mathsf{h} : \mathsf{T}}$$

Ty-val-Dest
$$\frac{}{\to\mathsf{h} :_{1\nu} \lfloor_{\mathsf{n}} \mathsf{T} \rfloor \Vdash \to\mathsf{h} : \lfloor_{\mathsf{n}} \mathsf{T} \rfloor}$$

Ty-val-Unit
$$\frac{}{\Vdash () : 1}$$

Ty-val-Fun
$$\frac{\Delta,\ \mathsf{x} :_{\mathsf{m}} \mathsf{T} \vdash u : \mathsf{U}}{\Delta \Vdash \lambda\, \mathsf{x}\ {}_{\mathsf{m}}\!\mapsto\ u : \mathsf{T} \multimap_{\mathsf{m}} \mathsf{U}}$$

Ty-val-Left
$$\frac{\Gamma \Vdash v_1 : \mathsf{T}_1}{\Gamma \Vdash \mathtt{Inl}\ v_1 : \mathsf{T}_1 \oplus \mathsf{T}_2}$$

Ty-val-Right
$$\frac{\Gamma \Vdash v_2 : \mathsf{T}_2}{\Gamma \Vdash \mathtt{Inr}\ v_2 : \mathsf{T}_1 \oplus \mathsf{T}_2}$$

Ty-val-Prod
$$\frac{\Gamma_1 \Vdash v_1 : \mathsf{T}_1 \qquad \Gamma_2 \Vdash v_2 : \mathsf{T}_2}{\Gamma_1 + \Gamma_2 \Vdash (v_1, v_2) : \mathsf{T}_1 \otimes \mathsf{T}_2}$$

Ty-val-Exp
$$\frac{\Gamma \Vdash v' : \mathsf{T}}{\mathsf{n}\cdot\Gamma \Vdash \mathsf{E}_{\mathsf{n}}\ v' : !_{\mathsf{n}} \mathsf{T}}$$

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

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

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

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

Ty-term-App
$$\frac{\Theta_1 \vdash t : \mathsf{T} \qquad \Theta_2 \vdash t' : \mathsf{T} \multimap_{\mathsf{m}} \mathsf{U}}{\mathsf{m}\cdot\Theta_1 + \Theta_2 \vdash t'\ t : \mathsf{U}}$$

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

Ty-term-PatS
$$\frac{\begin{array}{c}\Theta_1 \vdash t : \mathsf{T}_1 \oplus \mathsf{T}_2 \\ \Theta_2,\ \mathsf{x}_1 :_{\mathsf{m}} \mathsf{T}_1 \vdash u_1 : \mathsf{U} \\ \Theta_2,\ \mathsf{x}_2 :_{\mathsf{m}} \mathsf{T}_2 \vdash u_2 : \mathsf{U}\end{array}}{\mathsf{m}\cdot\Theta_1 + \Theta_2 \vdash t \rhd \mathsf{case}_{\mathsf{m}}\,\{\mathtt{Inl}\ \mathsf{x}_1 \mapsto u_1,\ \mathtt{Inr}\ \mathsf{x}_2 \mapsto u_2\} : \mathsf{U}}$$

Ty-term-PatP
$$\frac{\Theta_1 \vdash t : \mathsf{T}_1 \otimes \mathsf{T}_2 \qquad \Theta_2,\ \mathsf{x}_1 :_{\mathsf{m}} \mathsf{T}_1,\ \mathsf{x}_2 :_{\mathsf{m}} \mathsf{T}_2 \vdash u : \mathsf{U}}{\mathsf{m}\cdot\Theta_1 + \Theta_2 \vdash t \rhd \mathsf{case}_{\mathsf{m}}\,(\mathsf{x}_1, \mathsf{x}_2) \mapsto u : \mathsf{U}}$$

Ty-term-PatE
$$\frac{\Theta_1 \vdash t : !_{\mathsf{n}} \mathsf{T} \qquad \Theta_2,\ \mathsf{x} :_{\mathsf{m}\cdot\mathsf{n}} \mathsf{T} \vdash u : \mathsf{U}}{\mathsf{m}\cdot\Theta_1 + \Theta_2 \vdash t \rhd \mathsf{case}_{\mathsf{m}}\ \mathsf{E}_{\mathsf{n}}\ \mathsf{x} \mapsto u : \mathsf{U}}$$

Ty-term-Map
$$\frac{\Theta_1 \vdash t : \mathsf{U} \ltimes \mathsf{T} \qquad 1\!\uparrow\!\cdot\Theta_2,\ \mathsf{x} :_{1\nu} \mathsf{T} \vdash t' : \mathsf{T}'}{\Theta_1 + \Theta_2 \vdash t \rhd \mathsf{map}\ \mathsf{x} \mapsto t' : \mathsf{U} \ltimes \mathsf{T}'}$$

