A taste of Haskell?

Siddharth Bhat

IIIT Open Source Developers group

October 18th, 2019

What's programming like?

A lot like building a cathedral.

Why we pray

Why we pray

```
#include <iostream>
#include <climits>
using namespace std;

// f(x) == true ?
bool f(unsigned x) { return (x + 1) > x; }

int main() {
    cout << "f(0): " << f(0) << "\n";
}
```

Why we pray

```
#include <iostream>
#include <climits>
using namespace std;

// f(x) == true ?
bool f(unsigned x) { return (x + 1) > x; }

int main() {
    cout << "f(0): " << f(0) << "\n";
}
```

```
f(0): 1
```

Why we pray

```cpp
#include <iostream>
#include <climits>
using namespace std;

// f(x) == true ?
bool f(unsigned x) { return (x + 1) > x; }

int main() {
    cout << "f(0): " << f(0) << "\n";
}
```

```
f(0): 1
```

- $f : \mathbb{N} \to \mathbb{B}; f(x) \equiv \begin{cases} \texttt{true} & x+1 > x \\ \texttt{false} & \text{otherwise} \end{cases}$

Why we pray

```
#include <iostream>
#include <climits>
using namespace std;

// f(x) == true ?
bool f(unsigned x) { return (x + 1) > x; }

int main() {
    cout << "f(0): " << f(0) << "\n";
}
```

```
f(0): 1
```

- $f : \mathbb{N} \to \mathbb{B}; f(x) \equiv \begin{cases} \texttt{true} & x+1 > x \\ \texttt{false} & \text{otherwise} \end{cases}$
- $f(x) \equiv \texttt{true}$

Why we pray

```cpp
#include <iostream>
#include <climits>
using namespace std;

// f(x) == true ?
bool f(unsigned x) { return (x + 1) > x; }

int main() {
    cout << "f(0): " << f(0) << "\n";
}
```

```
f(0): 1
```

- $f : \mathbb{N} \to \mathbb{B}; f(x) \equiv \begin{cases} \texttt{true} & x+1 > x \\ \texttt{false} & \text{otherwise} \end{cases}$
- $f(x) \equiv \texttt{true}$

Always going to return `true`?

Why we pray

```
#include <iostream>
#include <climits>
using namespace std;

// f(x) == true ?
bool f(unsigned x) { return (x + 1) > x; }

int main() {
    cout << "f(0): " << f(0) << "\n";
}
```

```
f(0): 1
```

- $f : \mathbb{N} \to \mathbb{B}; f(x) \equiv \begin{cases} \texttt{true} & x+1 > x \\ \texttt{false} & \text{otherwise} \end{cases}$
- $f(x) \equiv \texttt{true}$

Always going to return `true`?

```
int main() {
    cout << "f(UINT32_MAX): " << f(UINT32_MAX) << "\n";
}
```

Why we pray

```
#include <iostream>
#include <climits>
using namespace std;

// f(x) == true ?
bool f(unsigned x) { return (x + 1) > x; }

int main() {
    cout << "f(0): " << f(0) << "\n";
}
```

```
f(0): 1
```

- $f : \mathbb{N} \to \mathbb{B}; f(x) \equiv \begin{cases} \texttt{true} & x+1 > x \\ \texttt{false} & \text{otherwise} \end{cases}$
- $f(x) \equiv \texttt{true}$

Always going to return `true`?

```
int main() {
    cout << "f(UINT32_MAX): " << f(UINT32_MAX) << "\n";
}
```

```
f(UINT32_MAX):0
```

Why we pray

```
#include <iostream>
#include <climits>
using namespace std;

// f(x) == true ?
bool f(unsigned x) { return (x + 1) > x; }

int main() {
    cout << "f(0): " << f(0) << "\n";
}
```

```
f(0): 1
```

- $f : \mathbb{N} \to \mathbb{B}; f(x) \equiv \begin{cases} \texttt{true} & x+1 > x \\ \texttt{false} & \text{otherwise} \end{cases}$
- $f(x) \equiv \texttt{true}$

Always going to return `true`?

```
int main() {
    cout << "f(UINT32_MAX): " << f(UINT32_MAX) << "\n";
}
```

```
f(UINT32_MAX):0
```

- $f : \mathbb{N} \to \mathbb{B}; f(x) \equiv \begin{cases} \texttt{true} & x +_{2^{32}} 1 > x \\ \texttt{false} & \text{otherwise} \end{cases}$

Why we pray

```
#include <iostream>
#include <climits>
using namespace std;

// f(x) == true ?
bool f(unsigned x) { return (x + 1) > x; }

int main() {
    cout << "f(0): " << f(0) << "\n";
}
```

```
f(0): 1
```

- $f : \mathbb{N} \to \mathbb{B}; f(x) \equiv \begin{cases} \texttt{true} & x+1 > x \\ \texttt{false} & \text{otherwise} \end{cases}$
- $f(x) \equiv \texttt{true}$

