Constrained Integers and Structural Types

Outline

1. Programming Languages
2. Types
3. Internals

Language Design

- Programming languages are tools for expressing computation
- Key ingrediants in computation: *State* and *Transitions*

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
```

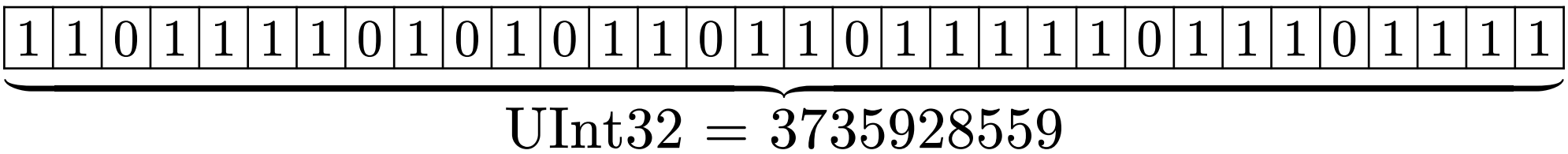
Programming Languages (Go)

```
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
}
```

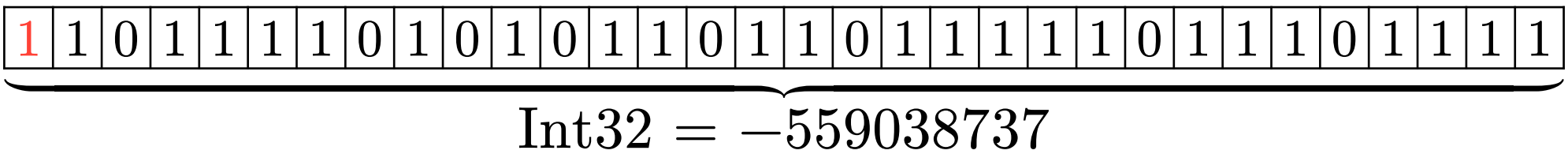
Types at a Low Level



Key Question about Integers:

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

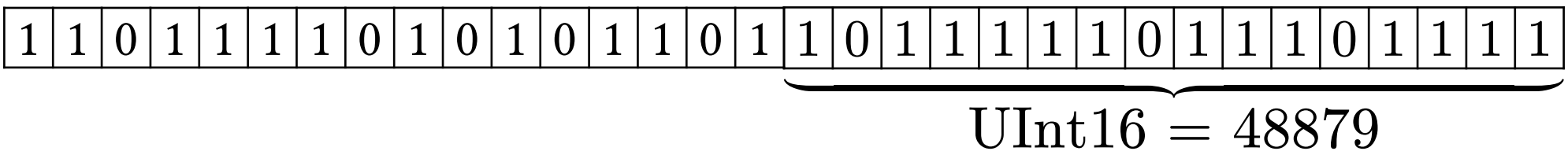
Types at a Low Level



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

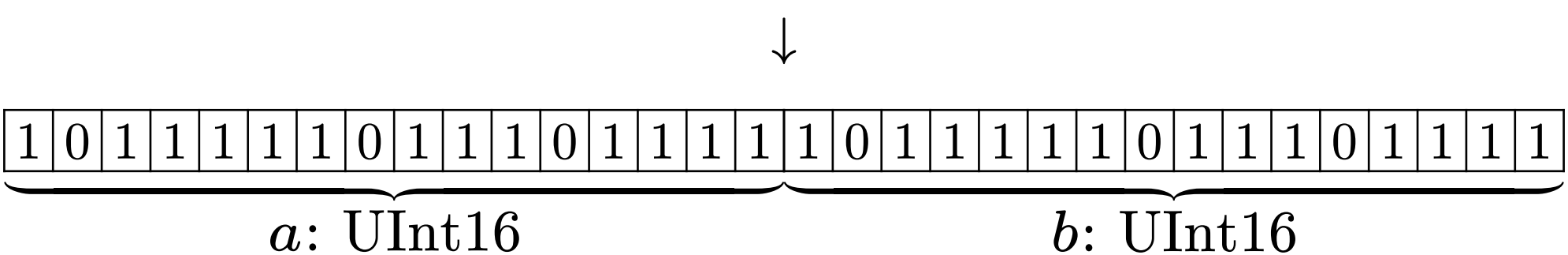
```
type Pair = {
  a: UInt16,
  b: UInt16
}
```

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

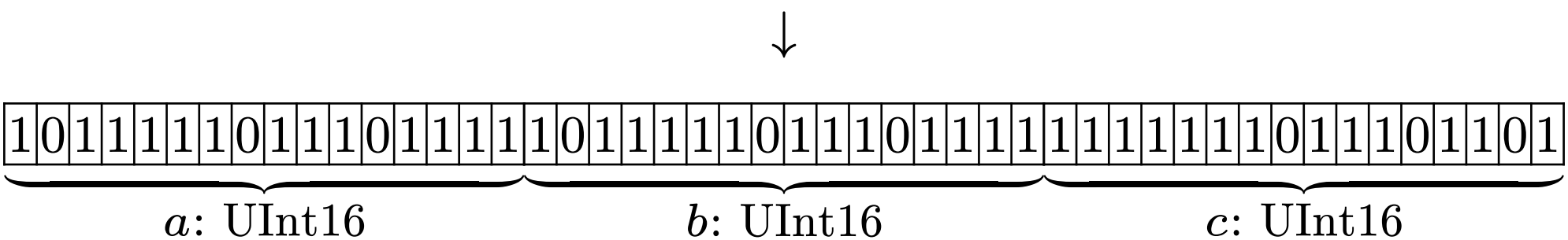
```
type Triple = {
  a: UInt16,
  b: UInt16,
  c: UInt16
}
```

Composition

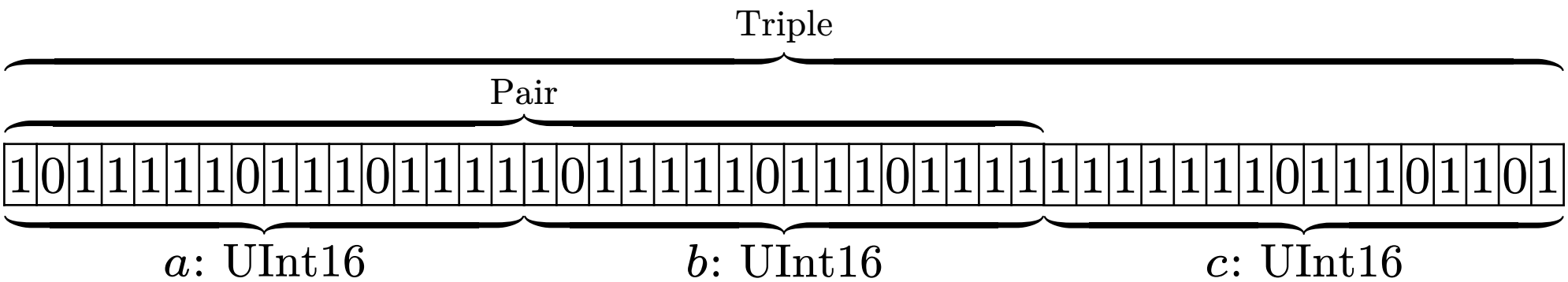
```
type Triple = {
  a: UInt16,
  b: UInt16,
  c: UInt16
}

