Formalization and Implementation of Safe Destination Passing in Pure Functional Programming Settings

Thomas BAGREL

PhD Defense — November 14th, 2025

Programming languages

- Imperative languages: instructions step-by-step

  how? | mutability | untracked side-effects

- Functional languages: compose expressions

  what? | immutability | purity | first-class functions

Functional languages – why?

Modeled after mathematical principles

Functional languages – why?

Modeled after mathematical principles

→ Easier to reason about behavior

Functional languages – why?

Modeled after mathematical principles

- → Easier to reason about behavior
- → Better safety guarantees

Functional languages – why?

Modeled after mathematical principles

→ Easier to reason about behavior

→ Better safety guarantees

Memory managed automatically: unpredictable overhead / hard to tune

Functional data structures

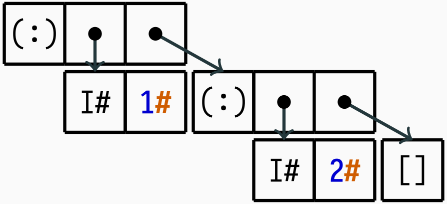
Data structures are heap-allocated; made of **linked** heap objects:

- a pointer to the **info table** (static struct describing the constructor)
- for each field, a pointer to this field's value (except for primitive types)

`[1, 2]`

a.k.a.

`(:) 1 ((:) 2 [])` ⇝



(:) is list "cons" constructor
[] is list "nil" constructor
I# is constructor for boxed integers
1# is the primitive/unboxed integer "one"

Building order in functional languages – current

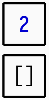
Data structures: immutable, thus built from the leaves up to the root.

- The value of a field must be an existing, fully constructed structure

Building order in functional languages – current

Data structures: immutable, thus built from the leaves up to the root.

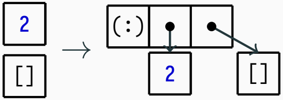
- The value of a field must be an existing, fully constructed structure



Building order in functional languages – current

Data structures: immutable, thus built from the leaves up to the root.

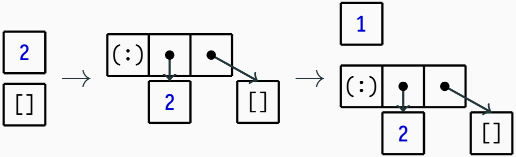
- The value of a field must be an existing, fully constructed structure



Building order in functional languages – current

Data structures: immutable, thus built from the leaves up to the root.

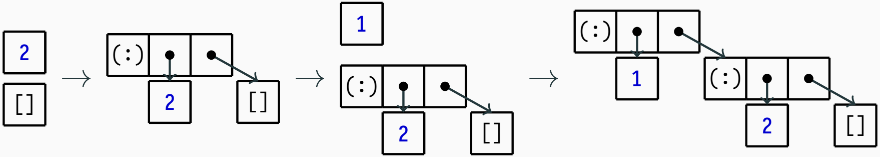
- The value of a field must be an existing, fully constructed structure



Building order in functional languages – current

Data structures: immutable, thus built from the leaves up to the root.

- The value of a field must be an existing, fully constructed structure



- Forces us to build structures in an order that might not be the most natural one.

Building order in functional languages – goal

What about lifting this limitation?

- Allowing pieces of data structures to be connected like Lego bricks in **any order**

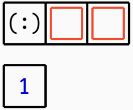
  pioneered by "A Functional Representation of Data Structures with a Hole", Minamide (1998)

□ *denotes a "hole" in the structure (an uninitialized memory cell)*

Building order in functional languages – goal

What about lifting this limitation?

- ▶ Allowing pieces of data structures to be connected like Lego bricks in **any order**
  pioneered by "A Functional Representation of Data Structures with a Hole", Minamide (1998)



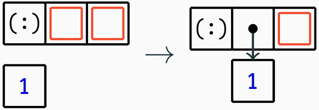
□ *denotes a "hole" in the structure (an uninitialized memory cell)*

Building order in functional languages – goal

What about lifting this limitation?

- Allowing pieces of data structures to be connected like Lego bricks in **any order**

  pioneered by “A Functional Representation of Data Structures with a Hole”, Minamide (1998)



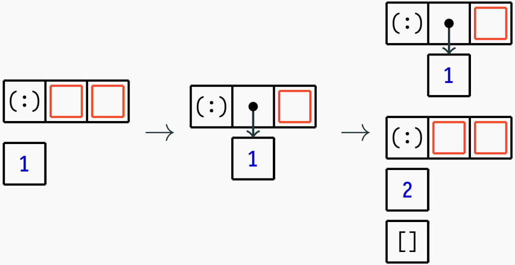
□ *denotes a “hole” in the structure (an uninitialized memory cell)*

Building order in functional languages – goal

What about lifting this limitation?

- Allowing pieces of data structures to be connected like Lego bricks in **any order**

  pioneered by “A Functional Representation of Data Structures with a Hole”, Minamide (1998)



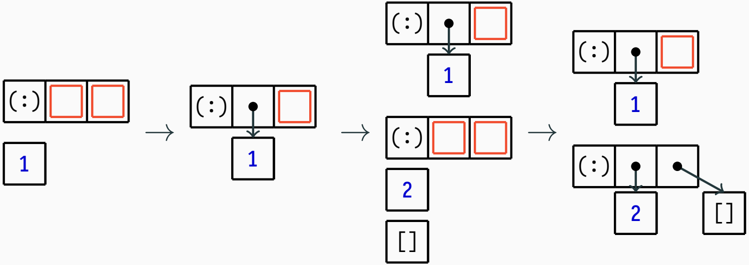
□ *denotes a “hole” in the structure (an uninitialized memory cell)*

Building order in functional languages – goal

What about lifting this limitation?

- Allowing pieces of data structures to be connected like Lego bricks in **any order**

  pioneered by "A Functional Representation of Data Structures with a Hole", Minamide (1998)



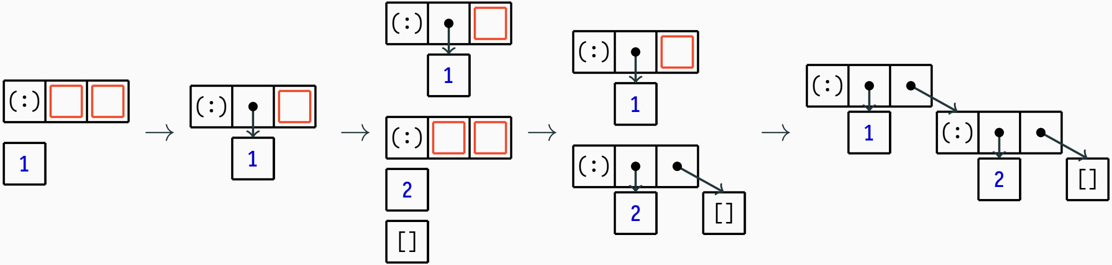
□ *denotes a "hole" in the structure (an uninitialized memory cell)*

Building order in functional languages – goal

What about lifting this limitation?

- Allowing pieces of data structures to be connected like Lego bricks in **any order**

  pioneered by “A Functional Representation of Data Structures with a Hole”, Minamide (1998)



🟥 *denotes a “hole” in the structure (an uninitialized memory cell)*

Challenges

Unitialized memory/**holes**:

- implies future **mutability**
- no read safety (risk of segfault)

Need proper functional abstraction to manipulate incomplete structures.

Destination passing	**(1st article)**
Safety concerns for functional DPS	(1st article)
Linear types	(1st article)
Safe yet?	(1st article)
Avoiding scope escape in Haskell	(1st article)
A more general solution: age control	(2nd article)
Formalization and proofs in Rocq	(2nd article)

Here comes destination passing style (DPS)

Coming from old C days:

Traditional style

```c
MyStruct * fooTrad() {
  MyStruct *res = malloc(sizeof(MyStruct));
  res->f1 = 1; res->f2 = 2;
  return res;
}
```

DPS style

```c
void fooDps(MyStruct *dest) {
  dest->f1 = 1; dest->f2 = 2;
}
```

Here comes destination passing style (DPS)

Coming from old C days:

Traditional style

```
MyStruct * fooTrad() {
  MyStruct *res = malloc(sizeof(MyStruct));
  res->f1 = 1; res->f2 = 2;
  return res;
}
```

DPS style

```
void fooDps(MyStruct *dest) {
  dest->f1 = 1; dest->f2 = 2;
}
```

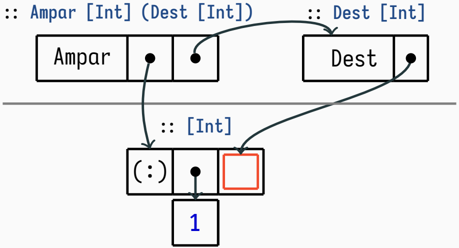
Caller is responsible for allocation in destination-passing-style function `fooDps`. More flexible:

- can allocate several slots at once
- can allocate on the stack (no `malloc`)

Functional DPS API – Types (1/2)

`Dest`ination: first-class typed wrapper for a raw pointer to a **hole**

- *breaks from Minamide's approach*
- only way to refer to and act on a hole (of an incomplete structure)



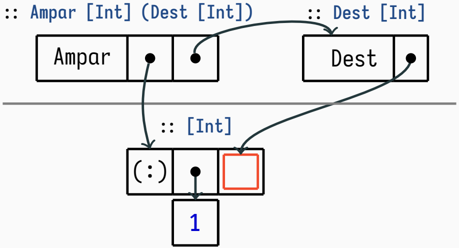
Functional DPS API – Types (1/2)

Destination: first-class typed wrapper for a raw pointer to a **hole**

- *breaks from Minamide's approach*
- only way to refer to and act on a hole (of an incomplete structure)

Ampar: first-class opaque wrapper for an **incomplete structure** and its destinations

