Destination-passing style programming: a Haskell implementation

Thomas Bagrel

Destination-passing style programming: a Haskell implementation

Thomas Bagrel[1]

[1]INRIA/LORIA, Vandœuvre-lès-Nancy, 54500, France
[1]Tweag, Paris, 75012, France

Destination-passing style programming introduces destinations, which represent the address of a write-once memory cell. Those destinations can be passed as function parameters, and thus enable the caller of a function to keep control over memory management: the body of the called function will just be responsible of filling that memory cell. This is especially useful in functional programming languages, in which the body of a function is typically responsible for allocation of the result value.

Programming with destination in Haskell is an interesting way to improve performance of critical parts of some programs, without sacrificing memory guarantees. Indeed, thanks to a linearly-typed API I present, a write-once memory cell cannot be left uninitialized before being read, and is still disposed of by the garbage collector when it is not in use anymore, eliminating the risk of uninitialized read, memory leak, or double-free errors that can arise when memory is managed manually.

In this article, I present an implementation of destinations for Haskell, which relies on so-called compact regions. I demonstrate, in particular, a simple parser example for which the destination-based version uses 35% less memory and time than its naive counterpart for large inputs.

1 Introduction

Destination-passing style (DPS) programming takes its source in the early days of imperative languages with manual memory management. In the C programming language, it's quite common for a function not to allocate memory itself for its result, but rather to receive a reference to a memory location where to write its result (often named *out parameter*). In that scheme, the caller of the function has control over allocation and disposal of memory for the function result, and thus gets to choose where the latter will be written.

DPS programming is an adaptation of this idea for functional languages, based on two core concepts: having arbitrary data structures with *holes* — that is to say, memory cells that haven't been filled yet — and *destinations*, which are pointers to those holes. A destination can be passed around, as a first-class object of the language (unlike holes), and it allows remote action on its associated hole: when one *fills* the destination with a value, that value is in fact written in the hole. As structures are allowed to have holes, they can be built from the root down, rather than from the leaves up. Indeed, children of a parent

node no longer have to be specified when the parent node is created; they can be left empty (which leaves holes in the parent node), and added later through destinations to those holes. It is thus possible to write very natural solutions to problems for which the usual functional bottom-up building approach is ill-fitting. On top of better expressiveness, DPS programming can lead to better time or space performance for critical parts of a program, allowing for example tail-recursive map, or efficient difference lists.

That being said, DPS programming is not about giving unlimited manual control over memory or using mutations without restrictions. The existence of a destination is directly linked to the existence of an accompanying hole: we say that a destination is *consumed* when it has already been used to write something in its associated hole. It must not be reused after that point, to ensure immutability and prevent a range of memory errors.

In this paper, I design a destination API whose memory safety (write-once model) is ensured through a linear type discipline. Linear type systems are based on Girard's Linear logic [Gir95], and introduce the concept of *linearity*: one can express through types that a function will *consume* its argument exactly once given the function result is *consumed* exactly once. Linearity helps to manage resources — such as destinations — that one should not forget to *consume* (e.g. forgetting to fill a hole before reading a structure), but also that shouldn't be reused several times.

The Haskell programming language is equipped with support for linear types through its main compiler, *GHC*, since version 9.0.1 [BBN+18]. But Haskell is also a *pure* functional language, which means that side effects are usually not safe to produce outside of monadic contexts. This led me to set a slightly more refined goal: I wanted to hide impure memory effects related to destinations behind a *pure* Haskell API, and make the whole safe through the linear type discipline. Although the proofs of type safety haven't been made yet, the early practical results seem to indicate that my API is safe, and its purity makes it more convenient to adopt DPS in a codebase compared to a monadic approach that would be more "contaminating".

The main contributions of this paper are

- a linearly-typed API for destinations that let us build and manipulate data structures with holes while exposing a pure interface (Section 4);
- a first implementation of destinations for Haskell relying on so-called compact regions (Section 5), together with a performance evaluation (Section 6). Implementation code is available in [Bag23a] and [Bag23b] (specifically `src/Compact/Pure/Internal.hs` and `bench/Bench`).

2 A short primer on linear types

Linear Haskell [BBN+18] introduces the linear function arrow, `a ⊸ b`, that guarantees that the argument of the function will be consumed exactly once when the result of the function is consumed exactly once. On the other hand, the regular function arrow `a → b` doesn't guarantee how many times its argument will be consumed when its result is consumed once.

A value is said to be *consumed once* (or *consumed linearly*) when it is pattern-matched on and its sub-components are consumed once; or when it is passed as an argument to a linear function whose result is consumed once. A function is said to be *consumed once* when it is applied to an argument and when the result is consumed exactly once. We say that a variable `x` is *used linearly* in an expression `u` when consuming `u` once implies consuming `x` exactly once. Linearity on function arrows thus creates a chain of requirements about consumption of values, which is usually bootstrapped by using the *scope function* trick, as detailed in Section 4.2.

Unrestricted values Linear Haskell introduces a wrapper named `Ur` which is used to indicate that a value in a linear context doesn't have to be used linearly. `Ur a` is equivalent to $!a$ in linear logic, and there is an equivalence between `Ur a ⊸ b` and `a → b`.

The value `(x, y)` is said to be consumed linearly only when both `x` and `y` are consumed exactly once; whereas `Ur x` is considered to be consumed once as long as one pattern-matches on it, even if `x` is not consumed exactly once after (it can be consumed several times or not at all). Conversely, both `x` and `y` are used linearly in `(x, y)`, whereas `x` is not used linearly in `Ur x`. As a result, only values already wrapped in `Ur` or coming from the left of a non-linear arrow can be put in another `Ur` without breaking linearity. The only exceptions are values of types that implement the `Movable` typeclass such as `Int` or `()`. `Movable` provides `move :: a ⊸ Ur a` so a value can escape linearity restrictions.

Operators Some Haskell operators are often used in the rest of this article:

`(<&>) :: Functor f ⇒ f a ⊸ (a ⊸ b) ⊸ f b` is the same as `fmap` with the order of the arguments flipped: `x <&> f = fmap f x`;

`(⨟) :: () ⊸ b ⊸ b` is used to chain a linear operation returning `()` with one returning a value of type `b` without breaking linearity;

`Class ⇒ ...` is notation for typeclass constraints (resolved implicitly by the compiler).

3 Motivating examples for DPS programming

The following subsections present three typical settings in which DPS programming brings expressiveness or performance benefits over a more traditional functional implementation.

3.1 Efficient difference lists

Linked lists are a staple of functional programming, but they aren't efficient for concatenation, especially when the concatenation calls are nested to the left.

In an imperative context, it would be quite easy to concatenate linked lists efficiently. One just has to keep both a pointer to the root and to the last *cons* cell of each list. Then, to concatenate two lists, one just has to mutate the last *cons* cell of the first one to point to the root of the second list.

It isn't possible to do so in an immutable functional context though. Instead, *difference lists* can be used: they are very fast to concatenate, and then to convert back into a list. They tend to emulate the idea of having a mutable (here, write-once) last *cons* cell. Usually, a difference list `x1 : ... : xn : □` is encoded by function `\ys -> x1 : ... : xn : ys` taking a last element `ys :: [a]` and returning a value of type `[a]` too.

With such a representation, concatenation is function composition: `f1 <> f2 = f1 . f2`, and we have `mempty = id`[1], `toList f = f []` and `fromList xs = \ys → xs ++ ys`.

In DPS, instead of encoding the concept of a write-once hole with a function, we can represent the hole of type `[a]` as a first-class object with a *destination* of type `Dest [a]`. A difference list now becomes an actual data structure in memory — not just a pending computation — that has two handles: one to the root of the list of type `[a]`, and one to the yet-to-be-filled hole in the last cons cell, represented by the destination of type `Dest [a]`.