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)

Compound 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)

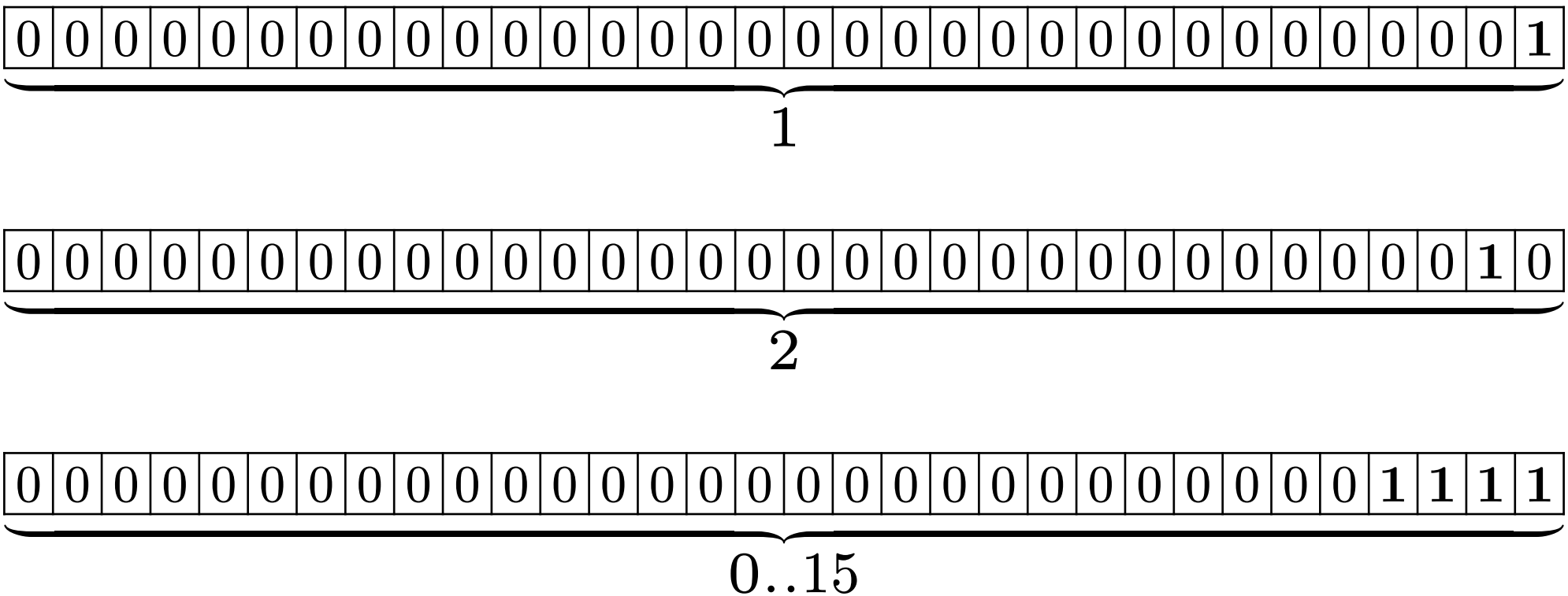
Compound Types: A sequence of *named* fields, each with its own *type*