- prevents reading incomplete structure and its holes
- only way to refer to and act on an incomplete structure



Functional DPS API – Types (2/2)

Every operation is done through `Ampar` and `Dest` types.

```
data Ampar s t = Ampar s t
```
(opaque)

- `s` is the type of the incomplete structure
- `t` is arbitrary structure carrying all the destinations to holes of `s`

E.g. `Ampar [Int] (Dest Int, Dest Int)`: list of ints with two missing values

Functional DPS API – Types (2/2)

Every operation is done through `Ampar` and `Dest` types.

```
data Ampar s t = Ampar s t
```
(opaque)

- `s` is the type of the incomplete structure
- `t` is arbitrary structure carrying all the destinations to holes of `s`

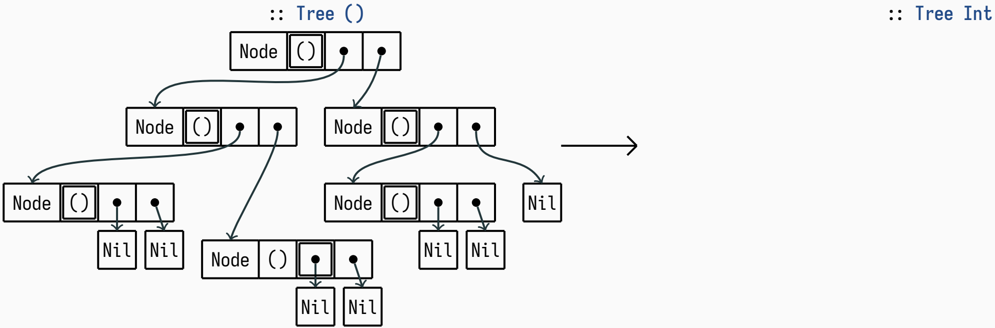
E.g. `Ampar [Int] (Dest Int, Dest Int)`: list of ints with two missing values

```
data Dest t = Dest Addr#
```
(opaque)

- `t` is the type of the hole that the destination references

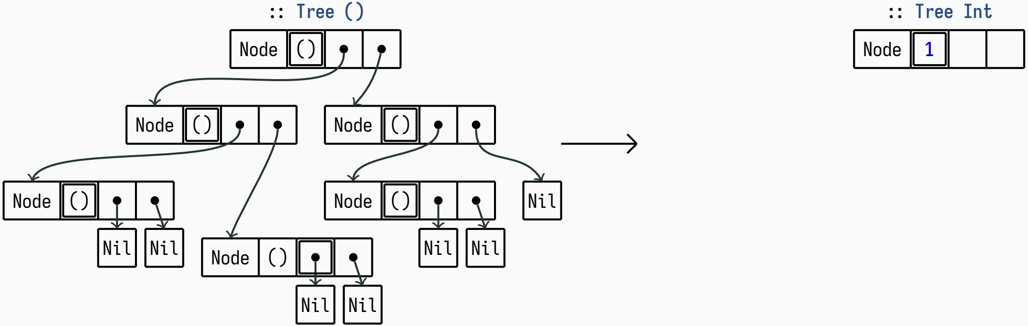
Example: Breadth-first tree relabeling in DPS (1/2)

```
data Tree t = Nil | Node t (Tree t) (Tree t)
```



Example: Breadth-first tree relabeling in DPS (1/2)

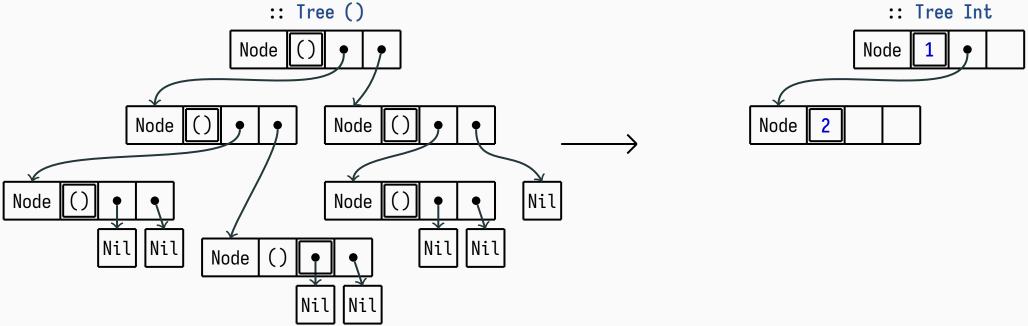
```
data Tree t = Nil | Node t (Tree t) (Tree t)
```



I represent boxed integer values in a compact way on the schema for clarity of the presentation.

Example: Breadth-first tree relabeling in DPS (1/2)

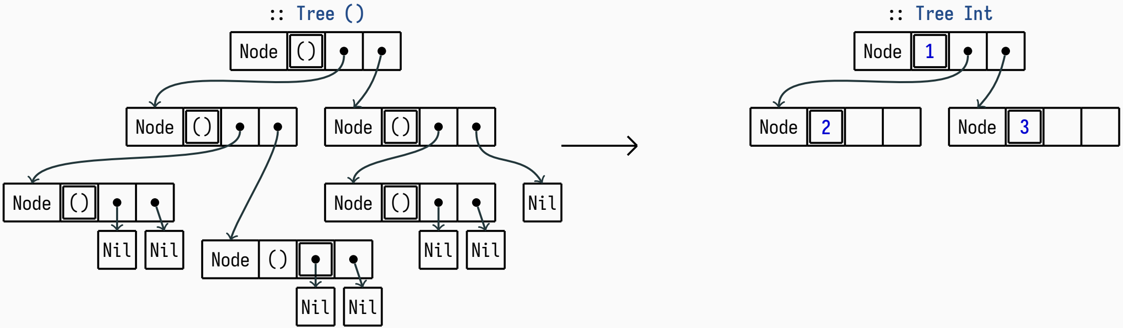
```
data Tree t = Nil | Node t (Tree t) (Tree t)
```



I represent boxed integer values in a compact way on the schema for clarity of the presentation.

Example: Breadth-first tree relabeling in DPS (1/2)

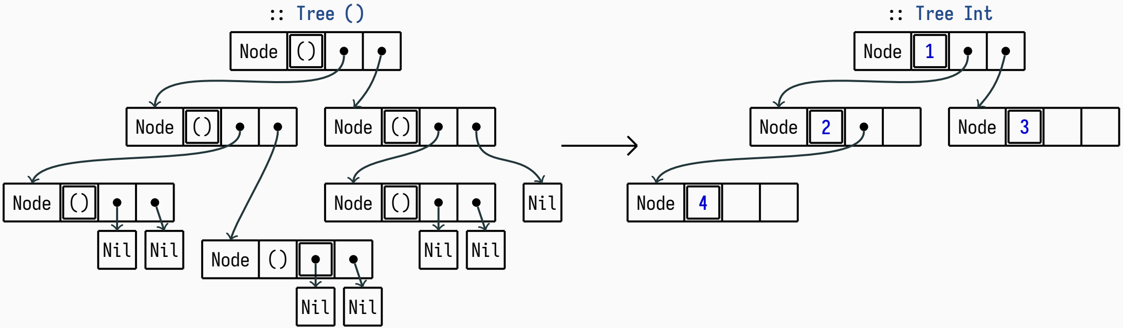
```
data Tree t = Nil | Node t (Tree t) (Tree t)
```



I represent boxed integer values in a compact way on the schema for clarity of the presentation.

Example: Breadth-first tree relabeling in DPS (1/2)

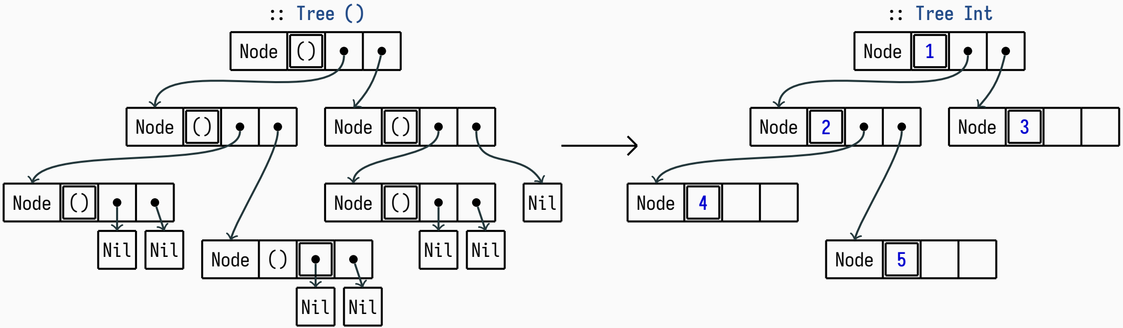
```
data Tree t = Nil | Node t (Tree t) (Tree t)
```



I represent boxed integer values in a compact way on the schema for clarity of the presentation.

Example: Breadth-first tree relabeling in DPS (1/2)

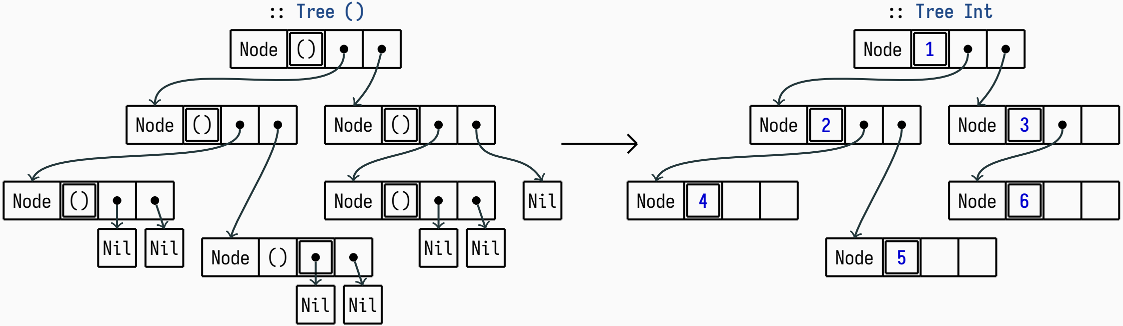
```
data Tree t = Nil | Node t (Tree t) (Tree t)
```



I represent boxed integer values in a compact way on the schema for clarity of the presentation.