With the function encoding, it isn't possible to read the list until a last element of type `[a]` has been supplied to complete it. With the destination representation, this constraint must persist: the actual list `[a]` shouldn't be readable until the accompanying destination is filled, as pattern-matching on the hole would lead to a dreaded *segmentation fault*. This constraint is embodied by the `Incomplete a b` type of our destination API: `b` is what needs to be linearly consumed to make the `a` readable. The `b` side often carries the destinations of

[1] `mempty` and `<>` are the usual notations for neutral element and binary operation of a monoid in Haskell.

a structure. A difference list is then `type DList a = Incomplete [a] (Dest [a])`: the `Dest [a]` must be filled (with a `[a]`) to get a readable `[a]`.

The implementation of destination-backed difference lists is presented in Table 1. More details about the API primitives used by this implementation are given in Section 4. For now, it's important to note that `fill` is a function taking a constructor as a type parameter (often used with `@` for type parameter application, and `'` to lift a constructor to a type).

- `alloc` (Figure 1) returns a `DList a` which is exactly an `Incomplete [a] (Dest [a])` structure. There is no data there yet and the list that will be fed in `Dest [a]` is exactly the list that the resulting `Incomplete` will hold. This is similar to the function encoding where `\x → x` represents the empty difference list;
- `append` (Figure 3) adds an element at the tail position of a difference list. For this, it first uses `fill @'(:)` to fill the hole at the end of the list represented by `d :: Dest [a]` with a hollow *cons* cell with two new holes pointed by `dh :: Dest a` and `dt :: Dest [a]`. Then, `fillLeaf` fills the hole represented by `dh` with the value of type `a` to append. The hole of the resulting difference list is the one pointed by `dt :: Dest [a]` which hasn't been filled yet.
- `concat` (Figure 4) concatenates two difference lists, `i1` and `i2`. It uses `fillComp` to fill the destination `dt1` of the first difference list with the root of the second difference list `i2`. The resulting `Incomplete` object hence has the same root as the first list, holds the elements of both lists, and inherits the hole of the second list. Memory-wise, `concat` just writes an address into a memory cell; no move is required.
- `toList` (Figure 2) completes the incomplete structure by plugging *nil* into its hole with `fill @'[]` (whose individual behavior is presented in Figure 6) and removes the `Incomplete` wrapper as the structure is now complete, using `fromIncomplete_'`.

To use this API safely, it is imperative that values of type `Incomplete` are used linearly. Otherwise we could first complete a difference list with `l = toList i`, then add a new cons cell with a hole to `i` with `append i x` (actually reusing the destination inside `i` for the second time). Doing that creates a hole inside `l`, although it is of type `[a]` so we are allowed to pattern-match on it (so we might get a segfault)! The simplified API of Table 1 doesn't actually enforce the required linearity properties, I'll address that in Section 4.2.

This implementation of difference list matches closely the intended memory behaviour, we can expect it to be more efficient than the functional encoding. We'll see in Section 6 that the prototype implementation presented in Section 5 cannot yet demonstrate these performance improvements.

3.2 Breadth-first tree traversal

Consider the problem, which Okasaki attributes to Launchbury [Oka00]

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

This problem admits a straightforward implementation if we're allowed to mutate trees. Nevertheless, a pure implementation is quite tricky [Oka00, Gib93] and a very elegant, albeit very clever, solution was proposed recently [GKSW23].

With destinations as first-class objects in our toolbelt, we can implement a solution that is both easy to come up with and efficient, doing only a single breadth-first traversal pass on the original tree. The main idea is to keep a queue of pairs of a tree to be relabeled and of the destination where the relabeled result is expected (as destinations can be stored in arbitrary containers!) and process each of them when their turn comes. The implementation provided in Table 2, implements the slightly more general `mapAccumBFS` which applies on each node of the tree a relabeling function that can depend on a state.

```
1  data [a] = {- nil constructor -} [] | {- cons constructor -} (:) a [a]
2
3  type DList a = Incomplete [a] (Dest [a])
4
5  alloc :: DList a  -- API primitive (simplified signature w.r.t. Section 4)
6
7  append :: DList a ⊸ a → DList a
8  append i x =
9    i <&> \d → case fill @'(:) d of
10     (dh, dt) → fillLeaf x dh ; dt
11
12 concat :: DList a ⊸ DList a ⊸ DList a
13 concat i1 i2 = i1 <&> \dt1 → fillComp i2 dt1
14
15 toList :: DList a ⊸ [a]
16 toList i = fromIncomplete_' (i <&> \dt → fill @'[] dt)
```

Table 1. Implementation of difference lists with destinations

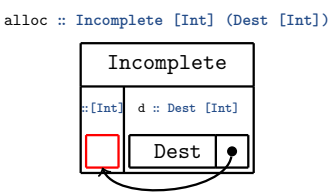


Figure 1. `alloc`

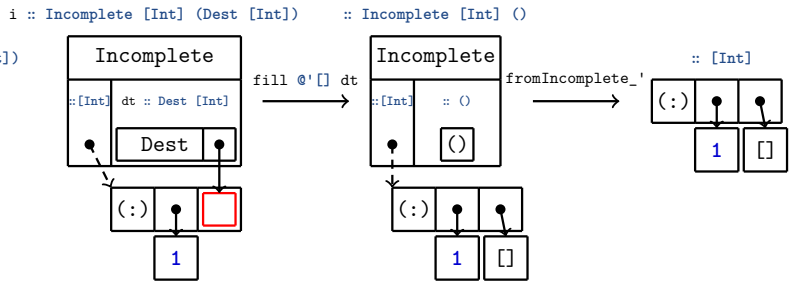


Figure 2. Memory behavior of `toList i`

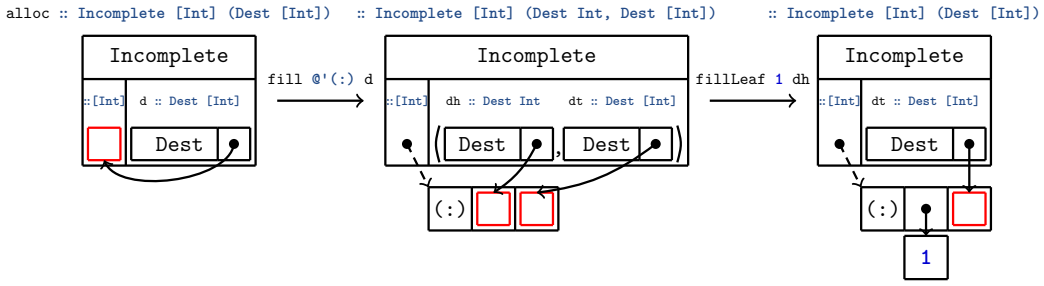


Figure 3. Memory behavior of `append alloc 1`

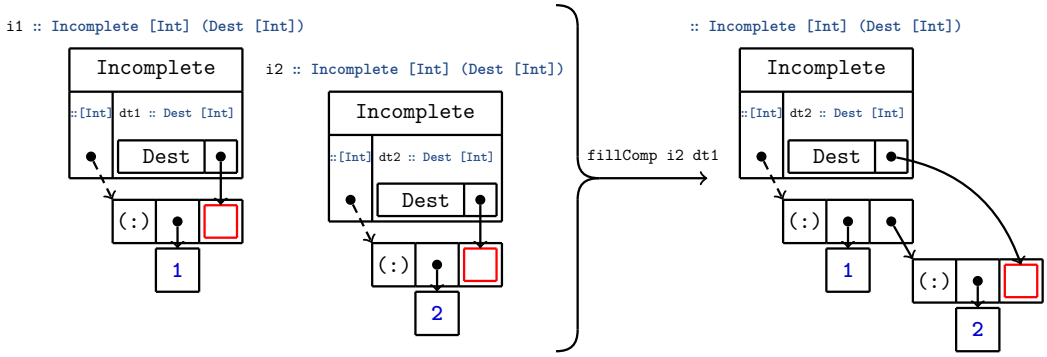


Figure 4. Memory behavior of `concat i1 i2` (based on `fillComp`)

```
1  data Tree a = Nil | Node a (Tree a) (Tree a)
2
3  relabelDPS :: Tree a → Tree Int
4  relabelDPS tree = fst (mapAccumBFS (\st _ → (st + 1, st)) 1 tree)
5
6  mapAccumBFS :: ∀ a b s. (s → a → (s, b)) → s → Tree a → (Tree b, s)
7  mapAccumBFS f s0 tree =
8    fromIncomplete' (    -- simplified alloc signature w.r.t. Section 4
9      alloc <&> \dtree → go s0 (singleton (Ur tree, dtree)))
10   where
11     go :: s → Queue (Ur (Tree a), Dest (Tree b)) ⊸ Ur s
12     go st q = case dequeue q of
13       Nothing → Ur st
14       Just ((utree, dtree), q') → case utree of
15         Ur Nil → fill @'Nil dtree ; go st q'
16         Ur (Node x tl tr) → case fill @'Node dtree of
17           (dy, dtl, dtr) →
18             let q'' = q' `enqueue` (Ur tl, dtl) `enqueue` (Ur tr, dtr)
19                 (st', y) = f st x
20               in fillLeaf y dy ; go st' q''
```