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

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

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

Ty-term-FillL
$$\frac{\Theta \vdash t : \lfloor_{\mathsf{n}} \mathsf{T}_1 \oplus \mathsf{T}_2 \rfloor}{\Theta \vdash t \triangleleft \mathtt{Inl} : \lfloor_{\mathsf{n}} \mathsf{T}_1 \rfloor}$$

Ty-term-FillR
$$\frac{\Theta \vdash t : \lfloor_{\mathsf{n}} \mathsf{T}_1 \oplus \mathsf{T}_2 \rfloor}{\Theta \vdash t \triangleleft \mathtt{Inr} : \lfloor_{\mathsf{n}} \mathsf{T}_2 \rfloor}$$

Ty-term-FillP
$$\frac{\Theta \vdash t : \lfloor_{\mathsf{n}} \mathsf{T}_1 \otimes \mathsf{T}_2 \rfloor}{\Theta \vdash t \triangleleft (,) : \lfloor_{\mathsf{n}} \mathsf{T}_1 \rfloor \otimes \lfloor_{\mathsf{n}} \mathsf{T}_2 \rfloor}$$

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

Ty-term-FillF
$$\frac{\Theta_1 \vdash t : \lfloor_{\mathsf{n}} \mathsf{T} \multimap_{\mathsf{m}} \mathsf{U} \rfloor \qquad \Theta_2,\ \mathsf{x} :_{\mathsf{m}} \mathsf{T} \vdash u : \mathsf{U}}{\Theta_1 + (1\!\uparrow\!\cdot\mathsf{n})\cdot\Theta_2 \vdash t \triangleleft (\lambda\, \mathsf{x}\ {}_{\mathsf{m}}\!\mapsto\ u) : 1}$$

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

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

Ty-sterm-Alloc
$$\frac{\texttt{DisposableOnly}\ \Theta}{\Theta \ {}^{s}\!\vdash\ \texttt{alloc} : \mathsf{T} \ltimes (\lfloor_{1_\nu}\, \mathsf{T}\rfloor)}$$

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

Ty-sterm-FillLeaf
$$\frac{\Theta_1 \vdash t : \lfloor_{\mathsf{n}}\, \mathsf{T}\rfloor \qquad \Theta_2 \vdash t' : \mathsf{T}}{\Theta_1 + (1\uparrow\cdot \mathsf{n})\cdot\Theta_2 \ {}^{s}\!\vdash\ t \triangleleft t' : 1}$$

Ty-sterm-Fun
$$\frac{\Theta_2,\ \mathsf{x} :_{\mathsf{m}} \mathsf{T} \vdash u : \mathsf{U}}{\Theta_2 \ {}^{s}\!\vdash\ {}^{s}\!\lambda\, \mathsf{x}_{\,\mathsf{m}}\!\mapsto\ u : \mathsf{T} \to_{\mathsf{m}} \mathsf{U}}$$

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

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

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

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

7 EVALUATION CONTEXTS AND SEMANTICS

7.1 Evaluation contexts forms

ectx, c	::=		Evaluation context component
	|	$t'\ \square$	
	|	$\square\ v$	
	|	$\square\ ; u$	
	|	$\square \rhd \texttt{case}_{\mathsf{m}}\ \{\texttt{Inl}\ \mathsf{x}_1 \mapsto u_1\,,\ \texttt{Inr}\ \mathsf{x}_2 \mapsto u_2\}$	
	|	$\square \rhd \texttt{case}_{\mathsf{m}}\ (\mathsf{x}_1\,,\, \mathsf{x}_2) \mapsto u$	
	|	$\square \rhd \texttt{case}_{\mathsf{m}}\ \mathsf{E}_{\mathsf{n}}\ \mathsf{x} \mapsto u$	
	|	$\square \rhd \texttt{map}\ \mathsf{x} \mapsto t'$	
	|	$\texttt{to}_{\ltimes}\ \square$	
	|	$\texttt{from}_{\ltimes}\ \square$	
	|	$\square \triangleleft ()$	
	|	$\square \triangleleft \texttt{Inl}$	
	|	$\square \triangleleft \texttt{Inr}$	
	|	$\square \triangleleft \mathsf{E}_{\mathsf{m}}$	
	|	$\square \triangleleft (,)$	
	|	$\square \triangleleft (\lambda\, \mathsf{x}_{\,\mathsf{m}}\!\mapsto\ u)$	
	|	$\square \triangleleft\!\bullet\ t'$	
	|	$v \triangleleft\!\bullet\ \square$	
	|	$\overset{\mathsf{op}}{\underset{\mathsf{H}}{}}\langle v_2\ {}_{\wedge}\square\rangle$	Open ampar, binding hole names in the
ectxs, C	::=		Evaluation context stack
	|	$\square$	Represent the empty stack / "identity" c
	|	$C \circ c$	Push c on top of C
	|	$C[\mathsf{h} :=_{\mathsf{H}}\ v]$	M Fill h in C with value v (that may conta