Example: Breadth-first tree relabeling in DPS (1/2)

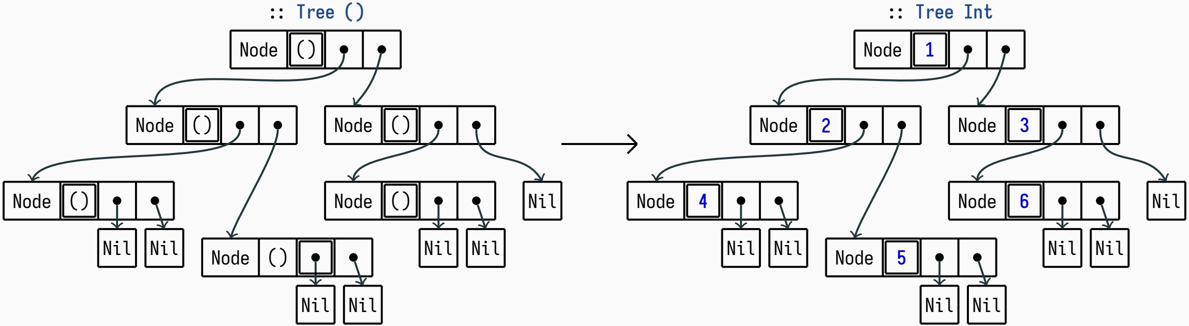
```
data Tree t = Nil | Node t (Tree t) (Tree t)
```



I represent boxed integer values in a compact way on the schema for clarity of the presentation.

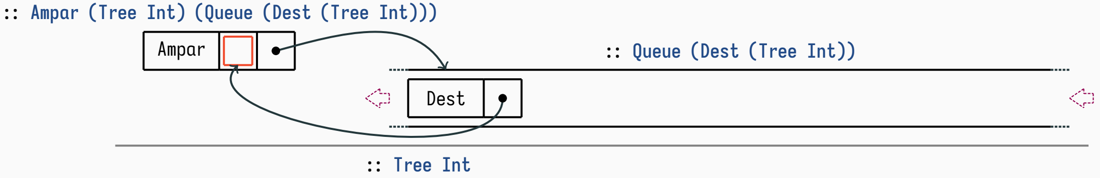
Example: Breadth-first tree relabeling in DPS (1/2)

```
data Tree t = Nil | Node t (Tree t) (Tree t)
```

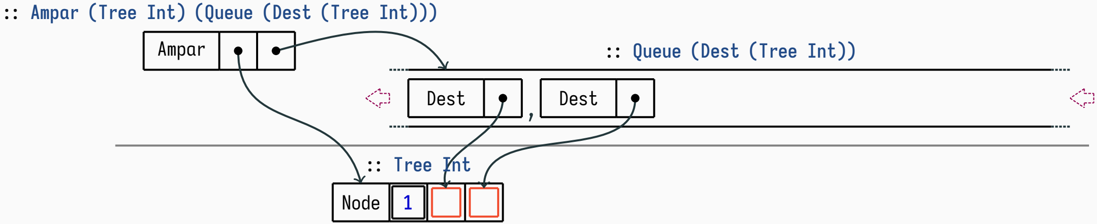


I represent boxed integer values in a compact way on the schema for clarity of the presentation.

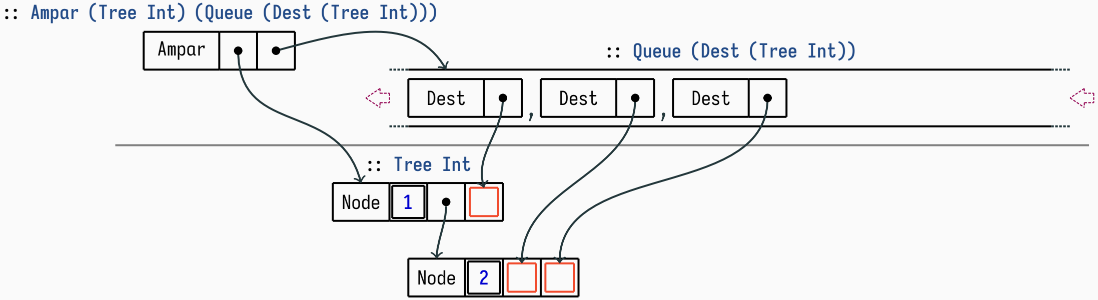
Example: Breadth-first tree building in DPS (2/2)


:: Ampar (Tree Int) (Queue (Dest (Tree Int)))
Ampar
:: Queue (Dest (Tree Int))
Dest
:: Tree Int

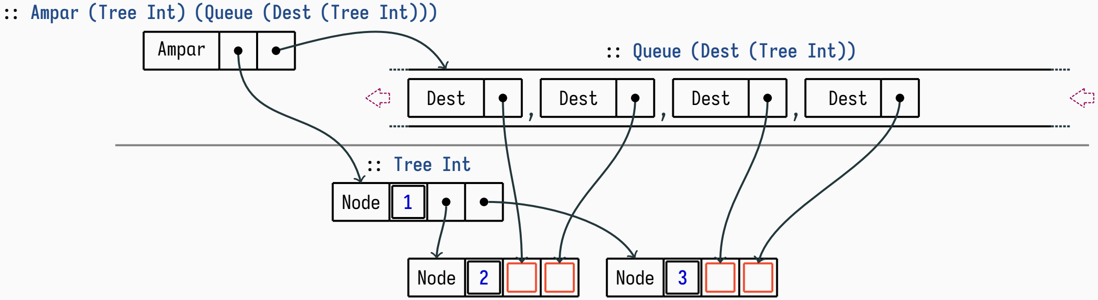
Example: Breadth-first tree building in DPS (2/2)


:: Ampar (Tree Int) (Queue (Dest (Tree Int)))
Ampar
:: Queue (Dest (Tree Int))
Dest
,
Dest
:: Tree Int
Node
1

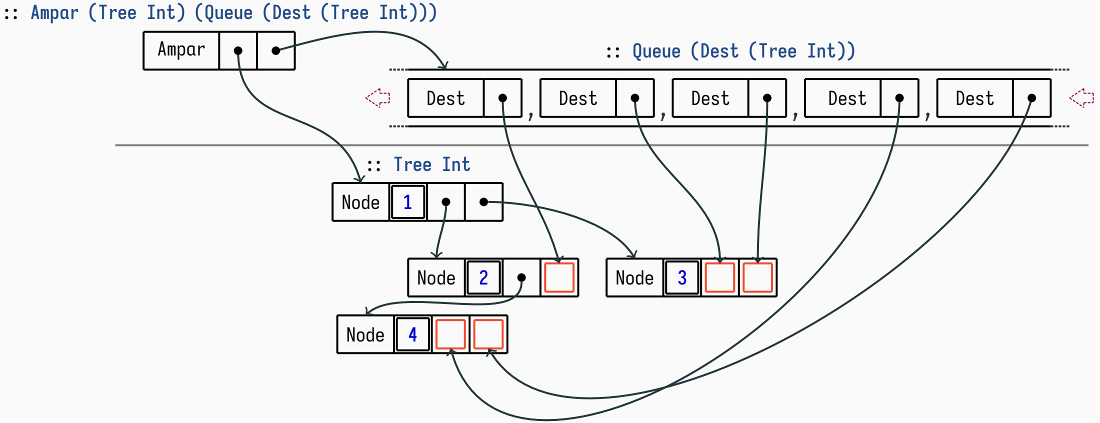
Example: Breadth-first tree building in DPS (2/2)


:: Ampar (Tree Int) (Queue (Dest (Tree Int)))
Ampar
:: Queue (Dest (Tree Int))
Dest
,
Dest
,
Dest
:: Tree Int
Node
1
Node
2

Example: Breadth-first tree building in DPS (2/2)


:: Ampar (Tree Int) (Queue (Dest (Tree Int)))
Ampar
:: Queue (Dest (Tree Int))
Dest
,
Dest
,
Dest
,
Dest
:: Tree Int
Node
1
Node
2
Node
3

Example: Breadth-first tree building in DPS (2/2)

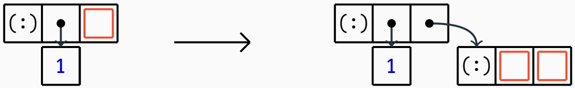

:: Ampar (Tree Int) (Queue (Dest (Tree Int)))
Ampar
:: Queue (Dest (Tree Int))
Dest
,
Dest
,
Dest
,
Dest
,
Dest
:: Tree Int
Node
1
Node
2
Node
3
Node
4

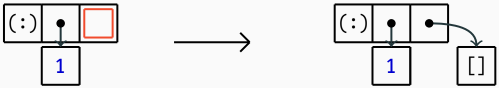
Functional DPS API – First attempt

```haskell
data Ampar s t
data Dest t
type family DestsOf 'Ctor t  -- returns dests corresponding to fields of Ctor

-- Functions on destinations (infix)
&fill @'Ctor :: Dest t             → DestsOf 'Ctor t
&fillComp    :: Dest s → Ampar s t → t
&fillLeaf    :: Dest t → t         → ()

-- Functions on Ampar
newAmpar   :: Ampar s (Dest s)
fromAmpar' :: Ampar s () → s
`updWith`  :: Ampar s t  → (t → u) → Ampar s u  -- (infix)
```

Functional DPS API – Functions (1/6)

- Allocate a hollow data constructor and plug it into a hole?