Always going to return `true`?

```
int main() {
    cout << "f(UINT32_MAX): " << f(UINT32_MAX) << "\n";
}
```

```
f(UINT32_MAX):0
```

- $f : \mathbb{N} \to \mathbb{B}; f(x) \equiv \begin{cases} \texttt{true} & x +_{2^{32}} 1 > x \\ \texttt{false} & \text{otherwise} \end{cases}$
- $+_{2^32} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}; x +_{2^{32}} y \equiv (x +_{\mathbb{N}} y) \% 2^{32}$

Why we pray

```
#include <iostream>
#include <climits>
using namespace std;

// f(x) == true ?
bool f(unsigned x) { return (x + 1) > x; }

int main() {
    cout << "f(0): " << f(0) << "\n";
}
```

```
f(0): 1
```

- $f : \mathbb{N} \to \mathbb{B}; f(x) \equiv \begin{cases} \texttt{true} & x+1 > x \\ \texttt{false} & \text{otherwise} \end{cases}$
- $f(x) \equiv \texttt{true}$

Always going to return `true`?

```
int main() {
    cout << "f(UINT32_MAX): " << f(UINT32_MAX) << "\n";
}
```

```
f(UINT32_MAX):0
```

- $f : \mathbb{N} \to \mathbb{B}; f(x) \equiv \begin{cases} \texttt{true} & x +_{2^{32}} 1 > x \\ \texttt{false} & \text{otherwise} \end{cases}$
- $+_{2^32} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}; x +_{2^{32}} y \equiv (x +_{\mathbb{N}} y)\%2^{32}$; $\texttt{UINT32\_MAX} +_{2^{32}} 1 = 0$

Why we pray: A second example

Why we pray: A second example

$$\forall x \in \mathbb{Z}, 2 * x = x + x$$

Why we pray: A second example

$$\forall x \in \mathbb{Z}, 2 * x = x + x$$

(Also holds in modular arithmetic)

Why we pray: A second example

$$\forall x \in \mathbb{Z}, 2 * x = x + x$$

(Also holds in modular arithmetic)

```
int main() {
  cout << 2 * ((int) getchar()) << "\n";
}
```

Why we pray: A second example

$$\forall x \in \mathbb{Z}, 2 * x = x + x$$

(Also holds in modular arithmetic)

```
int main() {
  cout << 2 * ((int) getchar()) << "\n";
}
```

```
int main() {
  cout << ((int)getchar() + (int)getchar()) << "\n";
}
```

Why we pray: A second example

$$\forall x \in \mathbb{Z}, 2 * x = x + x$$

(Also holds in modular arithmetic)

```
int main() {
  cout << 2 * ((int) getchar()) << "\n";
}
```

```
int main() {
  cout << ((int)getchar() + (int)getchar()) << "\n";
}
```

- `int getchar()`

Why we pray: A second example

$$\forall x \in \mathbb{Z}, 2 * x = x + x$$

(Also holds in modular arithmetic)

```
int main() {
  cout << 2 * ((int) getchar()) << "\n";
}
```

```
int main() {
  cout << ((int)getchar() + (int)getchar()) << "\n";
}
```

- `int getchar()`
- `getchar` : $\emptyset \rightarrow$ `int`

Why we pray: A second example

$$\forall x \in \mathbb{Z}, 2 * x = x + x$$

(Also holds in modular arithmetic)

```
int main() {
  cout << 2 * ((int) getchar()) << "\n";
}
```

```
int main() {
  cout << ((int)getchar() + (int)getchar()) << "\n";
}
```

- `int getchar()`
- `getchar` : $\emptyset \rightarrow$ `int`
- $\{\} \rightarrow \{1, 42, \ldots\}$

Why we pray: A second example

$$\forall x \in \mathbb{Z}, 2 * x = x + x$$

(Also holds in modular arithmetic)

```
int main() {
  cout << 2 * ((int) getchar()) << "\n";
}
```

```
int main() {
  cout << ((int)getchar() + (int)getchar()) << "\n";
}
```

- `int getchar()`
- `getchar` : $\emptyset \rightarrow$ `int`
- $\{\} \rightarrow \{1, 42, \ldots\}$
- Such a function can't return an output!

