

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, 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).

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

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

© 2024 Association for Computing Machinery.

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

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

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).

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

The destination-feeding primitive $\triangleleft \text{Inl}$ 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 represent 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:

$t_1 := \{h_1\} \langle h_1 \wedge \rightarrow h_1 \rangle \triangleright \text{map } d_1 \mapsto$
 $d_1 \triangleleft () \triangleright \text{case}_{1V} (d_2, d_3) \mapsto$
 $d_2 \triangleleft () \triangleright \text{case}_{1V} (d_4, d_5) \mapsto$
 $d_3 \triangleleft () \triangleright \text{case}_{1V} (d_6, d_7) \mapsto {}^s[d_4, d_5, d_6, d_7]$
 evaluates to:
 $\{h_4, h_5, h_6, h_7\} \langle ((h_4, h_5), (h_6, h_7)) \wedge {}^s[\rightarrow h_4, \rightarrow h_5, \rightarrow h_6, \rightarrow h_7] \rangle$

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:

$t_2 := t_1 \triangleright \text{map } d \mapsto$
 $d \triangleright \text{case}_{1V} {}^s[d_4, d_5, d_6, d_7] \mapsto$
 $d_5 \triangleleft \text{Inr } \triangleleft () ; d_7 \triangleleft \text{Inr } \triangleleft () ; {}^s[d_4, d_6]$
 reduces to
 $\{h_4, h_6\} \langle ((h_4, \text{Inr } ()), (h_6, \text{Inr } ())) \wedge {}^s[\rightarrow h_4, \rightarrow h_6] \rangle$

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:

$t_3 := t_2 \triangleright \text{map } d \mapsto$
 $d \triangleright \text{case}_{1V} {}^s[d_4, d_6] \mapsto$
 $d_4 \triangleleft \text{Inl } \triangleleft () ; d_6 \triangleleft \text{Inr } \triangleleft ()$
 t_3 reduces to $\{ \}\langle ((\text{Inl } ()), \text{Inr } ()), (\text{Inl } ()), \text{Inr } ())) \wedge () \rangle$ 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 Efficient queue implementation

TODO: in progress

Idea: the normal list is used in read-only fashion, and is kept in order, while the destination-based difference list is used in a write-only fashion, taking advantage of its $O(1)$ append operation. When the front list is exhausted, we just have to transform the difference list into a normal list in a $O(1)$ operation, instead of the usual $O(n)$ reverse operation to put the back list in order in the reference implementation.

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 `FILLEAF` 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 `FILLEAF`.

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

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

As a result, there are in fact two ways to consume a destination: 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 $\text{alloc}' : ([T] \multimap 1) \multimap T$. 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:

$\text{alloc}' (\lambda d \vdash t) \rightarrow \text{withTmpStore } \{h := _ \} \text{ do } \{t[d := \rightarrow h] ; \text{deref } \rightarrow h\}$

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

$\text{alloc}' (\lambda d \vdash d \triangleleft \text{Inl } \triangleleft ())$
 $\rightarrow \text{withTmpStore } \{h := _ \} \text{ do } \{\rightarrow h \triangleleft \text{Inl } \triangleleft () ; \text{deref } \rightarrow h\}$
 $\rightarrow \text{withTmpStore } \{h := \text{Inl } () \} \text{ do } \{\text{deref } \rightarrow h\}$
 $\rightarrow \text{Inl } ()$

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

$\text{alloc}' (\lambda dd \vdash \text{alloc}' (\lambda d \vdash dd \triangleleft d))$

```

→ withTmpStore {hd := _} do {alloc' (λ d1 ↦ →hd ◁ d); deref →hd}
→ withTmpStore {hd := _} do {withTmpStore {h := _} do {→hd ◁ h; deref →h}; deref →hd}
→ withTmpStore {hd := →h} do {withTmpStore {h := _} do {deref →h}; deref →hd}

```

The original term $\text{alloc}' (\lambda d d_1 \mapsto \text{alloc}' (\lambda d_1 \mapsto d d \triangleleft d))$ is well typed, as the inner call to alloc' returns a value of type 1 (as d is of type $[1]$) and consumes d linearly. However, we see that because $\rightarrow h$ 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 $\text{withTmpStore } \{h := _ \} \text{ do } \{ \text{deref } \rightarrow h \}$ 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 destination¹, as long as destinations of type $[]$ are allowed. And forbidding destinations of type $[]$ 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 $[]$ 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 v 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 $\uparrow^0 = v$ 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:

$$\begin{array}{c}
\text{TY-VAL-DEST}^* \\
\hline
\rightarrow h :_v [T] \vdash \rightarrow h : [T]
\end{array}
\qquad
\begin{array}{c}
\text{TY-TERM-FILLLEAF}^* \\
\Theta_1 \vdash t : [T] \\
\Theta_2 \vdash t' : T \\
\hline
\Theta_1 + \uparrow \cdot \Theta_2 \vdash t \triangleleft t' : 1
\end{array}$$

Typing a destination $\rightarrow h$ requires $\rightarrow h$ to have age v 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 $\uparrow \cdot \Theta_2$).

¹This 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.

Such a rule system prevents in particular the previous faulty expression $\rightarrow h d \triangleleft \rightarrow h$ where $\rightarrow h d$ 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 $1 \dots |T|$ in breadth-first order.

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

In [1], the author presents a DPS implementation that relies on safe, controlled mutations, 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 ($\langle \text{inputsubtree}, \langle \text{desttooutputsubtree} \rangle$) 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 implementation². 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 is presented as part of Figure 1.

The implementation of breadth-first traversal presented in Figure 2 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 `Tree T1` is consumed linearly. The stateful transformer is also linear in its two arguments. The state has to be 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 $S_{1v} \rightarrow !_{1\infty} S'$ so that the final state can be returned.