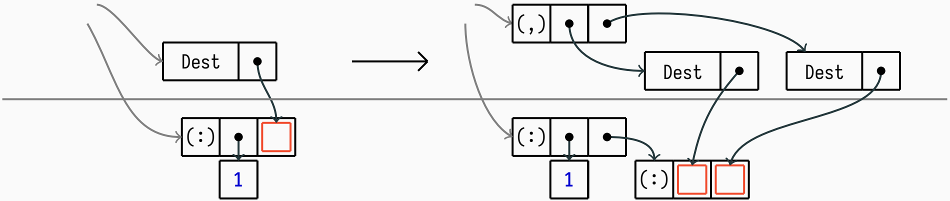




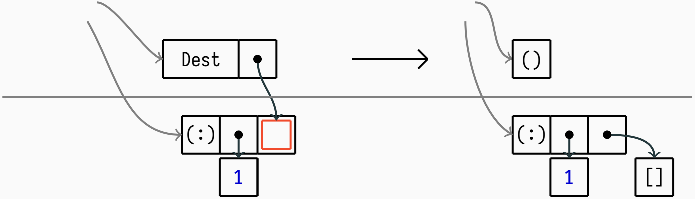
Functional DPS API – Functions (1/6)

- Allocate a hollow data constructor and plug it into a hole?

```
&fill @'(:) :: Dest [t]       →          (Dest t, Dest [t])
```

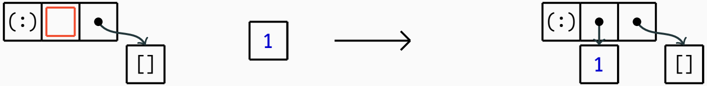


```
&fill @'[]  :: Dest [t]      →        ()
```



Functional DPS API – Functions (2/6)

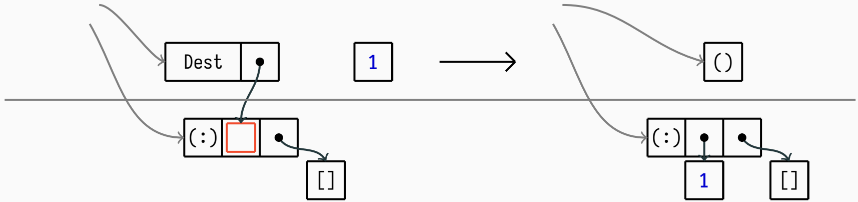
- Fill a hole with an already complete value?



Functional DPS API – Functions (2/6)

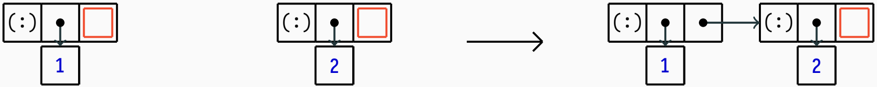
- Fill a hole with an already complete value?

```
&fillLeaf ::   Dest t   →   t      →           ()
```



Functional DPS API – Functions (3/6)

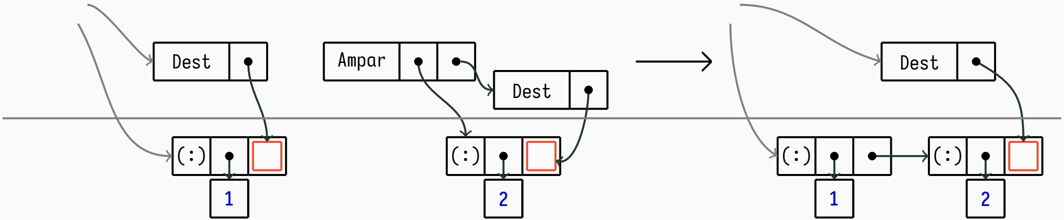
- Plug an incomplete structure into the hole of another one?



Functional DPS API – Functions (3/6)

- Plug an incomplete structure into the hole of another one?

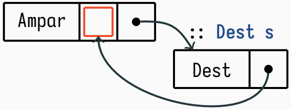
```
&fillComp :: Dest s  →  Ampar s t  →  t
```



Functional DPS API – Functions (4/6)

- Spawn new incomplete structure?

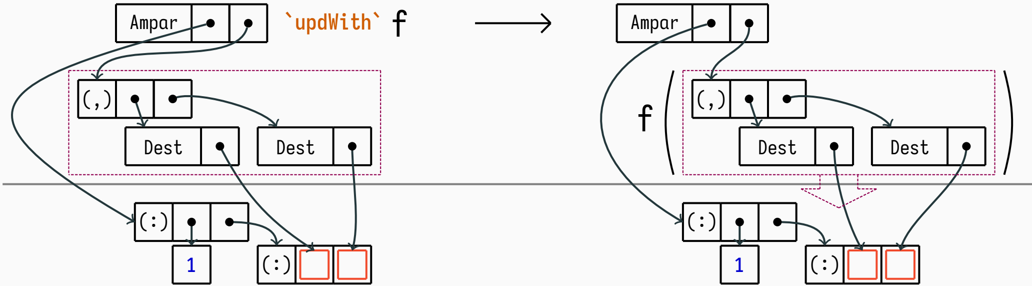
```
newAmpar ::    Ampar s (Dest s)
```



Functional DPS API – Functions (5/6)

- Access destinations of an Ampar?

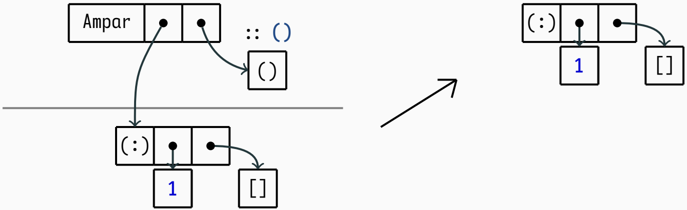
```
`updWith` :: Ampar s t  →  (t → u)  →  Ampar s u
```



Functional DPS API – Functions (6/6)

- Extract a completed structure?

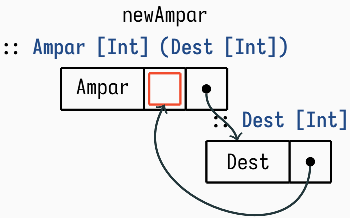
```
fromAmpar' ::    Ampar s ()          →          s
```



Destination passing	(1st article)
Safety concerns for functional DPS	**(1st article)**
Linear types	(1st article)
Safe yet?	(1st article)
Avoiding scope escape in Haskell	(1st article)
A more general solution: age control	(2nd article)
Formalization and proofs in Rocq	(2nd article)

Unrestricted use of destinations

```
fromAmpar' :: Ampar s () → s  -- I recall the signature

let apparentlyComplete :: Ampar [Int] () = newAmpar `updWith` \d → ()
 in fromAmpar' apparentlyComplete
```



newAmpar

:: Ampar [Int] (Dest [Int])

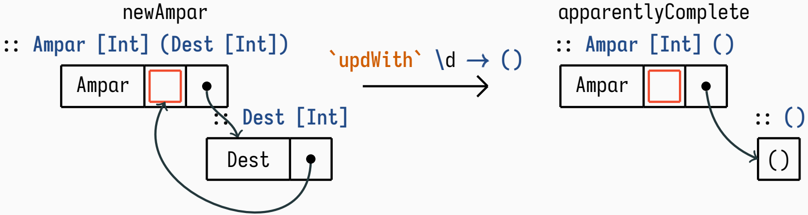
Ampar

:: Dest [Int]

Dest

Unrestricted use of destinations

```haskell
fromAmpar' :: Ampar s () → s  -- I recall the signature

let apparentlyComplete :: Ampar [Int] () = newAmpar `updWith` \d → ()
 in fromAmpar' apparentlyComplete
```



newAmpar

:: Ampar [Int] (Dest [Int])

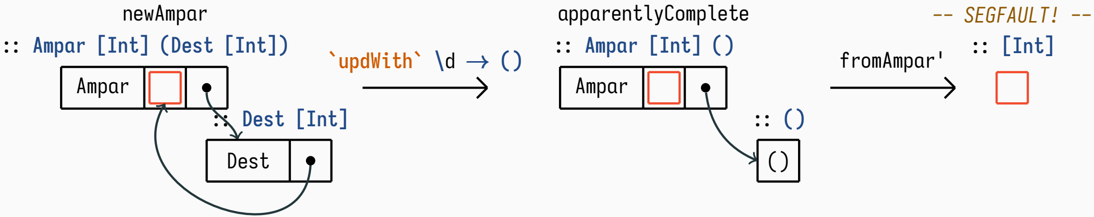
`updWith` \d → ()

apparentlyComplete

:: Ampar [Int] ()

Unrestricted use of destinations

```
fromAmpar' :: Ampar s () → s  -- I recall the signature

let apparentlyComplete :: Ampar [Int] () = newAmpar `updWith` \d → ()
 in fromAmpar' apparentlyComplete
```



Destination passing	(1st article)
Safety concerns for functional DPS	(1st article)
Linear types	**(1st article)**
Safe yet?	(1st article)
Avoiding scope escape in Haskell	(1st article)
A more general solution: age control	(2nd article)
Formalization and proofs in Rocq	(2nd article)

Linearity to the rescue

Linear types and linearity let us control resource consumption of functions.

Idea : **destinations are linear resources** i.e. must be consumed **exactly once**:

Linearity to the rescue

Linear types and linearity let us control resource consumption of functions.

Idea : **destinations are linear resources** i.e. must be consumed **exactly once**:

- consumed less than once → hole remained unfilled

Linearity to the rescue

Linear types and linearity let us control resource consumption of functions.