Why we pray: A second example

$$\forall x \in \mathbb{Z}, 2 * x = x + x$$

(Also holds in modular arithmetic)

```
int main() {
  cout << 2 * ((int) getchar()) << "\n";
}
```

```
int main() {
  cout << ((int)getchar() + (int)getchar()) << "\n";
}
```

- `int getchar()`
- `getchar` : $\emptyset \to$ `int`
- $\{\} \to \{1, 42, \ldots\}$
- Such a function can't return an output!
- `getchar` : $\{\star\} \to$ `int`

Why we pray: A second example

$$\forall x \in \mathbb{Z}, 2 * x = x + x$$

(Also holds in modular arithmetic)

```
int main() {
  cout << 2 * ((int) getchar()) << "\n";
}
```

```
int main() {
  cout << ((int)getchar() + (int)getchar()) << "\n";
}
```

- `int getchar()`
- `getchar` : $\emptyset \to$ `int`
- $\{\} \to \{1, 42, \ldots\}$
- Such a function can't return an output!
- `getchar` : $\{\star\} \to$ `int`
- $\{\star\} \to \{1, 42, \ldots\}$

Why we pray: A second example

$$\forall x \in \mathbb{Z}, 2 * x = x + x$$

(Also holds in modular arithmetic)

```
int main() {
  cout << 2 * ((int) getchar()) << "\n";
}
```

```
int main() {
  cout << ((int)getchar() + (int)getchar()) << "\n";
}
```

- `int getchar()`
- `getchar` : $\emptyset \to$ `int`
- $\{\} \to \{1, 42, \ldots\}$
- Such a function can't return an output!
- `getchar` : $\{\star\} \to$ `int`
- $\{\star\} \to \{1, 42, \ldots\}$
- $\{\star\} \to \{1, 42, \ldots\}\ \star \mapsto 42$

Why we pray: A second example

$$\forall x \in \mathbb{Z}, 2 * x = x + x$$

(Also holds in modular arithmetic)

```
int main() {
  cout << 2 * ((int) getchar()) << "\n";
}
```

```
int main() {
  cout << ((int)getchar() + (int)getchar()) << "\n";
}
```

- `int getchar()`
- `getchar` : $\emptyset \to$ `int`
- $\{\} \to \{1, 42, \ldots\}$
- Such a function can't return an output!
- `getchar` : $\{\star\} \to$ `int`
- $\{\star\} \to \{1, 42, \ldots\}$
- $\{\star\} \to \{1, 42, \ldots\}$ $\star \mapsto 42$
- Such a function will always return the same output!

Why we pray: A second example

$$\forall x \in \mathbb{Z}, 2 * x = x + x$$

(Also holds in modular arithmetic)

```
int main() {
  cout << 2 * ((int) getchar()) << "\n";
}
```

```
int main() {
  cout << ((int)getchar() + (int)getchar()) << "\n";
}
```

- `int getchar()`
- `getchar` : $\emptyset \to$ `int`
- $\{\} \to \{1, 42, \ldots\}$
- Such a function can't return an output!
- `getchar` : $\{\star\} \to$ `int`
- $\{\star\} \to \{1, 42, \ldots\}$
- $\{\star\} \to \{1, 42, \ldots\}$ $\star \mapsto 42$
- Such a function will always return the same output!
- `getchar` can't be a mathematical function.

Why we pray: A second example

$$\forall x \in \mathbb{Z}, 2 * x = x + x$$

(Also holds in modular arithmetic)

```
int main() {
  cout << 2 * ((int) getchar()) << "\n";
}
```

```
int main() {
  cout << ((int)getchar() + (int)getchar()) << "\n";
}
```

- `int getchar()`
- `getchar` : $\emptyset \to$ `int`
- $\{\} \to \{1, 42, \ldots\}$
- Such a function can't return an output!
- `getchar` : $\{\star\} \to$ `int`
- $\{\star\} \to \{1, 42, \ldots\}$
- $\{\star\} \to \{1, 42, \ldots\}$ $\star \mapsto 42$
- Such a function will always return the same output!
- `getchar` can't be a mathematical function.

Why we pray: A third example

```
int K (int x, int y) { return x; } // K(x, y) = x
```

Why we pray: A third example

```
int K (int x, int y) { return x; } // K(x, y) = x
int main() { cout << K(10, 20) << "\n" cout << 10; }
```

Why we pray: A third example

```
int K (int x, int y) { return x; } // K(x, y) = x

int main() { cout << K(10, 20) << "\n" cout << 10; }
```

K(10, 20) = 10

Why we pray: A third example

```
int K (int x, int y) { return x; } // K(x, y) = x

int main() { cout << K(10, 20) << "\n" cout << 10; }
```

$K(10, 20) = 10$

```
int err() { exit(1); return 0; } // start praying!
int main() { cout << K(10, err()) << "\n" cout << 10; }
```

Why we pray: A third example

```
int K (int x, int y) { return x; } // K(x, y) = x

int main() { cout << K(10, 20) << "\n" cout << 10; }
```

K(10, 20) = 10

```
int err() { exit(1); return 0; } // start praying!
int main() { cout << K(10, err()) << "\n" cout << 10; }
```

K(10, err()) $\neq$ 10

Why we pray: A third example

```
int K (int x, int y) { return x; } // K(x, y) = x

int main() { cout << K(10, 20) << "\n" cout << 10; }
```

K(10, 20) = 10

```
int err() { exit(1); return 0; } // start praying!
int main() { cout << K(10, err()) << "\n" cout << 10; }
```

K(10, err()) $\neq$ 10

- Mathematically, can replace $K(x, y)$ by x.