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

²This 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

```

type rec Int  $\triangleq$  1 $\oplus$ Int
operator zero : Int
    zero  $\triangleq$  Inl ()

operator succ : Int  $\rightarrow$  Int
    succ x  $\triangleq$   $\text{\textit{\textcircled{S}}}$ Inr x

type rec List T  $\triangleq$  1 $\oplus$ (T $\otimes$ (List T))
operator  $\triangleleft$ [] : [n] List T  $\rightarrow$  1
    d  $\triangleleft$  []  $\triangleq$  d  $\triangleleft$  Inl  $\triangleleft$  ()

operator  $\triangleleft$ (:) : [n] List T  $\rightarrow$  [n] T  $\otimes$  [n] List T
    d  $\triangleleft$  (:)  $\triangleq$  d  $\triangleleft$  Inr  $\triangleleft$  (:)

type DList T  $\triangleq$  (List T)  $\times$  ([1v] List T)
operator append : DList T  $\rightarrow$  T  $\rightarrow$  DList T
    x append y  $\triangleq$  x  $\triangleright$  map d  $\mapsto$  d  $\triangleleft$  (:)  $\triangleright$  case1v
        (dh, dt)  $\mapsto$  dh  $\triangleleft$  y ; dt

operator concat : DList T  $\rightarrow$  DList T  $\rightarrow$  DList T
    x concat x'  $\triangleq$  x  $\triangleright$  map d  $\mapsto$  d  $\triangleleft$  x'

operator toList : DList T  $\rightarrow$  List T
    toList x  $\triangleq$  fromK'(x  $\triangleright$  map d  $\mapsto$  d  $\triangleleft$  [])

type Queue T  $\triangleq$  (List T)  $\otimes$  (DList T)
operator singleton : T  $\rightarrow$  Queue T
    singleton x  $\triangleq$   $\text{\textit{\textcircled{S}}}$ ( $\text{\textit{\textcircled{S}}}$ Inr( $\text{\textit{\textcircled{S}}}$ (x, Inl ()), alloc)

operator enqueue : Queue T  $\rightarrow$  T  $\rightarrow$  Queue T
    x enqueue y  $\triangleq$  x  $\triangleright$  case1v(x1, x2)  $\mapsto$   $\text{\textit{\textcircled{S}}}$ (x1, x2 append y)

operator dequeue : Queue T  $\rightarrow$  1 $\oplus$ (T $\otimes$ Queue T)
    dequeue x  $\triangleq$  x  $\triangleright$  case1v
        (x, y)  $\mapsto$  x  $\triangleright$  case1v{
            Inl un  $\mapsto$  un ; toList y  $\triangleright$  case1v{
                Inl un  $\mapsto$   $\text{\textit{\textcircled{S}}}$ Inl un,
                Inr y'  $\mapsto$  y'  $\triangleright$  case1v
                    (y'1, y'2)  $\mapsto$   $\text{\textit{\textcircled{S}}}$ Inr( $\text{\textit{\textcircled{S}}}$ (y'1,  $\text{\textit{\textcircled{S}}}$ (y'2, alloc))
            },
            Inr x'  $\mapsto$  x'  $\triangleright$  case1v
                (x'1, y'1)  $\mapsto$   $\text{\textit{\textcircled{S}}}$ Inr( $\text{\textit{\textcircled{S}}}$ (x'1,  $\text{\textit{\textcircled{S}}}$ (y'1, y))
        }

type Tree T  $\triangleq$  1 $\oplus$ (T $\otimes$ ((Tree T)  $\otimes$  (Tree T)))
operator  $\triangleleft$ Nil : [n] Tree T  $\rightarrow$  1
    d  $\triangleleft$  Nil  $\triangleq$  d  $\triangleleft$  Inl  $\triangleleft$  ()

operator  $\triangleleft$ Node : [n] Tree T  $\rightarrow$  [n] T  $\otimes$  ([n] Tree T)  $\otimes$  ([n] Tree T)
    d  $\triangleleft$  Node  $\triangleq$  d  $\triangleleft$  Inr  $\triangleleft$  (,)  $\triangleright$  case1v(dv, dtlr)  $\mapsto$   $\text{\textit{\textcircled{S}}}$ (dv, dtlr  $\triangleleft$  (,))

```

Fig. 1. Boilerplate for breadth-first tree traversal

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

In fact, the type of hole abstraction $(T_1, T_2)\text{hfun}$ in 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 .

operator rec

```

go : ((!_{100}S) 1_V \to T_1 1_V \to (!_{100}S) \otimes T_2) \omega_V \to (!_{100}S) 1_V \to Queue (Tree T_1 \otimes [1_V Tree T_2]) 1_V \to (!_{100}S)
go f st q \triangleq dequeue q \triangleright case_{1_V} \{
  Inl un \mapsto un ; st,
  Inr x \mapsto x \triangleright case_{1_V}
    (x', q') \mapsto x' \triangleright case_{1_V}
      (tree, dtree) \mapsto tree \triangleright case_{1_V} \{
        Inl un \mapsto un ; dtree \triangleleft Nil ; go f st q',
        Inr y \mapsto y \triangleright case_{1_V}
          (y', y'') \mapsto y'' \triangleright case_{1_V}
            (tl, tr) \mapsto dtree \triangleleft Node \triangleright case_{1_V}
              (dv, dtlr) \mapsto dtlr \triangleright case_{1_V}
                (dtl, dtr) \mapsto f st y' \triangleright case_{1_V}
                  (st', y'') \mapsto
                    dv \triangleleft y'' ; go f st' (q' enqueue \textcolor{teal}{\textcircled{\text{!}}}(tl, dtl) enqueue \textcolor{teal}{\textcircled{\text{!}}}(tr, dtr))
              }
          }
      }
}

```

operator

```

mapAccumBFS : ((!_{100}S) 1_V \to T_1 1_V \to (!_{100}S) \otimes T_2) \omega_V \to (!_{100}S) 1_V \to Tree T_1 1_V \to Tree T_2 \otimes (!_{100}S)
mapAccumBFS f st tree \triangleq from_{\times} (alloc \triangleright map dtree \mapsto go f st (singleton \textcolor{teal}{\textcircled{\text{!}}}(tree, dtree)))

```

operator

```

relabelDPS : Tree 1 1_V \to (Tree Int) \otimes (!_{100}(!_{\omega_V} Int))
relabelDPS tree \triangleq mapAccumBFS
  (\lambda ex 1_V \mapsto \lambda un 1_V \mapsto un ; ex \triangleright case_{1_V}
    E_{100} ex' \mapsto ex' \triangleright case_{100}
      E_{\omega_V} st \mapsto \textcolor{teal}{\textcircled{\text{!}}}(E_{100} (\textcolor{teal}{\textcircled{\text{!}}}(E_{\omega_V} (succ st))), st))
    (\textcolor{teal}{\textcircled{\text{!}}}(E_{100} (\textcolor{teal}{\textcircled{\text{!}}}(E_{\omega_V} (succ zero))))
  tree

```

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

Now, in classical linear logic, we know we can transform linear implication $T_1 \multimap T_2$ into $T_1^\perp \wp T_2$. Doing so for the *wand* type $(T_1, T_2)\text{hfun}$ or $T_1 \multimap T_2$ gives $[T_1] \widehat{\wp} T_2$, where $[\cdot]$ is memory negation, and $\widehat{\wp}$ is a memory *par* (weaker than the CLL *par* that allows more “interaction” of its two sides).

