A taste of Haskell?

Siddharth Bhat

IIIT Open Source Developers group

October 18th, 2019

What's programming like?

A lot like building a cathedral.

What's programming like?

A lot like building a cathedral.

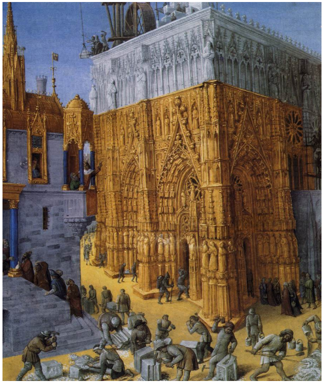

Figure: First we build

What's programming like?

A lot like building a cathedral.

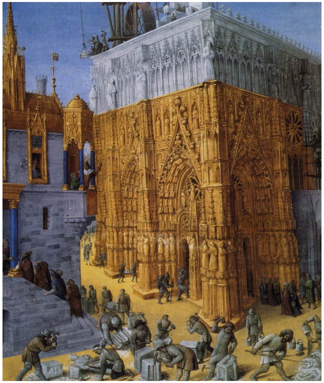

Figure: First we build

Figure: Then we pray

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

```
f(0): 1
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";
    cout << "f(UINT32_MAX: " << f(UINT32_MAX) << "\n";
}
```

```
f(0): 1
f(UINT32_MAX):0
```

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

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

```
f(0): 1
f(UINT32_MAX):0
```

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

Why we pray

```
#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";
    cout << "f(UINT32_MAX: " << f(UINT32_MAX) << "\n";
}
```

```
f(0): 1
f(UINT32_MAX):0
```

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

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

```
f(0): 1
f(UINT32_MAX):0
```

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

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

```
f(0): 1
f(UINT32_MAX):0
```

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

Why we pray: A second example

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

Why we pray: A second example

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

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

Why we pray: A second example

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

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

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

Why we pray: A second example

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

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

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

- `int getchar()`

Why we pray: A second example

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

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

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

- `int getchar()`
- $\texttt{getchar} : \emptyset \rightarrow \texttt{int}$

Why we pray: A second example

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

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

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

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

Why we pray: A second example

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

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

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

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

Why we pray: A second example

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

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

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

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

Why we pray: A second example

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

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

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

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

Why we pray: A second example

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

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

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

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

Why we pray: A second example

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

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

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

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

Why we pray: A second example

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

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

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

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

Why we pray: A second example

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

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

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

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

- `[1, 2,..10]`

A taste of Haskell

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

A taste of Haskell

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

A taste of Haskell

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

Philosophical differences

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

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

```
" ".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 :: [a] -> [[a]] -> [a]
```

`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.

Why should I learn haskell?

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

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

```
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
```

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

Foldable in detail

```haskell
class Foldable t where
  -- | 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:

```haskell
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

Fibonacci

```
let fib = 0:1:(zipWith (+) fib (tail fib))
```

Equational reasoning

```
k x y = x
k 10 (error "urk")
```

```
def k(x, y): return x
k(x, input())
```

What is `input` anyway?

$$\emptyset \mapsto \texttt{char}$$

Mathematically impossible!

Effects, or the "M" word

Keep every element, and drop every element.

```
powerset xs = filterM (const [True, False]) xs
```