Why we pray: A third example

```
int K (int x, int y) { return x; } // K(x, y) = x

int main() { cout << K(10, 20) << "\n" cout << 10; }
```

K(10, 20) = 10

```
int err() { exit(1); return 0; } // start praying!
int main() { cout << K(10, err()) << "\n" cout << 10; }
```

K(10, err()) $\neq$ 10

- Mathematically, can replace $K(x, y)$ by x.
- In C(++), impossible.

Why we pray: A third example

```
int K (int x, int y) { return x; } // K(x, y) = x

int main() { cout << K(10, 20) << "\n" cout << 10; }
```

`K(10, 20)` $= 10$

```
int err() { exit(1); return 0; } // start praying!
int main() { cout << K(10, err()) << "\n" cout << 10; }
```

`K(10, err())` $\neq 10$

- Mathematically, can replace $K(x, y)$ by x.
- In C(++), impossible.
- cannot **equationally reason** about programs.

Why we pray: A third example

```
int K (int x, int y) { return x; } // K(x, y) = x

int main() { cout << K(10, 20) << "\n" cout << 10; }
```

K(10, 20) = 10

```
int err() { exit(1); return 0; } // start praying!
int main() { cout << K(10, err()) << "\n" cout << 10; }
```

K(10, err()) $\neq$ 10

- Mathematically, can replace $K(x, y)$ by x.
- In C(++), impossible.
- cannot **equationally reason** about programs.
- **Can we** reason about programs?

Why we pray: A third example

```
int K (int x, int y) { return x; } // K(x, y) = x

int main() { cout << K(10, 20) << "\n" cout << 10; }
```

K(10, 20) = 10

```
int err() { exit(1); return 0; } // start praying!
int main() { cout << K(10, err()) << "\n" cout << 10; }
```

K(10, err()) $\neq$ 10

- Mathematically, can replace $K(x, y)$ by x.
- In C(++), impossible.
- cannot **equationally reason** about programs.
- **Can we** reason about programs?

Can we reason about C++?

Can we reason about C++?

- Short answer: Yes.

Can we reason about C++?

- Short answer: Yes.
- Longer answer: Yes. It's complicated.

Can we reason about C++?

- Short answer: Yes.
- Longer answer: Yes. It's complicated.
- Name dropping: Operational/Denotational semantics.

Can we reason about C++?

- Short answer: Yes.
- Longer answer: Yes. It's complicated.
- Name dropping: Operational/Denotational semantics.
- Name dropping: Hoare logic/Separation logic.

Can we reason about C++?

- Short answer: Yes.
- Longer answer: Yes. It's complicated.
- Name dropping: Operational/Denotational semantics.
- Name dropping: Hoare logic/Separation logic.

How do we escape having to pray?

How do we escape having to pray?

Define a language,

How do we escape having to pray?

Define a language, where anything that happens,

How do we escape having to pray?

Define a language, where anything that happens, is what **mathematics** says should happen.

How do we escape having to pray?

Define a language, where anything that happens, is what **mathematics** says should happen.

A taste of Haskell

- `let xs = [1, 2,..10]`

A taste of Haskell

- `let xs = [1, 2,..10]`
- `xs = range(10)`

A taste of Haskell

- `let xs = [1, 2,..10]`
- `xs = range(10)`
- `let nats = [1, 2..]`

A taste of Haskell

- `let xs = [1, 2,..10]`
- `xs = range(10)`
- `let nats = [1, 2..]`
- `xs = ??`

A taste of Haskell

- `let xs = [1, 2,..10]`
- `xs = range(10)`
- `let nats = [1, 2..]`
- `xs = ??`
- `take 2 nats`

A taste of Haskell

- `let xs = [1, 2,..10]`
- `xs = range(10)`
- `let nats = [1, 2..]`
- `xs = ??`
- `take 2 nats`
- `take 10 nats`

A taste of Haskell

- `let xs = [1, 2,..10]`
- `xs = range(10)`
- `let nats = [1, 2..]`
- `xs = ??`
- `take 2 nats`
- `take 10 nats`
- `take 100 nats`

A taste of Haskell

- `let xs = [1, 2,..10]`
- `xs = range(10)`
- `let nats = [1, 2..]`
- `xs = ??`
- `take 2 nats`
- `take 10 nats`
- `take 100 nats`
- Haskell is *lazily evaluated*

A taste of Haskell

- `let xs = [1, 2,..10]`
- `xs = range(10)`
- `let nats = [1, 2..]`
- `xs = ??`
- `take 2 nats`
- `take 10 nats`
- `take 100 nats`
- Haskell is *lazily evaluated*

Why laziness?

- `:t "foo"`

Why laziness?

- `:t "foo"`
- `"foo" :: [Char]`
- `take 2 "foo"`

Why laziness?

- `:t "foo"`
- `"foo" :: [Char]`
- `take 2 "foo"`
- `:t take`
- `take :: Int -> [a] -> [a]`

Why laziness?

