Linear Logic and Haskell

Lyra van Bokhoven

January 2025

Introduction

In traditional logic, proofs are free to copy:
$A, A \to B \vdash A \times B$ is a valid judgement.

Introduction

In traditional logic, proofs are free to copy:
$A, A \to B \vdash A \times B$ is a valid judgement.

Let A represent possession of a cake.
Let B represent a feeling of fullness.
Then, the judgement above would say that one can have a cake and eat it too.

Introduction

In traditional logic, proofs are free to copy:
$A, A \to B \vdash A \times B$ is a valid judgement.

Let A represent possession of a cake.
Let B represent a feeling of fullness.
Then, the judgement above would say that one can have a cake and eat it too.

To prevent cakes from being copied, we want a logic that gives us better control over resources. Linear logic provides such control.

Outline

Outline

Notation

A *proposition* is one of:

- a constant X,
- implication between propositions $A \to B$,
- conjunction between propositions $A \times B$, or
- disjunction between propositions $A + B$.

Notation

A *proposition* is one of:

- a constant X,
- implication between propositions $A \to B$,
- conjunction between propositions $A \times B$, or
- disjunction between propositions $A + B$.

An *assumption* is a sequence of 0 or more propositions.

Notation

A *proposition* is one of:

- a constant X,
- implication between propositions $A \to B$,
- conjunction between propositions $A \times B$, or
- disjunction between propositions $A + B$.

An *assumption* is a sequence of 0 or more propositions.
If A is a proposition and Γ is an assumption, then $\Gamma \vdash A$ is a *judgement*.

Notation

A *proposition* is one of:

- a constant X,
- implication between propositions $A \to B$,
- conjunction between propositions $A \times B$, or
- disjunction between propositions $A + B$.

An *assumption* is a sequence of 0 or more propositions.
If A is a proposition and Γ is an assumption, then $\Gamma \vdash A$ is a *judgement*.
A *rule* is a horizontal line with zero or more judgements above it, and one below it:

$$\frac{\Gamma \vdash A + B \qquad \Delta, A \vdash C \qquad \Delta, B \vdash C}{\Gamma, \Delta \vdash C}$$

The rules of intuitionistic logic: tautology

$$\frac{}{A \vdash A}\ \mathsf{Id}$$

The rules of intuitionistic logic: structural rules

$$\frac{\Gamma, \Delta \vdash A}{\Delta, \Gamma \vdash A}\ \text{Exchange}$$

$$\frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}\ \text{Contraction}$$

$$\frac{\Gamma \vdash B}{\Gamma, A \vdash B}\ \text{Weakening}$$

The rules of intuitionistic logic: structural rules

$$\frac{\Gamma, \Delta \vdash A}{\Delta, \Gamma \vdash A}\ \text{Exchange}$$

$$\frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}\ \text{Contraction}$$

$$\frac{\Gamma \vdash B}{\Gamma, A \vdash B}\ \text{Weakening}$$

Might seem trivial, but key difference with linear logic!

The rules of intuitionistic logic: logical rules ($\to$)

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \to\text{-I}$$

$$\frac{\Gamma \vdash A \to B \qquad \Delta \vdash A}{\Gamma, \Delta \vdash B} \to\text{-E}$$

The rules of intuitionistic logic: logical rules ($\to$)

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \to\text{-I}$$

$$\frac{\Gamma \vdash A \to B \qquad \Delta \vdash A}{\Gamma, \Delta \vdash B} \to\text{-E}$$

Sometimes just Γ instead of concatenating Γ and Δ:

$$\frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B} \to\text{-E}$$

Equivalent in intuitionistic logic.
In linear logic, this is seen as combining two different sets of resources.

