Constrained Integers and Structural Types

Outline

1. Programming Languages
2. Types
3. Internals

(1) Programming Languages

Language Design

- Programming languages are tools for expressing computation
- Key ingrediants in computation: *State* and *Transitions*
- Programming language is a human-friendly way to express both

Programming Languages (Ruby)

```
def indexOf(needle, haystack)
  haystack.chars.each_with_index do |chr, i|
    if chr == needle then
      return i
    end
  end

  return -1
end
```

Programming Languages (Go)

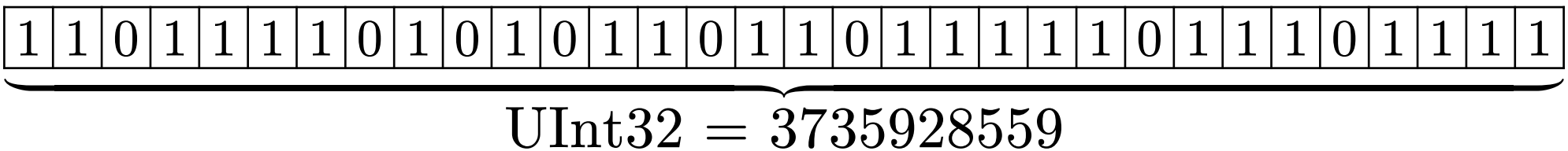
```
func indexOf(str string, c rune) int {
  runes := []rune(str);

  for i := 0; i < len(runes); i++ {
    if runes[i] == c { return i; }
  }

  return -1;
}
```

Programming Languages (Howlite)

```
func indexOf(s: &[char; NatI32], c: char): NatI32 | -1 {
  let i: UInt32 = 0;
  while i < s.len {
    if str[i] == c {
        return i;
    };
    i = i + 1;
  };
  -1
}
```

(2) Types

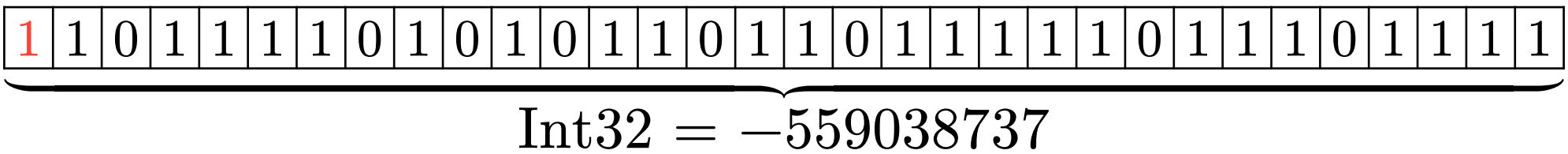
Types at a Low Level



Key Question about Integers:

- How many bits? (32)
- Does it have a sign bit? (no)

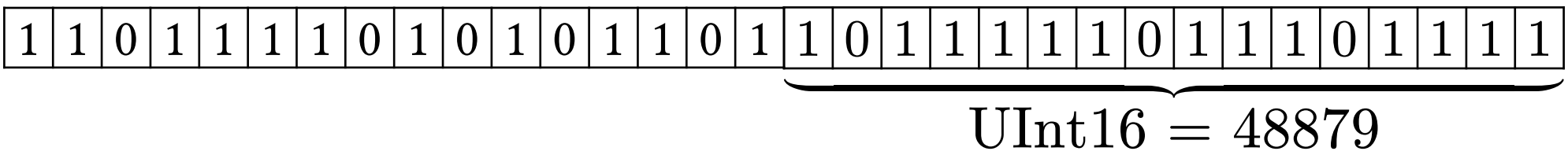
Types at a Low Level



Key Question about Integers:

- How many bits? (32)
- Does it have a sign bit? (yes)

Types at a Low Level



Key Question about Integers:

- How many bits? (16)
- Does it have a sign bit? (no)

Composition

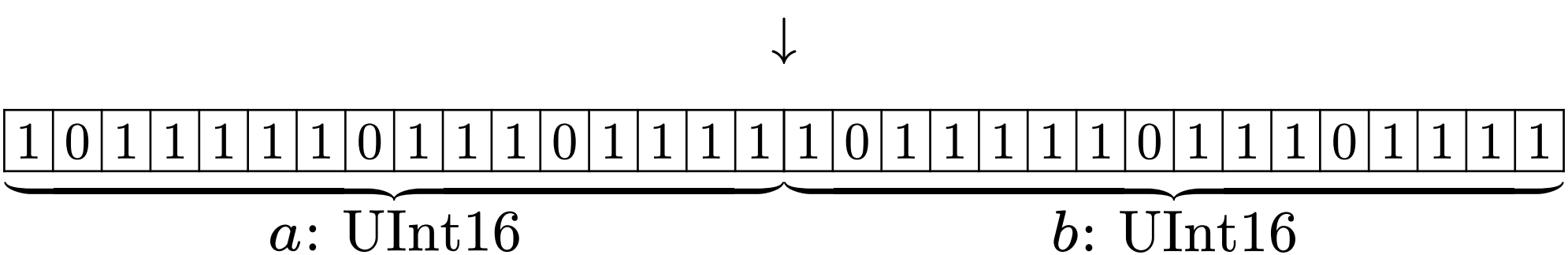
```
type Pair = {
  a: UInt16,
  b: UInt16
}
```