- `:t "foo"`
- `"foo" :: [Char]`
- `take 2 "foo"`
- `:t take`
- `take :: Int -> [a] -> [a]`
- take :: $\forall(\alpha \in \texttt{TYPE}), \texttt{Int} \rightarrow \texttt{List}(\alpha) \rightarrow \texttt{List}(\alpha)$

Why laziness?

- `:t "foo"`
- `"foo" :: [Char]`
- `take 2 "foo"`
- `:t take`
- `take :: Int -> [a] -> [a]`
- take :: $\forall(\alpha \in \texttt{TYPE}), \texttt{Int} \rightarrow \texttt{List}(\alpha) \rightarrow \texttt{List}(\alpha)$
- `take 1 'a'`

Why laziness?

- `:t "foo"`
- `"foo" :: [Char]`
- `take 2 "foo"`
- `:t take`
- `take :: Int -> [a] -> [a]`
- take :: $\forall(\alpha \in \mathtt{TYPE}), \mathtt{Int} \rightarrow \mathtt{List}(\alpha) \rightarrow \mathtt{List}(\alpha)$
- `take 1 'a'`
- `Prelude> take 1 'a'`

```
<interactive>:11:8: error:
    • Couldn't match expected type ‘[a]’ with actual type ‘Char’
    • In the second argument of ‘take’, namely ‘'a'’
      In the expression: take 1 'a'
      In an equation for ‘it’: it = take 1 'a'
    • Relevant bindings include it :: [a] (bound at <interactive>:11:1)
```

Why laziness?

- `:t "foo"`
- `"foo" :: [Char]`
- `take 2 "foo"`
- `:t take`
- `take :: Int -> [a] -> [a]`
- take :: $\forall(\alpha \in \mathtt{TYPE}), \mathtt{Int} \to \mathtt{List}(\alpha) \to \mathtt{List}(\alpha)$
- `take 1 'a'`
- `Prelude> take 1 'a'`

```
<interactive>:11:8: error:
    • Couldn't match expected type ‘[a]’ with actual type ‘Char’
    • In the second argument of ‘take’, namely ‘'a'’
      In the expression: take 1 'a'
      In an equation for ‘it’: it = take 1 'a'
    • Relevant bindings include it :: [a] (bound at <interactive>:11:1)
```

- If $f : A \to B$,

Why laziness?

- `:t "foo"`
- `"foo" :: [Char]`
- `take 2 "foo"`
- `:t take`
- `take :: Int -> [a] -> [a]`
- take :: $\forall(\alpha \in \mathtt{TYPE}), \mathtt{Int} \to \mathtt{List}(\alpha) \to \mathtt{List}(\alpha)$
- `take 1 'a'`
- `Prelude> take 1 'a'`

```
<interactive>:11:8: error:
    • Couldn't match expected type ‘[a]’ with actual type ‘Char’
    • In the second argument of ‘take’, namely ‘'a'’
      In the expression: take 1 'a'
      In an equation for ‘it’: it = take 1 'a'
    • Relevant bindings include it :: [a] (bound at <interactive>:11:1)
```

- If $f : A \to B$, can ask $f(a)$ for $a \in A$.

Philosophical differences

Philosophical differences

```
" ".join(["a", "b", "c", "d"])
```

Philosophical differences

```
" ".join(["a", "b", "c", "d"])
```

`str.join(iterable)`

Return a string which is the concatenation of the strings in *iterable*. A `TypeError` will be raised if there are any non-string values in *iterable*, including `bytes` `objects`. The separator between elements is the string providing this method.

Philosophical differences

```
intercalate " " ["a", "b", "c", "d"]
```

```
" ".join(["a", "b", "c", "d"])
```

`str.join(iterable)`

Return a string which is the concatenation of the strings in *iterable*. A `TypeError` will be raised if there are any non-string values in *iterable*, including `bytes` `objects`. The separator between elements is the string providing this method.

Philosophical differences

```
" ".join(["a", "b", "c", "d"])
```

```
str.join(iterable)
```

Return a string which is the concatenation of the strings in *iterable*. A `TypeError` will be raised if there are any non-string values in *iterable*, including `bytes` `objects`. The separator between elements is the string providing this method.

```
intercalate " " ["a", "b", "c", "d"]
```

```
intercalate ::
  String -> [String] -> String
```

Philosophical differences

```
" ".join(["a", "b", "c", "d"])
```

`str.join(iterable)`

Return a string which is the concatenation of the strings in *iterable*. A `TypeError` will be raised if there are any non-string values in *iterable*, including `bytes` `objects`. The separator between elements is the string providing this method.

```
intercalate " " ["a", "b", "c", "d"]
```

```
intercalate ::
  String -> [String] -> String
```

`intercalate xs xss` is *equivalent* to `(concat (intersperse xs xss))`. It inserts the list `xs` in between the lists in `xss` and concatenates the result.