The rules of intuitionistic logic: logical rules ($\times$)

$$\frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \times B} \times\text{-I}$$

$$\frac{\Gamma \vdash A \times B \qquad \Delta, A, B \vdash C}{\Gamma, \Delta \vdash C} \times\text{-E}$$

The rules of intuitionistic logic: logical rules ($\times$)

$$\frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \times B} \times\text{-I}$$

$$\frac{\Gamma \vdash A \times B \qquad \Delta, A, B \vdash C}{\Gamma, \Delta \vdash C} \times\text{-E}$$

Or, equivalently:

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \times B} \times\text{-I}'$$

$$\frac{\Gamma \vdash A \times B}{\Gamma \vdash A} \times\text{-E}'_1$$

$$\frac{\Gamma \vdash A \times B}{\Gamma \vdash B} \times\text{-E}'_2$$

The rules of intuitionistic logic: logical rules (+)

$$\frac{\Gamma \vdash A}{\Gamma \vdash A + B}\ +\text{-I}_1$$

$$\frac{\Gamma \vdash B}{\Gamma \vdash A + B}\ +\text{-I}_2$$

$$\frac{\Gamma \vdash A + B \qquad \Delta, A \vdash C \qquad \Delta, B \vdash C}{\Gamma, \Delta \vdash C}\ +\text{-E}$$

Example derivation: to have a cake and eat it too

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,\text{Id} \quad \dfrac{\dfrac{}{A \to B \vdash A \to B}\,\text{Id} \quad \dfrac{}{A \vdash A}\,\text{Id}}{A \to B, A \vdash B}\,{\to}\text{-E}}{A, A \to B, A \vdash A \times B}\,{\times}\text{-I}}{A \to B, A, A \vdash A \times B}\,\text{Exchange}}{A \to B, A \vdash A \times B}\,\text{Contraction}$$

Outline

Intuitionistic terms

We introduce a term for each logical rule (and the identity rule), so each connective will have a constructor and a destructor term.

Intuitionistic terms

We introduce a term for each logical rule (and the identity rule), so each connective will have a constructor and a destructor term.

Rule	Term
Id	x
$\to$-I	$\lambda x.u$
$\to$-E	$s(t)$
$\times$-I	(t, u)
$\times$-E	case s of $(x, y) \to v$
$+$-I$_1$	inl(t)
$+$-I$_2$	inr(u)
$+$-E	case s of inl$(x) \to v$; inr$(y) \to w$

Intuitionistic terms

We introduce a term for each logical rule (and the identity rule), so each connective will have a constructor and a destructor term.

Rule	Term
Id	x
$\to$-I	$\lambda x.u$
$\to$-E	$s(t)$
$\times$-I	(t, u)
$\times$-E	case s of $(x, y) \to v$
$+$-I$_1$	inl(t)
$+$-I$_2$	inr(u)
$+$-E	case s of inl$(x) \to v$; inr$(y) \to w$

Since proofs consist of rules, and rules are encoded by terms, a proof tree is uniquely encoded by its root term.

Notation

Assumptions are now written $x_1 : A_1, \ldots, x_n : A_n$
Assumptions must contain distinct variables (important when concatenating).
Judgements now have a term t on the right hand side: $\Gamma \vdash t : A$

Rules with terms

All rules now have terms. Some examples:

$$\frac{}{x : A \vdash x : A}\ \text{Id}$$

$$\frac{\Gamma, y : A, z : A \vdash u : B}{\Gamma, x : A \vdash u\,[x/y, x/z] : B}\ \text{Contraction}$$

$$\frac{\Gamma \vdash s : A \to B \qquad \Delta \vdash t : A}{\Gamma, \Delta \vdash s(t) : B}\ \to\text{-E}$$

Example derivation

$$\dfrac{\dfrac{\dfrac{}{y : A \vdash y : A}\,\text{Id} \quad \dfrac{\dfrac{}{f : A \to B \vdash f : A \to B}\,\text{Id} \quad \dfrac{}{z : A \vdash z : A}\,\text{Id}}{f : A \to B, z : A \vdash f(z) : B}\,{\to}\text{-E}}{y : A, f : A \to B, z : A \vdash (y, f(z)) : A \times B}\,{\times}\text{-I}}{f : A \to B, y : A, z : A \vdash (y, f(z)) : A \times B}\,\text{Exchange}}{f : A \to B, x : A \vdash (x, f(x)) : A \times B}\,\text{Contraction}$$

