Utrecht University

Linear type systems

Concepts of Programming Language Design

Tom Smeding
Gabriele Keller

What's wrong with this program?

- Some C++ code

```
int *array = new int[100];
for (int i = 0; i < 100; i++) {
  array[i] = i * i;
}
delete[] array;
printf("%d\n", array[99]);
```

What's wrong with this program?

- Some C++ code

```
int *array = new int[100];
for (int i = 0; i < 100; i++) {
  array[i] = i * i;
}

printf("%d\n", array[99]);
delete[] array;
delete[] array;
```

What's wrong with this program?

- Some C code

```
FILE *f = fopen("input.txt", "r");
int a;
fscanf(f, "%d", &a);
fclose(f);
int b;
fscanf(f, "%d", &b);
```

What's wrong with this program?

- Some Haskell code

```
main :: IO ()
main = do
  f <- openFile "input.txt" ReadMode
  line1 <- hGetLine f
  hClose f
  line2 <- hGetLine f
  putStrLn line1
  putStrLn line2
```

What's wrong with this program?

- Some C++ code

```
int *array = new int[100];
std::thread th1 = std::thread{[=]() {
  array[10] = 42;
  printf("%d\n", array[10]);
}};
std::thread th2 = std::thread{[=]() {
  array[10] = 100;
  printf("%d\n", array[10]);
}};
th1.join();
th2.join();
```

Tracking resources using types

Tracking resources using types

- Prevent use-after-free (memory) & use-after-close (resources)
  - Alternative: tracing garbage collection → unpredictable overhead
  - Alternative: reference counting → cannot deal with cycles

Tracking resources using types

- Prevent use-after-free (memory) & use-after-close (resources)
  - Alternative: tracing garbage collection → unpredictable overhead
  - Alternative: reference counting → cannot deal with cycles
- Allow mutable state (to go *fast*)
  - Alternative: state monad → no actual mutation + no parallel state access

Tracking resources using types

- Prevent use-after-free (memory) & use-after-close (resources)
  - Alternative: tracing garbage collection → unpredictable overhead
  - Alternative: reference counting → cannot deal with cycles
- Allow mutable state (to go *fast*)
  - Alternative: state monad → no actual mutation + no parallel state access
- Prevent unsafe concurrent access to mutable state
  - Alternative: immutability + message passing → not *fast*

Tracking resources using types

- Prevent use-after-free (memory) & use-after-close (resources)
  - Alternative: tracing garbage collection → unpredictable overhead
  - Alternative: reference counting → cannot deal with cycles
- Allow mutable state (to go *fast*)
  - Alternative: state monad → no actual mutation + no parallel state access
- Prevent unsafe concurrent access to mutable state
  - Alternative: immutability + message passing → not *fast*

- Downside: we're proving resource-correctness to the compiler

Tracking resources using types

- Prevent use-after-free (memory) & use-after-close (resources)
  - Alternative: tracing garbage collection → unpredictable overhead
  - Alternative: reference counting → cannot deal with cycles
- Allow mutable state (to go *fast*)
  - Alternative: state monad → no actual mutation + no parallel state access
- Prevent unsafe concurrent access to mutable state
  - Alternative: immutability + message passing → not *fast*

- Downside: we're proving resource-correctness to the compiler

Implementations of linear-like types

- Rust (affine types + ownership types)
- Haskell (linear types for resources – since GHC 9)
- Clean (uniqueness types for IO, safe array updates)
- Futhark (uniqueness types for safe array updates and parallelism)
- Idris (uniqueness types / linear types for resources)
- … and some more

A toy linear language

A toy linear language

Inspired by Philip Wadler's
Linear types can change the world

Syntax (no linearity yet)

- Language: some data type declarations, then an expression
- Data type declaration:
  - e.g. `data K = C1 T T … T | … | Cn T T … T` (with `T` already-defined types)
- Types: `K` or `T -> T`
- Expressions:
  - `x`
  - `(recfun f : (T -> T) x = e)`
  - `(e e)`
  - `(C e e … e)`
  - `(case e of C x1 … xn -> e`
    `...`
    `C x1 … xn -> e)`

Syntax (no linearity yet)

- Language: some data type declarations, then an expression
- Data type declaration:
  - e.g. `data K = C1 T T … T | … | Cn T T … T` (with `T` already-defined types)

No polymorphism! Let's keep it simple.

- Types: `K` or `T -> T`
- Expressions:
  - `x`
  - `(recfun f : (T -> T) x = e)`
  - `(e e)`
  - `(C e e … e)`
  - `(case e of C x1 … xn -> e`
    `...`
    `C x1 … xn -> e)`

Syntax (no linearity yet)

- Language: some data type declarations, then an expression
- Data type declaration:
  - e.g. `data K = C₁ T T … T | … | Cₙ T T … T` (with `T` already-defined types)

No polymorphism! Let's keep it simple.

- Types: `K` or `T -> T`
- Expressions:
  - `x`
  - `(recfun f : (T -> T) x = e)`
  - `(e e)`
  - `(C e e … e)`
  - `(case e of C x1 … xn -> e`
    `...`
    `C x1 … xn -> e)`

```
λ(x : T) -> e
→
recfun _ : (T -> U) x = e
```

```
let x = e₁ in e₂
→
(λ(x : T) -> e₂) e₁
```

Syntax (no linearity yet)

```
data Bool = True | False
data Nat = Z | S Nat
data Thing = A Bool | B Nat
data List = Nil | Cons Thing List

let length = recfun length : (List -> Nat) l =
                case l of
                  Nil -> Z
                  Cons x l' -> S (length l')
in length (Cons (A True) (Cons (B (S Z)) Nil))
```

Syntax (no linearity yet)

```
data Bool = True | False
data Nat = Z | S Nat
data Thing = A Bool | B Nat
data List = Nil | Cons Thing List

let length = recfun length : (List -> Nat) l =
                case l of
                  Nil -> Z
                  Cons x l' -> S (length l')
in length (Cons (A True) (Cons (B (S Z)) Nil))

      → S (S Z)
```

Normal typing rules

$$\frac{}{A, x : T \vdash x : T}\ \text{Var}$$

Normal typing rules

$$\frac{}{A, x : T \vdash x : T}\ \text{Var}$$

$$\frac{A, f : T \to U, x : T \vdash e : U}{A \vdash (\texttt{recfun}\ f : (T \to U)\ x = e) : T \to U}\ \text{Recfun}\ (f, x \notin A)$$