Table 2. Implementation of breadth-first tree traversal with destinations

Note that the signatures of `mapAccumBFS` and `relabelDPS` don't involve linear types. Linear types only appear in the inner loop `go`, which manipulates destinations. Linearity enforces the fact that every destination ever put in the queue is eventually filled at some point, which guarantees that the output tree is complete after the function has run.

As the state-transforming function `s → a → (s, b)` is non-linear, the nodes of the original tree won't be used in a linear fashion. However, we want to store these nodes together with their accompanying destinations in a queue of pairs, and destinations used to construct the queue have to be used linearly (because of `<&>` signature, which initially gives the root destination). That forces the queue to be used linearly, so the pairs too, so pairs' components too. Thus, we wrap the nodes of the input tree in the `Ur` wrapper, whose linear consumption allows for unrestricted use of its inner value, as detailed in Section 2.

This example shows how destinations can be used even in a non-linear setting in order to improve the expressiveness of the language. This more natural and less convoluted implementation of breadth-first traversal also presents great performance gains compared to the fancy functional implementation from [GKSW23], as detailed in Section 6.

3.3 Deserializing, lifetime, and garbage collection

In client-server applications, the following pattern is very frequent: the server receives a request from a client with a serialized payload, the server then deserializes the payload, runs some code, and respond to the request. Most often, the deserialized payload is kept alive for the entirety of the request handling. In a garbage collected language, there's a real cost to this: the garbage collector (GC) will traverse the deserialized payload again and again, although we know that all its internal pointers are live for the duration of the request.

Instead, we'd rather consider the deserialized payload as a single heap object, which doesn't need to be traversed, and is freed as a block. GHC supports this use-case with a feature named *compact regions* [YCA+15]. Compact regions contain normal heap objects, but the GC never follows pointers into a compact region. The flipside is that a compact region can only be collected when all of the objects it contains are dead.

With compact regions, we would first deserialize the payload normally, in the GC heap, then copy it into a compact region and only keep a reference to the copy. That way, internal

```
1  parseSList :: ByteString → Int → [SExpr] → Either Error SExpr
2  parseSList bs i acc = case bs !? i of
3    Nothing → Left (UnexpectedEOFSList i)
4    Just x → if
5      | x == ')' → Right (SList i (reverse acc))
6      | isSpace x → parseSList bs (i + 1) acc
7      | otherwise → case parseSExpr bs i of
8          Left err → Left err
9          Right child → parseSList bs (endPos child + 1) (child : acc)
```

Table 3. Implementation of the S-expression parser without destinations

```
1  parseSListDPS :: ByteString → Int → Dest [SExpr] ⊸ Either Error Int
2  parseSListDPS bs i d = case bs !? i of
3    Nothing → fill @'[] d ⨟ Left (UnexpectedEOFSList i)
4    Just x → if
5      | x == ')' → fill @'[] d ⨟ Right i
6      | isSpace x → parseSListDPS bs (i + 1) d
7      | otherwise →
8          case fill @'(:) d of
9            (dh, dt) -> case parseSExprDPS bs i dh of
10             Left err → fill @'[] dt ⨟ Left err
11             Right endPos → parseSListDPS bs (endPos + 1) dt
```

Table 4. Implementation of the S-expression parser with destinations

pointers of the region copy will never be followed by the GC, and that copy will be collected as a whole later on, whereas the original in the GC heap will be collected immediately.

However, we are still allocating two copies of the deserialized payload. This is wasteful, it would be much better to allocate directly in the region, but this isn't part of the original compact region API. Destinations are one way to accomplish this. In fact, as I'll explain in Section 5, my implementation of DPS for Haskell is backed by compact regions because they provide more freedom to do low-level memory operations without interfering with GC.

Given a payload serialized as S-expressions, let's see how using destinations and compact regions for the parser can lead to greater performance. S-expressions are parenthesized lists whose elements are separated by spaces. These elements can be of several types: int, string, symbol (a textual token with no quotes around it), or a list of other S-expressions.

Parsing an S-expression can be done naively with mutually recursive functions:

- `parseSExpr` scans the next character, and either dispatches to `parseSList` if it encounters an opening parenthesis, or to `parseSString` if it encounters an opening quote, or eventually parses the string into a number or symbol;
- `parseSList` calls `parseSExpr` to parse the next token, and then calls itself again until reaching a closing parenthesis, accumulating the parsed elements along the way.

Only the implementation of `parseSList` will be presented here as it is enough for our purpose, but the full implementation of both the naive and destination-based versions of the whole parser can be found in `src/Compact/Pure/SExpr.hs` of [Bag23b] .

The implementation presented in Table 3 is quite standard: the accumulator `acc` collects the nodes that are returned by `parseSExpr` in the reverse order (because it's the natural building order for a linked list without destinations). When the end of the list is reached (line 5), the accumulator is reversed, wrapped in the `SList` constructor, and returned.

We will see that destinations can bring very significative performance gains with only very little stylistic changes in the code. Accumulators of tail-recursive functions just have to be changed into destinations. Instead of writing elements into a list that will be reversed

```
1  data Token
2  consume   ::      Token ⊸ ()
3  dup2      ::      Token ⊸ (Token, Token)
4  withToken :: ∀ a. (Token ⊸ Ur a) ⊸ Ur a
5
6  data Incomplete a b
7  fmap                ::  ∀ a b c. (b ⊸ c) ⊸ Incomplete a b ⊸ Incomplete b c
8  alloc               ::  ∀ a.     Token ⊸ Incomplete a (Dest a)
9  intoIncomplete      ::  ∀ a.     Token ⊸ a → Incomplete a ()
10 fromIncomplete_     ::  ∀ a.     Incomplete a () ⊸ Ur a
11 fromIncomplete      ::  ∀ a b.   Incomplete a (Ur c) ⊸ Ur (a, c)
12
13 data Dest a
14 type family DestsOf lCtor a -- returns dests associated to fields of constructor
15 fill     :: ∀ lCtor a. Dest a ⊸ DestsOf lCtor a
16 fillComp :: ∀ a b.     Incomplete a b ⊸ Dest a ⊸ b
17 fillLeaf :: ∀ a.       a → Dest a ⊸ ()
```

Table 5. Destination API for Haskell

at the end as we did before, the program in the destination style will directly write the elements into their final location.

Code for `parseSListDPS` is presented in Table 4. Let's see what changed compared to the naive implementation:

- even for error cases, we are forced to consume the destination that we receive as an argument (to stay linear), hence we write some sensible default data to it (see line 3);
- the `SExpr` value resulting from `parseSExprDPS` is not collected by `parseSListDPS` but instead written directly into its final location by `parseSExprDPS` through the passing and filling of destination `dh` (see line 9);
- adding an element of type `SExpr` to the accumulator `[SExpr]` is replaced with adding a new cons cell with `fill @'(:)` into the hole represented by `Dest [SExpr]`, writing an element to the *head* destination, and then doing a recursive call with the *tail* destination passed as an argument (which has type `Dest [SExpr]` again);
- instead of reversing and returning the accumulator at the end of the processing, it is enough to complete the list by writing a nil element to the tail destination (with `fill @'[]`, see line 5), as the list has been built in a top-down approach;
- DPS functions return the offset of the next character to read instead of a parsed value.