Philosophical differences

```
" ".join(["a", "b", "c", "d"])
```

```
str.join(iterable)
```

Return a string which is the concatenation of the strings in *iterable*. A `TypeError` will be raised if there are any non-string values in *iterable*, including `bytes` `objects`. The separator between elements is the string providing this method.

```
intercalate " " ["a", "b", "c", "d"]
```

```
intercalate ::
  String -> [String] -> String
```

`intercalate xs xss` is *equivalent* to `(concat (intersperse xs xss))`. It inserts the list `xs` in between the lists in `xss` and concatenates the result.

```
intercalate ::
  [Char] -> [[Char]] -> [Char]
intercalate :: [a] -> [[a]] -> [a]
```

Philosophical differences

```
" ".join(["a", "b", "c", "d"])
```

`str.join(iterable)`

Return a string which is the concatenation of the strings in *iterable*. A `TypeError` will be raised if there are any non-string values in *iterable*, including `bytes` objects. The separator between elements is the string providing this method.

```
intercalate " " ["a", "b", "c", "d"]
```

```
intercalate ::
  String -> [String] -> String
```

`intercalate xs xss` is *equivalent* to `(concat (intersperse xs xss))`. It inserts the list `xs` in between the lists in `xss` and concatenates the result.

```
intercalate ::
  [Char] -> [[Char]] -> [Char]
intercalate :: [a] -> [[a]] -> [a]
```

```
intercalate [1, 2]
  [[3,4], [30, 40], [300, 400]]
```

Philosophical differences

```
" ".join(["a", "b", "c", "d"])
```

`str.join(iterable)`

Return a string which is the concatenation of the strings in *iterable*. A `TypeError` will be raised if there are any non-string values in *iterable*, including `bytes` objects. The separator between elements is the string providing this method.

```
intercalate " " ["a", "b", "c", "d"]
```

```
intercalate ::
  String -> [String] -> String
```

`intercalate xs xss` is *equivalent* to `(concat (intersperse xs xss))`. It inserts the list `xs` in between the lists in `xss` and concatenates the result.

```
intercalate ::
  [Char] -> [[Char]] -> [Char]
intercalate :: [a] -> [[a]] -> [a]
```

```
intercalate [1, 2]
  [[3,4], [30, 40], [300, 400]]
```

```
= [3, 4, 1, 2, 30, 40, 1, 2, 300, 400]
```

Philosophical differences

```
" ".join(["a", "b", "c", "d"])
```

`str.join(iterable)`

Return a string which is the concatenation of the strings in *iterable*. A `TypeError` will be raised if there are any non-string values in *iterable*, including `bytes` objects. The separator between elements is the string providing this method.

docs.python.org/3/library/stdtypes.html#str.join

```
intercalate " " ["a", "b", "c", "d"]
```

```
intercalate ::
  String -> [String] -> String
```

`intercalate xs xss` is *equivalent* to `(concat (intersperse xs xss))`. It inserts the list `xs` in between the lists in `xss` and concatenates the result.

```
intercalate ::
  [Char] -> [[Char]] -> [Char]
intercalate :: [a] -> [[a]] -> [a]
```

```
intercalate [1, 2]
  [[3,4], [30, 40], [300, 400]]
```

```
= [3, 4, 1, 2, 30, 40, 1, 2, 300, 400]
```

hackage.haskell.org/package/base-4.14.0.0/docs/Data-List.html#v:intercalate

Summing over a list of numbers

```
sum([1, 2, 3, 4])
```

Summing over a list of numbers

```
sum([1, 2, 3, 4])
```

```
sum(iterable, /, start=0)
```

Sums *start* and the items of an *iterable* from left to right and returns the total. The *iterable*'s items are normally numbers, and the start value is not allowed to be a string.

Summing over a list of numbers

```
sum([1, 2, 3, 4])
```

```
sum(iterable, /, start=0)
```

Sums *start* and the items of an *iterable* from left to right and returns the total. The *iterable*'s items are normally numbers, and the start value is not allowed to be a string.

```
sum [1, 2, 3, 4]
```

Summing over a list of numbers

```
sum([1, 2, 3, 4])
```

```
sum(iterable, /, start=0)
```

Sums *start* and the items of an *iterable* from left to right and returns the total. The *iterable*'s items are normally numbers, and the start value is not allowed to be a string.

```
sum [1, 2, 3, 4]
```

```
sum :: (Foldable t, Num a) => t a -> a
sum :: [Int] -> Int]
```

Summing over a list of numbers

```
sum([1, 2, 3, 4])
```

```
sum(iterable, /, start=0)
```

Sums *start* and the items of an *iterable* from left to right and returns the total. The *iterable*'s items are normally numbers, and the start value is not allowed to be a string.

```
sum [1, 2, 3, 4]
```

```
sum :: (Foldable t, Num a) => t a -> a
sum :: [Int] -> Int]
```

The `sum` function computes the sum of the numbers of a structure.