Normal typing rules

$$\frac{}{A, x : T \vdash x : T}\ \text{Var}$$

$$\frac{A, f : T \to U, x : T \vdash e : U}{A \vdash (\texttt{recfun}\ f : (T \to U)\ x = e) : T \to U}\ \text{Recfun}\ (f, x \notin A)$$

$$\frac{A \vdash e_1 : T \to U \qquad A \vdash e_2 : T}{A \vdash (e_1\ e_2) : U}\ \text{App}$$

Normal typing rules

$$\frac{}{A, x : T \vdash x : T}\ \text{Var}$$

$$\frac{A, f : T \to U, x : T \vdash e : U}{A \vdash (\texttt{recfun}\ f : (T \to U)\ x = e) : T \to U}\ \text{Recfun}\ (f, x \notin A)$$

$$\frac{A \vdash e_1 : T \to U \qquad A \vdash e_2 : T}{A \vdash (e_1\ e_2) : U}\ \text{App}$$

$$\frac{\begin{array}{c}[\text{data}\ K = \cdots \mid C\ T_1\ \ldots\ T_k \mid \cdots] \\ A \vdash e_1 : T_1 \quad \cdots \quad A \vdash e_k : T_k\end{array}}{A \vdash (C\ e_1\ \ldots\ e_k) : K}\ \text{Con}$$

Normal typing rules

$$\frac{}{A, x : T \vdash x : T}\ \text{Var}$$

$$\frac{A, f : T \to U, x : T \vdash e : U}{A \vdash (\texttt{recfun}\ f : (T \to U)\ x = e) : T \to U}\ \text{Recfun}\ (f, x \notin A)$$

$$\frac{A \vdash e_1 : T \to U \qquad A \vdash e_2 : T}{A \vdash (e_1\ e_2) : U}\ \text{App}$$

$$\frac{\begin{array}{c}[\text{data}\ K = \cdots \mid C\ T_1\ \ldots\ T_k \mid \cdots] \\ A \vdash e_1 : T_1 \quad \cdots \quad A \vdash e_k : T_k\end{array}}{A \vdash (C\ e_1\ \ldots\ e_k) : K}\ \text{Con}$$

$$\frac{\begin{array}{c}[\text{data}\ K = C_1\ T_{1,1}\ \ldots\ T_{1,k_1} \mid \cdots \mid C_n\ T_{n,1}\ \ldots\ T_{n,k_n}] \\ A, x_{1,1} : T_{1,1}, \ldots, x_{1,k_1} : T_{1,k_1} \vdash e_1 : U \\ \ldots \\ A \vdash e : K \qquad A, x_{n,1} : T_{n,1}, \ldots, x_{n,k_n} : T_{n,k_n} \vdash e_n : U\end{array}}{A \vdash (\textbf{case}\ e\ \textbf{of}\ \{C_1\ x_{1,1}\ \ldots\ x_{1,k_1} \to e_1 \mid \cdots \mid C_n\ x_{n,1}\ \ldots\ x_{n,k_n} \to e_n\}) : U}\ \text{Case}\ (x_{i,j} \notin A)$$

Examples

```
data Nat = Z | S Nat
data List = Nil | Cons Nat List
data Pair = Pair Nat Nat

λ(x : Nat) -> x
```

Examples

```
data Nat = Z | S Nat
data List = Nil | Cons Nat List
data Pair = Pair Nat Nat
```

```
λ(x : Nat) -> x
```

```
recfun id : (Nat -> Nat) x = x
```

Examples

```
data Nat = Z | S Nat
data List = Nil | Cons Nat List
data Pair = Pair Nat Nat
```

```
λ(x : Nat) -> x
λ(x : Nat) -> λ(y : Nat) -> Pair x y
```

```
recfun id : (Nat -> Nat) x = x
```

Examples

```
data Nat = Z | S Nat
data List = Nil | Cons Nat List
data Pair = Pair Nat Nat
```

```
λ(x : Nat) -> x
λ(x : Nat) -> λ(y : Nat) -> Pair x y
```

```
recfun id : (Nat -> Nat) x = x
```

```
recfun pair : (Nat -> Nat -> Nat) x =
  recfun _ : (Nat -> Nat) y =
    Pair x y
```

Examples

```
data Nat = Z | S Nat
data List = Nil | Cons Nat List
data Pair = Pair Nat Nat
```

```
λ(x : Nat) -> x
λ(x : Nat) -> λ(y : Nat) -> Pair x y
```

```
recfun id : (Nat -> Nat) x = x
```

```
recfun pair : (Nat -> Nat -> Nat) x =
  recfun _ : (Nat -> Nat) y =
    Pair x y
```

```
recfun add : (Nat -> Nat -> Nat) m =
  λ(n : Nat) -> case m of Z -> n
                          S m' -> S (add m' n)
```

Examples

```
data Nat = Z | S Nat
data List = Nil | Cons Nat List
data Pair = Pair Nat Nat
```

```
λ(x : Nat) -> x
λ(x : Nat) -> λ(y : Nat) -> Pair x y
```

```
recfun id : (Nat -> Nat) x = x
```

```
recfun pair : (Nat -> Nat -> Nat) x =
  recfun _ : (Nat -> Nat) y =
    Pair x y
```

```
recfun add : (Nat -> Nat -> Nat) m =
  λ(n : Nat) -> case m of Z -> n
                          S m' -> S (add m' n)
```

```
recfun length : (List -> Nat) l = case l of Nil -> Z
                                            Cons n l' -> S (length l')
```

Examples

```
data Nat = Z | S Nat
data List = Nil | Cons Nat List
data Pair = Pair Nat Nat
```

```
λ(x : Nat) -> x
λ(x : Nat) -> λ(y : Nat) -> Pair x y
```

```
recfun id : (Nat -> Nat) x = x
```

```
recfun pair : (Nat -> Nat -> Nat) x =
  recfun _ : (Nat -> Nat) y =
    Pair x y
```

```
recfun add : (Nat -> Nat -> Nat) m =
  λ(n : Nat) -> case m of Z -> n
                          S m' -> S (add m' n)
```

```
recfun length : (List -> Nat) l = case l of Nil -> Z
                                            Cons n l' -> S (length l')
```

```
recfun sum : (List -> Nat) l = case l of Nil -> Z
                                         Cons n l' -> add n (sum l')
```

What's wrong with this program?