7.2 Typing of evaluation contexts and commands

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\textsc{Sem-PatS-Foc}\ \frac{\texttt{NotVal}\ t}{C[t \rhd \mathtt{case}_{\mathtt{m}}\,\{\mathtt{Inl}\,\mathtt{x}_1 \mapsto u_1,\ \mathtt{Inr}\,\mathtt{x}_2 \mapsto u_2\}] \longrightarrow (C \circ (\square \rhd \mathtt{case}_{\mathtt{m}}\,\{\mathtt{Inl}\,\mathtt{x}_1 \mapsto u_1,\ \mathtt{Inr}\,\mathtt{x}_2 \mapsto u_2\}))[t]}$$

$$\textsc{Sem-PatS-Unfoc}\ \frac{}{(C \circ (\square \rhd \mathtt{case}_{\mathtt{m}}\,\{\mathtt{Inl}\,\mathtt{x}_1 \mapsto u_1,\ \mathtt{Inr}\,\mathtt{x}_2 \mapsto u_2\}))[v] \longrightarrow C[v \rhd \mathtt{case}_{\mathtt{m}}\,\{\mathtt{Inl}\,\mathtt{x}_1 \mapsto u_1,\ \mathtt{Inr}\,\mathtt{x}_2 \mapsto u_2\}]}$$

$$\textsc{Sem-PatL-Red}\ \frac{}{C[(\mathtt{Inl}\ v_1) \rhd \mathtt{case}_{\mathtt{m}}\,\{\mathtt{Inl}\,\mathtt{x}_1 \mapsto u_1,\ \mathtt{Inr}\,\mathtt{x}_2 \mapsto u_2\}] \longrightarrow C[u_1[\mathtt{x}_1 := v_1]]}$$

Sem-PatR-Red

$$\frac{}{C[(\mathtt{Inr}\ v_2) \rhd \mathtt{case}_{\mathtt{m}}\ \{\mathtt{Inl}\ \mathtt{x}_1 \mapsto u_1\,,\ \mathtt{Inr}\ \mathtt{x}_2 \mapsto u_2\}] \longrightarrow C[u_2[\mathtt{x}_2 := v_2]]}$$

Sem-PatP-Foc

$$\frac{\mathtt{NotVal}\ t}{C[t \rhd \mathtt{case}_{\mathtt{m}}\ (\mathtt{x}_1\,,\ \mathtt{x}_2) \mapsto u] \longrightarrow (C \circ (\square \rhd \mathtt{case}_{\mathtt{m}}\ (\mathtt{x}_1\,,\ \mathtt{x}_2) \mapsto u))[t]}$$

Sem-PatP-Unfoc

$$\frac{}{(C \circ (\square \rhd \mathtt{case}_{\mathtt{m}}\ (\mathtt{x}_1\,,\ \mathtt{x}_2) \mapsto u))[v] \longrightarrow C[v \rhd \mathtt{case}_{\mathtt{m}}\ (\mathtt{x}_1\,,\ \mathtt{x}_2) \mapsto u]}$$

Sem-PatP-Red

$$\frac{}{C[(v_1\,,\ v_2) \rhd \mathtt{case}_{\mathtt{m}}\ (\mathtt{x}_1\,,\ \mathtt{x}_2) \mapsto u] \longrightarrow C[u[\mathtt{x}_1 := v_1][\mathtt{x}_2 := v_2]]}$$

Sem-PatE-Foc

$$\frac{\mathtt{NotVal}\ t}{C[t \rhd \mathtt{case}_{\mathtt{m}}\ \mathtt{E}_{\mathtt{n}}\ \mathtt{x} \mapsto u] \longrightarrow (C \circ (\square \rhd \mathtt{case}_{\mathtt{m}}\ \mathtt{E}_{\mathtt{n}}\ \mathtt{x} \mapsto u))[t]}$$