Outline

Linear logic

We want greater control over resources:

- No discarding (contraction)
- No adding out of nowhere (weakening)

To achieve this, we have two types of assumptions:

- Linear assumptions disallow contraction and weakening, written $\langle A \rangle$. This means that A must be used once.
- Intuitionistic assumptions allow contraction and weakening, written $[A]$. This means that A can be used arbitrarily much.

Symbol summary

We use some new symbols:

symbol	name	usage	pronunciation
$\multimap$	lollipop	$A \multimap B$	"Consume A yielding B"
$\otimes$	tensor	$A \otimes B$	"Both A and B"
$\&$	with	$A \,\&\, B$	"Choose from A and B"
$\oplus$	disjunction	$A \oplus B$	"Either A or B"
!	bang	$!A$	"Of course A"

Let A and B again represent posession of a cake and a feeling of fullness.

Example

Let A mean "I have 10 euros".
Let B mean "I have a pizza".
Let C mean "I have a cake".
Then we can add axioms $\langle A \rangle \vdash B$ and $\langle A \rangle \vdash C$: *One* 10 euro note can be used to get a pizza/cake.

Now, $\langle A \rangle, \langle A \rangle \vdash B \otimes C$ means that for 20 euros, I can buy a pizza and a cake.
$\langle A \rangle \vdash B \,\&\, C$ means that for 10 euros, I can choose between a pizza and a cake.
$\langle A \rangle \vdash B \oplus C$ means that for 10 euros, I can buy a pizza or a cake, but I have no choice. For example, because the bakery is closed.

Example

Let B mean "I have a pizza".
Let C mean "I have a cake".
Let D mean "I am happy.".

Then $\langle B \otimes C \rangle \vdash D$ means I am happy when I have both a pizza and a cake.
$\langle B \,\&\, C \rangle \vdash D$ means I am happy when I have a choice between a pizza and a cake.
$\langle B \oplus C \rangle \vdash D$ means I am happy when I have either a pizza or a cake, and don't care which.

Example

Let A mean "I have 10 euros".
Let B mean "I have a pizza".
Let C mean "I have a cake".
Let D mean "I am happy."

Then $\langle !A \rangle \vdash !B$ means that with one infinite supply of 10 euro notes, I can buy an infinite amount of pizza.
$\langle !B \rangle \vdash D$ means that I am happy when given an infinite supply of pizza.
$[B] \vdash D$ also means that I am happy when given an infinite supply of pizza.

Example

Let A mean "I have 10 euros".
Let B mean "I have a pizza".
Let C mean "I have a cake".
Let D mean "I am happy."

Then $\langle !A \rangle \vdash !B$ means that with one infinite supply of 10 euro notes, I can buy an infinite amount of pizza.
$\langle !B \rangle \vdash D$ means that I am happy when given an infinite supply of pizza.
$[B] \vdash D$ also means that I am happy when given an infinite supply of pizza.
In fact, $\langle !X \rangle$ and $[X]$ prove the exact same propositions.

The rules of linear logic: tautology

$$\frac{}{\langle A \rangle \vdash A}\ \langle \mathsf{Id} \rangle$$

$$\frac{}{[A] \vdash A}\ [\mathsf{Id}]$$

We want to be able to introduce both linear and intuitionistic assumptions.

The rules of linear logic: structural rules

$$\frac{\Gamma, \Delta \vdash A}{\Delta, \Gamma \vdash A}\ \text{Exchange}$$

$$\frac{\Gamma, [A], [A] \vdash B}{\Gamma, [A] \vdash B}\ \text{Contraction}$$

$$\frac{\Gamma \vdash B}{\Gamma, [A] \vdash B}\ \text{Weakening}$$

Note that contraction and weakening only work with intuitionistic assumptions.