- Some C++ code

```
int *array = new int[100];
for (int i = 0; i < 100; i++) {
  array[i] = i * i;
}
delete[] array;
printf("%d\n", array[99]);
```

Idea

- Add a new category of types:
  - Types whose values *must be used exactly once*

Idea

- Add a new category of types:
  - Types whose values *must be used exactly once*
- Arrays:
  - `new : Int → a → Array a`
  - `set : Int → a → Array a → Array a`
  - `get : Int → Array a → (a, Array a)`
  - `free : Array a → ()`

Idea

- Add a new category of types:
  - Types whose values *must be used exactly once*
- Arrays:
  - `new : Int → a → Array a`
  - `set : Int → a → Array a → Array a`
  - `get : Int → Array a → (a, Array a)`
  - `free : Array a → ()`
- Files:
  - `open : FileMode → FilePath → File`
  - `putchar : Char → File → File`
  - `getchar : File → (Char, File)`
  - `close : File → ()`

Additional syntax

- Language: some data type declarations, then an expression
- Data type declaration:
  - e.g. `data` K = C_1 T T … T | … | C_n T T … T (with T already defined **nonlinear** types)
    e.g. `data` `i` K = `i` C_1 T T … T | … | `i` C_n T T … T
- Types: K **`or` `i` K** or T `->` T **`or` T `i` `->` T**
- Expressions:
  - x
  - (**`recfun`** f : (T -> T) x = e)
  - (e e)
  - (C e e … e)
  - (**`case`** e **`of`** C x_1 … x_n -> e
    ...
    C x_1 … x_n -> e)

inclamation

noun

1. Exclamation.

   Similar: **exclamation**

The GNU version of the Collaborative International Dictionary of English • More at Wordnik

Additional syntax

- Language: some data type declarations, then an expression
- Data type declaration:
  - e.g. `data` K = C_1 T T … T | … | C_n T T … T (with T already defined **nonlinear** types)
    e.g. `data i` K = `i` C_1 T T … T | … | `i` C_n T T … T
- Types: K **`or i`K** or `T -> T` **`or T i -> T`**
- Expressions:
  - `x`
  - `(recfun f : (T -> T) x = e)`
  - `(e e)`
  - `(C e e … e)`
  - `(case e of C` x_1 … x_n `-> e`
    `...`
    `C` x_1 … x_n `-> e)`

Additional syntax

- **Linear** data types can contain anything (including nonlinear types)
- **Nonlinear** data types can *only* contain nonlinear types

- Language: some data type declarations, then an expression
- Data type declaration:
  - e.g. `data` K = C_1 T T … T | … | C_n T T … T (with T already defined **nonlinear** types)
    e.g. `data i` K = `i` C_1 T T … T | … | `i` C_n T T … T
- Types: K **`or i`K** or `T -> T` **or T `i ->` T**
- Expressions:
  - `x`
  - `(`**`recfun`** `f : (T -> T) x = e)`
  - `(e e)`
  - `(C e e … e)`
  - `(`**`case`** `e` **`of`** `C` x_1 … x_n `-> e`
    `...`
    `C` x_1 … x_n `-> e)`

Additional syntax

- **Linear** data types can contain anything (including nonlinear types)
- **Nonlinear** data types can *only* contain nonlinear types

- Language: some data type declarations, then an expression
- Data type declaration:
  - e.g. `data` K = C_1 T T … T | … | C_n T T … T (with T already defined **nonlinear** types)
    e.g. `data i` K = `i` C_1 T T … T | … | `i` C_n T T … T
- Types: K **or `i` K** or `T -> T` **or `T i -> T`**
- Expressions:
  - `x`
  - `(recfun f : (T -> T) x = e)`
  - `(e e)`
  - `(C e e … e)`
  - `(case e of C` x_1 … x_n `-> e`
    `...`
    `C` x_1 … x_n `-> e)`

Linear values must be used exactly once.

Additional syntax

- **Linear** data types can contain anything (including nonlinear types)
- **Nonlinear** data types can *only* contain nonlinear types

- Language: some data type declarations, then an expression
- Data type declaration:
  - e.g. `data` K = C_1 T T … T | … | C_n T T … T (with T already defined **nonlinear** types)
    e.g. `data` `i` K = `i` C_1 T T … T | … | `i` C_n T T … T
- Types: K **or `i` K** or `T -> T` **or `T i -> T`**
- Expressions:
  - `x`
  - `(recfun f : (T -> T) x = e)`
  - `(e e)`
  - `(C e e … e)`
  - `(case e of C` x_1 … x_n `-> e`
    `...`
    `C` x_1 … x_n `-> e)`

Linear values must be used exactly once.

The “linear types” are `i` K and `T i -> T`

Additional syntax

- **Linear** data types can contain anything (including nonlinear types)
- **Nonlinear** data types can *only* contain nonlinear types

- Language: some data type declarations, then an expression
- Data type declaration:
  - e.g. `data` K = C_1 T T … T | … | C_n T T … T (with T already defined **nonlinear** types)
    e.g. `data i` K = `i` C_1 T T … T | … | `i` C_n T T … T
- Types: K **or `i` K** or T `->` T **or T `i ->` T**
- Expressions:
  - x
  - (**`recfun`** f : (T -> T) x = e)
  - (e e)
  - (C e e … e)
  - (**`case`** e **`of`** C x_1 … x_n -> e … C x_1 … x_n -> e)

Linear values must be used exactly once.

The “linear types” are `i` K and T `i ->` T

- (`i`λ(x : T) -> e) : T `i ->` T
- (`i` e e)
- (`i`C e e … e)
- (**`case`** e **`of`** `i`C x_1 … x_n -> e … `i`C x_1 … x_n -> e)

Examples

```
data Nat = Z | S Nat
data iNat = iZ | iS iNat
data iPair = iPair iNat iNat

λ(n : Nat) -> n
```

Examples

```
data Nat = Z | S Nat
data iNat = iZ | iS iNat
data iPair = iPair iNat iNat

λ(n : Nat) -> n
```

✓

Examples

```
data Nat = Z | S Nat
data iNat = iZ | iS iNat
data iPair = iPair iNat iNat

λ(n : Nat) -> n
λ(n : iNat) -> n
```

✓

Examples

```
data Nat = Z | S Nat
data iNat = iZ | iS iNat
data iPair = iPair iNat iNat

λ(n : Nat) -> n          ✓
λ(n : iNat) -> n         ✓
```

Examples

```
data Nat = Z | S Nat
data ¡Nat = ¡Z | ¡S ¡Nat
data ¡Pair = ¡Pair ¡Nat ¡Nat

λ(n : Nat) -> n          ✓
λ(n : ¡Nat) -> n         ✓
λ(n : ¡Nat) -> ¡Pair n n
```