Thanks to that new implementation which is barely longer (in terms of lines of code) than the naive one, the program runs almost twice as fast, mostly because garbage-collection time goes to almost zero. The detailed benchmark is available in Section 6.

4 API Design

Table 5 presents my pure API for functional DPS programming. This API is sufficient to implement all the examples of Section 3. This section explains its various parts in detail.

4.1 The `Incomplete` type

The main design principle behind DPS structure building is that no structure can be read before all its destinations have been filled. That way, incomplete data structures can be freely passed around and stored, but need to be completed before any pattern-matching can be made on them.

Hence we introduce a new data type `Incomplete a b` where `a` stands for the type of the structure being built, and `b` is the type of what needs to be linearly consumed before the structure can be read. The idea is that one can map over the `b` side, which will contain destinations or containers with destinations inside, until there is no destination left but just a non-linear value that can safely escape (e.g. `()`, `Int`, or something wrapped in `Ur`). When destinations from the `b` side are consumed, the structure on the `a` side is built little by little in a top-down fashion, as we showed in Figures 3 and 4. And when no destination remains on the `b` side, the value of type `a` no longer has holes, thus is ready to be released/read.

It can be released in two ways: with `fromIncomplete_`, the value on the `b` side must be unit (`()`), and just the complete `a` is returned, wrapped in `Ur`. With `fromIncomplete`, the type on the `b` side must be of the form `Ur c`, and then a pair `Ur (a, c)` is returned.

It is actually safe to wrap the structure that has been built in `Ur` because its leaves either come from non-linear sources (as `fillLeaf :: a → Dest a ⊸ ()` consumes its first argument non-linearly) or are made of 0-ary constructors added with `fill`, both of which can be used in an unrestricted fashion safely. Variants `fromIncomplete_'` and `fromIncomplete'` from the beginning of this article just drop the `Ur` wrapper.

Conversely, the function `intoIncomplete` takes a non-linear argument of type `a` and wraps it into an `Incomplete` with no destinations left to be consumed.

4.2 Ensuring write-once model for holes with linear types

Types aren't linear by themselves in Linear Haskell. Instead, functions can be made to use their arguments linearly or not. So in direct style, where the consumer of a resource isn't tied to the resource creation site, there is no way to state that the resource must be used exactly once:

```
createR           :: Resource -- no way to force the result to be used exactly once
consumeR          :: Resource ⊸ ()
exampleShouldFail :: () =
  let x = createR in consumeR x ; consumeR x -- valid even if x is consumed twice
```

The solution is to force the consumer of a resource to become explicit at the creation site of the resource, and to check through its signature that it is indeed a linear continuation:

```
withR       :: (Resource ⊸ a) ⊸ a
consumeR    :: Resource ⊸ ()
exampleFail :: () = withR (\x → consumeR x ; consumeR x) -- not linear
```

The `Resource` type is in positive position in the signature of `withR`, so that the function should somehow know how to produce a `Resource`, but this is opaque for the user. What matters is that a resource can only be accessed by providing a linear continuation to `withR`.

Still, this is not enough; because `\x → x` is indeed a linear continuation, one could use `withR (\x → x)` to leak a `Resource`, and then use it in a non-linear fashion in the outside world. Hence we must forbid the resource from appearing anywhere in the return type of the continuation. To do that, we ask the return type to be wrapped in `Ur`: because the resource comes from the left of a linear arrow, and doesn't implement `Movable`, it cannot be wrapped in `Ur` without breaking linearity (see Section 2). On the other hand, a `Movable` value of type `()` or `Int` can be returned:

```
withR'       :: (Resource ⊸ Ur a) ⊸ Ur a
consumeR     :: Resource ⊸ ()
exampleOk'   :: Ur ()       = withR' (\x → let u :: () = consumeR x' in move u)
exampleFail' :: Ur Resource = withR' (\x → Ur x) -- not linear
```

This explicit *scope function* trick will no longer be necessary when linear constraints will land in GHC (see [SKB+22]). In the meantime, this principle has been used to ensure safety of the DPS implementation in Haskell.

Ensuring linear use of `Incomplete` objects If an `Incomplete` object is used linearly, then its destinations will be written to exactly once; this is ensured by the signature of `fmap` for `Incomplete`s. So we need to ensure that `Incomplete` objects are used linearly. For that, we introduce a new type `Token`. A token can be linearly exchanged one-for-one with an `Incomplete` of any type with `alloc`, linearly duplicated with `dup2`, or linearly deleted with `consume`. However, it cannot be linearly stored in `Ur` as it doesn't implement `Movable`.

As in the example above, we just ensure that `withToken :: (Token ⊸ Ur a) ⊸ Ur a` is the only source of `Token`s around. Now, to produce an `Incomplete`, one must get a token first, so has to be in the scope of a continuation passed to `withToken`. Putting either a `Token` or `Incomplete` in `Ur` inside the continuation would make it non-linear. So none of them can escape the scope as is, but a structure built from an `Incomplete` and finalized with `fromIncomplete` would be automatically wrapped in `Ur`, thus could safely escape[2].

4.3 Filling functions for destinations

The last part of the API is the one in charge of actually building the structures in a top-down fashion. To fill a hole represented by `Dest a`, three functions are available:

`fillLeaf :: ∀ a. a → Dest a ⊸ ()` uses a value of type `a` to fill the hole represented by the destination. The destination is consumed linearly, but the value to fill the hole isn't (as indicated by the first non-linear arrow). Memory-wise, the address of the object `a` is written into the memory cell pointed to by the destination (see Figure 7).

`fillComp :: ∀ a b. Incomplete a b ⊸ Dest a ⊸ b` is used to plug two `Incomplete` objects together. The target `Incomplete` isn't represented in the signature of the function. Instead, only the target hole that will receive the address of the child is represented by `Dest a`; and `Incomplete a b` in the signature refers to the child object. A call to `fillComp` always takes place in the scope of `fmap`/`<&>` over the parent object:

```
1 parent :: Incomplete BigStruct (Dest SmallStruct, Dest OtherStruct)
2 child :: Incomplete SmallStruct (Dest Int)
3 comp = parent <&> \(ds, extra) → fillComp child ds
4        :: Incomplete BigStruct (Dest Int, Dest OtherStruct)
```

The resulting structure `comp` is morally a `BigStruct` like `parent`, that inherited the hole from the child structure (`Dest Int`) and still has its other hole (`Dest OtherStruct`) to be filled. An example of memory behavior of `fillComp` in action can be seen in Figure 4.

`fill :: ∀ lCtor a. Dest a ⊸ DestsOf lCtor a` lets us build structures using layers of hollow constructors. It takes a constructor as a type parameter (`lCtor`) and allocates a hollow heap object that has the same header/tag as the specified constructor but unspecified fields. The address of the allocated hollow constructor is written in the destination that is passed to `fill`. As a result, one hole is now filled, but there is one new hole in the structure for each field left unspecified in the hollow constructor that is now part of the bigger structure. So `fill` returns one destination of matching type for each of the fields of the constructor. An example of the memory behavior of `fill @'(:) :: Dest [a] ⊸ (Dest a, Dest [a])` is given in Figure 5 and the one of `fill @'[] :: Dest [a] ⊸ ()` is given in Figure 6.

`DestsOf` is a type family (i.e. a function operating on types) whose role is to map a constructor to the type of destinations for its fields. For example, `DestsOf '[] [a] = ()` and `DestsOf '(:) [a] = (Dest a, Dest [a])`. More generally, there is a duality between the type of a constructor `Ctor` $:: (f_1 \dots f_n) \to a$ and of the associated destination-filling functions `fill @'Ctor :: Dest a ⊸ (Dest` $f_1 \dots$ `Dest` f_n`)`. Destination-based data building can be seen as more general than the usual bottom-up constructor approach, as we can recover `Ctor` from the associated function `fill @'Ctor`, but not the reverse:

[2]This is why the `fromIncomplete'` and `fromIncomplete_'` variants aren't that useful in the actual memory-safe API (which differs slightly from the simplified examples of Section 3): here the built structure would be stuck in the scope function without its `Ur` escape pass.