Composition

```
type Pair = {
  a: UInt16,
  b: UInt16
}

let pair: Pair = #{ a: 0xDEAD, b: 0xBEEF };
```

Composition

```
type Pair = {
  a: UInt16,
  b: UInt16
}

let pair: Pair = #{ a: 0xDEAD, b: 0xBEEF };
```



Composition

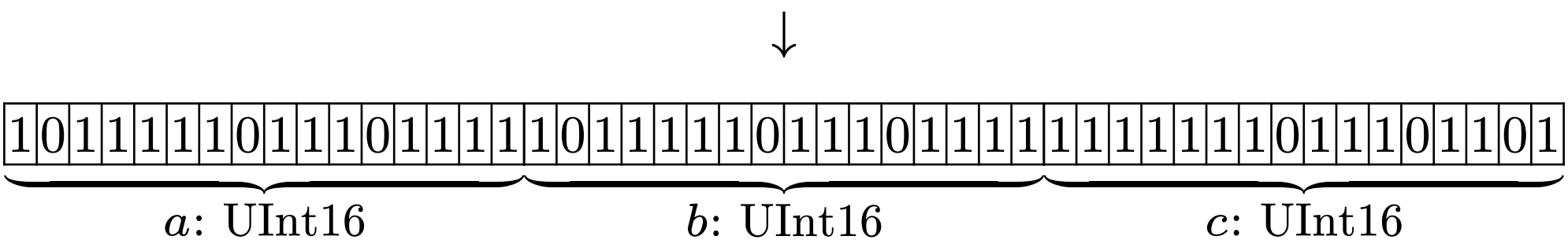
```
type Triple = {
  a: UInt16,
  b: UInt16,
  c: UInt16
}
```

Composition

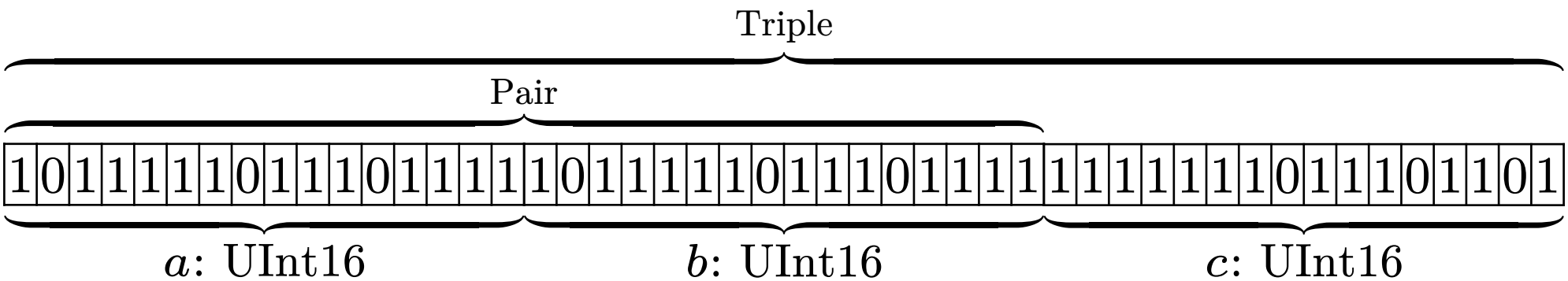
```
type Triple = {
  a: UInt16,
  b: UInt16,
  c: UInt16
}

let triple: Triple = #{ a: 0xDEAD, b: 0xBEEF, c: 0xFEED };
```

Composition

```
type Triple = {
  a: UInt16,
  b: UInt16,
  c: UInt16
}

let triple: Triple = #{ a: 0xDEAD, b: 0xBEEF, c: 0xFEED };
```



Subtyping




```
type Triple = { a: UInt16, b: UInt16, c: UInt16 }

type Pair   = { a: UInt16, b: UInt16 }
```

Adding Another Layer

The story so far

- Programs have *State*
- A *Type System* is a way to describe a program's state

Integer Types: *length* (# of bits) and *sign* (can it be negative)

Structure Types: A sequence of *named* fields, each with its own *type*

Adding Another Layer

The story so far

- Programs have *State*
- A *Type System* is a way to describe a program's state

Integer Types: *length* (# of bits) and *sign* (can it be negative)

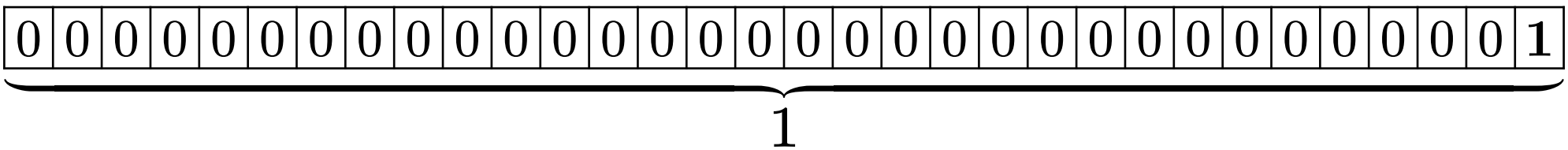
Structure Types: A sequence of *named* fields, each with its own *type*