Sem-PatE-Unfoc

$$\frac{}{(C \circ (\square \rhd \mathtt{case}_{\mathtt{m}}\ \mathtt{E}_{\mathtt{n}}\ \mathtt{x} \mapsto u))[v] \longrightarrow C[v \rhd \mathtt{case}_{\mathtt{m}}\ \mathtt{E}_{\mathtt{n}}\ \mathtt{x} \mapsto u]}$$

Sem-PatE-Red

$$\frac{}{C[\mathtt{E}_{\mathtt{n}}\ v' \rhd \mathtt{case}_{\mathtt{m}}\ \mathtt{E}_{\mathtt{n}}\ \mathtt{x} \mapsto u] \longrightarrow C[u[\mathtt{x} := v']]}$$

Sem-Map-Foc

$$\frac{\mathtt{NotVal}\ t}{C[t \rhd \mathtt{map}\ \mathtt{x} \mapsto t'] \longrightarrow (C \circ (\square \rhd \mathtt{map}\ \mathtt{x} \mapsto t'))[t]}$$

Sem-Map-Unfoc

$$\frac{}{(C \circ (\square \rhd \mathtt{map}\ \mathtt{x} \mapsto t'))[v] \longrightarrow C[v \rhd \mathtt{map}\ \mathtt{x} \mapsto t']}$$

Sem-Map-Red-OpenAmpar-Foc

$$\frac{\mathtt{h}' = \mathit{max}(\mathit{hvars}(C))+1}{C[{}_{\mathtt{H}}\langle v_2 \wedge v_1\rangle \rhd \mathtt{map}\ \mathtt{x} \mapsto t'] \longrightarrow (C \circ ({}^{\mathrm{op}}_{\mathtt{H}\pm\mathtt{h}'}\langle v_2[\mathtt{H}\pm\mathtt{h}'] \wedge \square\rangle))[t'[\mathtt{x} := v_1[\mathtt{H}\pm\mathtt{h}']]]}$$

Sem-OpenAmpar-Unfoc

$$\frac{}{(C \circ {}^{\mathrm{op}}_{\mathtt{H}}\langle v_2 \wedge \square\rangle)[v_1] \longrightarrow C[{}_{\mathtt{H}}\langle v_2 \wedge v_1\rangle]}$$

Sem-ToA-Foc

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

Sem-ToA-Unfoc

$$\frac{}{(C \circ (\mathtt{to}_{\ltimes}\ \square))[v_2] \longrightarrow C[\mathtt{to}_{\ltimes}\ v_2]}$$

Sem-ToA-Red

$$\frac{}{C[\mathtt{to}_{\ltimes}\ v_2] \longrightarrow C[\{\}\langle v_2 \wedge ()\rangle]}$$

Sem-FromA-Foc

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

Sem-FromA-Unfoc

$$\frac{}{(C \circ (\mathtt{from}_{\ltimes}\ \square))[v] \longrightarrow C[\mathtt{from}_{\ltimes}\ v]}$$

Sem-FromA-Red
$$C[\mathtt{from}_{\ltimes\ \{\}}\langle v_2 \wedge_{\mathtt{E}_{1\infty}} v_1\rangle] \longrightarrow C[(v_2, \mathtt{E}_{1\infty}\ v_1)]$$

Sem-FillU-Foc
$$\frac{\mathtt{NotVal}\ t}{C[t \triangleleft ()] \longrightarrow (C \circ (\square \triangleleft ()))[t]}$$

Sem-FillU-Unfoc
$$(C \circ (\square \triangleleft ()))[v] \longrightarrow C[v \triangleleft ()]$$

Sem-FillU-Red
$$C[\to \mathtt{h} \triangleleft ()] \longrightarrow C[\mathtt{h} :=_{\{\}} ()][()]$$

Sem-FillL-Foc
$$\frac{\mathtt{NotVal}\ t}{C[t \triangleleft \mathtt{Inl}] \longrightarrow (C \circ (\square \triangleleft \mathtt{Inl}))[t]}$$

Sem-FillL-Unfoc
$$(C \circ (\square \triangleleft \mathtt{Inl}))[v] \longrightarrow C[v \triangleleft \mathtt{Inl}]$$

Sem-FillL-Red
$$\frac{\mathtt{h}' = max(hvars(C) \cup \{\mathtt{h}\})+1}{C[\to \mathtt{h} \triangleleft \mathtt{Inl}] \longrightarrow C[\mathtt{h} :=_{\{\mathtt{h}'+1\}} \mathtt{Inl}\ (\mathtt{h}'+1)][\to(\mathtt{h}'+1)]}$$