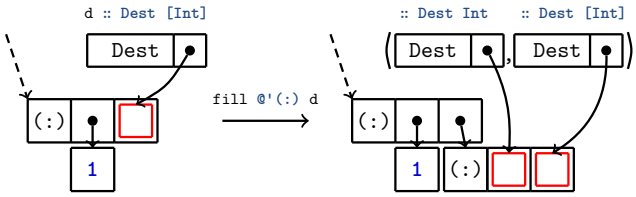


Figure 5. Memory behavior of `fill @'(:) :: Dest [a] ⊸ (Dest a, Dest [a])`

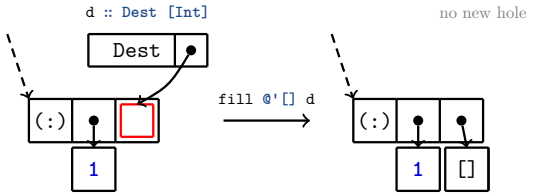


Figure 6. Memory behavior of `fill @'[] :: Dest [a] ⊸ ()`

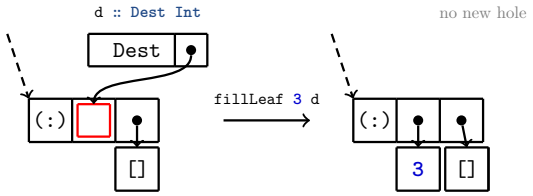


Figure 7. Memory behavior of `fillLeaf :: a → Dest [a] ⊸ ()`

```
1 Ctor :: (f1 ... fn) → a
2 Ctor (x1 ... xn) = fromIncomplete_' (
3   alloc <&> \(d :: Dest a) → case fill @'Ctor d of
4     (dx1 ... dxn) → fillLeaf x1 dx1 ⨟ ... ⨟ fillLeaf xn dxn)
```

5 Implementing destinations in Haskell

Having incomplete structures in the memory inherently introduces a lot of tension with both the garbage collector and compiler. Indeed, the GC assumes that every heap object it traverses is well-formed, whereas incomplete structures are absolutely ill-formed: they contain uninitialized pointers, which the GC should absolutely not follow.

The tension with the compiler is of lesser extent. The compiler can make some optimizations because it assumes that every object is immutable, while DPS programming breaks that guarantee by mutating constructors after they have been allocated (albeit only one update can happen). Fortunately, these errors are easily detected when implementing the API, and fixed by asking GHC not to inline specific parts of the code (with pragmas).

5.1 Compact Regions

As I teased in Section 3.3, *compact regions* from [YCA+15] make it very convenient to implement DPS programming in Haskell. A compact region represents a memory area in the Haskell heap that is almost fully independent from the GC and the rest of the garbage-collected heap. For the GC, each compact region is seen as a single heap object with a single lifetime. The GC can efficiently check whether there is at least one pointer in the garbage-collected heap that points into the region, and while this is the case, the region is kept alive. When this condition is no longer matched, the whole region is discarded. The

result is that the GC won't traverse any node from the region: it is treated as one opaque block (even though it is actually implemented as a chain of blocks of the same size, that doesn't change the principle). Also, compact regions are immobile in memory; the GC won't move them, so a destination can just be implemented as a raw pointer (type `Addr#` in Haskell): `data Dest r a = Dest Addr#`

By using compact regions to implement DPS programming, we completely elude the concerns of tension between the garbage collector and incomplete structures we want to build. Instead, we get two extra restrictions. First, every structure in a region must be in a fully-evaluated form. Regions are strict, and a heap object that is copied to a region is first forced into normal form. This might not always be a win; sometimes laziness, which is the default *modus operandi* of the garbage-collected heap, might be preferable.

Secondly, data in a region cannot contain pointers to the garbage-collected heap, or pointers to other regions: it must be self-contained. That forces us to slightly modify the API, to add a phantom type parameter `r` which tags each object with the identifier of the region it belongs to. There are two related consequences: `fillLeaf` has to copy each *leaf* value from the garbage-collected heap into the region in which it will be used as a leaf; and `fillComp` can only plug together two `Incomplete`s that come from the same region.

A typeclass `Region r` is also needed to carry around the details about a region that are required for the implementation. This typeclass has a single method `reflect`, not available to the user, that returns the `RegionInfo` structure associated to identifier `r`.

The `withRegion` function is the new addition to the modified API presented in Table 6 (the `Token` type and its associated functions `dup2` and `consume` are unchanged). `withRegion` is mostly a refinement over the `withToken` function from Table 5. It receives a continuation in which `r` must be a free type variable. It then spawns both a new compact region and a fresh type `r` (not a variable), and uses the `reflection` library to provide an instance of `Region r` on-the-fly that links `r` and the `RegionInfo` for the new region, and calls the continuation at type `r`. This is fairly standard practice since [LPJ94].

5.2 Representation of `Incomplete` objects

Ideally, as we detailed in the API, we want `Incomplete r a b` to contains an `a` and a `b`, and let the `a` free when the `b` is fully consumed (or linearly transformed into `Ur c`). So the most straightforward implementation for `Incomplete` would be a pair `(a, b)`, where `a` in the pair is only partially complete.

It is also natural for `alloc` to return an `Incomplete r a (Dest a)`: there is nothing more here than an empty memory cell (named *root receiver*) of type `a` which the associated destination of type `Dest a` points to, as presented in Figure 1. A bit like the identity function, whatever goes in the hole is exactly what will be retrieved in the `a` side.

If `Incomplete r a b` is represented by a pair `(a, b)`, then the root receiver should be the first field of the pair. However, the root receiver must be in the region, otherwise the GC might follow the garbage pointer that lives inside; whereas the `Incomplete` wrapper must be in the garbage-collected heap so that it can sometimes be optimized away by the compiler, and always deallocated as soon as possible.

One potential solution is to represent `Incomplete r a b` by a pair `(Ur a, b)` where `Ur` is allocated inside the region and its field `a` serves as the root receiver. With this approach, the issue of `alloc` representation is solved, but every `Incomplete` will now allocate a few words in the region (to host the `Ur` constructor) that won't be collected by the GC for a long time even if the parent `Incomplete` is collected. This makes `intoIncomplete` quite inefficient memory-wise too, as the `Ur` wrapper is useless for already complete structures.

The desired outcome is to only allocate a root receiver in the region for actual incomplete structures, and skip that allocation for already complete structures that are turned into an `Incomplete` object, while preserving a same type for both use-cases. This is made possible by replacing the `Ur` wrapper inside the `Incomplete` by an indirection object (`stg_IND` label)

```
type Region r :: Constraint
withRegion :: ∀ a. (∀ r. Region r ⇒ Token ⊸ Ur a) ⊸ Ur a

data Incomplete r a b
fmap            :: ∀ r a b c. (b ⊸ c) ⊸ Incomplete r a b ⊸ Incomplete r b c
alloc           :: ∀ r a.     Region r ⇒ Token ⊸ Incomplete r a (Dest r a)
intoIncomplete  :: ∀ r a.     Region r ⇒ Token ⊸ a → Incomplete r a ()
fromIncomplete_ :: ∀ r a.     Region r ⇒ Incomplete r a () ⊸ Ur a
fromIncomplete  :: ∀ r a b.   Region r ⇒ Incomplete r a (Ur c) ⊸ Ur (a, c)

data Dest r a
type family DestsOf lCtor r a
fill     :: ∀ lCtor r a. Region r ⇒ Dest r a ⊸ DestsOf lCtor r a
fillComp :: ∀ r a b.     Region r ⇒ Incomplete r a b ⊸ Dest r a ⊸ b
fillLeaf :: ∀ r a.       Region r ⇒ a → Dest r a ⊸ ()
```

Table 6. Destination API using compact regions

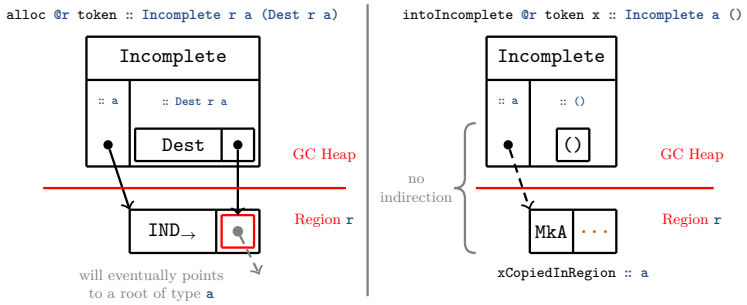


Figure 8. Memory behaviour of `alloc` and `intoIncomplete` in the region implementation

for the actually-incomplete case. `Incomplete r a b` will be represented by a pair `(a, b)` allocated in the garbage-collected heap, with slight variations as illustrated in Figure 8:

- in the pair `(a, b)` returned by `alloc`, the `a` side points to an indirection object (a sort of constructor with one field, whose resulting type `a` is the same as the field type `a`), that is allocated in the region, and serves as the root receiver;
- in the pair `(a, b)` returned by `intoIncomplete`, the `a` side directly points to the object of type `a` that has been copied to the region.