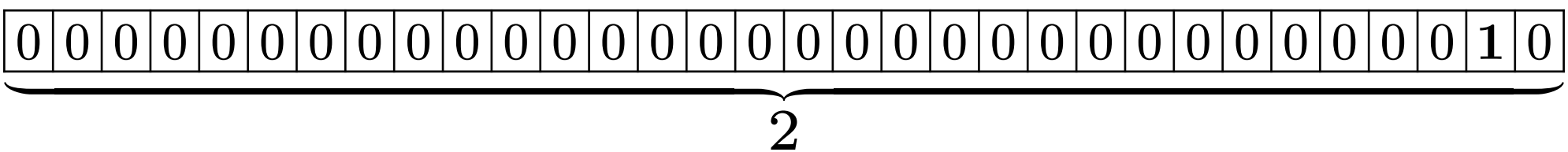
Static Types for Single States

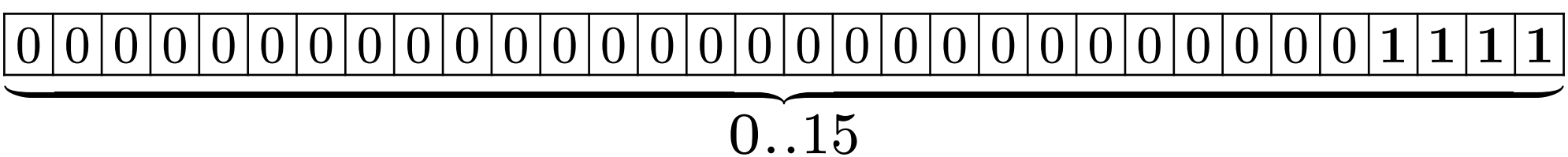
"This is a 32-bit integer"

"This is an 32-integer, and it has a value of 1"

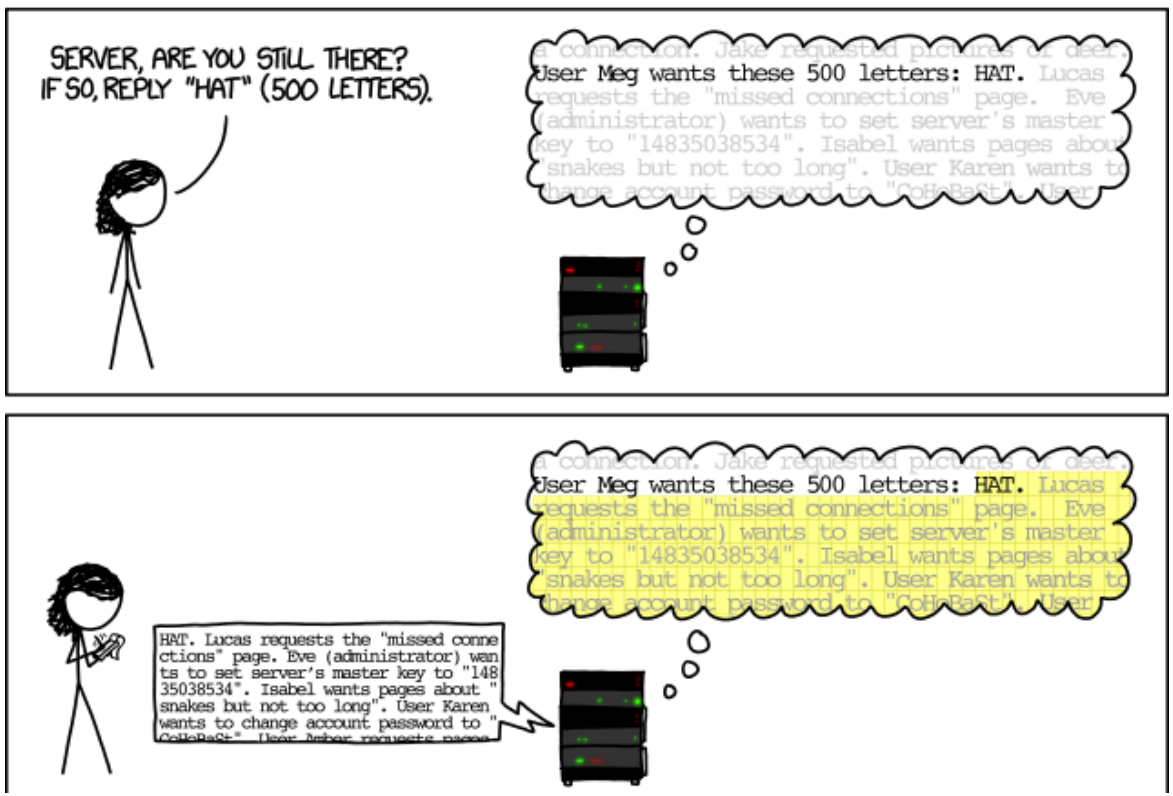
Only Using a Few Bits







Intermission



Why?