Sem-FillR-Foc
$$\frac{\mathtt{NotVal}\ t}{C[t \triangleleft \mathtt{Inr}] \longrightarrow (C \circ (\square \triangleleft \mathtt{Inr}))[t]}$$

Sem-FillR-Unfoc
$$(C \circ (\square \triangleleft \mathtt{Inr}))[v] \longrightarrow C[v \triangleleft \mathtt{Inr}]$$

Sem-FillR-Red
$$\frac{\mathtt{h}' = max(hvars(C) \cup \{\mathtt{h}\})+1}{C[\to \mathtt{h} \triangleleft \mathtt{Inr}] \longrightarrow C[\mathtt{h} :=_{\{\mathtt{h}'+1\}} \mathtt{Inr}\ (\mathtt{h}'+1)][\to(\mathtt{h}'+1)]}$$

Sem-FillE-Foc
$$\frac{\mathtt{NotVal}\ t}{C[t \triangleleft \mathtt{E}_{\mathtt{m}}] \longrightarrow (C \circ (\square \triangleleft \mathtt{E}_{\mathtt{m}}))[t]}$$

Sem-FillE-Unfoc
$$(C \circ (\square \triangleleft \mathtt{E}_{\mathtt{m}}))[v] \longrightarrow C[v \triangleleft \mathtt{E}_{\mathtt{m}}]$$

Sem-FillE-Red
$$\frac{\mathtt{h}' = max(hvars(C) \cup \{\mathtt{h}\})+1}{C[\to \mathtt{h} \triangleleft \mathtt{E}_{\mathtt{m}}] \longrightarrow C[\mathtt{h} :=_{\{\mathtt{h}'+1\}} \mathtt{E}_{\mathtt{m}}\ (\mathtt{h}'+1)][\to(\mathtt{h}'+1)]}$$

Sem-FillP-Foc
$$\frac{\mathtt{NotVal}\ t}{C[t \triangleleft (,)] \longrightarrow (C \circ (\square \triangleleft (,)))[t]}$$

Sem-FillP-Unfoc
$$(C \circ (\square \triangleleft (,)))[v] \longrightarrow C[v \triangleleft (,)]$$

Sem-FillP-Red
$$\frac{\mathtt{h}' = max(hvars(C) \cup \{\mathtt{h}\})+1}{C[\to \mathtt{h} \triangleleft (,)] \longrightarrow C[\mathtt{h} :=_{\{\mathtt{h}'+1,\mathtt{h}'+2\}} ((\mathtt{h}'+1), (\mathtt{h}'+2))][(\to(\mathtt{h}'+1), \to(\mathtt{h}'+2))]}$$

Sem-FillF-Foc
$$\frac{\mathtt{NotVal}\ t}{C[t \triangleleft (\lambda\, \mathtt{x}_{\mathtt{m}} \mapsto\ u)] \longrightarrow (C \circ (\square \triangleleft (\lambda\, \mathtt{x}_{\mathtt{m}} \mapsto\ u)))[t]}$$

Sem-FillF-Unfoc
$$(C \circ (\square \triangleleft (\lambda\, \mathtt{x}_{\mathtt{m}} \mapsto\ u)))[v] \longrightarrow C[v \triangleleft (\lambda\, \mathtt{x}_{\mathtt{m}} \mapsto\ u)]$$

Sem-FillF-Red
$$C[\to \mathtt{h} \triangleleft (\lambda\, \mathtt{x}_{\mathtt{m}} \mapsto\ u)] \longrightarrow C[\mathtt{h} :=_{\{\}} {}^{\forall}\lambda\, \mathtt{x}_{\mathtt{m}} \mapsto\ u][()]$$

Sem-FillComp-Foc1
$$\frac{\mathtt{NotVal}\ t}{C[t \triangleleft\bullet t'] \longrightarrow (C \circ (\square \triangleleft\bullet t'))[t]}$$

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

$$\textsc{Sem-FillComp-Unfoc2}\ \frac{}{(C \circ (v \triangleleft\bullet\ \Box))[v'] \longrightarrow C[v \triangleleft\bullet\ v']} \qquad \textsc{Sem-FillComp-Red}\ \frac{\mathsf{h}' = max(hvars(C) \cup \{\mathsf{h}\}) + 1}{C[\rightarrow \mathsf{h} \triangleleft\bullet_{\mathsf{H}} \langle v_2 \wedge v_1 \rangle] \longrightarrow C[\mathsf{h} :=_{(\mathsf{H} \triangleq \mathsf{h}')}\ v_2[\mathsf{H} \triangleq \mathsf{h}']][v_1[\mathsf{H} \triangleq \mathsf{h}']]}$$

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] 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