Examples

```
data Nat = Z | S Nat
data iNat = iZ | iS iNat
data iPair = iPair iNat iNat

λ(n : Nat) -> n          ✓
λ(n : iNat) -> n         ✓
λ(n : iNat) -> iPair n n ✗
```

Examples

```
data Nat = Z | S Nat
data iNat = iZ | iS iNat
data iPair = iPair iNat iNat

λ(n : Nat) -> n          ✓
λ(n : iNat) -> n         ✓
λ(n : iNat) -> iPair n n ✗
λ(n : iNat) -> Z
```

Examples

```
data Nat = Z | S Nat
data iNat = iZ | iS iNat
data iPair = iPair iNat iNat

λ(n : Nat) -> n          ✓
λ(n : iNat) -> n         ✓
λ(n : iNat) -> iPair n n ✗
λ(n : iNat) -> Z         ✗
```

Examples

```
data Nat = Z | S Nat
data iNat = iZ | iS iNat
data iPair = iPair iNat iNat

λ(n : Nat) -> n          ✓
λ(n : iNat) -> n         ✓
λ(n : iNat) -> iPair n n ✗
λ(n : iNat) -> Z         ✗
iλ(n : Nat) -> n
```

Examples

```
data Nat = Z | S Nat
data iNat = iZ | iS iNat
data iPair = iPair iNat iNat

λ(n : Nat) -> n          ✓
λ(n : iNat) -> n         ✓
λ(n : iNat) -> iPair n n ✗
λ(n : iNat) -> Z         ✗
iλ(n : Nat) -> n         ✓
```

Examples

```
data Nat = Z | S Nat
data ¡Nat = ¡Z | ¡S ¡Nat
data ¡Pair = ¡Pair ¡Nat ¡Nat

λ(n : Nat) -> n          ✓
λ(n : ¡Nat) -> n         ✓
λ(n : ¡Nat) -> ¡Pair n n ✗
λ(n : ¡Nat) -> Z         ✗
¡λ(n : Nat) -> n         ✓

data ¡List = ¡Nil | ¡Cons Nat ¡List
recfun length : (¡List -> Nat) l = case l of ¡Nil -> Z
                                             ¡Cons x l' -> S (length l')
```

Examples

```
data Nat = Z | S Nat
data iNat = iZ | iS iNat
data iPair = iPair iNat iNat

λ(n : Nat) -> n          ✓
λ(n : iNat) -> n         ✓
λ(n : iNat) -> iPair n n ✗
λ(n : iNat) -> Z         ✗
iλ(n : Nat) -> n         ✓

data iList = iNil | iCons Nat iList
recfun length : (iList -> Nat) l = case l of iNil -> Z
                                             iCons x l' -> S (length l')

data iList2 = iNil2 | iCons2 iNat iList2
recfun length2 : (iList2 -> Nat) l =
  case l of iNil2 -> Z
            iCons2 x l' -> S (length2 l')
```

Example: append

```
data Nat = Z | S Nat
data List = Nil | Cons Nat List


recfun append : (List -> List -> List) l1 =
  λ(l2 : List) ->
    case l1 of Nil -> l2
               Cons x l1' -> Cons x (append l1' l2)
```

Example: append

```
data Nat = Z | S Nat
data List = Nil | Cons Nat List
data ¡List = ¡Nil | ¡Cons Nat ¡List

recfun append : (List -> List -> List) l1 =
  λ(l2 : List) ->
    case l1 of Nil -> l2
               Cons x l1' -> Cons x (append l1' l2)

recfun appendLin : (¡List -> ¡List -> ¡List) l1 =
  λ(l2 : ¡List) ->
    case l1 of ¡Nil -> l2
               ¡Cons x l1' -> ¡Cons x (appendLin l1' l2)
```

Example: append

```
data Nat = Z | S Nat
data ¡List = ¡Nil | ¡Cons Nat ¡List

recfun appendLin : (¡List -> ¡List -> ¡List) l1 =
  λ(l2 : ¡List) ->
    case l1 of ¡Nil -> l2
               ¡Cons x l1' -> ¡Cons x (appendLin l1' l2)
```

Example: append

```
data Nat = Z | S Nat
data ¡List = ¡Nil | ¡Cons Nat ¡List

recfun appendLin : (¡List -> ¡List -> ¡List) l1 =
  λ(l2 : ¡List) ->
    case l1 of ¡Nil -> l2
               ¡Cons x l1' -> ¡Cons x (appendLin l1' l2)



let l1 = ... in let l2 = ... in let l3 = ... in
let appL1 = appendLin l1
in g (appL1 l2) (appL1 l3)
```

Example: append

```
data Nat = Z | S Nat
data iList = iNil | iCons Nat iList

recfun appendLin : (iList -> iList -> iList) l1 =
  λ(l2 : iList) ->
    case l1 of iNil -> l2
               iCons x l1' -> iCons x (appendLin l1' l2)



let l1 = ... in let l2 = ... in let l3 = ... in
let appL1 = appendLin l1
in g (appL1 l2) (appL1 l3)
```

Example: append

```
data Nat = Z | S Nat
data iList = iNil | iCons Nat iList

recfun appendLin : (iList -> (iList i-> iList)) l1 =
  iλ(l2 : iList) ->
    case l1 of iNil -> l2
               iCons x l1' -> iCons x (append l1' l2)



let l1 = ... in let l2 = ... in let l3 = ... in
let appL1 = appendLin l1
in g (appL1 l2) (appL1 l3)
```

Example: append

```
data Nat = Z | S Nat
data iList = iNil | iCons Nat iList

recfun appendLin : (iList -> (iList i-> iList)) l1 =
  iλ(l2 : iList) ->
    case l1 of iNil -> l2
               iCons x l1' -> iCons x (append l1' l2)



let l1 = ... in let l2 = ... in let l3 = ... in
let appL1 = appendLin l1
in g (appL1 l2) (appL1 l3)
```

↑ *Error: appL1 is a linear value and cannot be used multiple times*

Example: append

```
data Nat = Z | S Nat
data iList = iNil | iCons Nat iList

recfun appendLin : (iList -> (iList i-> iList)) l1 =
  iλ(l2 : iList) ->
    case l1 of iNil -> l2
               iCons x l1' -> iCons x (append l1' l2)



let l1 = ... in let l2 = ... in let l3 = ... in
let appL1 = appendLin l1
in g (appL1 l2) (appL1 l3)
```