1. **Static bounds checks**

```
let buffer: [UInt8; 1024] = ...;
let start: UInt32 = 256;
let end: UInt32 = /* user input */;
&buffer[start..end];
^^^^^^^^^^^^^^^^^^^^

Error! Potential out-of-bounds read.
```

Why?

1. **Static bounds checks**

```
let buffer: [UInt8; 1024] = ...;
let start: UInt32 = 256;
let end: 256..1023 = /* user input */;
&buffer[start..end];
^^^^^^^^^^^^^^^^^^^
Ok now :)
```

Why?

1. Static bounds checks
2. **Expressiveness**

```
// get the N'th bit of a 32-bit int
func bit(n: UInt32, bit: UInt8): bool

// get the N'th bit of a 32-bit int
func bit(n: UInt32, bit: 0..31): 0|1
```

Why?

1. Static bounds checks
2. **Expressiveness**

```
type T = { t: 1, payload: UInt32 }
       | { t: 2, payload: Bool }
       | { t: 3, payload: &String }
```

Why?

1. Static bounds checks
2. Expressiveness
3. **Type Narrowing**

```
type T = { t: 1, payload: UInt32 }
       | { t: 2, payload: Bool }
       | { t: 3, payload: &String }
```

```
type T = {
  t: 1 | 2 | 3,
  payload: UInt32 | Bool | &String
}
```

Why?

1. Static bounds checks
2. Expressiveness
3. **Type Narrowing**

```
type T = { t: 1, payload: UInt32 }
       | { t: 2, payload: Bool }
       | { t: 3, payload: &String }
```

If some instance of `T` has...

- `t = 1` then `payload` is `UInt32`
- `t = 2` then `payload` is `Bool`
- `t = 3` then `payload` is `&String`

(3) Implementation

Storing Sets of Integers

Continuous Ranges: includes every integer between L and H.

Storing Sets of Integers

Continuous Ranges: includes every integer between L and H.

Small Sets: includes any set of integers between O and $O + 8192$.

Storing Sets of Integers

Continuous Ranges: includes every integer between L and H.

Small Sets: includes any set of integers between O and $O + 8192$.

Stripe Sets: A collection of *Step Ranges*

Step Range: includes every N^{th} integer between L and H

Small Set

Example

$O = 500$	0	0	0	0	0	0	1	0	0	1	...	0
$O +$	0	1	2	3	4	5	6	7	8	9	...	8191

This set includes 506, 509.

Small Set

Example

$O = 500$

	0	0	0	0	0	0	1	0	0	1	...	0
$O+$	0	1	2	3	4	5	6	7	8	9	...	8191

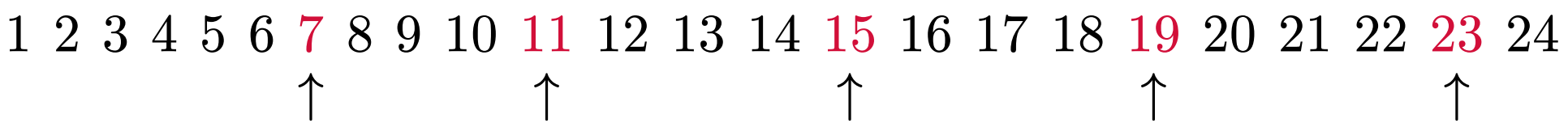
This set includes 506, 509.

Nice Properties

1. Fast set operations (subset, set subtract, union, etc)
2. Reasonably fast element-wise operations.

Step Range

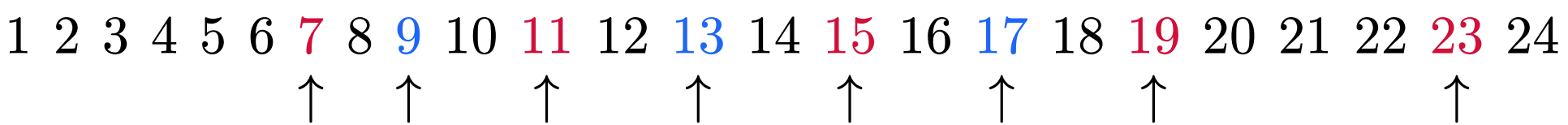
Set: $L = 7$, and $H = 23$, with step 4.



Stripe Sets

Ranges:

- $L = 7$, and $H = 23$, with step 4.
- $L = 9$, and $H = 17$, with step 2.



Nice Properties

1. can represent arbitrary sets without too much memory
2. Addition and subtraction are efficient (mostly).

Problems with Stripe Sets

- Bit-wise operations are slow.
- Stripe sets get fragmented.
- Stripe sets are used too often when other options are better

Type Checking

```
let x: 0..10 = /* ... */
let y: 0..10 = /* ... */
let z: 0..10 = /* ... */
```