The rules of linear logic: logical rules ($\multimap$)

$$\frac{\Gamma, \langle A \rangle \vdash B}{\Gamma \vdash A \multimap B} \multimap\text{-I}$$

$$\frac{\Gamma \vdash A \multimap B \qquad \Delta \vdash A}{\Gamma, \Delta \vdash B} \multimap\text{-E}$$

Note that these rules are the same as those for $\to$, except assumptions are linear.

The rules of linar logic: logical rules ($\otimes$)

$$\frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B} \otimes\text{-I}$$

$$\frac{\Gamma \vdash A \otimes B \qquad \Delta, \langle A \rangle, \langle B \rangle \vdash C}{\Gamma, \Delta \vdash C} \otimes\text{-E}$$

Note that these rules are the same as those for $\times$, except assumptions are linear.

The rules of linar logic: logical rules (&)

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \,\&\, B}\ \&\text{-I}$$

$$\frac{\Gamma \vdash A \,\&\, B}{\Gamma \vdash A}\ \&\text{-E}_1$$

$$\frac{\Gamma \vdash A \,\&\, B}{\Gamma \vdash B}\ \&\text{-E}_2$$

Note that these rules are the same as the alternative rules for $\times$.
They are no longer equivalent in linear logic!

The rules of linear logic: logical rules ($\oplus$)

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \oplus B}\ \oplus\text{-I}_1$$

$$\frac{\Gamma \vdash B}{\Gamma \vdash A \oplus B}\ \oplus\text{-I}_2$$

$$\frac{\Gamma \vdash A \oplus B \qquad \Delta, \langle A \rangle \vdash C \qquad \Delta, \langle B \rangle \vdash C}{\Gamma, \Delta \vdash C}\ \oplus\text{-E}$$

Note that these rules are the same as those for $+$, except assumptions are linear.

The rules of linear logic: logical rules (!)

$$\frac{[\Gamma] \vdash A}{[\Gamma] \vdash\ !A}\ \text{!-I}$$

$$\frac{\Gamma \vdash\ !A \qquad \Delta, [A] \vdash B}{\Gamma, \Delta \vdash B}\ \text{!-E}$$

Note that these rules again use intuitionistic assumptions.

Example derivation: to have a cake and eat it too

$$\dfrac{\dfrac{\dfrac{}{\langle !A\rangle \vdash !A}\,\langle\text{Id}\rangle \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[A]\vdash A}\,[\text{Id}]}{[A]\vdash !A}\,!\text{-I} \quad \dfrac{}{\langle A \multimap B\rangle \vdash A \multimap B}\,\langle\text{Id}\rangle \quad \dfrac{}{[A]\vdash A}\,[\text{Id}]}{\langle A \multimap B\rangle, [A] \vdash B}\,\multimap\text{-E}}{[A], \langle A \multimap B\rangle, [A] \vdash !A \otimes B}\,\times\text{-I}}{\langle A \multimap B\rangle, [A], [A] \vdash !A \otimes B}\,\text{Exchange}}{\langle A \multimap B\rangle, [A] \vdash !A \otimes B}\,\text{Contraction}}{\langle !A\rangle, \langle A \multimap B\rangle \vdash !A \otimes B}\,!\text{-E}$$

Outline

Linear terms

Same idea as intuitionistic terms:

rule	term
$\langle$Id$\rangle$ and [Id]	x
$\multimap$-I	$\lambda\langle x\rangle.u$
$\multimap$-E	$s\langle t\rangle$
$\otimes$-I	$\langle t, u\rangle$
$\otimes$-E	case s of $\langle x, y\rangle \to v$
&-I	$\langle\!\langle t, u\rangle\!\rangle$
&-E$_1$	fst$\langle s\rangle$
&-E$_2$	snd$\langle s\rangle$
$\oplus$-I$_1$	inl$\langle t\rangle$
$\oplus$-I$_2$	inr$\langle u\rangle$
$\oplus$-E	case s of inl$\langle x\rangle \to v$; inr$\langle y\rangle \to w$
!-I	$!t$
!-E	case s of $!x \to u$

Outline

$\lambda^q_{\to}$

$\lambda^q_{\to}$ is the core calculus formalizing most of linear haskell. It formalizes all key features.