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

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

In destination calculus though, we don’t have an involutive memory negation $[\cdot]$. If we are provided with a destination of destination $\rightarrow h' : [T]$, we know that some structure is expecting to store a destination of type $[T]$. If ever that structure is consumed, then the destination stored 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 $\rightarrow h : [T]$, and write $\rightarrow h$ to the memory slot pointed to by $\rightarrow h'$, then we can get back a value of type T at h if ever the structure pointed to by $\rightarrow h'$ is consumed. Thus, a destination of destination is only equivalent to the promise of an eventual value, not an immediate usable one.

As a result, in destination calculus, we cannot have the same kind of cut rule as in CLL. This is, in fact, the part of destination calculus that was the hardest to design, and the source of a lot of early errors. For a destination of type $\llbracket T \rrbracket$, both storing it through a destination of destination $\llbracket \llbracket T \rrbracket \rrbracket$ 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 $\llbracket T \rrbracket$ originating from $T \hat{\times} \llbracket T \rrbracket$ (through a destination of destination of type $\llbracket \llbracket T \rrbracket \rrbracket$) should not allow to free up the T , as it would in a CLL-like setting.

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

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

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

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

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

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

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

Destinations from previous scopes are inert. In destination calculus, scopes are delimited by the map operation over ampars. Anytime a map happens, we enter a new scope, and any preexisting destination or variable see its age increased by one (\uparrow). As soon as a destination or variable is no longer of age 0 (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, ex, st, tree, tl, tr, dtree, f, dh, dt, dv, dtlr, dtl, dtr, 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 $h \in H$
	$\max(H)$	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 $H, H_1, H_2 \dots$

hvars, H ::= Set of hole names

	$\{h_1, \dots, h_k\}$		
	$H_1 \cup H_2$	M	Union of sets
	$H\#h'$	M	Shift all names from H by h' .
	$hvars(\Gamma)$	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 $\{h\} \langle h_\wedge \rightarrow h \rangle$ as part of the user syntax as a way to spawn a fresh memory cell to be later filled using destination-feeding primitives (see *alloc* in Section 5.4).

<i>term, t, u</i>	::=	Term
	v	Value
	x	Variable
	$t' t$	Application
	$t ; u$	Pattern-match on unit
	$t \triangleright \text{case}_m \{ \text{Inl } x_1 \mapsto u_1, \text{Inr } x_2 \mapsto u_2 \}$	Pattern-match on sum
	$t \triangleright \text{case}_m (x_1, x_2) \mapsto u$	Pattern-match on product
	$t \triangleright \text{case}_m E_n x \mapsto u$	Pattern-match on exponential
	$t \triangleright \text{map } x \mapsto t'$	Map over the right side of ampar
	$\text{to}_\times u$	Wrap into a trivial ampar
	$\text{from}_\times t$	Convert ampar to a pair
	$t \triangleleft ()$	Fill destination with unit

	$t \triangleleft \text{Inl}$	Fill destination with left variant
	$t \triangleleft \text{Inr}$	Fill destination with right variant
	$t \triangleleft E_m$	Fill destination with exponential constructor
	$t \triangleleft (,)$	Fill destination with product constructor
	$t \triangleleft (\lambda x_m \mapsto u)$	Fill destination with function
	$t \triangleleft \bullet t'$	Fill destination with root of other ampar
	$t[x := v]$	M Shift hole names inside v by H if they belong to H .

val, v	$::=$	Value
	h	Hole
	$\rightarrow h$	Destination
	$()$	Unit
	$\lambda x_m \mapsto u$	Function with no free variable
	$\text{Inl } v$	Left variant for sum
	$\text{Inr } v$	Right variant for sum
	$E_m v$	Exponential
	(v_1, v_2)	Product
	$H \langle v_2 \wedge v_1 \rangle$	Ampar
	$v[H \pm h']$	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 $!_{\omega a}(T_1 \otimes T_2) \simeq (!_{\omega a} T_1) \otimes (!_{\omega a} T_2)$, 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.

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

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

The remaining term operators $\triangleleft(), \triangleleft \text{Inl}, \triangleleft \text{Inr}, \triangleleft E_m, \triangleleft (,), \triangleleft (\lambda x_m \mapsto u)$ 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.

$\text{from}_{\times'}$ is a simpler variant of from_{\times} that allows to extract the right side of an ampar when the right side has been fully consumed. We implement it in terms of from_{\times} 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.

<i>term</i>	$::=$		Syntactic sugar for terms
	alloc	M	Evaluate to a fresh new ampar
	$t \triangleleft t'$	M	Fill destination with supplied term
	$\text{from}'_{\times} t$	M	Extract left side of ampar when right side is unit
	$^s\lambda x_m \mapsto u$	M	Allocate function
	$^s\text{Inl } t$	M	Allocate left variant
	$^s\text{Inr } t$	M	Allocate right variant
	$^sE_m t$	M	Allocate exponential
	$^s(t_1, t_2)$	M	Allocate product

$\text{alloc} \triangleq \{1\}\langle 1_{\wedge} \rightarrow 1 \rangle$	$t \triangleleft t' \triangleq t \triangleleft (\text{to}_{\times} t')$
$\text{from}'_{\times} t \triangleq (\text{from}_{\times} (t \triangleright \text{map } \text{un} \mapsto \text{un} ; E_{1\infty} ())) \triangleright \text{case}_{1\nu} (st, ex) \mapsto ex \triangleright \text{case}_{1\nu} E_{1\infty} \text{un} \mapsto \text{un} ; st$	$^s\lambda x_m \mapsto u \triangleq \text{from}'_{\times} (\text{alloc} \triangleright \text{map } d \mapsto d \triangleleft (\lambda x_m \mapsto u))$
$^s\text{Inl } t \triangleq \text{from}'_{\times} (\text{alloc} \triangleright \text{map } d \mapsto d \triangleleft \text{Inl} \triangleleft t)$	$^s\text{Inr } t \triangleq \text{from}'_{\times} (\text{alloc} \triangleright \text{map } d \mapsto d \triangleleft \text{Inr} \triangleleft t)$
$^sE_m t \triangleq \text{from}'_{\times} (\text{alloc} \triangleright \text{map } d \mapsto d \triangleleft E_m \triangleleft t)$	$^s(t_1, t_2) \triangleq \text{from}'_{\times} (\text{alloc} \triangleright \text{map } d \mapsto (d \triangleleft (.)) \triangleright \text{case}_{1\nu} (d_1, d_2) \mapsto d_1 \triangleleft t_1 ; d_2 \triangleleft t_2)$