↑ *Error: appL1 is a linear value and cannot be used multiple times*

Examples

```
data iNat = iZ | iS iNat
data iPair = iPair iNat iNat

recfun add : (iNat -> (iNat i-> iNat)) m =
  iλ(n : iNat) -> case m of iZ -> n
                            iS m' -> iS (add m' n)

data iList2 = iNil2 | iCons2 iNat iList2
recfun sum : (iList2 -> iNat) l =
  case l of iNil2 -> Z
            iCons2 n l' -> add n (sum l')
```

Typing rules for linearity

Making copy/kill explicit

$$\frac{}{A, x : T \vdash x : T}\ \text{Var}$$

$$\downarrow$$

$$\frac{}{x : T \vdash x : T}\ \text{Var}$$

$$\frac{A, x : T, x : T \vdash e : U}{A, x : T \vdash e : U}\ \text{Copy (nonlinear } T\text{)}$$

$$\frac{A \vdash e : U}{A, x : T \vdash e : U}\ \text{Kill (nonlinear } T\text{)}$$

Making copy/kill explicit

$$\frac{A, f : T \to U, x : T \vdash e : U}{A \vdash (\texttt{recfun}\ f : (T \to U)\ x = e) : T \to U}\ \text{Recfun}\ (f, x \notin A)$$

$$\downarrow$$

$$\frac{A, f : T \to U, x : T \vdash e : U}{A \vdash (\texttt{recfun}\ f : (T \to U)\ x = e) : T \to U}\ \text{Recfun}\ (f, x \notin A,\ \text{nonlinear}\ A)$$

Making copy/kill explicit

$$\frac{A \vdash e_1 : T \to U \qquad A \vdash e_2 : T}{A \vdash (e_1\ e_2) : U}\ \text{App}$$

$$\downarrow$$

$$\frac{A \vdash e_1 : T \to U \qquad B \vdash e_2 : T}{A, B \vdash (e_1\ e_2) : U}\ \text{App}$$

Assuming the environment is an unordered list

Making copy/kill explicit

$$\frac{[\text{data } K = \cdots \mid C\ T_1\ \ldots\ T_k \mid \cdots] \quad A \vdash e_1 : T_1 \quad \cdots \quad A \vdash e_k : T_k}{A \vdash (C\ e_1\ \ldots\ e_k) : K}\ \text{Con}$$

$$\downarrow$$

$$\frac{[\text{data } K = \cdots \mid C\ T_1\ \ldots\ T_k \mid \cdots] \quad A_1 \vdash e_1 : T_1 \quad \cdots \quad A_k \vdash e_k : T_k}{A_1, \ldots, A_k \vdash (C\ e_1\ \ldots\ e_k) : K}\ \text{Con}$$

Assuming the environment is an unordered list

Making copy/kill explicit

$$\frac{\begin{array}{c}[\text{data } K = C_1\ T_{1,1}\ \ldots\ T_{1,k_1} \mid \cdots \mid C_n\ T_{n,1}\ \ldots\ T_{n,k_n}] \\ A, x_{1,1} : T_{1,1}, \ldots, x_{1,k_1} : T_{1,k_1} \vdash e_1 : U \\ \cdots \\ A \vdash e : K \qquad A, x_{n,1} : T_{n,1}, \ldots, x_{n,k_n} : T_{n,k_n} \vdash e_n : U\end{array}}{A \vdash (\textbf{case } e \textbf{ of } \{C_1\ x_{1,1}\ \ldots\ x_{1,k_1} \to e_1 \mid \cdots \mid C_n\ x_{n,1}\ \ldots\ x_{n,k_n} \to e_n\}) : U}\ \text{Case } (x_{i,j} \notin A)$$

$$\downarrow$$

$$\frac{\begin{array}{c}[\text{data } K = C_1\ T_{1,1}\ \ldots\ T_{1,k_1} \mid \cdots \mid C_n\ T_{n,1}\ \ldots\ T_{n,k_n}] \\ B, x_{1,1} : T_{1,1}, \ldots, x_{1,k_1} : T_{1,k_1} \vdash e_1 : U \\ \cdots \\ A \vdash e : K \qquad B, x_{n,1} : T_{n,1}, \ldots, x_{n,k_n} : T_{n,k_n} \vdash e_n : U\end{array}}{A, B \vdash (\textbf{case } e \textbf{ of } \{C_1\ x_{1,1}\ \ldots\ x_{1,k_1} \to e_1 \mid \cdots \mid C_n\ x_{n,1}\ \ldots\ x_{n,k_n} \to e_n\}) : U}\ \text{Case } (x_{i,j} \notin B)$$

Assuming the environment is an unordered list

Nonlinear typing rules

$$\frac{}{x : T \vdash x : T}\ \text{Var}$$

$$\frac{A, x : T, x : T \vdash x : T}{A, x : T \vdash e : U}\ \text{Copy (nonlinear } T\text{)}$$

$$\frac{A \vdash e : U}{A, x : T \vdash e : U}\ \text{Kill (nonlinear } T\text{)}$$

$$\frac{A, f : T \to U, x : T \vdash e : U}{A \vdash (\texttt{recfun}\ f : (T \to U)\ x = e) : T \to U}\ \text{Recfun } (f, x \notin A,\ \text{nonlinear } A)$$

$$\frac{A \vdash e_1 : T \to U \qquad B \vdash e_2 : T}{A, B \vdash (e_1\ e_2) : U}\ \text{App}$$

$$\frac{\begin{array}{c}[\text{data } K = \cdots \mid C\ T_1\ \ldots\ T_k \mid \cdots] \\ A_1 \vdash e_1 : T_1 \quad \cdots \quad A_k \vdash e_k : T_k\end{array}}{A_1, \ldots, A_k \vdash (C\ e_1\ \ldots\ e_k) : K}\ \text{Con}$$

$$\frac{\begin{array}{c}[\text{data } K = C_1\ T_{1,1}\ \ldots\ T_{1,k_1} \mid \cdots \mid C_n\ T_{n,1}\ \ldots\ T_{n,k_n}] \\ B, x_{1,1} : T_{1,1}, \ldots, x_{1,k_1} : T_{1,k_1} \vdash e_1 : U \\ \ldots \\ A \vdash e : K \qquad B, x_{n,1} : T_{n,1}, \ldots, x_{n,k_n} : T_{n,k_n} \vdash e_n : U\end{array}}{A, B \vdash (\textbf{case}\ e\ \textbf{of}\ \{C_1\ x_{1,1}\ \ldots\ x_{1,k_1} \to e_1 \mid \cdots \mid C_n\ x_{n,1}\ \ldots\ x_{n,k_n} \to e_n\}) : U}\ \text{Case } (x_{i,j} \notin B)$$