Summing over a list of numbers

```
sum([1, 2, 3, 4])
```

```
sum(iterable, /, start=0)
```

Sums *start* and the items of an *iterable* from left to right and returns the total. The *iterable*'s items are normally numbers, and the start value is not allowed to be a string.

- `sum [1, 2, 3]`

```
sum [1, 2, 3, 4]
```

```
sum :: (Foldable t, Num a) => t a -> a
sum :: [Int] -> Int]
```

The `sum` function computes the sum of the numbers of a structure.

Summing over a list of numbers

```
sum([1, 2, 3, 4])
```

```
sum(iterable, /, start=0)
```

Sums *start* and the items of an *iterable* from left to right and returns the total. The *iterable*'s items are normally numbers, and the start value is not allowed to be a string.

- `sum [1, 2, 3]`
- `sum [1.1, 2.1, -3.2]`

```
sum [1, 2, 3, 4]
```

```
sum :: (Foldable t, Num a) => t a -> a
sum :: [Int] -> Int]
```

The `sum` function computes the sum of the numbers of a structure.

Summing over a list of numbers

```
sum([1, 2, 3, 4])
```

```
sum(iterable, /, start=0)
```

Sums *start* and the items of an *iterable* from left to right and returns the total. The *iterable*'s items are normally numbers, and the start value is not allowed to be a string.

- `sum [1, 2, 3]`
- `sum [1.1, 2.1, -3.2]`
- `let minus_1_by_12 = sum [1, 2..]`

```
sum [1, 2, 3, 4]
```

```
sum :: (Foldable t, Num a) => t a -> a
sum :: [Int] -> Int]
```

The `sum` function computes the sum of the numbers of a structure.

Summing over a list of numbers

```
sum([1, 2, 3, 4])
```

```
sum(iterable, /, start=0)
```

Sums *start* and the items of an *iterable* from left to right and returns the total. The *iterable*'s items are normally numbers, and the start value is not allowed to be a string.

- `sum [1, 2, 3]`
- `sum [1.1, 2.1, -3.2]`
- `let minus_1_by_12 = sum [1, 2..]`

https://docs.python.org/3/library/functions.html#sum

```
sum [1, 2, 3, 4]
```

```
sum :: (Foldable t, Num a) => t a -> a
sum :: [Int] -> Int]
```

The `sum` function computes the sum of the numbers of a structure.

hackage.haskell.org/package/base-4.14.0.0/docs/Data-List.html#v:intercalate

Num?

```
sum :: (Foldable t, Num a) => t a -> a
```

Num?

```
sum :: (Foldable t, Num a) => t a -> a
```

```
class  Num a  where -- Typeclass: `a` is a Num if...
```

Num?

```
sum :: (Foldable t, Num a) => t a -> a
class  Num a  where -- Typeclass: `a` is a Num if...
    (+), (-), (*)       :: a -> a -> a
```

Num?

```
sum :: (Foldable t, Num a) => t a -> a

class  Num a  where -- Typeclass: `a` is a Num if...
    (+), (-), (*)       :: a -> a -> a
    -- | Unary negation.
    negate              :: a -> a
    -- | Absolute value.
    abs                 :: a -> a
```

Num?

```
sum :: (Foldable t, Num a) => t a -> a
class  Num a  where -- Typeclass: `a` is a Num if...
    (+), (-), (*)       :: a -> a -> a
    -- | Unary negation.
    negate              :: a -> a
    -- | Absolute value.
    abs                 :: a -> a
    -- | Sign of a number.
    -- The functions 'abs' and 'signum' should satisfy the law:
    --
    -- > abs x * signum x == x
    --
    -- For real numbers, the 'signum' is either `-1` (negative), `0` (zero)
    -- or `1` (positive).
    ...
```

Num?

```
sum :: (Foldable t, Num a) => t a -> a
class  Num a  where -- Typeclass: `a` is a Num if...
    (+), (-), (*)       :: a -> a -> a
    -- | Unary negation.
    negate              :: a -> a
    -- | Absolute value.
    abs                 :: a -> a
    -- | Sign of a number.
    -- The functions 'abs' and 'signum' should satisfy the law:
    --
    -- > abs x * signum x == x
    --
    -- For real numbers, the 'signum' is either `-1` (negative), `0` (zero)
    -- or `1` (positive).
    ...
```

- Associativity of + `(x + y) + z = x + (y + z)`

Num?

```
sum :: (Foldable t, Num a) => t a -> a
class  Num a  where -- Typeclass: `a` is a Num if...
    (+), (-), (*)       :: a -> a -> a
    -- | Unary negation.
    negate              :: a -> a
    -- | Absolute value.
    abs                 :: a -> a
    -- | Sign of a number.
    -- The functions 'abs' and 'signum' should satisfy the law:
    --
    -- > abs x * signum x == x
    --
    -- For real numbers, the 'signum' is either `-1` (negative), `0` (zero)
    -- or `1` (positive).
    ...
```