Idea : **destinations are linear resources** i.e. must be consumed **exactly once**:

- consumed less than once → hole remained unfilled
- consumed more than once → order of evaluation becomes semantically relevant

Linearity to the rescue

Linear types and linearity let us control resource consumption of functions.

Idea : **destinations are linear resources** i.e. must be consumed **exactly once**:

- consumed less than once → hole remained unfilled
- consumed more than once → order of evaluation becomes semantically relevant

Unconsumed destination = witnesses a remaining hole

Linear Haskell overview (1/2)

Haskell supports linear types since GHC 9.0.1

follows from "Linear Haskell: practical linearity in a higher-order polymorphic language",
Bernardy et al. (2018)

`s` $\multimap$ `t` = Linear function from `s` to `t`

means that if the result of type `t` is consumed exactly once, then the argument of type `s` is consumed exactly once too.

Linear Haskell overview (2/2)

A value is **consumed once** when:

- **pattern-matched on**, and all its fields are consumed once
- **used as an argument** of a linear function whose results is consumed once
- *(function)* **applied** to give a result which is consumed once

Linear Haskell overview (2/2)

A value is **consumed once** when:

- **pattern-matched on**, and all its fields are consumed once
- **used as an argument** of a linear function whose results is consumed once
- *(function)* **applied** to give a result which is consumed once

Examples (assuming the value on the right is consumed):

`x` linear in `(x, y)` ? → **YES**

Linear Haskell overview (2/2)

A value is **consumed once** when:

- **pattern-matched on**, and all its fields are consumed once
- **used as an argument** of a linear function whose results is consumed once
- *(function)* **applied** to give a result which is consumed once

Examples (assuming the value on the right is consumed):

`x` linear in `(x, y)` ? → **YES**

`x` linear in `(x, x)` ? → **NO**

Linear Haskell overview (2/2)

A value is **consumed once** when:

- **pattern-matched on**, and all its fields are consumed once
- **used as an argument** of a linear function whose results is consumed once
- *(function)* **applied** to give a result which is consumed once

Examples (assuming the value on the right is consumed):

`x`	linear in	`(x, y)` ?	→ **YES**
`x`	linear in	`(x, x)` ?	→ **NO**
`x`	linear in	`case y of {Nothing -> x ; Just v -> v}` ?	→ **NO**

Linear Haskell overview (2/2)

A value is **consumed once** when:

- **pattern-matched on**, and all its fields are consumed once
- **used as an argument** of a linear function whose results is consumed once
- *(function)* **applied** to give a result which is consumed once

Examples (assuming the value on the right is consumed):

`x`	linear in	`(x, y)` ?	→ **YES**
`x`	linear in	`(x, x)` ?	→ **NO**
`x`	linear in	`case y of {Nothing -> x ; Just v -> v}` ?	→ **NO**
`x`	linear in	`case y of {Nothing -> (x, 0) ; Just v -> (x, v)}` ?	→ **YES**

Linear Haskell overview (2/2)

A value is **consumed once** when:

- **pattern-matched on**, and all its fields are consumed once
- **used as an argument** of a linear function whose results is consumed once
- *(function)* **applied** to give a result which is consumed once

Examples (assuming the value on the right is consumed):

`x`	linear in	`(x, y)` ?	→ **YES**
`x`	linear in	`(x, x)` ?	→ **NO**
`x`	linear in	`case y of {Nothing -> x ; Just v -> v}` ?	→ **NO**
`x`	linear in	`case y of {Nothing -> (x, 0) ; Just v -> (x, v)}` ?	→ **YES**
`x`	linear in	`f x` where `f :: t → u` ?	→ **NO**

Linear Haskell overview (2/2)

A value is **consumed once** when:

- **pattern-matched on**, and all its fields are consumed once
- **used as an argument** of a linear function whose results is consumed once
- *(function)* **applied** to give a result which is consumed once

Examples (assuming the value on the right is consumed):

`x`	linear in	`(x, y)`		?	→ YES
`x`	linear in	`(x, x)`		?	→ NO
`x`	linear in	`case y of {Nothing -> x ; Just v -> v}`		?	→ NO
`x`	linear in	`case y of {Nothing -> (x, 0) ; Just v -> (x, v)}`		?	→ YES
`x`	linear in	`f x`	where `f :: t → u`	?	→ NO
`x`	linear in	`fLin x`	where `fLin :: t ⊸ u`	?	→ YES

Linear Haskell overview (2/2)

A value is **consumed once** when:

- **pattern-matched on**, and all its fields are consumed once
- **used as an argument** of a linear function whose results is consumed once
- *(function)* **applied** to give a result which is consumed once

Examples (assuming the value on the right is consumed):

`x`	linear in	`(x, y)` ?	→ **YES**
`x`	linear in	`(x, x)` ?	→ **NO**
`x`	linear in	`case y of {Nothing -> x ; Just v -> v}` ?	→ **NO**
`x`	linear in	`case y of {Nothing -> (x, 0) ; Just v -> (x, v)}` ?	→ **YES**
`x`	linear in	`f x` where `f :: t → u` ?	→ **NO**
`x`	linear in	`fLin x` where `fLin :: t ⊸ u` ?	→ **YES**
`x`	linear in	`Ur x` ?	→ **NO**

Linear scopes and resources

Linearity chains consumption requirements, but it needs to be bootstrapped.

Trick: scoping function

```
withResource :: (Resource ⊸ Ur t) ⊸ Ur t
```

which is the only **producer** of `Resource` values

User must pass a **linear function** consuming the resource in this scope

Linear scopes and resources

Linearity chains consumption requirements, but it needs to be bootstrapped.

Trick: scoping function

```
withResource :: (Resource ⊸ Ur t) ⊸ Ur t
```

which is the only **producer** of `Resource` values

User must pass a **linear function** consuming the resource in this scope

- `withResource (\res -> linearConsume res ⨟ Ur ())` is **OK**

Linear scopes and resources

Linearity chains consumption requirements, but it needs to be bootstrapped.

Trick: scoping function

```
withResource :: (Resource ⊸ Ur t) ⊸ Ur t
```

which is the only **producer** of `Resource` values

User must pass a **linear function** consuming the resource in this scope

- `withResource (\res -> linearConsume res ⨟ Ur ())` is **OK**
- `withResource (\res -> res)` or
  `withResource (\res -> Ur res)` are **REJECTED** (cannot leak)

Updating the API with linearity

Trick to easier manage linear resources and their scopes: linearity-infectious **Token**:

```haskell
data Token
dup  :: Token ⊸ (Token, Token)
drop :: Token ⊸ ()
withToken :: (Token ⊸ Ur t) ⊸ Ur t
```

Updating the API with linearity

Trick to easier manage linear resources and their scopes: linearity-infectious `Token`:

```haskell
data Token
dup  :: Token ⊸ (Token, Token)
drop :: Token ⊸ ()
withToken :: (Token ⊸ Ur t) ⊸ Ur t
```

Making sure `Ampar`s are used linearly, e.g.:

```
Old: newAmpar ::          Ampar s (Dest s)
New: newAmpar :: Token ⊸ Ampar s (Dest s)
```

Updating the API with linearity

Trick to easier manage linear resources and their scopes: linearity-infectious **Token**:

```
data Token
dup  :: Token ⊸ (Token, Token)
drop :: Token ⊸ ()
withToken :: (Token ⊸ Ur t) ⊸ Ur t
```

Making sure **Ampar**s are used linearly, e.g.:

```
Old: newAmpar ::         Ampar s (Dest s)
New: newAmpar :: Token ⊸ Ampar s (Dest s)
```

```
Old: updWith :: Ampar s t → (t → u) → Ampar s u
New: updWith :: Ampar s t ⊸ (t ⊸ u) ⊸ Ampar s u
```

Updating the API with linearity

Trick to easier manage linear resources and their scopes: linearity-infectious **Token**:

```
data Token
dup  :: Token ⊸ (Token, Token)
drop :: Token ⊸ ()
withToken :: (Token ⊸ Ur t) ⊸ Ur t
```

Making sure **Ampar**s are used linearly, e.g.:

```
Old: newAmpar ::         Ampar s (Dest s)
New: newAmpar :: Token ⊸ Ampar s (Dest s)
```

```
Old: updWith :: Ampar s t → (t → u) → Ampar s u
New: updWith :: Ampar s t ⊸ (t ⊸ u) ⊸ Ampar s u
```

Dests can only ever be accessed through `updWith`, which takes a linear function `t ⊸ u` now

Functional DPS API – Second attempt, with linearity

```
data Ampar s t
data Dest t
type family DestsOf 'Ctor t  -- returns dests corresponding to fields of Ctor

&fill @'Ctor :: Dest t             ⊸ DestsOf 'Ctor t
&fillComp    :: Dest s ⊸ Ampar s t ⊸ t
&fillLeaf    :: Dest t ⊸ t         ⊸ ()

newAmpar     :: Token          ⊸ Ampar s (Dest s)
toAmpar      :: Token   ⊸ s    ⊸ Ampar s ()
tokenBesides :: Ampar s t      ⊸ (Ampar s t, Token)
fromAmpar    :: Ampar s (Ur t) ⊸ (s, Ur t)
fromAmpar'   :: Ampar s ()     ⊸ s
`updWith`    :: Ampar s t      ⊸ (t ⊸ u) ⊸ Ampar s u
```

Destination passing	(1st article)
Safety concerns for functional DPS	(1st article)
Linear types	(1st article)
Safe yet?	**(1st article)**
Avoiding scope escape in Haskell	(1st article)
A more general solution: age control	(2nd article)
Formalization and proofs in Rocq	(2nd article)

Scope escape – Example

```haskell
let outer :: Ampar (Dest ()) ()
    outer = (newAmpar tok1) `updWith` \(dOuter :: Dest (Dest ())) →
              let inner :: Ampar () ()
                  inner = (newAmpar tok2) `updWith` \(dInner :: Dest ()) →
                            dOuter &fillLeaf dInner
               in fromAmpar' inner
 in fromAmpar' outer
```