Type Checking

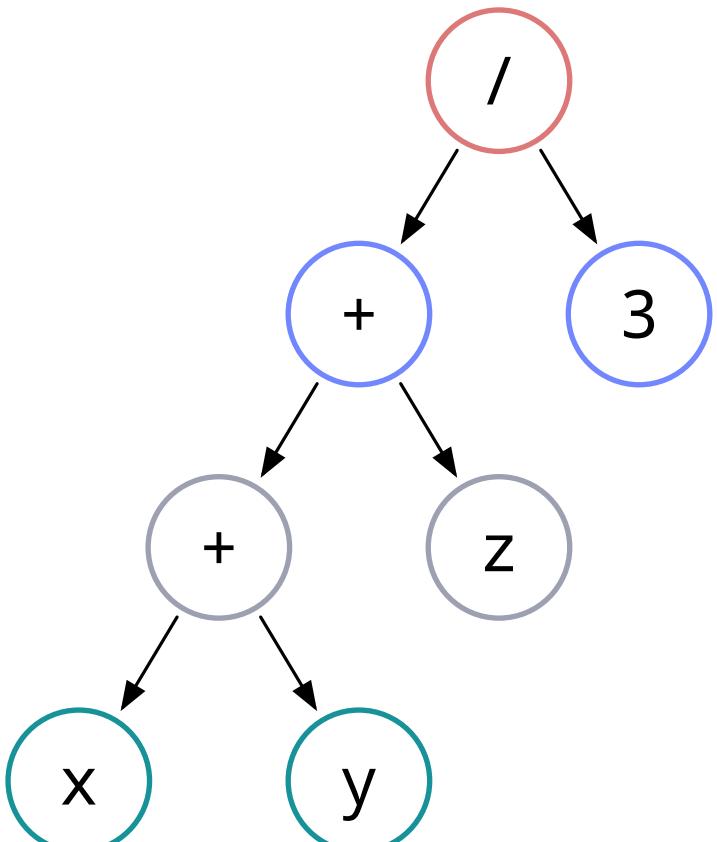
```
let x: 0..10 = /* ... */
let y: 0..10 = /* ... */
let z: 0..10 = /* ... */


(x + y + z) / 3
```

Type Checking

```
let x: 0..10 = /* ... */
let y: 0..10 = /* ... */
let z: 0..10 = /* ... */
```

```
(x + y + z) / 3
```

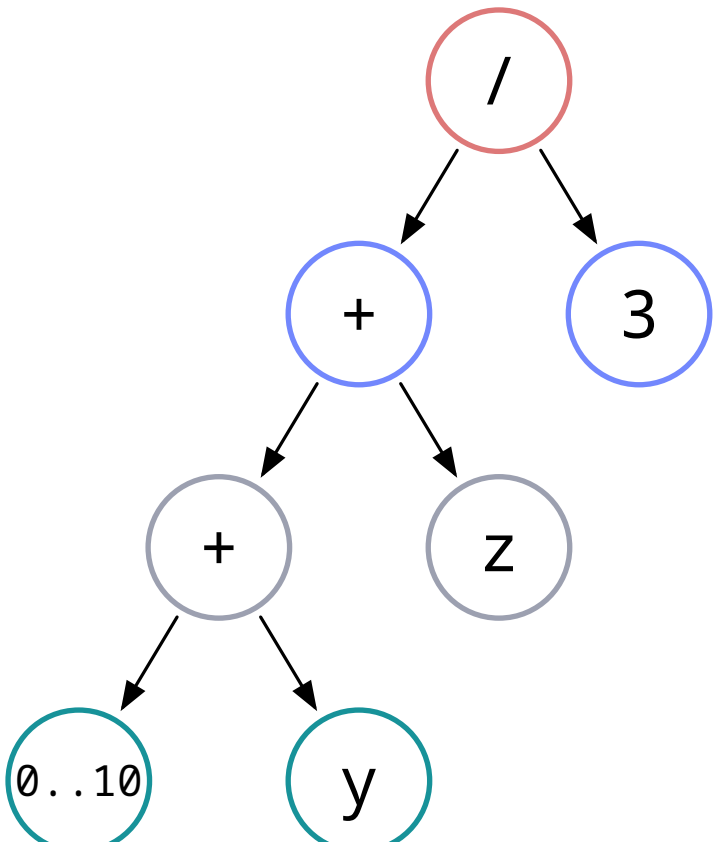
parses to... →



Type Checking

```
let x: 0..10 = /* ... */
let y: 0..10 = /* ... */
let z: 0..10 = /* ... */

(x + y + z) / 3
```