Table 1. Desugaring of syntactic sugar forms for terms

6 TYPE SYSTEM

6.1 Syntax for types, modes, and typing contexts

$\text{type, } T, U, S$	$::=$	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
	$\llbracket_m T \rrbracket$	Destination
$\text{mode, } m, n$	$::=$	Mode (Semiring)
	pa	Pair of a multiplicity and age
	ω	Error case (incompatible types, multiplicities, or ages)
$\text{mul, } p$	$::=$	Multiplicity (Semiring, first component of modality)
	1	Linear use
	ω	Non-linear use
$\text{age, } a$	$::=$	Age (Semiring, second component of modality)
	ν	Born now
	\uparrow	One scope older
	∞	Infinitely old / static
$\text{ctx, } \Gamma, \Delta, \Theta$	$::=$	Typing context
	$x :_m T$	Variable typing binding
	$h :_n T$	Hole typing binding
	$\rightarrow h :_m \llbracket_n T \rrbracket$	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 ω)
	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
	$\Gamma[H:h']$	M Shift hole/dest names by h' if they belong to H

6.2 Typing of terms and values

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

$\frac{\text{TY-VAL-HOLE}}{h :_{1v} T \vdash h : T}$	$\frac{\text{TY-VAL-DEST}}{\rightarrow h :_{1v} [n] T \vdash \rightarrow h : [n] T}$	$\frac{\text{TY-VAL-UNIT}}{\vdash () : 1}$
$\frac{\text{TY-VAL-FUN}}{\Delta, x :_{\text{m}} T \vdash u : U}{\Delta \vdash \lambda x_{\text{m}} \mapsto u : T_{\text{m}} \rightarrow U}$	$\frac{\text{TY-VAL-LEFT}}{\Gamma \vdash v_1 : T_1}{\Gamma \vdash \text{Inl } v_1 : T_1 \oplus T_2}$	$\frac{\text{TY-VAL-RIGHT}}{\Gamma \vdash v_2 : T_2}{\Gamma \vdash \text{Inr } v_2 : T_1 \oplus T_2}$
$\frac{\text{TY-VAL-PROD}}{\Gamma_1 \vdash v_1 : T_1, \Gamma_2 \vdash v_2 : T_2}{\Gamma_1 + \Gamma_2 \vdash (v_1, v_2) : T_1 \otimes T_2}$	$\frac{\text{TY-VAL-EXP}}{\Gamma \vdash v' : T}{n \cdot \Gamma \vdash \text{En } v' : !n T}$	$\frac{\text{TY-VAL-AMPAR}}{\text{LinOnly } \Delta_3, \text{FinAgeOnly } \Delta_3, \uparrow \Delta_1, \Delta_3 \vdash v_1 : T, \Delta_2, (\rightarrow^{-1} \Delta_3) \vdash v_2 : U}{\Delta_1, \Delta_2 \vdash \text{hvars}(\rightarrow^{-1} \Delta_3) \langle v_2 \wedge v_1 \rangle : U \ltimes T}$

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

$\frac{\text{TY-TERM-VAL} \quad \text{DisposableOnly } \Theta \quad \Delta \vdash v : T}{\Theta, \Delta \vdash v : T}$	$\frac{\text{TY-TERM-VAR} \quad \text{DisposableOnly } \Theta \quad 1v <: m}{\Theta, x : mT \vdash x : T}$	$\frac{\text{TY-TERM-APP} \quad \Theta_1 \vdash t : T \quad \Theta_2 \vdash t' : T_m \rightarrow U}{m \cdot \Theta_1 + \Theta_2 \vdash t' t : U}$
TY-TERM-PATS		
$\frac{\text{TY-TERM-PATU} \quad \Theta_1 \vdash t : 1 \quad \Theta_2 \vdash u : U}{\Theta_1 + \Theta_2 \vdash t ; u : U}$	$\frac{\Theta_1 \vdash t : T_1 \oplus T_2 \quad \Theta_2, x_1 : mT_1 \vdash u_1 : U \quad \Theta_2, x_2 : mT_2 \vdash u_2 : U}{m \cdot \Theta_1 + \Theta_2 \vdash t \triangleright \text{case}_m \{ \text{Inl } x_1 \mapsto u_1, \text{Inr } x_2 \mapsto u_2 \} : U}$	
$\frac{\text{TY-TERM-PATP} \quad \Theta_1 \vdash t : T_1 \otimes T_2 \quad \Theta_2, x_1 : mT_1, x_2 : mT_2 \vdash u : U}{m \cdot \Theta_1 + \Theta_2 \vdash t \triangleright \text{case}_m (x_1, x_2) \mapsto u : U}$	$\frac{\text{TY-TERM-PATE} \quad \Theta_1 \vdash t : !nT \quad \Theta_2, x : mnT \vdash u : U}{m \cdot \Theta_1 + \Theta_2 \vdash t \triangleright \text{case}_m \text{En } x \mapsto u : U}$	
$\frac{\text{TY-TERM-MAP} \quad \Theta_1 \vdash t : U \ltimes T \quad \uparrow \Theta_2, x : 1vT \vdash t' : T'}{\Theta_1 + \Theta_2 \vdash t \triangleright \text{map } x \mapsto t' : U \ltimes T'}$	$\frac{\text{TY-TERM-TOA} \quad \Theta \vdash u : U}{\Theta \vdash \text{to}_{\ltimes} u : U \ltimes 1}$	$\frac{\text{TY-TERM-FROMA} \quad \Theta \vdash t : U \ltimes (!_{1\infty} T)}{\Theta \vdash \text{from}_{\ltimes} t : U \otimes (!_{1\infty} T)}$
$\frac{\text{TY-TERM-FILLU} \quad \Theta \vdash t : [n] 1}{\Theta \vdash t \triangleleft () : 1}$	$\frac{\text{TY-TERM-FILLL} \quad \Theta \vdash t : [n] T_1 \oplus T_2}{\Theta \vdash t \triangleleft \text{Inl} : [n] T_1}$	$\frac{\text{TY-TERM-FILLR} \quad \Theta \vdash t : [n] T_1 \oplus T_2}{\Theta \vdash t \triangleleft \text{Inr} : [n] T_2}$
$\frac{\text{TY-TERM-FILLE} \quad \Theta \vdash t : [n] !_{n'} T}{\Theta \vdash t \triangleleft \text{En}' : [n' \cdot n] T}$	$\frac{\text{TY-TERM-FILLF} \quad \Theta_1 \vdash t : [n] T_m \rightarrow U \quad \Theta_2, x : mT \vdash u : U}{\Theta_1 + (\uparrow \cdot n) \cdot \Theta_2 \vdash t \triangleleft (\lambda x_m \mapsto u) : 1}$	$\frac{\text{TY-TERM-FILLCOMP} \quad \Theta_1 \vdash t : [n] U \quad \Theta_2 \vdash t' : U \ltimes T}{\Theta_1 + (\uparrow \cdot n) \cdot \Theta_2 \vdash t \triangleleft \bullet t' : T}$