- Same result as the normal typing rules
- Explicit copy & kill
- Requirement that certain types are “nonlinear”

Example

$$plus : N \to N \to N, x : N \vdash plus\ x\ x :$$

Example

$$\dfrac{\dfrac{\dfrac{}{plus : N \to N \to N \vdash plus : N \to N \to N}\,\text{Var} \quad \dfrac{}{x : N \vdash x : N}\,\text{Var}}{plus : N \to N \to N, x : N \vdash plus\ x : N \to N}\,\text{App} \quad \dfrac{}{x : N \vdash x : N}\,\text{Var}}{plus : N \to N \to N, x : N, x : N \vdash plus\ x\ x : N}\,\text{App}}{plus : N \to N \to N, x : N \vdash plus\ x\ x : N}\,\text{Copy}$$

Linear typing rules

$$\frac{}{x : T \vdash x : T}\ \text{Var}$$

$$\frac{A, x : T, x : T \vdash e : U}{A, x : T \vdash e : U}\ \text{Copy (nonlinear } T\text{)}$$

$$\frac{A \vdash e : U}{A, x : T \vdash e : U}\ \text{Kill (nonlinear } T\text{)}$$

These are still good!

(Which was the intent in the first place)

Linear typing rules

$$\frac{A, f : T \to U, x : T \vdash e : U}{A \vdash (\texttt{recfun}\ f : (T \to U)\ x = e) : T \to U}\ \text{Recfun}\ (f, x \notin A,\ \text{nonlinear}\ A)$$

$$+$$

$$\frac{A, x : T \vdash e : U}{A \vdash (\text{¡}\lambda(x : T) \to e) : T \text{¡}\!\to U}\ \text{LLam}\ (f, x \notin A)$$

Linear typing rules

$$\frac{A \vdash e_1 : T \to U \qquad B \vdash e_2 : T}{A, B \vdash (e_1\ e_2) : U}\ \text{App}$$

$$+$$

$$\frac{A \vdash e_1 : T \mathbin{\mathsf{i}\!\to} U \qquad B \vdash e_2 : T}{A, B \vdash (\mathsf{i} e_1\ e_2) : U}\ \text{LApp}$$

Linear typing rules

$$\frac{[\text{data } K = \cdots \mid C\ T_1\ \ldots\ T_k \mid \cdots] \qquad A_1 \vdash e_1 : T_1 \quad \cdots \quad A_k \vdash e_k : T_k}{A_1, \ldots, A_k \vdash (C\ e_1\ \ldots\ e_k) : K}\ \text{Con}$$

Nonlinear data type, hence fields are also nonlinear!

$+$

$$\frac{[\text{data } \text{¡}K = \cdots \mid \text{¡}C\ T_1\ \ldots\ T_k \mid \cdots] \qquad A_1 \vdash e_1 : T_1 \quad \cdots \quad A_k \vdash e_k : T_k}{A_1, \ldots, A_k \vdash (\text{¡}C\ e_1\ \ldots\ e_k) : K}\ \text{LCon}$$

Linear data type, fields might be linear.

Linear typing rules

$$\frac{\begin{array}{c}\left[\text{data } K = C_1\ T_{1,1}\ \ldots\ T_{1,k_1} \mid \cdots \mid C_n\ T_{n,1}\ \ldots\ T_{n,k_n}\right] \\ B, x_{1,1} : T_{1,1}, \ldots, x_{1,k_1} : T_{1,k_1} \vdash e_1 : U \\ \ldots \\ A \vdash e : K \qquad B, x_{n,1} : T_{n,1}, \ldots, x_{n,k_n} : T_{n,k_n} \vdash e_n : U\end{array}}{A, B \vdash (\textbf{case}\ e\ \textbf{of}\ \{C_1\ x_{1,1}\ \ldots\ x_{1,k_1} \to e_1 \mid \cdots \mid C_n\ x_{n,1}\ \ldots\ x_{n,k_n} \to e_n\}) : U}\ \text{Case}\ (x_{i,j} \notin B)$$

$$+$$

$$\frac{\begin{array}{c}\left[\text{data } \text{¡}K = \text{¡}C_1\ T_{1,1}\ \ldots\ T_{1,k_1} \mid \cdots \mid \text{¡}C_n\ T_{n,1}\ \ldots\ T_{n,k_n}\right] \\ B, x_{1,1} : T_{1,1}, \ldots, x_{1,k_1} : T_{1,k_1} \vdash e_1 : U \\ \ldots \\ A \vdash e : \text{¡}K \qquad B, x_{n,1} : T_{n,1}, \ldots, x_{n,k_n} : T_{n,k_n} \vdash e_n : U\end{array}}{A, B \vdash (\textbf{case}\ e\ \textbf{of}\ \{\text{¡}C_1\ x_{1,1}\ \ldots\ x_{1,k_1} \to e_1 \mid \cdots \mid \text{¡}C_n\ x_{n,1}\ \ldots\ x_{n,k_n} \to e_n\}) : U}\ \text{LCase}\ (x_{i,j} \notin B)$$

Full type system

$$\frac{}{x : T \vdash x : T}\ \text{Var} \qquad \frac{A, f : T \to U, x : T \vdash e : U}{A \vdash (\mathtt{recfun}\ f : (T \to U)\ x = e) : T \to U}\ \text{Recfun}\ (f, x \notin A,\ \text{nonlinear}\ A)$$

$$\frac{A, x : T, x : T \vdash e : U}{A, x : T \vdash e : U}\ \text{Copy (nonlinear } T) \qquad \frac{A \vdash e_1 : T \to U \qquad B \vdash e_2 : T}{A, B \vdash (e_1\ e_2) : U}\ \text{App}$$

$$\frac{A \vdash e : U}{A, x : T \vdash e : U}\ \text{Kill (nonlinear } T) \qquad \frac{[\text{data } K = \cdots \mid C\ T_1\ \ldots\ T_k \mid \cdots] \quad A_1 \vdash e_1 : T_1 \quad \cdots \quad A_k \vdash e_k : T_k}{A_1, \ldots, A_k \vdash (C\ e_1\ \ldots\ e_k) : K}\ \text{Con}$$