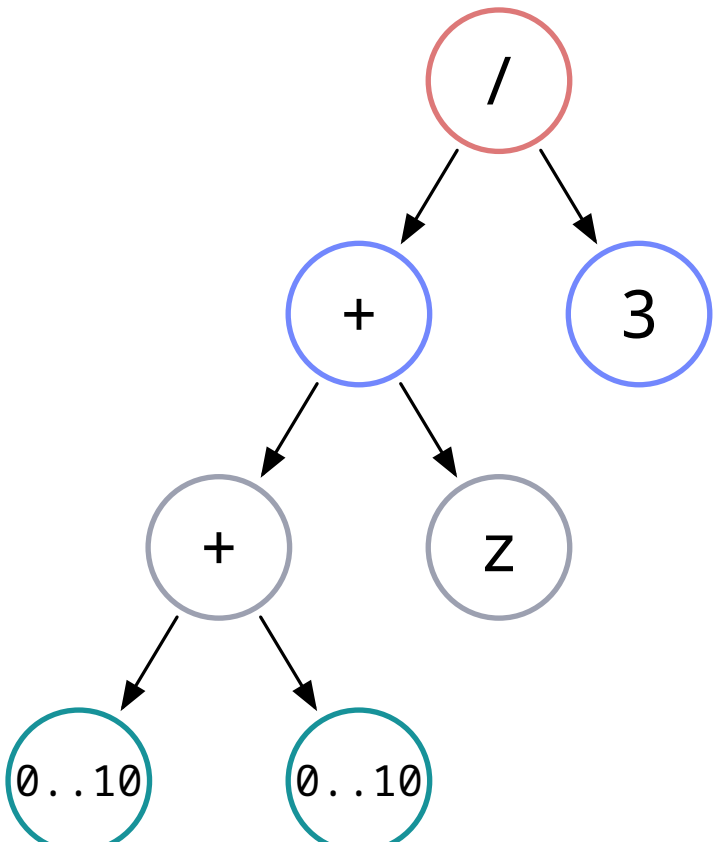
Begin with the bottom leaves



Type Checking

```
let x: 0..10 = /* ... */
let y: 0..10 = /* ... */
let z: 0..10 = /* ... */

(x + y + z) / 3
```

Begin with the bottom leaves

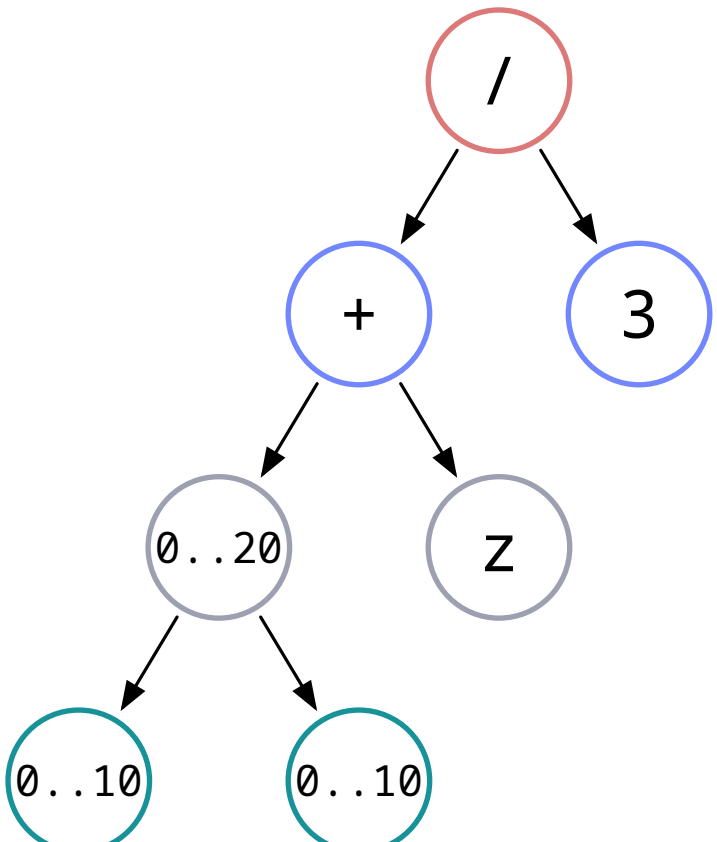


Type Checking

```
let x: 0..10 = /* ... */
let y: 0..10 = /* ... */
let z: 0..10 = /* ... */
```

```
(x + y + z) / 3
```

Perform operation based on child types

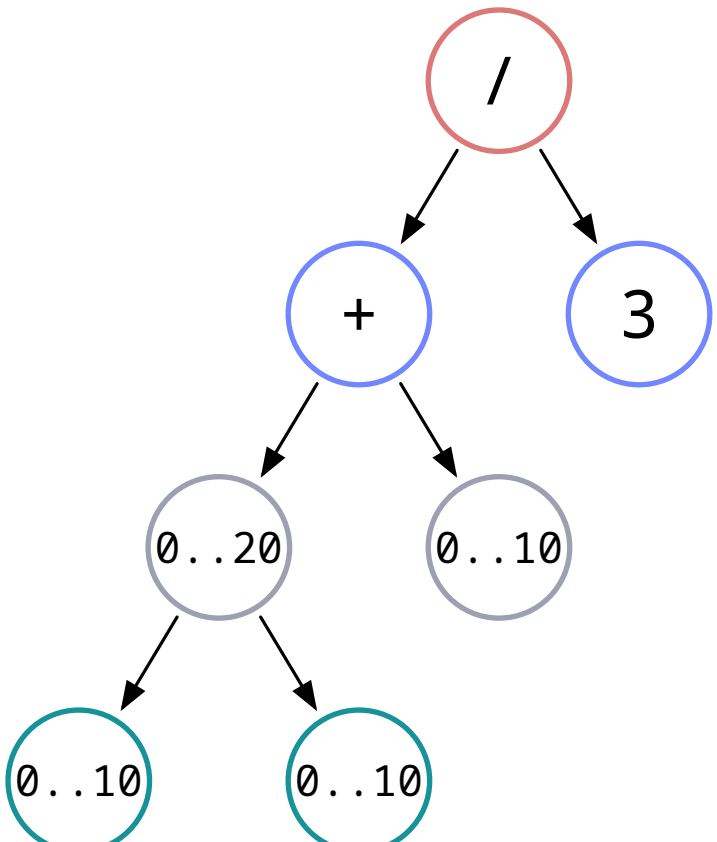


Type Checking

```
let x: 0..10 = /* ... */
let y: 0..10 = /* ... */
let z: 0..10 = /* ... */
```

```
(x + y + z) / 3
```