6.3 Derived typing rules for syntactic sugar forms

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

$$\begin{array}{c}
 \text{TY-STERM-ALLOC} \\
 \hline
 \text{DisposableOnly } \Theta \\
 \hline
 \Theta \vdash^s \text{alloc} : T \ltimes (\lfloor_{1\nu} T \rfloor)
 \end{array}
 \quad
 \begin{array}{c}
 \text{TY-STERM-FROMA'} \\
 \hline
 \Theta \vdash t : T \ltimes 1 \\
 \hline
 \Theta \vdash^s \text{from}'_{\ltimes} t : T
 \end{array}
 \quad
 \begin{array}{c}
 \text{TY-STERM-FILLLEAF} \\
 \hline
 \Theta_1 \vdash t : \lfloor_n T \rfloor \\
 \Theta_2 \vdash t' : T \\
 \hline
 \Theta_1 + (\uparrow_n) \cdot \Theta_2 \vdash^s t \triangleleft t' : 1
 \end{array}$$

$$\begin{array}{c}
 \text{TY-STERM-FUN} \\
 \hline
 \Theta_2, x : \lfloor_m T \rfloor \vdash u : U \\
 \hline
 \Theta_2 \vdash^s \lambda x_{\lfloor_m \rfloor} \mapsto u : T_{\lfloor_m \rfloor} \rightarrow U
 \end{array}
 \quad
 \begin{array}{c}
 \text{TY-STERM-LEFT} \\
 \hline
 \Theta_2 \vdash t : T_1 \\
 \hline
 \Theta_2 \vdash^s \text{Inl } t : T_1 \oplus T_2
 \end{array}
 \quad
 \begin{array}{c}
 \text{TY-STERM-RIGHT} \\
 \hline
 \Theta_2 \vdash t : T_2 \\
 \hline
 \Theta_2 \vdash^s \text{Inr } t : T_1 \oplus T_2
 \end{array}$$

$$\begin{array}{c}
 \text{TY-STERM-EXP} \\
 \hline
 \Theta_2 \vdash t : T \\
 \hline
 \lfloor_m \Theta_2 \vdash^s \text{E}_m t : \lfloor_m T \rfloor
 \end{array}
 \quad
 \begin{array}{c}
 \text{TY-STERM-PROD} \\
 \hline
 \Theta_{21} \vdash t_1 : T_1 \\
 \Theta_{22} \vdash t_2 : T_2 \\
 \hline
 \Theta_{21} + \Theta_{22} \vdash^s \text{E}(t_1, t_2) : T_1 \otimes T_2
 \end{array}$$

7 EVALUATION CONTEXTS AND SEMANTICS

7.1 Evaluation contexts forms

 $ectx, c$
 $::=$

- $t' \square$
- $\square \nu$
- $\square ; u$
- $\square \triangleright \text{case}_m \{ \text{Inl } x_1 \mapsto u_1, \text{Inr } x_2 \mapsto u_2 \}$
- $\square \triangleright \text{case}_m (x_1, x_2) \mapsto u$
- $\square \triangleright \text{case}_m E_n x \mapsto u$
- $\square \triangleright \text{map } x \mapsto t'$
- $\text{to}_{\ltimes} \square$
- $\text{from}_{\ltimes} \square$
- $\square \triangleleft ()$
- $\square \triangleleft \text{Inl}$
- $\square \triangleleft \text{Inr}$
- $\square \triangleleft E_m$
- $\square \triangleleft (,)$
- $\square \triangleleft (\lambda x_{\lfloor_m \rfloor} \mapsto u)$
- $\square \triangleleft \bullet t'$
- $\nu \triangleleft \bullet \square$
- $\text{op}_H \langle \nu_2 \wedge \square \rangle$

Evaluation context component

 $ectxs, C$
 $::=$

- \square
- $C \circ c$
- $C[\text{h} :=_{\text{H}} \nu]$

Evaluation context stack

Represent the empty stack / "identity"

 Push c on top of C
 M Fill h in C with value ν (that may contain M)

7.2 Typing of evaluation contexts and commands

$$\boxed{\Delta \vdash C : T \multimap U_0}$$

(Typing judgment for evaluation contexts)

$$\begin{array}{c}
\text{TY-ECTXS-ID} \\
\hline
\vdash \square : U_0 \multimap U_0
\end{array}
\quad
\begin{array}{c}
\text{TY-ECTXS-APP-FOC1} \\
\text{m}\Delta_1, \Delta_2 \vdash C : U \multimap U_0 \\
\Delta_2 \vdash t' : T \multimap U \\
\hline
\Delta_1 \vdash C \circ (t' \square) : T \multimap U_0
\end{array}
\quad
\begin{array}{c}
\text{TY-ECTXS-APP-FOC2} \\
\text{m}\Delta_1, \Delta_2 \vdash C : U \multimap U_0 \\
\Delta_1 \vdash v : T \\
\hline
\Delta_2 \vdash C \circ (\square v) : (T \multimap U) \multimap U_0
\end{array}$$

$$\begin{array}{c}
\text{TY-ECTXS-PATU-FOC} \\
\Delta_1, \Delta_2 \vdash C : U \multimap U_0 \\
\Delta_2 \vdash u : U \\
\hline
\Delta_1 \vdash C \circ (\square ; u) : 1 \multimap U_0
\end{array}$$

$$\begin{array}{c}
\text{TY-ECTXS-PATS-FOC} \\
\text{m}\Delta_1, \Delta_2 \vdash C : U \multimap U_0 \\
\Delta_2, x_1 : \text{m}T_1 \vdash u_1 : U \\
\Delta_2, x_2 : \text{m}T_2 \vdash u_2 : U \\
\hline
\Delta_1 \vdash C \circ (\square \triangleright \text{case}_m \{ \text{Inl } x_1 \mapsto u_1, \text{Inr } x_2 \mapsto u_2 \}) : (T_1 \oplus T_2) \multimap U_0
\end{array}$$