Minimal example of type-checked code that produce scope-escape/segfault

Scope escape – Example

```
let outer :: Ampar (Dest ()) ()
    outer = (newAmpar tok1) `updWith` \(dOuter :: Dest (Dest ())) →
              let inner :: Ampar () ()
                  inner = (newAmpar tok2) `updWith` \(dInner :: Dest ()) →
                            dOuter &fillLeaf dInner
               in fromAmpar' inner
 in fromAmpar' outer
```

Minimal example of type-checked code that produce scope-escape/segfault

Scope escape – Bigger picture

Not *just* an artificial edge-case

- Linear API are often based on scope-delimited resource consumption
- Any form of (linear) storage for linear resources breaks that

```
storeAway :: t ⊸ ()  -- resource goes away magically
```

Destination passing	(1st article)
Safety concerns for functional DPS	(1st article)
Linear types	(1st article)
Safe yet?	(1st article)
Avoiding scope escape in Haskell	**(1st article)**
A more general solution: age control	(2nd article)
Formalization and proofs in Rocq	(2nd article)

With just Haskell type system (1/2)

Solution 1: destinations can only store non-linear data

Old: `fillLeaf :: t ⊸  Dest t ⊸ ()`

New: `fillLeaf :: t → UDest t ⊸ ()`

Old: `fromAmpar'  ::  Ampar s () ⊸ s`

New: `fromUAmpar' :: UAmpar s () ⊸ Ur s`

In other words, `UAmpar s t` builds an unrestricted `s`

Used in my library `linear-dest` *from article “DPS: a Haskell implementation”, Bagrel, JFLA 2024*

With just Haskell type system (2/2)

Solution 2: destinations for linear data needs new primitives instead of `fillLeaf` / `fillComp`

Back to Minamide-like system where one operate on `Ampar`s directly instead of `Dest`s (to prevent scope escape)

- With some complications, can work with multiple holes
- Still allows for efficient queue of `Dest`s in BF-traversal

Developed in Chapter 4 of the PhD manuscript (unpublished)

Destination passing	(1st article)
Safety concerns for functional DPS	(1st article)
Linear types	(1st article)
Safe yet?	(1st article)
Avoiding scope escape in Haskell	(1st article)
A more general solution: age control	**(2nd article)**
Formalization and proofs in Rocq	(2nd article)

Age system

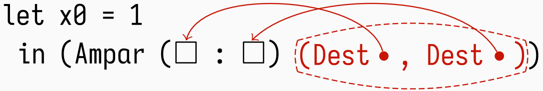
Principle: track the age of resources.

- Age $\uparrow^0$ says the resource (variable) comes from the innermost **`updWith`** scope.
- When entering a new **`updWith`** scope, all existing resources ages are multiplied by $\uparrow$ (scope number increased by 1)

Developed in “Destination Calculus: A Linear λ-Calculus for Purely Functional Memory Writes”,
Bagrel & Spiwack, OOPSLA1 2025

Age system – Example

Colors indicate the scope in which each object lives.



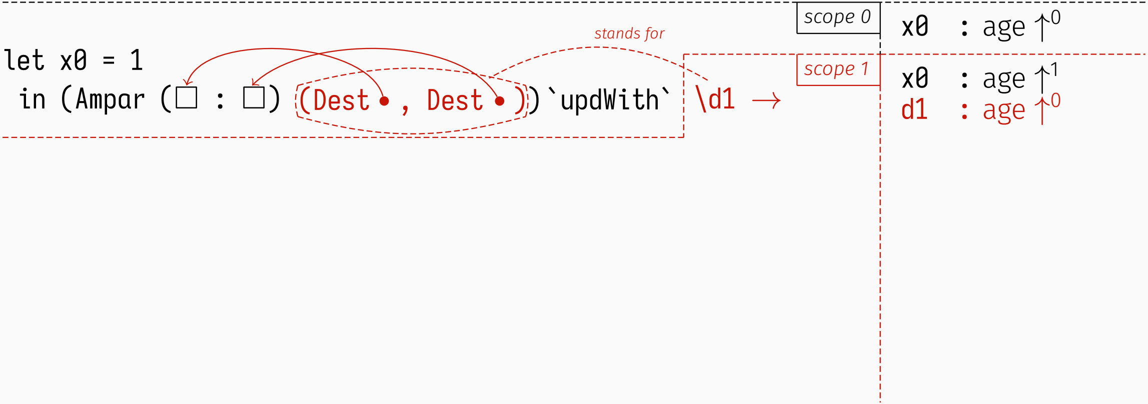
```
let x0 = 1
 in (Ampar (□ : □) (Dest • , Dest • ))
```

scope 0

x0 : age $\uparrow^{0}$

Age system – Example

Colors indicate the scope in which each object lives.



```
let x0 = 1
 in (Ampar (□ : □) (Dest • , Dest • )) `updWith` \d1 →
```

stands for

scope 0

x0 : age $\uparrow^{0}$

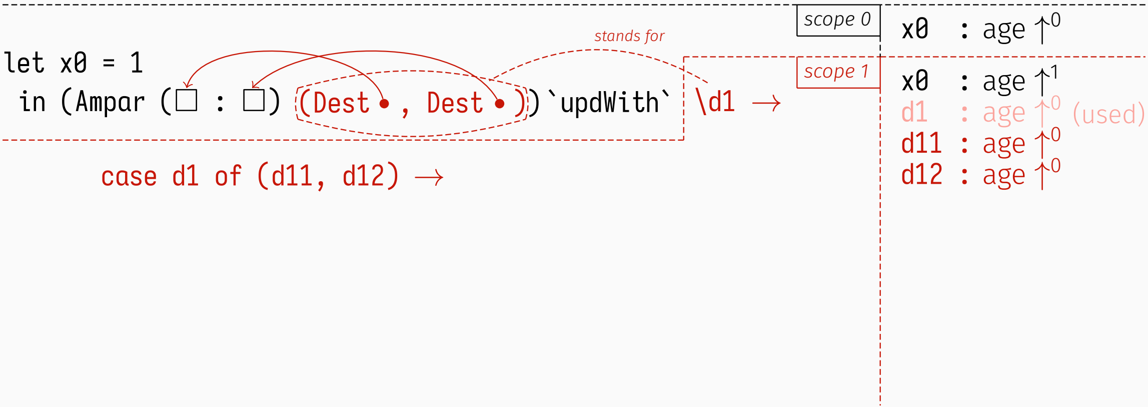
scope 1

x0 : age $\uparrow^{1}$

d1 : age $\uparrow^{0}$

Age system – Example

Colors indicate the scope in which each object lives.

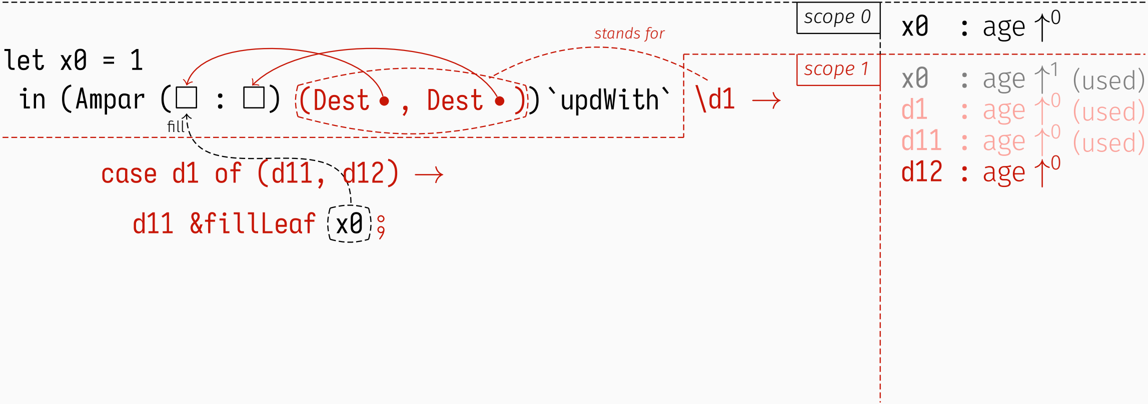


```
let x0 = 1
 in (Ampar (□ : □) (Dest • , Dest • )) `updWith` \d1 →

      case d1 of (d11, d12) →
```

```
x0  : age ↑0
x0  : age ↑1
d1  : age ↑0 (used)
d11 : age ↑0
d12 : age ↑0
```

Age system – Example

Colors indicate the scope in which each object lives.