Multiplicity

$\lambda^q_{\rightarrow}$ uses multiplicity (often denoted using π and μ) to deal with the difference between linear and intuitionistic statements.
A multiplicity can be 1 (linear) or ω (intuitionistic) in normal linear haskell.

Contexts

Contexts (called "assumptions" before), are variables with associated type and multiplicity:

$$\Gamma = x :_\mu A$$

is a context with a variable x of type A, which behaves like an intuitionistic assumption if $\mu = \omega$, and like a linear assumption if $\mu = 1$.

Arrow type

Arrow types now have a multiplicity: $A \to_\pi B$:

- If $\pi = 1$, the function must use A exactly once.
- If $\pi = \omega$, there are no restrictions on how often the function uses A.

Can sometimes say that the function takes π copies of A to produce a B, or similar terminology.
We may write $\multimap$ for $\to_1$ and $\to$ for $\to_\omega$.

Arrow type

Arrow types now have a multiplicity: $A \to_\pi B$:

- If $\pi = 1$, the function must use A exactly once.
- If $\pi = \omega$, there are no restrictions on how often the function uses A.

Can sometimes say that the function takes π copies of A to produce a B, or similar terminology.
We may write $\multimap$ for $\to_1$ and $\to$ for $\to_\omega$.

For example, if we have f :: Int $\multimap$ Int, then f x $= 2 \cdot$ x is valid, but f x $=$ x $+$ x is not.
However, g :: Int $\to$ Int with g x $=$ x $+$ x is valid.

Datatype declaration

Datatype declarations are of the form

$$\text{data } D\ p_1, \ldots, p_n \text{ where } \left(c_k : A_1 \to_{\pi_1} \ldots A_{n_k} \to_{\pi_{n_k}} D\right)_{k=1}^{m}$$

This declares a datatype D parameterized over multiplicities $p_1, \ldots, p_n$.
There are m constructors c_k, each with n_k arguments.
Arguments of a constructor have a multiplicity, just like arguments of a function. All arguments of multiplicity 1 are used exactly once in the constructor, arguments of multiplicity ω can be used any number of times.
All multiplicities π_i must be among $p_1, \ldots, p_n$.

Judgement

The judgement $\Gamma \vdash t : A$ means that each linear assumption $(x :_1 B) \in \Gamma$ is used exactly once to derive $t : A$.
Intuitionistic assumptions $((y :_\omega C))$ have no restrictions on how often they are used in deriving $t : A$.

Multiplicity operations

Multiplicities can be added and multiplied:

- $1 + 1 = 1 + \omega = \omega + 1 = \omega + \omega = \omega$
- $1 \cdot 1 = 1$
- $1 \cdot \omega = \omega \cdot 1 = \omega \cdot \omega = \omega$

Context operations

Variables are still unique in contexts. Instead of concatenating contexts, we now add them (the first applicable rule is used):

- $(x :_\pi A, \Gamma) + (x :_\mu A, \Delta) = x :_{\pi+\mu} A, (\Gamma + \Delta)$
- $(x :_\pi A, \Gamma) + \Delta = x :_\pi A, (\Gamma + \Delta)$
- $() + \Delta = \Delta$

We can scale contexts with multiplicities:

- $\pi(x :_\mu A, \Gamma) = x :_{\pi\mu} A, \pi\Gamma$

Example: variable rule

$$\frac{}{\omega\Gamma + x :_1 A \vdash x : A}\ \text{var}$$

The assumptions in Γ are not used exactly once to derive $x : A$, so they must all have multiplicity ω.
In linear logic, $x :_\omega A \vdash x : A$ had its own rule ([Id]). In $\lambda^q_\to$, it is an instance of the same rule: if $\Gamma = (x :_1 A)$, the context becomes $\omega(x :_1 A) + (x :_1 A) = (x :_\omega A) + (x :_1 A) = x :_\omega A$.