$$\frac{\begin{array}{c}[\text{data } K = C_1\ T_{1,1}\ \ldots\ T_{1,k_1} \mid \cdots \mid C_n\ T_{n,1}\ \ldots\ T_{n,k_n}] \\ B, x_{1,1} : T_{1,1}, \ldots, x_{1,k_1} : T_{1,k_1} \vdash e_1 : U \\ \ldots \\ A \vdash e : K \qquad B, x_{n,1} : T_{n,1}, \ldots, x_{n,k_n} : T_{n,k_n} \vdash e_n : U\end{array}}{A, B \vdash (\mathbf{case}\ e\ \mathbf{of}\ \{C_1\ x_{1,1}\ \ldots\ x_{1,k_1} \to e_1 \mid \cdots \mid C_n\ x_{n,1}\ \ldots\ x_{n,k_n} \to e_n\}) : U}\ \text{Case}\ (x_{i,j} \notin B)$$

$$\frac{A, x : T \vdash e : U}{A \vdash (\mathfrak{i}\lambda(x : T) \to e) : T \mathfrak{i}\to U}\ \text{LLam}\ (f, x \notin A) \qquad \frac{A \vdash e_1 : T \mathfrak{i}\to U \qquad B \vdash e_2 : T}{A, B \vdash (\mathfrak{i}e_1\ e_2) : U}\ \text{LApp}$$

$$\frac{[\text{data } \mathfrak{i}K = \cdots \mid \mathfrak{i}C\ T_1\ \ldots\ T_k \mid \cdots] \quad A_1 \vdash e_1 : T_1 \quad \cdots \quad A_k \vdash e_k : T_k}{A_1, \ldots, A_k \vdash (\mathfrak{i}C\ e_1\ \ldots\ e_k) : K}\ \text{LCon}$$

$$\frac{\begin{array}{c}[\text{data } \mathfrak{i}K = \mathfrak{i}C_1\ T_{1,1}\ \ldots\ T_{1,k_1} \mid \cdots \mid \mathfrak{i}C_n\ T_{n,1}\ \ldots\ T_{n,k_n}] \\ B, x_{1,1} : T_{1,1}, \ldots, x_{1,k_1} : T_{1,k_1} \vdash e_1 : U \\ \ldots \\ A \vdash e : \mathfrak{i}K \qquad B, x_{n,1} : T_{n,1}, \ldots, x_{n,k_n} : T_{n,k_n} \vdash e_n : U\end{array}}{A, B \vdash (\mathbf{case}\ e\ \mathbf{of}\ \{\mathfrak{i}C_1\ x_{1,1}\ \ldots\ x_{1,k_1} \to e_1 \mid \cdots \mid \mathfrak{i}C_n\ x_{n,1}\ \ldots\ x_{n,k_n} \to e_n\}) : U}\ \text{LCase}\ (x_{i,j} \notin B)$$

Example

$$set : \mathtt{I} \to \mathtt{F} \to \text{¡Array} \to \text{¡Array}, arr : \text{¡Array} \vdash (\text{¡}\lambda(x : \mathtt{F}) \to set\ 2\ x\ arr) : ?$$

$$\frac{}{x : T \vdash x : T}\ \text{Var} \qquad \frac{A \vdash e_1 : T \to U \qquad B \vdash e_2 : T}{A, B \vdash (e_1\ e_2) : U}\ \text{App} \qquad \frac{}{\varepsilon \vdash 4 : \mathtt{Int}} \qquad \frac{}{\varepsilon \vdash 0.0 : \mathtt{Float}}$$

$$\frac{A, x : T, x : T \vdash e : U}{A, x : T \vdash e : U}\ \text{Copy (nonlinear } T\text{)} \qquad \frac{A \vdash e : U}{A, x : T \vdash e : U}\ \text{Kill (nonlinear } T\text{)}$$

Example

$$\dfrac{\dfrac{\dfrac{\dfrac{}{set \vdash set : \mathtt{I} \to \mathtt{F} \to \text{¡Array} \to \text{¡Array}}\,\textsc{Var} \quad \dfrac{}{\epsilon \vdash 2 : \mathtt{I}}\,\textsc{Num}}{set \vdash set\ 2 : \mathtt{F} \to \text{¡Array} \to \text{¡Array}}\,\textsc{App} \quad \dfrac{}{x : \mathtt{F} \vdash x : \mathtt{F}}\,\textsc{Var}}{set, x : \mathtt{F} \vdash set\ 2\ x : \text{¡Array} \to \text{¡Array}}\,\textsc{App} \quad \dfrac{}{arr \vdash arr : \text{¡Array}}\,\textsc{Var}}{set, arr, x : \mathtt{F} \vdash set\ 2\ x\ arr : \text{¡Array}}\,\textsc{App}}{set, arr \vdash (\text{¡}\lambda(x : \mathtt{F}) \to set\ 2\ x\ arr) : \mathtt{F} \text{¡}\to \text{¡Array}}\,\textsc{LLam}$$

$$set : \mathtt{I} \to \mathtt{F} \to \text{¡Array} \to \text{¡Array}, arr : \text{¡Array} \vdash (\text{¡}\lambda(x : \mathtt{F}) \to set\ 2\ x\ arr) : ?$$

$$\frac{}{x : T \vdash x : T}\ \text{Var} \qquad \frac{A \vdash e_1 : T \to U \qquad B \vdash e_2 : T}{A, B \vdash (e_1\ e_2) : U}\ \text{App} \qquad \frac{}{\varepsilon \vdash 4 : \texttt{Int}} \qquad \frac{}{\varepsilon \vdash 0.0 : \texttt{Float}}$$

$$\frac{A, x : T, x : T \vdash e : U}{A, x : T \vdash e : U}\ \text{Copy (nonlinear } T) \qquad \frac{A \vdash e : U}{A, x : T \vdash e : U}\ \text{Kill (nonlinear } T)$$