The implementation of `fromIncomplete_` is then relatively straightforward. It allocates a hollow `Ur` □ in the region, writes the address of the complete structure into it, and returns the `Ur` (an alternative would have been to use a regular `Ur` allocated in the GC heap).

5.3 Deriving `fill` for all constructors with `Generics`

The `fill @lCtor @r @a` function should plug a new hollow constructor `Ctor` □ `:: a` into the hole of an existing incomplete structure, and return one destination object per new hole in the structure (corresponding to the unspecified fields of the new hollow constructor). Naively, we would need one `fill` function per constructor, but that cannot be realistically implemented. Instead, we have to generalize all `fill` functions into a typeclass `Fill lCtor a`, and derive an instance of the typeclass (i.e. implement `fill`) generically for any constructor, based only on statically-known information about that constructor.

In Section 5.4, we will see how to allocate a hollow heap object for a specified constructor (which is known at compile-time). The only other information we need to implement `fill` generically is the shape of the constructor, and more precisely the number and type of its

fields. So we will leverage `GHC.Generics` to find the required information.

`GHC.Generics` is a built-in Haskell library that provides compile-time inspection of a type metadata through the `Generic` typeclass: list of constructors, their fields, memory representation, etc. And that typeclass can be derived automatically for any type! Here's, for example, the `Generic` representation of `Maybe a`:

```
repl> :k! Rep (Maybe a) () -- display the Generic representation of Maybe a
M1 D (MetaData "Maybe" "GHC.Maybe" "base" False) (
  M1 C (MetaCons "Nothing" PrefixI False) U1
  :+: M1 C (MetaCons "Just" PrefixI False) (M1 S [...] (K1 R a)))
```

We see that there are two different constructors (indicated by `M1 C ...` lines): `Nothing` has zero fields (indicated by `U1`) and `Just` has one field of type `a` (indicated by `K1 R a`).

With a bit of type-level programming[3], we can extract the parts of that representation which are related to the constructor `lCtor` and use them inside the instance head of `Fill lCtor a` so the implementation of `fill` can depend on them. That's how we can give the proper types to the destinations returned by that function for a specified constructor. The `DestsOf lCtor a :: Type` type family also uses the generic representation of `a` to extract what it needs to know about `lCtor` and its fields.

5.4 Changes to GHC internals and RTS

We will see here how to allocate a hollow heap object for a given constructor, but let's first take a detour to give more context about the internals of the compiler.

Haskell's runtime system (RTS) is written in a mix of C and C--. The RTS has many roles, among which managing threads, organizing garbage collection or managing compact regions. It also defines various primitive operations, named *external primops*, that expose the RTS capabilities as normal functions. Despite all its responsibilities, however, the RTS is not responsible for the allocation of normal constructors (built in the garbage-collected heap). One reason is that it doesn't have all the information needed to build a constructor heap object, namely, the info table associated to the constructor.

The info table is what defines both the layout and behavior of a heap object. All heap objects representing a same constructor (let's say `Just`) have the same info table, even when the associated types are different (e.g. `Maybe Int` and `Maybe Bool`). Heap objects representing this constructor point to a label `<ctor>_con_info` that will be later resolved by the linker into an actual pointer to the shared info table.

The RTS is in fact a static piece of code that is compiled once when GHC is built. So the RTS has no direct way to access the information emitted during the compilation of a program. In other words, when the RTS runs, it has no way to inspect the program that it runs and info table labels have long been replaced by actual pointers so it cannot find them itself. But it is the one which knows how to allocate space inside a compact region.

As a result, I need to add two new primitives to GHC to allocate a hollow constructor:

- one *external primop* to allocate space inside a compact region for a hollow constructor. This primop has to be implemented inside the RTS for the aforementioned reasons;
- one *internal primop* (internal primops are macros which generates C-- code) that will be resolved into a normal albeit static value representing the info table pointer of a given constructor. This value will be passed as an argument to the external primop.

All the alterations to GHC that will be showed here are available in full form in [Bag23a].

External primop: allocate a hollow constructor in a region The implementation of the external primop is presented in Table 7. The `stg_compactAddHollowzh` function (whose equivalent on the Haskell side is `compactAddHollow#`) is mostly a glorified call to

[3] see `src/Compact/Pure/Internal.hs:418` in [Bag23b]

the `ALLOCATE` macro defined in the `Compact.cmm` file, which tries to do a pointer-bumping allocation in the current block of the compact region if there is enough space, and otherwise add a new block to the region.

As announced, this primop takes the info table pointer of the constructor to allocate as its second parameter (`W_ info`) because it cannot access that information itself. The info table pointer is then written to the first word of the heap object in the call to `SET_HDR`.

Internal primop: reify an info table label into a runtime value The only way, in Haskell, to pass a constructor to a primop so that the primop can inspect it, is to lift the constructor into a type-level literal. It's common practice to use a `Proxy a` (the unit type with a phantom type parameter) to pass the type `a` as an input to a function. Unfortunately, due to a quirk of the compiler, primops don't have access to the type of their arguments. They can, however, access their return type. So I'm using a phantom type `InfoPtrPlaceholder# a` as the return type, to pass the contructor as an input!

The gist of this implementation is presented in Table 8. The primop `reifyInfoPtr#` pattern-matches on the type `resTy` of its return value. In the case it reads a string literal, it resolves the primop call into the label `stg_<name>` (this is used in particular to retrieve `stg_IND` to allocate indirection heap objects). In the case it reads a lifted data constructor, it resolves the primop call into the label which corresponds to the info table pointer of that constructor. The returned `InfoPtrPlaceholder# a` can later be converted back to an `Addr#` using the `unsafeCoerceAddr` function.

As an example, here is how to allocate a hollow `Just` constructor in a compact region:

```
1 hollowJust :: Maybe a = compactAddHollow#
2   compactRegion#
3   (unsafeCoerceAddr (reifyInfoPtr# (# #) :: InfoPtrPlaceholder# 'Just ))
```

Built-in type family to go from a lifted constructor to the associated symbol The internal primop `reifyInfoPtr#` that we introduced above takes as input a constructor lifted into a type-level literal, so this is also what `fill` will use to know which constructor it should operate with. But `DestsOf` have to find the metadata of a constructor in the `Generic` representation of a type, in which only the constructor name appears.

So we added a new type family `LCtorToSymbol` inside GHC that inspects its (type-level) parameter representing a constructor, fetches its associated `DataCon` structure, and returns a type-level string (kind `Symbol`) carrying the constructor name, as presented in Table 9.

6 Evaluating the performance of DPS programming

Benchmarking methodology All over this article, I talked about programs in both naive style and DPS style. With DPS programs, the result is stored in a compact region, which also forces strictness i.e. the structure is automatically in fully evaluated form.

For naive versions, we have a choice to make on how to fully evaluate the result: either force each chunk of the result inside the GC heap (using `Control.DeepSeq.force`), or copy the result in a compact region that is strict by default (using `Data.Compact.compact`).