$$\begin{array}{c}
\text{TY-ECTXS-PATP-FOC} \\
\text{m}\Delta_1, \Delta_2 \vdash C : U \multimap U_0 \\
\Delta_2, x_1 : \text{m}T_1, x_2 : \text{m}T_2 \vdash u : U \\
\hline
\Delta_1 \vdash C \circ (\square \triangleright \text{case}_m (x_1, x_2) \mapsto u) : (T_1 \otimes T_2) \multimap U_0
\end{array}$$

$$\begin{array}{c}
\text{TY-ECTXS-PATE-FOC} \\
\text{m}\Delta_1, \Delta_2 \vdash C : U \multimap U_0 \\
\Delta_2, x : \text{m-m}T \vdash u : U \\
\hline
\Delta_1 \vdash C \circ (\square \triangleright \text{case}_m \text{Em}' x \mapsto u) : !\text{m}'T \multimap U_0
\end{array}
\quad
\begin{array}{c}
\text{TY-ECTXS-MAP-FOC} \\
\Delta_1, \Delta_2 \vdash C : U \ltimes T' \multimap U_0 \\
1\uparrow\Delta_2, x : !vT \vdash t' : T' \\
\hline
\Delta_1 \vdash C \circ (\square \triangleright \text{map } x \mapsto t') : (U \ltimes T) \multimap U_0
\end{array}$$

$$\begin{array}{c}
\text{TY-ECTXS-TOA-FOC} \\
\Delta \vdash C : (U \ltimes 1) \multimap U_0 \\
\hline
\Delta \vdash C \circ (\text{to}_\ltimes \square) : U \multimap U_0
\end{array}
\quad
\begin{array}{c}
\text{TY-ECTXS-FROMA-FOC} \\
\Delta \vdash C : (U \otimes (!_{1\infty}T)) \multimap U_0 \\
\hline
\Delta \vdash C \circ (\text{from}_\ltimes \square) : (U \ltimes (!_{1\infty}T)) \multimap U_0
\end{array}$$

$$\begin{array}{c}
\text{TY-ECTXS-FILLU-FOC} \\
\Delta \vdash C : 1 \multimap U_0 \\
\hline
\Delta \vdash C \circ (\square \triangleleft ()) : \llbracket n \ 1 \rrbracket \multimap U_0
\end{array}
\quad
\begin{array}{c}
\text{TY-ECTXS-FILLL-FOC} \\
\Delta \vdash C : \llbracket n \ T_1 \rrbracket \multimap U_0 \\
\hline
\Delta \vdash C \circ (\square \triangleleft \text{Inl}) : \llbracket n \ T_1 \oplus T_2 \rrbracket \multimap U_0
\end{array}$$

$$\begin{array}{c}
\text{TY-ECTXS-FILLR-FOC} \\
\Delta \vdash C : \llbracket n \ T_2 \rrbracket \multimap U_0 \\
\hline
\Delta \vdash C \circ (\square \triangleleft \text{Inr}) : \llbracket n \ T_1 \oplus T_2 \rrbracket \multimap U_0
\end{array}
\quad
\begin{array}{c}
\text{TY-ECTXS-FILLP-FOC} \\
\Delta \vdash C : (\llbracket n \ T_1 \rrbracket \otimes \llbracket n \ T_2 \rrbracket) \multimap U_0 \\
\hline
\Delta \vdash C \circ (\square \triangleleft (,)) : \llbracket n \ T_1 \otimes T_2 \rrbracket \multimap U_0
\end{array}$$

$$\begin{array}{c}
\text{TY-ECTXS-FILLE-FOC} \\
\Delta \vdash C : \llbracket m \text{-} n \ T \rrbracket \multimap U_0 \\
\hline
\Delta \vdash C \circ (\square \triangleleft \text{Em}) : \llbracket n \ !\text{m}T \rrbracket \multimap U_0
\end{array}
\quad
\begin{array}{c}
\text{TY-ECTXS-FILLF-FOC} \\
\Delta_1, (1\uparrow\text{-}n) \cdot \Delta_2 \vdash C : 1 \multimap U_0 \\
\Delta_2, x : \text{m}T \vdash u : U \\
\hline
\Delta_1 \vdash C \circ (\square \triangleleft (\lambda x \text{m} \mapsto u)) : \llbracket n \ T \multimap U \rrbracket \multimap U_0
\end{array}$$

$$\begin{array}{c}
\text{TY-ECTXS-FILLCOMP-FOC1} \\
\frac{\Delta_1, (1\uparrow \cdot n) \cdot \Delta_2 \vdash C : T \multimap U_0 \quad \Delta_2 \vdash t' : U \ltimes T}{\Delta_1 \vdash C \circ (\Box \blacktriangleleft t') : [n U] \multimap U_0} \\
\\
\text{TY-ECTXS-FILLCOMP-FOC2} \\
\frac{\Delta_1, (1\uparrow \cdot n) \cdot \Delta_2 \vdash C : T \multimap U_0 \quad \Delta_1 \vdash v : [n U]}{\Delta_2 \vdash C \circ (v \blacktriangleleft \Box) : U \ltimes T \multimap U_0} \\
\\
\text{TY-ECTXS-OPENAMPAR-FOC} \\
\frac{\text{hvars}(C) \text{ \textcolor{red}{\#\#} hvars}(\rightarrow^{-1} \Delta_3) \quad \text{LinOnly } \Delta_3 \quad \text{FinAgeOnly } \Delta_3 \quad \Delta_1, \Delta_2 \vdash C : (U \ltimes T') \multimap U_0 \quad \Delta_2, \rightarrow^{-1} \Delta_3 \Vdash v_2 : U}{1\uparrow \cdot \Delta_1, \Delta_3 \vdash C \circ (\overset{\text{op}}{\text{hvars}(\rightarrow^{-1} \Delta_3)} \langle v_2 \wedge \Box \rangle) : T' \multimap U_0}
\end{array}$$

$$\boxed{\vdash C[t] : T}$$

(Typing judgment for commands)

$$\begin{array}{c}
\text{TY-CMD} \\
\frac{\Delta \vdash C : T \multimap U_0 \quad \Delta \vdash t : T}{\vdash C[t] : U_0}
\end{array}$$