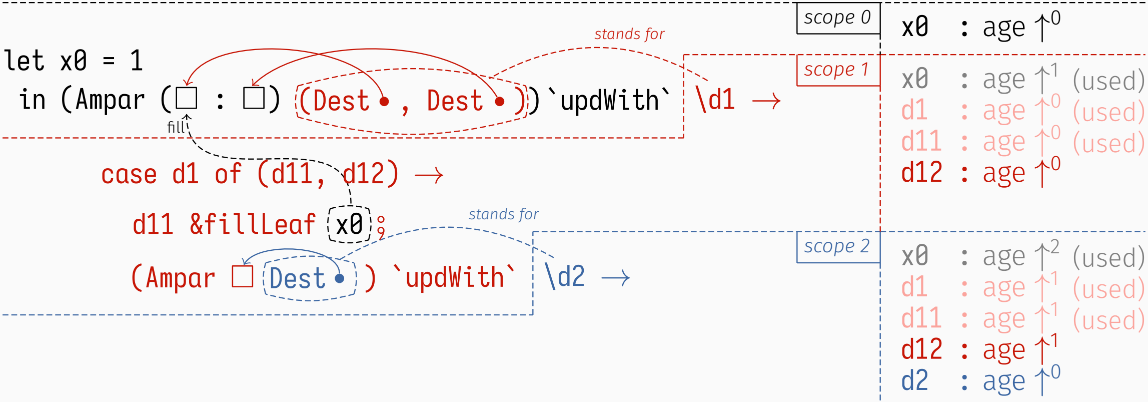


```
let x0 = 1
 in (Ampar (□ : □) (Dest • , Dest • )) `updWith` \d1 →
      case d1 of (d11, d12) →
        d11 &fillLeaf x0 ⨟
```

```
x0  : age ↑0
x0  : age ↑1 (used)
d1  : age ↑0 (used)
d11 : age ↑0 (used)
d12 : age ↑0
```

Age system – Example

Colors indicate the scope in which each object lives.



```
let x0 = 1
 in (Ampar (□ : □) (Dest • , Dest • )) `updWith` \d1 →
      case d1 of (d11, d12) →
        d11 &fillLeaf x0 ⨾
        (Ampar □ Dest • ) `updWith` \d2 →
```

scope 0

```
x0  : age ↑0
```

scope 1

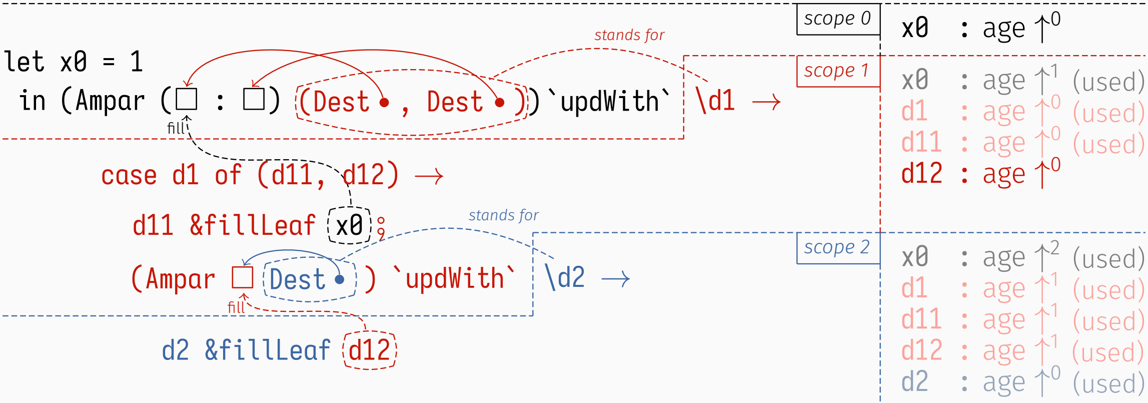
```
x0  : age ↑1 (used)
d1  : age ↑0 (used)
d11 : age ↑0 (used)
d12 : age ↑0
```

scope 2

```
x0  : age ↑2 (used)
d1  : age ↑1 (used)
d11 : age ↑1 (used)
d12 : age ↑1
d2  : age ↑0
```

Age system – Example

Colors indicate the scope in which each object lives.



```
let x0 = 1
 in (Ampar (□ : □) (Dest • , Dest • )) `updWith` \d1 →
      case d1 of (d11, d12) →
        d11 &fillLeaf x0 ;
        (Ampar □ Dest • ) `updWith` \d2 →
          d2 &fillLeaf d12
```

scope 0

x0 : age $\uparrow^{0}$

scope 1

x0 : age $\uparrow^{1}$ (used)
d1 : age $\uparrow^{0}$ (used)
d11 : age $\uparrow^{0}$ (used)
d12 : age $\uparrow^{0}$

scope 2

x0 : age $\uparrow^{2}$ (used)
d1 : age $\uparrow^{1}$ (used)
d11 : age $\uparrow^{1}$ (used)
d12 : age $\uparrow^{1}$ (used)
d2 : age $\uparrow^{0}$ (used)

Age control prevents scope escape

New rule: The LHS of **&fillLeaf** (destination being filled) must be exactly one scope **younger** than the RHS (content being stored away)

```
let outer :: Ampar (Dest ()) ()
    outer = (newAmpar tok1) `updWith` \(dOuter :: ↑^0 Dest (Dest ())) →
              let inner :: Ampar () ()
                  inner = (newAmpar tok2) `updWith` \(dInner :: ↑^0 Dest ()) →
                            (dOuter :: ↑^1) &fillLeaf (dInner :: ↑^0) -- REJECTED
               in fromAmpar' inner
 in fromAmpar' outer
```

Destination passing	(1st article)
Safety concerns for functional DPS	(1st article)
Linear types	(1st article)
Safe yet?	(1st article)
Avoiding scope escape in Haskell	(1st article)
A more general solution: age control	(2nd article)
Formalization and proofs in Rocq	**(2nd article)**

Formalization

Essence of *“Destination Calculus: A Linear λ-Calculus for Purely Functional Memory Writes”, Bagrel & Spiwack, OOPSLA1 2025*:

- Formal language λ_d
- Built from the ground up with destination-passing in mind
- Equipped with linear types and age control
- Mechanical proof of type safety (progress + preservation) in Rocq

→ Prove definitively that it is possible to safely program with destinations in a purely functional language!

Modal type system to track linearity and ages

Many similarities with type system and rules of *"Linear Haskell [...]", Bernardy et al., 2018*:

- Every variable is annotated with a **mode**
- Modes indicates how variables can be used
- Following insight from *"Bounded Linear Types in a Resource Semiring", Ghica & al., 2014* and *"Coeffects: a calculus of context-dependent computation", Petricek & al., 2014*, modes as a **semiring** and operations on modes are lifted to typing contexts

→ Easy to add age control without changing much to linear type system rules!

Combining linearity and ages in a same semiring

Modes are elements of a semiring $(M, +, \cdot, 1)$

- $+$ is used to combine modes when a same variable is used in multiple sub-expressions
- $\cdot$ is used to combine modes when a substitution/function composition happens

Combining linearity and ages in a same semiring

Modes are elements of a semiring $(M, +, \cdot, 1)$

- $+$ is used to combine modes when a same variable is used in multiple sub-expressions
- $\cdot$ is used to combine modes when a substitution/function composition happens

Linearity is represented as semiring
$(\{l, \omega\}, +, \cdot, l)$

Combining linearity and ages in a same semiring

Modes are elements of a semiring $(M, +, \cdot, 1)$

- $+$ is used to combine modes when a same variable is used in multiple sub-expressions
- $\cdot$ is used to combine modes when a substitution/function composition happens

Linearity is represented as semiring
$(\{l, \omega\}, +, \cdot, l)$

Ages are represented as semiring
$(\{\uparrow^n \mid n \in \mathbb{N}, \infty\}, \overset{\infty}{=}, \cdot, \uparrow^0)$

Combining linearity and ages in a same semiring

Modes are elements of a semiring $(M, +, \cdot, 1)$

- $+$ is used to combine modes when a same variable is used in multiple sub-expressions
- $\cdot$ is used to combine modes when a substitution/function composition happens

Linearity is represented as semiring
$(\{l, \omega\}, +, \cdot, l)$

Ages are represented as semiring
$(\{\uparrow^n \mid n \in \mathbb{N}, \infty\}, \overset{\infty}{=}, \cdot, \uparrow^0)$

$\longrightarrow$ Product of semirings is a semiring!

Type system – balancing destinations and holes

I ensure holes and destinations are balanced using a “subtractive” typing technique:

- The presence of a hole **provides** a special binding in the typing context

  $\{\rightarrow h :_{m} \texttt{Dest}_{n}\, t\} \;\dashv\; \texttt{OpAmpar}\, \boxed{h}\, [\,] :: u_1 \rightarrowtail u_2$

- The use of a destination **requires** a special binding in the typing context (same as a variable use)

  $\{\rightarrow h :_{m} \texttt{Dest}_{n}\, t\} \;\vdash\; \rightarrow h\ \texttt{\&fillLeaf}\ () :: ()$

→ A closed term/program is necessarily well-balanced wrt. destinations and holes.

Semantics

Theoretical language λ_d is equipped with small-step semantics.

- Controlled mutations resulting from destination filling are implemented as substitutions on the evaluation context E

  Syntactic manipulation of the evaluation context as a stack
- No need for full-blown memory model

Named hole substitution lemma is the most technical part of the proofs

Type safety

Proved using fairly standard techniques (progress + preservation).

Most of the heavy lifting is done through lemmas and theorems about typing contexts due to the algebraic structure of the type system.

Theorem (Type preservation)

$$\mathtt{E} \quad \mathtt{e} \quad :: \mathtt{t} \qquad\qquad \mathtt{E} \quad \mathtt{e} \qquad\qquad \mathtt{E}' \quad \mathtt{e}' \qquad\qquad\qquad \mathtt{E}' \quad \mathtt{e}' \quad :: \mathtt{t}$$

Theorem (Progress)

$$\mathtt{E} \quad \mathtt{e} \quad :: \mathtt{t} \qquad\qquad \mathtt{v} \;\; \mathtt{E} \;\; \mathtt{e} \qquad\qquad \mathtt{v} \qquad\qquad\qquad \mathtt{E}' \;\; \mathtt{e}' \;\; \mathtt{E} \quad \mathtt{e} \qquad\qquad \mathtt{E}' \quad \mathtt{e}'$$

Conclusion (1/3) – Goal reached!

→ Safe functional destination-passing is definitely doable!