Example

$$\dfrac{\dfrac{\dfrac{\dfrac{}{set \vdash set : \mathtt{I} \to \mathtt{F} \to \text{¡Array} \to \text{¡Array}}\,\textsc{Var} \quad \dfrac{}{\epsilon \vdash 2 : \mathtt{I}}\,\textsc{Num}}{set \vdash set\ 2 : \mathtt{F} \to \text{¡Array} \to \text{¡Array}}\,\textsc{App} \quad \dfrac{}{x : \mathtt{F} \vdash x : \mathtt{F}}\,\textsc{Var}}{set, x : \mathtt{F} \vdash set\ 2\ x : \text{¡Array} \to \text{¡Array}}\,\textsc{App} \quad \dfrac{}{arr \vdash arr : \text{¡Array}}\,\textsc{Var}}{set, arr, x : \mathtt{F} \vdash set\ 2\ x\ arr : \text{¡Array}}\,\textsc{App}}{set, arr \vdash (\text{¡}\lambda(x : \mathtt{F}) \to set\ 2\ x\ arr) : \mathtt{F} \text{ ¡}\to \text{¡Array}}\,\textsc{LLam}$$

$$set : \mathtt{I} \to \mathtt{F} \to \text{¡Array} \to \text{¡Array}, arr : \text{¡Array} \vdash (\text{¡}\lambda(x : \mathtt{F}) \to set\ 2\ x\ arr) : ?$$

What if the lambda was a normal one?

$$\frac{}{x : T \vdash x : T}\ \text{Var} \qquad \frac{A \vdash e_1 : T \to U \qquad B \vdash e_2 : T}{A, B \vdash (e_1\ e_2) : U}\ \text{App} \qquad \frac{}{\varepsilon \vdash 4 : \mathtt{Int}} \qquad \frac{}{\varepsilon \vdash 0.0 : \mathtt{Float}}$$

$$\frac{A, x : T, x : T \vdash e : U}{A, x : T \vdash e : U}\ \text{Copy (nonlinear } T\text{)} \qquad \frac{A \vdash e : U}{A, x : T \vdash e : U}\ \text{Kill (nonlinear } T\text{)}$$

Example: append

```
data Nat = Z | S Nat
data iList = iNil | iCons Nat iList

recfun appendLin : (iList -> (iList i-> iList)) l1 =
  iλ(l2 : iList) ->
    case l1 of iNil -> l2
               iCons x l1' -> iCons x (append l1' l2)
```

```
let l1 = ... in let l2 = ... in let l3 = ... in
let appL1 = appendLin l1
in g (appL1 l2) (appL1 l3)
```

↑ *Error: appL1 is a linear value and cannot be used multiple times*

Implementations

- Rust:
  - Additional rules for *borrowing* values
    - Via explicit mutable and non-mutable *references*
    - Lifetimes
  - “Ownership types”
- Haskell: retrofitted linear function types into the existing language
  - No `i` like in the example type system; instead: functions that use their *argument* exactly once
  - Took a lot of redesigns
- The toy language presented here: “uniqueness types”
  - https://github.com/tomsmeding/linlang

Conclusions & take-aways

Conclusions & take-aways

- **Performance** of **mutable data structures** in **pure**, functional languages
  - When doing **updates**, **order of operations** must be clear
- IO monad (Haskell): sequentialise everything
- Linear types: local sequentialisation

Conclusions & take-aways

- Main point of linear types: **make duplicating and discarding values *explicit***
- Then we can carefully decide where to allow and where to prohibit
- Can put nonlinear values inside a linear data structure
- **Cannot** put linear values inside a nonlinear data structure
  - Otherwise we can duplicate/discard the linear values by passing the containing data structure around!
- Examples:

Conclusions & take-aways

- Main point of linear types: **make duplicating and discarding values *explicit***
- Then we can carefully decide where to allow and where to prohibit
- Can put nonlinear values inside a linear data structure
- **Cannot** put linear values inside a nonlinear data structure
  - Otherwise we can duplicate/discard the linear values by passing the containing data structure around!
- Examples:
  - `data iLinPair = iMkLinPair Int Float`

Conclusions & take-aways

- Main point of linear types: **make duplicating and discarding values *explicit***
- Then we can carefully decide where to allow and where to prohibit
- Can put nonlinear values inside a linear data structure
- **Cannot** put linear values inside a nonlinear data structure
  - Otherwise we can duplicate/discard the linear values by passing the containing data structure around!
- Examples:
  - `data ¡LinPair = ¡MkLinPair Int Float`
  - `data Pair = MkPair ¡Array Float`

Conclusions & take-aways

- Main point of linear types: **make duplicating and discarding values *explicit***
- Then we can carefully decide where to allow and where to prohibit
- Can put nonlinear values inside a linear data structure
- **Cannot** put linear values inside a nonlinear data structure
  - Otherwise we can duplicate/discard the linear values by passing the containing data structure around!
- Examples:
  - `data iLinPair = iMkLinPair Int Float`
  - ~~`data Pair = MkPair iArray Float`~~

Conclusions & take-aways

- Main point of linear types: **make duplicating and discarding values *explicit***
- Then we can carefully decide where to allow and where to prohibit
- Can put nonlinear values inside a linear data structure
- **Cannot** put linear values inside a nonlinear data structure
  - Otherwise we can duplicate/discard the linear values by passing the containing data structure around!
- Examples:
  - `data ¡LinPair = ¡MkLinPair Int Float`
  - ~~`data Pair = MkPair ¡Array Float`~~
  - `arr : ¡Array ⊢ (¡λx -> set 3 x arr)`

Conclusions & take-aways

- Main point of linear types: **make duplicating and discarding values *explicit***
- Then we can carefully decide where to allow and where to prohibit
- Can put nonlinear values inside a linear data structure
- **Cannot** put linear values inside a nonlinear data structure
  - Otherwise we can duplicate/discard the linear values by passing the containing data structure around!
- Examples:
  - `data ¡LinPair = ¡MkLinPair Int Float`
  - ~~`data Pair = MkPair ¡Array Float`~~
  - `arr : ¡Array ⊢ (¡λx -> set 3 x arr)`
  - `arr : ¡Array ⊢ (λx -> set 3 x arr)`

Conclusions & take-aways

- Main point of linear types: **make duplicating and discarding values *explicit***
- Then we can carefully decide where to allow and where to prohibit
- Can put nonlinear values inside a linear data structure
- **Cannot** put linear values inside a nonlinear data structure
  - Otherwise we can duplicate/discard the linear values by passing the containing data structure around!
- Examples:
  - `data ¡LinPair = ¡MkLinPair Int Float`
  - ~~`data Pair = MkPair ¡Array Float`~~
  - `arr : ¡Array ⊢ (¡λx -> set 3 x arr)`
  - ~~`arr : ¡Array ⊢ (λx -> set 3 x arr)`~~

Conclusions & take-aways

- The example type system was *very carefully* constructed to be correct
  - No rule allowed accidental duplication of linear values
  - We remembered to not put linear values inside a non-linear container

- Designing a programming language is all about trade-offs:
  - Efficiency / safety / convenience / simplicity