Move up a level,
get leaf type

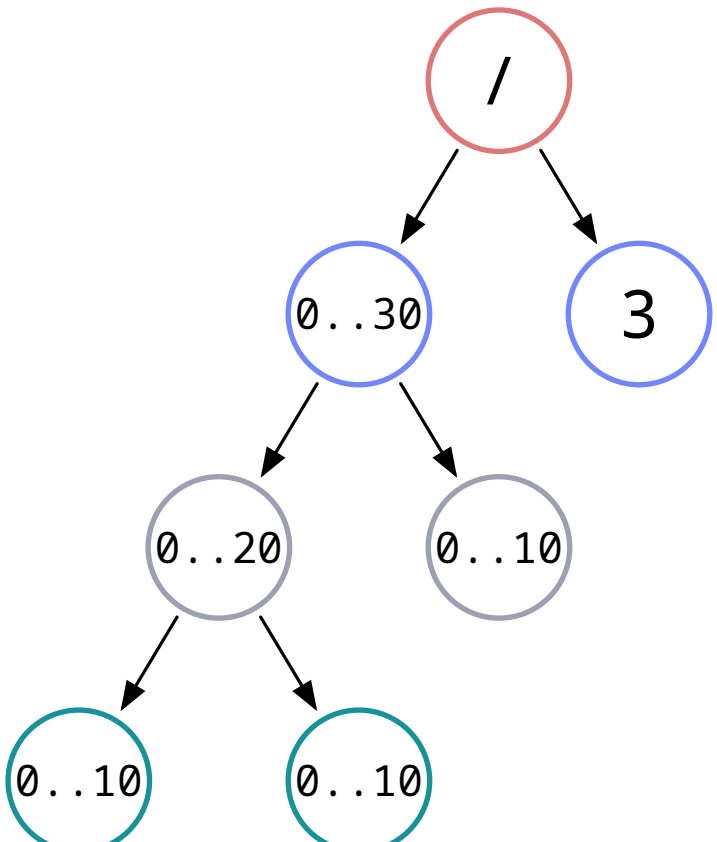


Type Checking

```
let x: 0..10 = /* ... */
let y: 0..10 = /* ... */
let z: 0..10 = /* ... */
```

```
(x + y + z) / 3
```

Perform operation based on child types

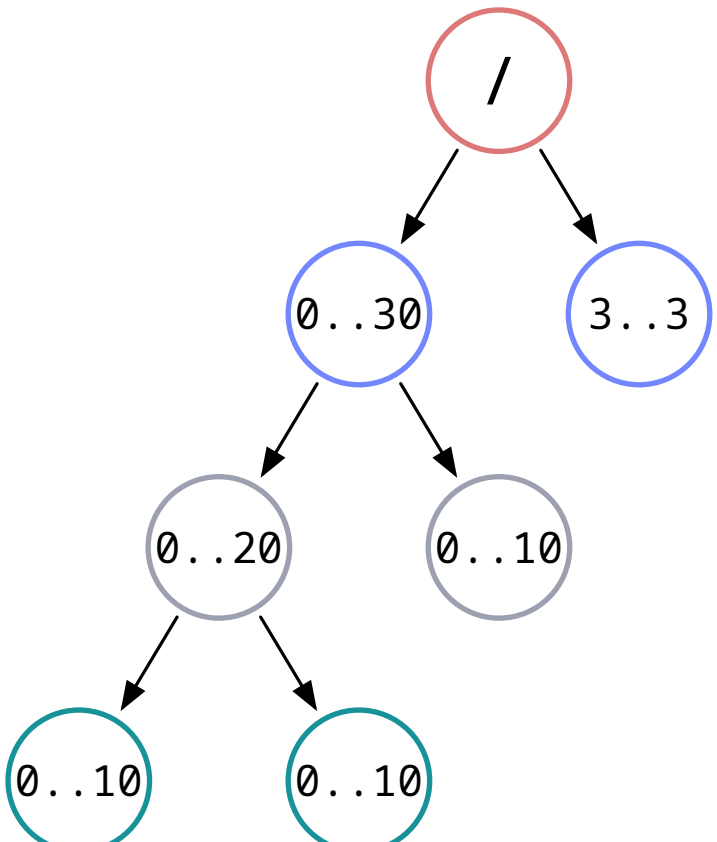


Type Checking

```
let x: 0..10 = /* ... */
let y: 0..10 = /* ... */
let z: 0..10 = /* ... */
```

```
(x + y + z) / 3
```