- Associativity of + `(x + y) + z = x + (y + z)`
- Commutativity of +: `x + y = y + x`

Num?

```
sum :: (Foldable t, Num a) => t a -> a
class  Num a  where -- Typeclass: `a` is a Num if...
    (+), (-), (*)       :: a -> a -> a
    -- | Unary negation.
    negate              :: a -> a
    -- | Absolute value.
    abs                 :: a -> a
    -- | Sign of a number.
    -- The functions 'abs' and 'signum' should satisfy the law:
    --
    -- > abs x * signum x == x
    --
    -- For real numbers, the 'signum' is either `-1` (negative), `0` (zero)
    -- or `1` (positive).
    ...
```

- Associativity of + `(x + y) + z = x + (y + z)`
- Commutativity of +: `x + y = y + x`
- negate gives the additive inverse: `x + negate x = fromInteger 0`

Num?

```
sum :: (Foldable t, Num a) => t a -> a
class  Num a  where -- Typeclass: `a` is a Num if...
    (+), (-), (*)       :: a -> a -> a
    -- | Unary negation.
    negate              :: a -> a
    -- | Absolute value.
    abs                 :: a -> a
    -- | Sign of a number.
    -- The functions 'abs' and 'signum' should satisfy the law:
    --
    -- > abs x * signum x == x
    --
    -- For real numbers, the 'signum' is either `-1` (negative), `0` (zero)
    -- or `1` (positive).
    ...
```

- Associativity of + `(x + y) + z = x + (y + z)`
- Commutativity of +: `x + y = y + x`
- negate gives the additive inverse: `x + negate x = fromInteger 0`

Num?

```
sum :: (Foldable t, Num a) => t a -> a
class  Num a  where -- Typeclass: `a` is a Num if...
    (+), (-), (*)       :: a -> a -> a
    -- | Unary negation.
    negate              :: a -> a
    -- | Absolute value.
    abs                 :: a -> a
    -- | Sign of a number.
    -- The functions 'abs' and 'signum' should satisfy the law:
    --
    -- > abs x * signum x == x
    --
    -- For real numbers, the 'signum' is either `-1` (negative), `0` (zero)
    -- or `1` (positive).
    ...
```

- Associativity of + `(x + y) + z = x + (y + z)`
- Commutativity of +: `x + y = y + x`
- negate gives the additive inverse: `x + negate x = fromInteger 0`

https://hackage.haskell.org/package/base-4.14.0.0/docs/Prelude.html#t:Num

Foldable?

```
sum :: (Foldable t, Num a) => t a -> a
```

Foldable?

```
sum :: (Foldable t, Num a) => t a -> a
```

An object one can "fold over" / "accumulate answers on".

Foldable?

```
sum :: (Foldable t, Num a) => t a -> a
```

An object one can "fold over" / "accumulate answers on". A list, a set, a binary tree, ...

- `let x = Data.Set.fromList [1, 2, 3]`

Foldable?

```
sum :: (Foldable t, Num a) => t a -> a
```

An object one can "fold over" / "accumulate answers on". A list, a set, a binary tree, ...

- `let x = Data.Set.fromList [1, 2, 3]`
- `fromList [1,2,3]`

Foldable?

```
sum :: (Foldable t, Num a) => t a -> a
```

An object one can "fold over" / "accumulate answers on". A list, a set, a binary tree, ...

- `let x = Data.Set.fromList [1, 2, 3]`
- `fromList [1,2,3]`
- `let y = union x x`

Foldable?

```
sum :: (Foldable t, Num a) => t a -> a
```

An object one can "fold over" / "accumulate answers on". A list, a set, a binary tree, ...

- `let x = Data.Set.fromList [1, 2, 3]`
- `fromList [1,2,3]`
- `let y = union x x`
- `fromList [1,2,3]`

Foldable?

```
sum :: (Foldable t, Num a) => t a -> a
```

An object one can "fold over" / "accumulate answers on". A list, a set, a binary tree, ...

- `let x = Data.Set.fromList [1, 2, 3]`
- `fromList [1,2,3]`
- `let y = union x x`
- `fromList [1,2,3]`
- `sum y`
- 6

Foldable in detail

```
sum :: (Foldable t, Num a) => t a -> a
```

Foldable in detail

```
sum :: (Foldable t, Num a) => t a -> a
```

```
class Foldable t where -- a data structure t is foldable if...
  -- | Map each element of the structure to a monoid, and combine the results.
  foldMap :: Monoid m => (a -> m) -> t a -> m
```

Foldable instances are expected to satisfy the following laws:

```
foldr f z t = appEndo (foldMap (Endo . f) t ) z
foldl f z t = appEndo (getDual (foldMap (Dual . Endo . flip f) t)) z
fold = foldMap id
length = getSum . foldMap (Sum . const  1)
```

https://hackage.haskell.org/package/base-4.14.0.0/docs/Data-Foldable.html#t:Foldable