Example: abstraction rule

$$\frac{\Gamma, x :_\pi A \vdash t : B}{\Gamma \vdash \lambda_\pi (x : A).t : A \to_\pi B}\ \text{abs}$$

If we need π copies of x to derive t, then we can make a function that uses π copies of x and makes a t.

Example: application rule

$$\frac{\Gamma \vdash t : A \to_\pi B \qquad \Delta \vdash u : A}{\Gamma + \pi\Delta \vdash t\ u : B}\ \text{app}$$

We need π copies of u, which we can derive if we have π copies of Δ.

Arrow type: example

Suppose we have a function f :: $s \multimap t$. Then, we can define a function g :: $s \to t$ as g x = f x.

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \mathsf{f} : s \to_1 t \qquad \dfrac{}{\mathsf{x} :_\omega s \vdash \mathsf{x} : s}\ \text{var}}{\Gamma, \mathsf{x} :_\omega s \vdash \mathsf{f}\ \mathsf{x} : t}\ \text{app}}{\Gamma \vdash \lambda_\omega(\mathsf{x} : s).\mathsf{f}\ \mathsf{x} : s \to_\omega t}\ \text{abs}$$

So any linear function can be turned into an intuitionistic one, but not vice versa.

Arrow type: example

Suppose we have a function f :: $s \multimap t$. Then, we can define a function g :: $s \to t$ as g x = f x.

$$\dfrac{\dfrac{\Gamma \vdash \mathsf{f} : s \to_1 t \qquad \dfrac{}{\mathsf{x} :_\omega s \vdash \mathsf{x} : s}\ \text{var}}{\Gamma, \mathsf{x} :_\omega s \vdash \mathsf{f}\ \mathsf{x} : t}\ \text{app}}{\Gamma \vdash \lambda_\omega(\mathsf{x} : s).\mathsf{f}\ \mathsf{x} : s \to_\omega t}\ \text{abs}$$

So any linear function can be turned into an intuitionistic one, but not vice versa.

Linear arrow types are not more restrictive, just more expressive (they give guarantees about how they use their arguments).

Outline

A bit more formal

If the application of a function to an argument is consumed exactly once, then that argument must be consumed exactly once.

To consume exactly once means:

- For an atomic base type (like Int or Ptr), to evaluate it.
- For a function value, to apply it to one argument and consume its result exactly once.
- For an algebraic datatype, to pattern-match on it, and consume all its *linear* components exactly once.

A bit more formal: example

Suppose we have f :: [Int] $\multimap$ [Int], defined as f xs = repeat $1 +\!\!+$ xs.
This seems invalid, it will never consume xs.
However, only if the application of this function to a variable is consumed exactly once do we have that xs must be consumed exactly once.

Linear datatypes

We can define pairs either as $a \multimap b \multimap (a, b)$, or as $a \to b \to (a, b)$. The constructor consumes each argument once. Making this explicit is the cleanest choice.

Linear datatypes

We can define pairs either as $a \multimap b \multimap (a, b)$, or as $a \to b \to (a, b)$. The constructor consumes each argument once. Making this explicit is the cleanest choice.

Every intuitionistic function using or constructing a pair will continue to work if we replace Haskell's pair constructor with the linear one: only if we guarantee to consume the pair exactly once do we need to consume the elements of the pair exactly once.

Linear functions with a linear pair as argument provide an additional guarantee: they use each element exactly once. This is crucial for storing linear values (such as mutable arrays, which we will see soon) in a pair.