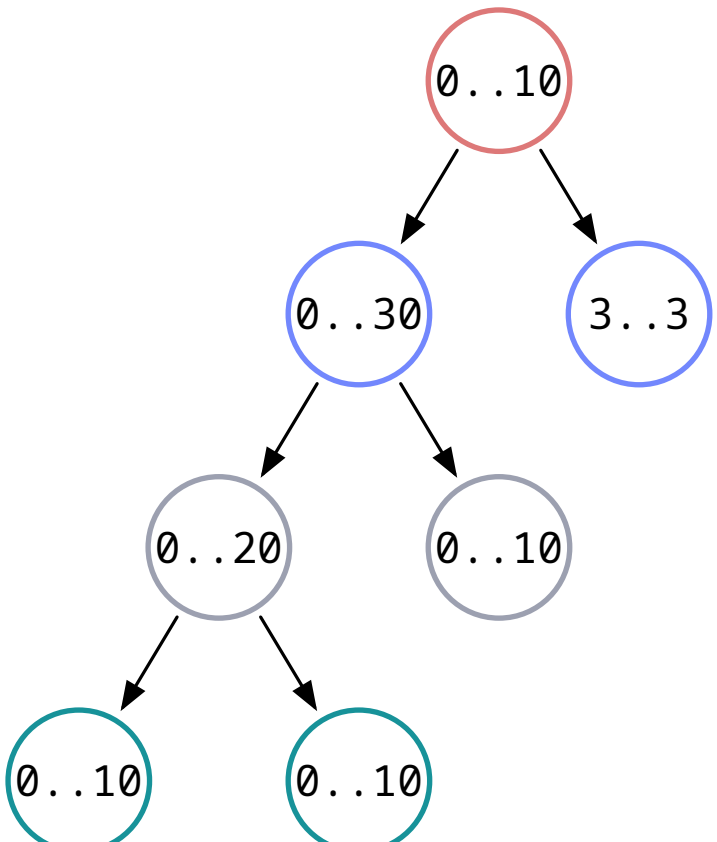
Move up a level,
get leaf type



Type Checking

```
let x: 0..10 = /* ... */
let y: 0..10 = /* ... */
let z: 0..10 = /* ... */

(x + y + z) / 3
```

Calculate type of the entire expression



Type Narrowing

```
if x + 2 < y { ... }
```

Type Narrowing - Producing Linear Sums

```
if x + 2 < y { ... }
```

1. Try to build a linear out of each side of the condition

Type Narrowing - Producing Linear Sums

```
if x + 2 < y { ... }
```

1. Try to build a linear out of each side of the condition

   So we have $x + 2$ on the left and y on the right.

Type Narrowing - Producing Linear Sums

```
if x + 2 < y { ... }
```

1. Try to build a linear out of each side of the condition

   So we have $x + 2$ on the left and y on the right.

2. Convert into a simple inequality or equation

   The above ineq. is converted into $x - y + 2 < 0$

Type Narrowing - Determining Implications

3. Convert the type of x & y into constraints

What we know:

- $x - y + 2 < 0$

Type Narrowing - Determining Implications

3. Convert the type of x & y into constraints

   Suppose `x: 0..10`
   and `y: 2 | 4 | 6 | 8`

What we know:

- $x - y + 2 < 0$

Type Narrowing - Determining Implications

3. Convert the type of x & y into constraints

Suppose `x: 0..10`
and `y: 2 | 4 | 6 | 8`

What we know:

- $x - y + 2 < 0$
- $0 \leq x \leq 10$
- $y \leq 8$
- $y = 2n_1 + 2$

Type Narrowing - Determining Implications

3. Convert the type of x & y into constraints

   Suppose `x: 0..10`
   and `y: 2 | 4 | 6 | 8`

4. Repeatedly solve constraints with *aries*

What we know:

- $x - y + 2 < 0$
- $0 \leq x \leq 10$
- $y \leq 8$
- $y = 2n_1 + 2$

Type Narrowing - Determining Implications

3. Convert the type of x & y into constraints

   Suppose `x: 0..10`
   and `y: 2 | 4 | 6 | 8`

4. Repeatedly solve constraints with *aries*

What we know:

- $x - y + 2 < 0$
- $0 \leq x \leq 10$
- $y \leq 8$
- $y = 2n_1 + 2$

Type Narrowing - Determining Implications

3. Convert the type of x & y into constraints

   Suppose `x: 0..10`
   and `y: 2 | 4 | 6 | 8`

4. Repeatedly solve constraints with *aries*
5. Reassign variables

```
if x + 2 < y {
  // y : 4 | 6 | 8 and x : 0..5
```

What we know:

- $x - y + 2 < 0$
- $0 \leq x \leq 10$
- $y \leq 8$
- $y = 2n_1 + 2$
- $y = 2n_2 + 4$
- $x \leq 5$

Type Narrowing - Determining Implications

3. Convert the type of x & y into constraints

   Suppose `x: 0..10`
   and `y: 2 | 4 | 6 | 8`

4. Repeatedly solve constraints with *aries*
5. Reassign variables

```
if x + 2 < y {
  // y : 4 | 6 | 8 and x : 0..5
```

6. Process consequences

What we know:

- $x - y + 2 < 0$
- $0 \leq x \leq 10$
- $y \leq 8$
- $y = 2n_1 + 2$
- $y = 2n_2 + 4$
- $x \leq 5$

What can we Learn?

- Range types are likely practical
- Sets of discrete integers are not
- We need a different representation for bit operations

Questions?