- Build immutable structures in a more flexible order
- Incomplete structures as first-class citizens

Conclusion (2/3) – Applications

Direct applications:

- Enable new algorithms with less copying (difference lists, BF tree traversal)
- Prototype already available in Haskell (`linear-dest` library)
- Zero-copy interface to compact regions in GHC (**proposal submitted!**)

Conclusion (3/3) – Trade-offs of the different approaches

Trade-offs to be found between complexity of the type system and expressivity of the destination-passing interface:

- Lower end: Haskell type system only, simple interface, destinations can’t store linear data – still very useful!
- Higher end: safe-proven theoretical language with no limit on destination use (but needs a bespoke type system)

Further work: Use the theoretical language as a framework to prove safety of less expressive destination-passing systems

Contributions

- "Destination-passing style programming: a Haskell implementation" (Article + Impl)

  JFLA 2024, `inria.hal.science/hal-04406360`

- "A zero-copy interface to compact regions powered by destinations" (Talk)

  HIW 2024, `www.youtube.com/watch?v=UIw0s-yEkfw`

- "Destination Calculus: A Linear λ-Calculus for Purely Functional Memory Writes" (Article)

  Co-authored with A. Spiwack, OOPSLA1 2025, `dl.acm.org/doi/10.1145/3720423`

- "Primitives for zero-copy compact regions" (GHC Proposal)

  Work in progress, `github.com/ghc-proposals/ghc-proposals/pull/683`

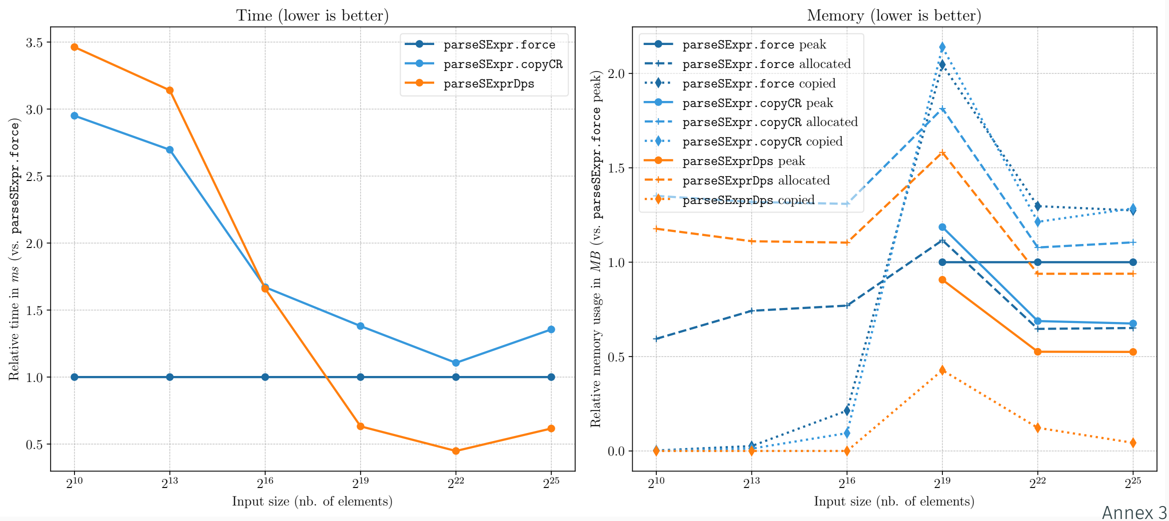
Thank you for your attention! I'll be happy to answer your questions.

Functional DPS API – scope-escape free

```
data UAmpar s t
data UDest t
type family UDestsOf 'Ctor t  -- returns dests corresponding to fields of Ctor

&fill @'Ctor :: UDest t              ⊸ UDestsOf 'Ctor t
&fillComp    :: UDest s ⊸ UAmpar s t ⊸ t
&fillLeaf    :: UDest t ⊸ t          → ()

newUAmpar    :: Token           ⊸ UAmpar s (Dest s)
toUAmpar     :: Token   ⊸ s     → UAmpar s ()
tokenBesides :: UAmpar s t      ⊸ (UAmpar s t, Token)
fromUAmpar   :: UAmpar s (Ur t) ⊸ Ur (s, t)
fromUAmpar'  :: UAmpar s ()     ⊸ Ur s
`updWith`    :: UAmpar s t      ⊸ (t ⊸ u) ⊸ UAmpar s u
```

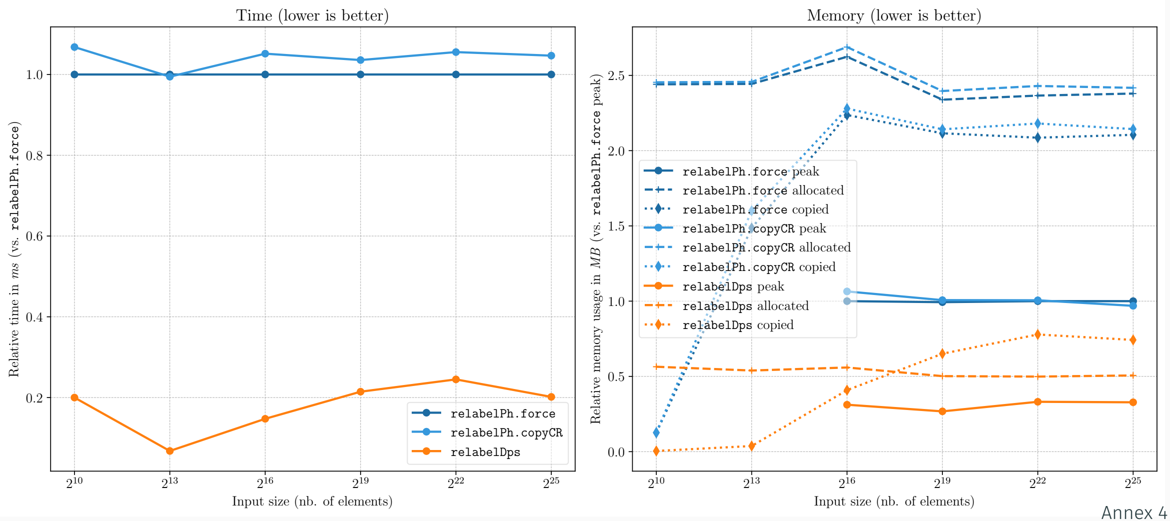
Benchmarks of zero-copy compact region implementation

S-expression parser



Benchmarks of zero-copy compact region implementation

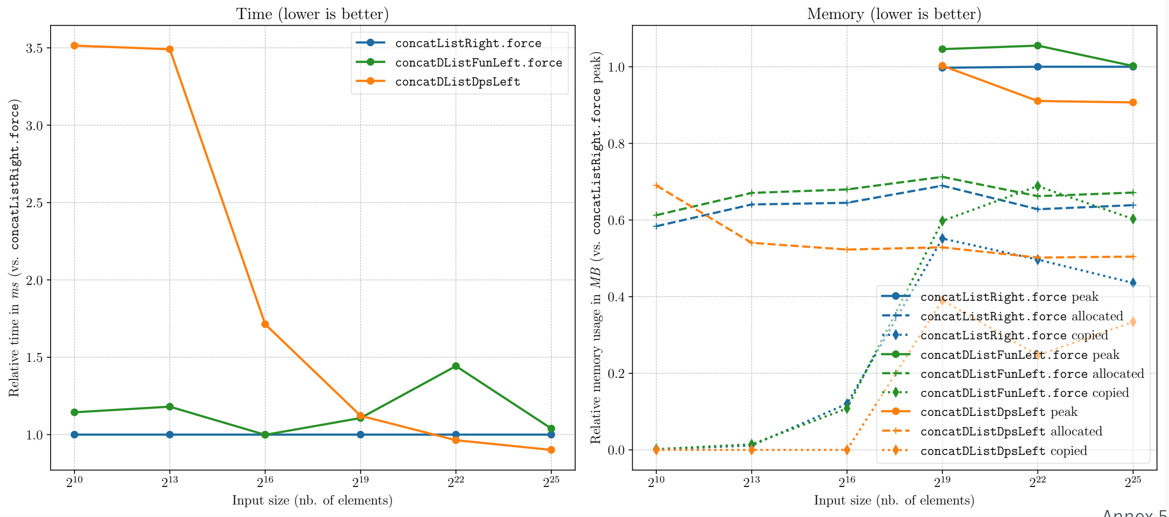
Tree relabeling



Annex 4

Benchmarks of zero-copy compact region implementation

Iterated list concatenation



Ampar: origin of the name

In classical linear logic, the linear implication $\multimap$ can decomposed into $\cdot^{\perp}$ and $\parr$:

$$\mathtt{t} \multimap \mathtt{u} \quad \equiv \quad \mathtt{t}^{\perp} \parr \mathtt{u} \quad \equiv \quad \mathtt{u} \parr \mathtt{t}^{\perp}$$

Minamide shows that a structure of type $\mathtt{u}$ with a hole of type $\mathtt{t}$ can be represented as a form of linear function $\mathtt{t} \overset{\mathtt{mem}}{\multimap} \mathtt{u}$

So we decompose it into $\mathtt{u} \overset{\mathtt{mem}}{\parr} \mathtt{t}^{\overset{\mathtt{mem}}{\perp}}$.

Actually,

- $\mathtt{t}^{\overset{\mathtt{mem}}{\perp}}$ is $\mathtt{Dest}\ \mathtt{t}$
- $\overset{\mathtt{mem}}{\parr}$ is $\mathtt{Ampar}$ type constructor (*asymetrical memory par*)

The *asymetrical* aspect comes from the fact that we lose some nice properties of the original $\parr$ connective because we are not in a classical setting.

Age system VS Rust lifetimes

Ages in λ_d are:

- relative (relative offsets through modality $\texttt{Mod}_{\texttt{m}}$)
- locally exact
- equalities and strict inequalities

Rust lifetimes are:

- global/absolute
- lifetime subtyping = (local) loss of information
- supports only large inequalities

To avoid scope escape, we want to know that **something will die strictly before another** instead of that **something will live at least as long as another**.