Linear datatypes

We can define pairs either as $a \multimap b \multimap (a, b)$, or as $a \to b \to (a, b)$. The constructor consumes each argument once. Making this explicit is the cleanest choice.

Every intuitionistic function using or constructing a pair will continue to work if we replace Haskell's pair constructor with the linear one: only if we guarantee to consume the pair exactly once do we need to consume the elements of the pair exactly once.

Linear functions with a linear pair as argument provide an additional guarantee: they use each element exactly once. This is crucial for storing linear values (such as mutable arrays, which we will see soon) in a pair.

All of the above also holds for lists.

Unrestricted datatype

$$\mathbf{data}\ \text{Unrestricted a}\ \mathbf{where}\ \{\ \text{Unrestricted :: a} \to \text{Unrestricted a}\ \}$$

Let us distinguish two cases:

- Unrestricted a is consumed exactly once
- Unrestricted a is consumed an arbitrary number of times

In both cases, a is consumed an arbitrary number of times, and can therefore not be a linear argument.

Outline

Safe mutable arrays

"Mutable" means that something can be modified ("mutated"). Haskell implements immutable arrays as follows:

```
array :: Int → [(Int, a)] → Array a
array size pairs = runST
  (do {ma ← newMArray size
      ; forM_ pairs (write ma)
      ; unsafeFreeze ma})
```

An array of some size is created, then filled with (index, value) pairs, and then frozen to make it immutable.

Safe mutable arrays

"Mutable" means that something can be modified ("mutated").
Haskell implements immutable arrays as follows:

```
array :: Int → [(Int, a)] → Array a
array size pairs = runST
  (do {ma ← newMArray size
      ; forM_ pairs (write ma)
      ; unsafeFreeze ma})
```

An array of some size is created, then filled with (index, value) pairs, and then frozen to make it immutable.
This approach has some undesirable properties:

- Safe freezing requires copying the array, which would be slow. Instead, the programmer is obligated to not mutate the array after freezing.
- We need to use the ST monad, which is overly sequential.

Safe mutable arrays

$$\text{newMArray} :: \text{Int} \to (\text{MArray a} \multimap \text{Unrestricted b}) \multimap \text{b}$$

$$\text{write} :: \text{MArray a} \multimap (\text{Int}, \text{a}) \to \text{MArray a}$$

$$\text{read} :: \text{MArray a} \multimap \text{Int} \to (\text{MArray a}, \text{Unrestricted a})$$

$$\text{freeze} :: \text{MArray a} \multimap \text{Unrestricted (Array a)}$$

Then we can define array as follows:

$$\text{array size pairs} = \text{newMArray size } (\lambda \text{ma} \multimap \text{freeze (foldl write ma pairs)})$$

Safe mutable arrays

$$\text{newMArray} :: \text{Int} \to (\text{MArray a} \multimap \text{Unrestricted b}) \multimap \text{b}$$

$$\text{write} :: \text{MArray a} \multimap (\text{Int}, \text{a}) \to \text{MArray a}$$

$$\text{read} :: \text{MArray a} \multimap \text{Int} \to (\text{MArray a}, \text{Unrestricted a})$$

$$\text{freeze} :: \text{MArray a} \multimap \text{Unrestricted (Array a)}$$

Then we can define array as follows:

$$\text{array size pairs} = \text{newMArray size } (\lambda \text{ma} \multimap \text{freeze (foldl write ma pairs)})$$

This is safe because:

- newMArray creates only 1 MArray.
- This MArray must be used as a linear argument, so it cannot be copied.
- The MArray cannot be returned (b must be Unrestricted).
- Freezing the MArray consumes it.

Safe mutable arrays

Benefits:

- No more ST monad, so more parallelism.
- More library code is statically typechecked (freeze), less obligation for the programmer.