In programs where there is no particular long-lived piece of data, having the result of the function copied into a compact region isn't particularly desirable since it will generally inflate memory allocations. So we use `force` to benchmark the naive version of those programs (the associated benchmark names are denoted with a "*" suffix).

Concatenating lists and difference lists We compared three implementations. `foldr (++)*` has calls to `(++)` nested to the right, giving the most optimal context for list concatenation (it should run in $\mathcal{O}(n)$ time). `foldl' concat`λ^* uses function-backed

```
// compactAddHollow#
//   :: Compact# → Addr# → State# RealWorld → (# State# RealWorld, a #)
stg_compactAddHollowzh(P_ compact, W_ info) {
    W_ pp, ptrs, nptrs, size, tag, hp;
    P_ to, p; p = NULL;  // p isn't actually used by ALLOCATE macro
    again: MAYBE_GC(again); STK_CHK_GEN();

    pp = compact + SIZEOF_StgHeader + OFFSET_StgCompactNFData_result;
    ptrs  = TO_W_(%INFO_PTRS(%STD_INFO(info)));
    nptrs  = TO_W_(%INFO_NPTRS(%STD_INFO(info)));
    size = BYTES_TO_WDS(SIZEOF_StgHeader) + ptrs + nptrs;

    ALLOCATE(compact, size, p, to, tag);
    P_[pp] = to;
    SET_HDR(to, info, CCS_SYSTEM);
  #if defined(DEBUG)
    ccall verifyCompact(compact);
  #endif
    return (P_[pp]);
}
```

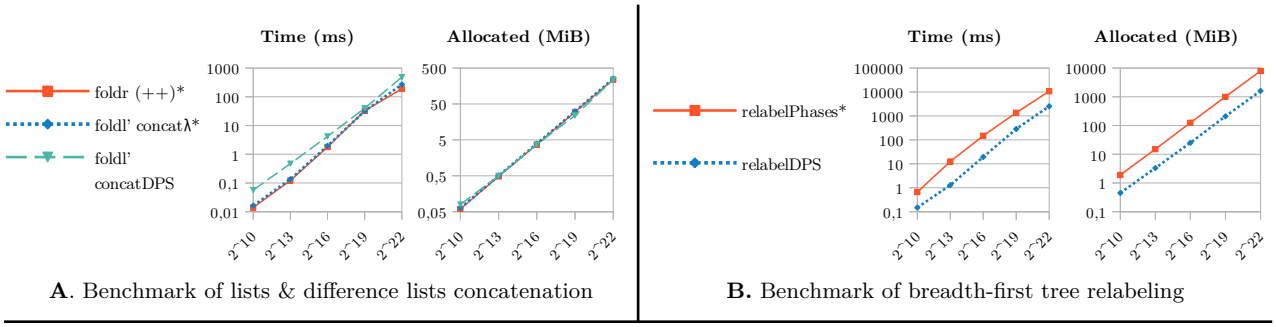
Table 7. `compactAddHollow#` implementation in `rts/Compact.cmm`

```
case primop of
  [...]
  ReifyStgInfoPtrOp → \_ →  -- we don't care about the function argument (# #)
    opIntoRegsTy $ \[res] resTy → emitAssign (CmmLocal res) $ case resTy of
      -- when 'a' is a Symbol, and extracts the symbol value in 'sym'
      TyConApp _addrLikeTyCon [_typeParamKind, LitTy (StrTyLit sym)] →
          CmmLit (CmmLabel (
            mkCmmInfoLabel rtsUnitId (fsLit "stg_" `appendFS` sym)))
      -- when 'a' is a lifted data constructor, extracts it as a DataCon
      TyConApp _addrLikeTyCon [_typeParamKind, TyConApp tyCon _]
        | Just dataCon ← isPromotedDataCon_maybe tyCon →
          CmmLit (CmmLabel (
            mkConInfoTableLabel (dataConName dataCon) DefinitionSite))
      _ → [...] -- error when no pattern matches
```

Table 8. `reifyInfoPtr#` implementation in `compiler/GHC/StgToCmm/Prim.hs`

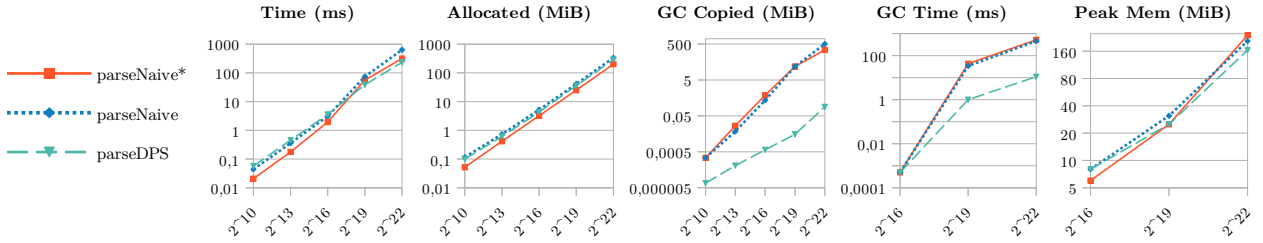
```
matchFamLCtorToSymbol :: [Type] → Maybe (CoAxiomRule, [Type], Type)
matchFamLCtorToSymbol [kind, ty]
  | TyConApp tyCon _ ← ty, Just dataCon ← isPromotedDataCon_maybe tyCon =
      let symbolLit = (mkStrLitTy . occNameFS . occName . getName $ dataCon)
       in Just (axLCtorToSymbolDef, [kind, ty], symbolLit)
matchFamLCtorToSymbol tys = Nothing

axLCtorToSymbolDef =
  mkBinAxiom "LCtorToSymbolDef" typeLCtorToSymbolTyCon Just
    (\case { TyConApp tyCon _ → isPromotedDataCon_maybe tyCon ; _ → Nothing })
    (\_ dataCon → Just (mkStrLitTy . occNameFS . occName . getName $ dataCon))
```

Table 9. `LCtorToSymbol` implementation in `compiler/GHC/Builtin/Types/Literal.hs`



A. Benchmark of lists & difference lists concatenation

B. Benchmark of breadth-first tree relabeling



	Time (ms)					Allocated (MiB)					GC Copied (MiB)					GC Time (ms)			Peak Mem (MiB)		
Size	2^{10}	2^{13}	2^{16}	2^{19}	2^{22}	2^{10}	2^{13}	2^{16}	2^{19}	2^{22}	2^{10}	2^{13}	2^{16}	2^{19}	2^{22}	2^{16}	2^{19}	2^{22}	2^{16}	2^{19}	2^{22}
parseNaive*	0.0	0.2	1.9	56.0	315.0	0.1	0.4	3.2	25.0	201.0	0.0	0.0	0.7	28.0	235.0	0.0	43.0	517.0	6.0	25.0	240.0
parseNaive	0.0	0.4	3.1	74.0	633.0	0.1	0.7	5.2	41.0	334.0	0.0	0.0	0.4	28.0	503.0	0.0	34.0	455.0	8.0	31.0	207.0
parseDPS	0.1	0.5	3.6	**38.4**	**233.0**	0.1	0.6	4.3	36.0	289.0	**0.0**	**0.0**	**0.0**	**0.0**	**0.2**	**0.0**	**1.0**	**11.0**	8.0	**25.0**	**164.0**

C. Benchmark of S-expressions parser

Figure 9. Benchmarks performed on AMD EPYC 7401P @ 2.0 GHz (single core, `-N1 -O2`)

difference lists, and `foldl' concatDPS` uses destination-backed ones, so both should run in $\mathcal{O}(n)$ even if calls to concat are nested to the left.

We see in part **A** of Figure 9 that the destination-backed difference lists have a comparable memory use as the two other linear implementations, while being quite slower (by a factor 2-4) on all datasets. We would expect better results though for a DPS implementation outside of compact regions because those cause extra copying.

Breadth-first relabeling We see in part **B** of Figure 9 that the destination-based tree traversal is almost one order of magnitude more efficient, both time-wise and memory-wise, compared to the implementation based on *Phases* applicatives presented in [GKSW23].