7.3 Small-step semantics

$$\boxed{C[t] \longrightarrow C'[t']}$$

(Small-step evaluation of commands)

$$\begin{array}{c}
\text{SEM-APP-FOC1} \\
\frac{\text{NotVal } t}{C[t' t] \longrightarrow (C \circ (t' \Box))[t]} \\
\\
\text{SEM-APP-UNFOC1} \\
\frac{}{(C \circ (t' \Box))[v] \longrightarrow C[t' v]} \\
\\
\text{SEM-APP-FOC2} \\
\frac{\text{NotVal } t'}{C[t' v] \longrightarrow (C \circ (\Box v))[t']} \\
\\
\text{SEM-APP-UNFOC2} \\
\frac{}{(C \circ (\Box v))[v'] \longrightarrow C[v' v]} \\
\\
\text{SEM-APP-RED} \\
\frac{}{C[(\lambda x_{\mathfrak{m}} \mapsto u) v] \longrightarrow C[u[x := v]]} \\
\\
\text{SEM-PATU-FOC} \\
\frac{\text{NotVal } t}{C[t; u] \longrightarrow (C \circ (\Box; u))[t]} \\
\\
\text{SEM-PATU-UNFOC} \\
\frac{}{(C \circ (\Box; u))[v] \longrightarrow C[v; u]} \\
\\
\text{SEM-PATU-RED} \\
\frac{}{C[(); u] \longrightarrow C[u]}
\end{array}$$

SEM-PATS-FOC

$$\frac{\text{NotVal } t}{C[t \triangleright \text{case}_{\mathfrak{m}} \{ \text{Inl } x_1 \mapsto u_1, \text{Inr } x_2 \mapsto u_2 \}] \longrightarrow (C \circ (\Box \triangleright \text{case}_{\mathfrak{m}} \{ \text{Inl } x_1 \mapsto u_1, \text{Inr } x_2 \mapsto u_2 \}))[t]}$$

SEM-PATS-UNFOC

$$\frac{}{(C \circ (\Box \triangleright \text{case}_{\mathfrak{m}} \{ \text{Inl } x_1 \mapsto u_1, \text{Inr } x_2 \mapsto u_2 \}))[v] \longrightarrow C[v \triangleright \text{case}_{\mathfrak{m}} \{ \text{Inl } x_1 \mapsto u_1, \text{Inr } x_2 \mapsto u_2 \}]}$$

SEM-PATL-RED

$$\frac{}{C[(\text{Inl } v_1) \triangleright \text{case}_{\mathfrak{m}} \{ \text{Inl } x_1 \mapsto u_1, \text{Inr } x_2 \mapsto u_2 \}] \longrightarrow C[u_1[x_1 := v_1]]}$$

SEM-PATR-RED

$$\overline{C[(\text{Inr } v_2) \triangleright \text{case}_m \{ \text{Inl } x_1 \mapsto u_1, \text{Inr } x_2 \mapsto u_2 \}]} \longrightarrow C[u_2[x_2 := v_2]]$$

SEM-PATP-FOC

$$\frac{\text{NotVal } t}{\overline{C[t \triangleright \text{case}_m (x_1, x_2) \mapsto u]} \longrightarrow (C \circ (\Box \triangleright \text{case}_m (x_1, x_2) \mapsto u))[t]}$$

SEM-PATP-UNFOC

$$\overline{(C \circ (\Box \triangleright \text{case}_m (x_1, x_2) \mapsto u))[v]} \longrightarrow C[v \triangleright \text{case}_m (x_1, x_2) \mapsto u]$$

SEM-PATP-RED

$$\overline{C[(v_1, v_2) \triangleright \text{case}_m (x_1, x_2) \mapsto u]} \longrightarrow C[u[x_1 := v_1][x_2 := v_2]]$$

SEM-PATF-FOC

$$\frac{\text{NotVal } t}{\overline{C[t \triangleright \text{case}_m E_n x \mapsto u]} \longrightarrow (C \circ (\Box \triangleright \text{case}_m E_n x \mapsto u))[t]}$$

SEM-PATF-UNFOC

$$\overline{(C \circ (\Box \triangleright \text{case}_m E_n x \mapsto u))[v]} \longrightarrow C[v \triangleright \text{case}_m E_n x \mapsto u]$$

SEM-PATF-RED

$$\overline{C[E_n v' \triangleright \text{case}_m E_n x \mapsto u]} \longrightarrow C[u[x := v']]$$

SEM-MAP-FOC

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

SEM-MAP-UNFOC

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

SEM-MAP-RED-OPENAMPAR-FOC

$$\frac{\text{h}' = \text{max}(hvars(C))+1}{\overline{C[\text{H}\langle v_2 \wedge v_1 \rangle \triangleright \text{map } x \mapsto t']} \longrightarrow (C \circ (\overset{\text{op}}{\text{H}}_{\text{h}'} \langle v_2 [\text{H}\text{h}'] \wedge \Box \rangle)))[t'[x := v_1 [\text{H}\text{h}']]]}$$

SEM-OPENAMPAR-UNFOC

$$\overline{(C \circ \overset{\text{op}}{\text{H}} \langle v_2 \wedge \Box \rangle)[v_1]} \longrightarrow C[\text{H}\langle v_2 \wedge v_1 \rangle]$$

SEM-ToA-FOC

$$\frac{\text{NotVal } u}{\overline{C[\text{to}_{\times} u]} \longrightarrow (C \circ (\text{to}_{\times} \Box))[u]}$$

SEM-ToA-UNFOC

$$\overline{(C \circ (\text{to}_{\times} \Box))[v_2]} \longrightarrow C[\text{to}_{\times} v_2]$$

SEM-ToA-RED

$$\overline{C[\text{to}_{\times} v_2]} \longrightarrow C[\{\}_{\{v_2 \wedge ()\}}]$$

SEM-FROMA-FOC

$$\frac{\text{NotVal } t}{\overline{C[\text{from}_{\times} t]} \longrightarrow (C \circ (\text{from}_{\times} \Box))[t]}$$