Linear input/output

Traditional file I/O in Haskell is defined as follows:

$$\mathbf{type}\ \mathsf{File}$$
$$\mathsf{openFile} :: \mathsf{FilePath} \to \mathsf{IO}\ \mathsf{File}$$
$$\mathsf{readLine} :: \mathsf{File} \to \mathsf{IO}\ \mathsf{ByteString}$$
$$\mathsf{closeFile} :: \mathsf{File} \to \mathsf{IO}\ ()$$

Linear input/output

Traditional file I/O in Haskell is defined as follows:

$$\begin{aligned}
&\mathbf{type}\ \mathsf{File}\\
&\mathsf{openFile} :: \mathsf{FilePath} \to \mathsf{IO}\ \mathsf{File}\\
&\mathsf{readLine} :: \mathsf{File} \to \mathsf{IO}\ \mathsf{ByteString}\\
&\mathsf{closeFile} :: \mathsf{File} \to \mathsf{IO}\ ()
\end{aligned}$$

This does not prevent reading from a closed file, or not closing a file at all.

Linear input/output

Traditional file I/O in Haskell is defined as follows:

$$\mathbf{type}\ \mathsf{File}$$
$$\mathsf{openFile} :: \mathsf{FilePath} \to \mathsf{IO}\ \mathsf{File}$$
$$\mathsf{readLine} :: \mathsf{File} \to \mathsf{IO}\ \mathsf{ByteString}$$
$$\mathsf{closeFile} :: \mathsf{File} \to \mathsf{IO}\ ()$$

Linear file I/O is fairly similar:

$$\mathbf{type}\ \mathsf{File}$$
$$\mathsf{openFile} :: \mathsf{FilePath} \to \mathsf{IO}_{\mathsf{L}}\ 1\ \mathsf{File}$$
$$\mathsf{readLine} :: \mathsf{File} \multimap \mathsf{IO}_{\mathsf{L}}\ 1\ (\mathsf{File}, \mathsf{Unrestricted}\ \mathsf{ByteString})$$
$$\mathsf{closeFile} :: \mathsf{File} \multimap \mathsf{IO}_{\mathsf{L}}\ \omega\ ()$$

Linear input/output

Traditional file I/O in Haskell is defined as follows:

$$\begin{aligned}&\mathbf{type}\ \mathsf{File}\\&\mathsf{openFile} :: \mathsf{FilePath} \to \mathsf{IO}\ \mathsf{File}\\&\mathsf{readLine} :: \mathsf{File} \to \mathsf{IO}\ \mathsf{ByteString}\\&\mathsf{closeFile} :: \mathsf{File} \to \mathsf{IO}\ ()\end{aligned}$$

Linear file I/O is fairly similar:

$$\begin{aligned}&\mathbf{type}\ \mathsf{File}\\&\mathsf{openFile} :: \mathsf{FilePath} \to \mathsf{IO}_\mathsf{L}\ 1\ \mathsf{File}\\&\mathsf{readLine} :: \mathsf{File} \multimap \mathsf{IO}_\mathsf{L}\ 1\ (\mathsf{File}, \mathsf{Unrestricted}\ \mathsf{ByteString})\\&\mathsf{closeFile} :: \mathsf{File} \multimap \mathsf{IO}_\mathsf{L}\ \omega\ ()\end{aligned}$$

The IO monad now contains information about whether the value should be used linearly or not, solving the aforementioned issues. But how?

Linear input/output

Since monads have an explicit continuation, we can say whether this continuation must be a linear function. The $\mathsf{IO_L}$ monad is defined as follows:

$$\mathbf{type}\ \mathsf{IO_L\ p\ a}$$

$$\mathsf{return_{IO_L} :: a \to_p\ IO_L\ p\ a}$$

$$\mathsf{bind_{IO_L} :: IO_L\ p\ a \multimap (a \to_p\ IO_L\ q\ b) \multimap IO_L\ q\ b}$$

Conclusion

We have seen linear logic, how it relates to intuitionistic logic, and how it is applied to a programming language to be used in practice.