Parsing S-expressions In part **C** of Figure 9, we compare the naive implementation of the S-expression parser and the DPS one (see Section 3.3). For this particular program, where using compact regions might reduce the future GC load of the application, it is relevant to benchmark the naive version twice: once with `force` and once with `compact`.

The DPS version starts by being less efficient than the naive versions for small inputs, but gets an edge as soon as garbage collection kicks in (on datasets of size $\leq 2^{16}$, no garbage collection cycle is required as the heap size stays small).

On the largest dataset ($2^{22} \simeq$ 4MiB file), the DPS version still makes about 45% more allocations than the starred naive version, but uses 35% less memory at its peak, and more importantly, spends 47× less time in garbage collection. As a result, the DPS version only takes 0.55-0.65× the time spent by the naive versions, thanks to garbage collection savings. All of this also indicates that most of the data allocated in the GC heap by the DPS version just lasts one generation and thus can be discarded very early by the GC, without needing to be copied into the next generation, unlike most nodes allocated by the naive versions.

Finally, copying the result of the naive version to a compact region (for future GC savings) incurs a significant time and memory penalty, that the DPS version offers to avoid.

7 Related work

The idea of functional data structures with write-once holes is not new. Minamide already proposed in [Min98] a variant of λ-calculus with support for *hole abstractions*, which can be represented in memory by an incomplete structure with one hole and can be composed efficiently with each other (as with `fillComp` in Figure 4). With such a framework, it is fully possible to implement destination-backed difference lists for example.

However, in Minamide's work, there is no concept of destination: the hole in a structure can only be filled if one has the structure itself at hand. On the other hand, our approach introduces destinations, as a way to interact with a hole remotely, even when one doesn't have a handle to the associated structure. Because destinations are first-class objects, they can be passed around or stored in collections or other structure, while preserving memory safety. This is the major step forward that our paper presents.

More recently, [PP13] introduced the Mezzo programming language, in which mutable data structures can be freezed into immutable ones after having been completed. This principle is used to some extend in their list standard library module, to mimic a form of DPS programming. An earlier appearance of DPS programming as a mean to achieve better performance in a mutable language can also be seen in [Lar89].

Finally, both [SFPJV17] and [BCS21] use DPS programming to make list or array processing algorithms more efficient in a functional, immutable context, by turning non tail-recursive functions into tail-recursive DPS ones. More importantly, they present an automatized way to go from a naive program to its tail-recursive version. However, holes/destinations are only supported at an intermediary language level, while both [Min98] and our present work support safe DPS programming in user-land. In a broader context, [LLS23] presents a system in which linearity is used to identify where destructive updates can be made, so as to reuse the same constructor instead of deallocating and reallocating one; but this optimization technique is still mostly invisible for the user, unlike ours which is made explicit.

8 Conclusion and future work

Programming with destinations definitely has a place in the realm of functional programming, as the recent adoption of *Tail Modulo Cons* [BCS21] in the OCaml compiler shows. In this paper, we have shown how destination-passing style programming can be used in user-land in Haskell safely, thanks to a linear type discipline. Adopting DPS programming opens the way for more natural and efficient programs in a variety of contexts, where the major points are being able to build structures in a top-down fashion, manipulating and composing incomplete structures, and managing holes in these structures through first-class objects (destinations). Our DPS implementation relies only on a few alterations to the compiler, thanks to *compact regions* that are already available as part of GHC. Simultaneously, it allows to build structures in those regions without copying, which wasn't possible before.

There are two limitations that we would like to lift in the future. First, DPS programming could be useful outside of compact regions: destinations could probably be used to manipulate the garbage-collected heap (with proper read barriers in place), or other forms of secluded memory areas that aren't traveled by the GC (RDMA, network serialized buffers, etc.). Secondly, at the moment, the type of `fillLeaf` implies that we can't store destinations (which are always linear) in a difference list implemented as in Section 3.1, whereas we can store them in a regular list or queue (like we do, for instance, in Section 3.2). This unwelcome restriction ensures memory safety but it's quite coarse grain. In the future we'll be trying to have a more fine-grained approach that would still ensure safety.

References

[Bag23a] Bagrel, Thomas: GHC with support for hollow constructor allocation. Software Heritage, `swh:1:dir:84c7e717fd5f189c6b6222e0fc92d0a82d755e7c; origin=https://github.com/tweag/ghc; visit=swh:1:snp:141fa3c28e01574deebb6cc91693c75f49717c32; anchor=swh:1:rev:184f838b352a0d546e574bdeb83c8c190e9dfdc2`, 2023. Accessed: 2023-10-19.

[Bag23b] Bagrel, Thomas: `linear-dest`, a Haskell library that adds supports for DPS programming. Software Heritage, `swh:1:rev:0e7db2e6b24aad348837ac78d8137712c1d8d12a; origin=https://github.com/tweag/linear-dest; visit=swh:1:snp:c0eb2661963bb176204b46788f4edd26f72ac83c`, 2023. Accessed: 2023-10-19.

[BBN+18] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones and Arnaud Spiwack: Linear Haskell: practical linearity in a higher-order polymorphic language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–29, January 2018. arXiv:1710.09756 [cs].

[BCS21] Frédéric Bour, Basile Clément and Gabriel Scherer: Tail Modulo Cons. *arXiv:2102.09823 [cs]*, February 2021. arXiv: 2102.09823.

[Gib93] Jeremy Gibbons: Linear-time Breadth-first Tree Algorithms: An Exercise in the Arithmetic of Folds and Zips. (No. 71), 1993. Number: No. 71.

[Gir95] J.-Y. Girard: Linear Logic: its syntax and semantics. *In* Jean-Yves Girard, Yves Lafont and Laurent Regnier, éditeurs: *Advances in Linear Logic*, pages 1–42. Cambridge University Press, Cambridge, 1995.

[GKSW23] Jeremy Gibbons, Donnacha Oisín Kidney, Tom Schrijvers and Nicolas Wu: Phases in Software Architecture. *In Proceedings of the 1st ACM SIGPLAN International Workshop on Functional Software Architecture*, pages 29–33, Seattle WA USA, August 2023. ACM.

[Lar89] James Richard Larus: *Restructuring symbolic programs for concurrent execution on multiprocessors*. phd, University of California, Berkeley, 1989. AAI9006407.

[LLS23] Anton Lorenzen, Daan Leijen and Wouter Swierstra: FP^2: Fully in-Place Functional Programming. *Proceedings of the ACM on Programming Languages*, 7(ICFP):275–304, August 2023.

[LPJ94] John Launchbury and Simon L. Peyton Jones: Lazy functional state threads. *In Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94, pages 24–35, New York, NY, USA, June 1994. Association for Computing Machinery.

[Min98] Yasuhiko Minamide: 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, pages 75–84, New York, NY, USA, January 1998. Association for Computing Machinery.

[Oka00] Chris Okasaki: Breadth-first numbering: lessons from a small exercise in algorithm design. *In Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, pages 131–136, New York, NY, USA, September 2000. Association for Computing Machinery.

[PP13] Jonathan Protzenko and François Pottier: Programming with Permissions in Mezzo. *In Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pages 173–184, September 2013. arXiv:1311.7242 [cs].

[SFPJV17] Amir Shaikhha, Andrew Fitzgibbon, Simon Peyton Jones and Dimitrios Vytiniotis: Destination-passing style for efficient memory management. *In Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*, pages 12–23, Oxford UK, September 2017. ACM.

[SKB+22] Arnaud Spiwack, Csongor Kiss, Jean-Philippe Bernardy, Nicolas Wu and Richard A. Eisenberg: Linearly qualified types: generic inference for capabilities and uniqueness. *Proceedings of the ACM on Programming Languages*, 6(ICFP): 95:137–95:164, August 2022.

[YCA+15] Edward Z. Yang, Giovanni Campagna, Ömer S. Aacan, Ahmed El-Hassany, Abhishek Kulkarni and Ryan R. Newton: Efficient communication and collection with compact normal forms. *In Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, pages 362–374, Vancouver BC Canada, August 2015. ACM.