SEM-FROMA-UNFOC

$$\overline{(C \circ (\text{from}_{\times} \Box))[v]} \longrightarrow C[\text{from}_{\times} v]$$

$$\begin{array}{c}
\text{SEM-FROMA-RED} \\
\hline
C[\text{from}_{\times} \{ \} \langle v_2 \text{ } \lambda E_{1\infty} v_1 \rangle] \longrightarrow C[(v_2, E_{1\infty} v_1)]
\end{array}
\qquad
\begin{array}{c}
\text{SEM-FILLU-FOC} \\
\text{NotVal } t \\
\hline
C[t \triangleleft ()] \longrightarrow (C \circ (\Box \triangleleft ())) [t]
\end{array}$$

$$\begin{array}{c}
\text{SEM-FILLU-UNFOC} \\
\hline
(C \circ (\Box \triangleleft ())) [v] \longrightarrow C[v \triangleleft ()]
\end{array}
\qquad
\begin{array}{c}
\text{SEM-FILLU-RED} \\
\hline
C[\rightarrow h \triangleleft ()] \longrightarrow C[h :=_{\{ \} } ()] [()]
\end{array}$$

$$\begin{array}{c}
\text{SEM-FILLL-FOC} \\
\text{NotVal } t \\
\hline
C[t \triangleleft \text{Inl}] \longrightarrow (C \circ (\Box \triangleleft \text{Inl})) [t]
\end{array}
\qquad
\begin{array}{c}
\text{SEM-FILLL-UNFOC} \\
\hline
(C \circ (\Box \triangleleft \text{Inl})) [v] \longrightarrow C[v \triangleleft \text{Inl}]
\end{array}$$

$$\begin{array}{c}
\text{SEM-FILLR-RED} \\
h' = \text{max}(hvars(C) \cup \{h\}) + 1 \\
\hline
C[\rightarrow h \triangleleft \text{Inl}] \longrightarrow C[h :=_{\{h'+1\}} \text{Inl } (h'+1)] [\rightarrow (h'+1)]
\end{array}
\qquad
\begin{array}{c}
\text{SEM-FILLR-FOC} \\
\text{NotVal } t \\
\hline
C[t \triangleleft \text{Inr}] \longrightarrow (C \circ (\Box \triangleleft \text{Inr})) [t]
\end{array}$$

$$\begin{array}{c}
\text{SEM-FILLR-UNFOC} \\
\hline
(C \circ (\Box \triangleleft \text{Inr})) [v] \longrightarrow C[v \triangleleft \text{Inr}]
\end{array}
\qquad
\begin{array}{c}
\text{SEM-FILLR-RED} \\
h' = \text{max}(hvars(C) \cup \{h\}) + 1 \\
\hline
C[\rightarrow h \triangleleft \text{Inr}] \longrightarrow C[h :=_{\{h'+1\}} \text{Inr } (h'+1)] [\rightarrow (h'+1)]
\end{array}$$

$$\begin{array}{c}
\text{SEM-FILLE-FOC} \\
\text{NotVal } t \\
\hline
C[t \triangleleft E_m] \longrightarrow (C \circ (\Box \triangleleft E_m)) [t]
\end{array}
\qquad
\begin{array}{c}
\text{SEM-FILLE-UNFOC} \\
\hline
(C \circ (\Box \triangleleft E_m)) [v] \longrightarrow C[v \triangleleft E_m]
\end{array}$$

$$\begin{array}{c}
\text{SEM-FILLE-RED} \\
h' = \text{max}(hvars(C) \cup \{h\}) + 1 \\
\hline
C[\rightarrow h \triangleleft E_m] \longrightarrow C[h :=_{\{h'+1\}} E_m (h'+1)] [\rightarrow (h'+1)]
\end{array}
\qquad
\begin{array}{c}
\text{SEM-FILLP-FOC} \\
\text{NotVal } t \\
\hline
C[t \triangleleft (,)] \longrightarrow (C \circ (\Box \triangleleft (,))) [t]
\end{array}$$

$$\begin{array}{c}
\text{SEM-FILLP-UNFOC} \\
\hline
(C \circ (\Box \triangleleft (,))) [v] \longrightarrow C[v \triangleleft (,)]
\end{array}$$

$$\begin{array}{c}
\text{SEM-FILLP-RED} \\
h' = \text{max}(hvars(C) \cup \{h\}) + 1 \\
\hline
C[\rightarrow h \triangleleft (,)] \longrightarrow C[h :=_{\{h'+1, h'+2\}} ((h'+1), (h'+2))] [\rightarrow (h'+1), \rightarrow (h'+2)]
\end{array}$$

$$\begin{array}{c}
\text{SEM-FILLF-FOC} \\
\text{NotVal } t \\
\hline
C[t \triangleleft (\lambda x_{\text{m}} \mapsto u)] \longrightarrow (C \circ (\Box \triangleleft (\lambda x_{\text{m}} \mapsto u))) [t]
\end{array}$$

$$\begin{array}{c}
\text{SEM-FILLF-UNFOC} \\
\hline
(C \circ (\Box \triangleleft (\lambda x_{\text{m}} \mapsto u))) [v] \longrightarrow C[v \triangleleft (\lambda x_{\text{m}} \mapsto u)]
\end{array}$$

$$\begin{array}{c}
\text{SEM-FILLF-RED} \\
\hline
C[\rightarrow h \triangleleft (\lambda x_{\text{m}} \mapsto u)] \longrightarrow C[h :=_{\{ \} } \lambda x_{\text{m}} \mapsto u] [()]
\end{array}
\qquad
\begin{array}{c}
\text{SEM-FILLCOMP-FOC1} \\
\text{NotVal } t \\
\hline
C[t \triangleleft \bullet t'] \longrightarrow (C \circ (\Box \triangleleft \bullet t')) [t]
\end{array}$$

$$\begin{array}{c}
\text{SEM-FILLCOMP-UNFOC1} \\
\hline
(C \circ (\Box \triangleleft \bullet t'))[v] \longrightarrow C[v \triangleleft \bullet t']
\end{array}
\qquad
\begin{array}{c}
\text{SEM-FILLCOMP-FOC2} \\
\text{NotVal } t' \\
\hline
C[v \triangleleft \bullet t'] \longrightarrow (C \circ (v \triangleleft \bullet \Box))[t']
\end{array}$$

$$\begin{array}{c}
\text{SEM-FILLCOMP-UNFOC2} \\
\hline
(C \circ (v \triangleleft \bullet \Box))[v'] \longrightarrow C[v \triangleleft \bullet v']
\end{array}
\qquad
\begin{array}{c}
\text{SEM-FILLCOMP-RED} \\
h' = \text{max}(hvars(C) \cup \{h\}) + 1 \\
\hline
C[\rightarrow h \triangleleft \bullet h(v_2 \wedge v_1)] \longrightarrow C[h :=_{(H \neq h')} v_2[H \neq h']][v_1[H \neq h']]
\end{array}$$

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>