Static Types for Single States

"This is a 32-bit integer"

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

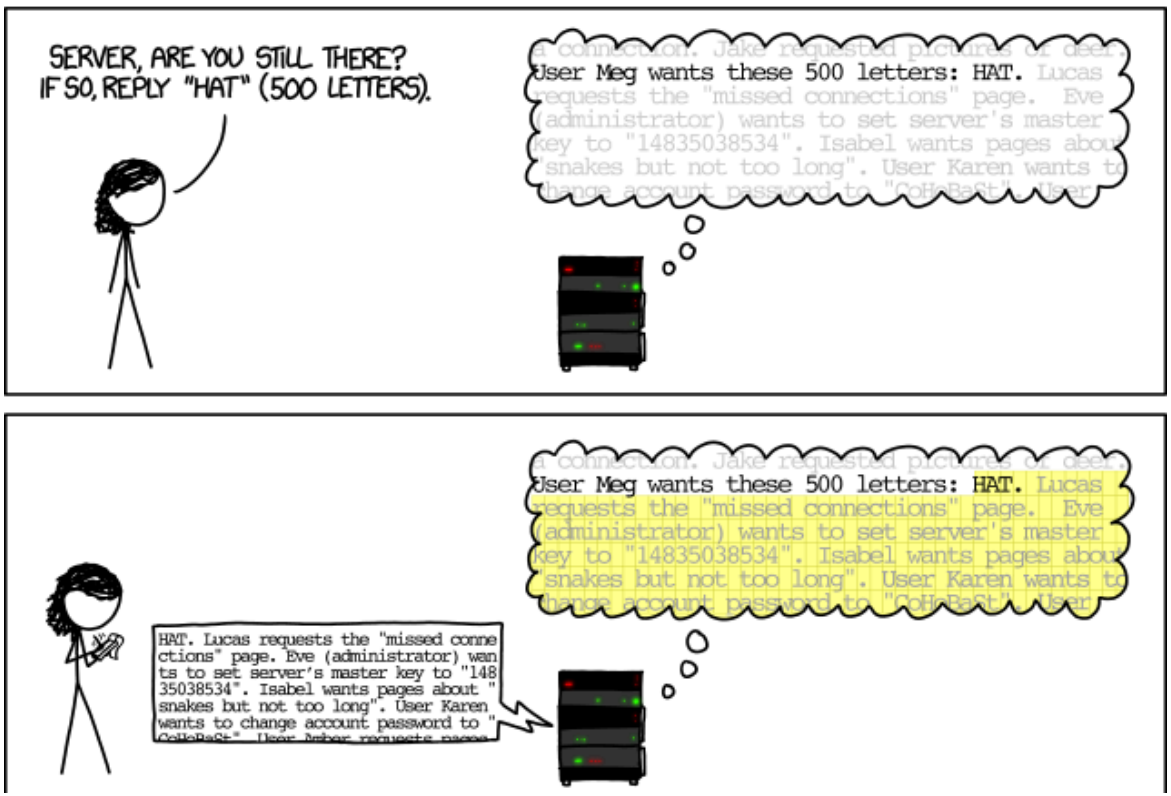
Only Using a Few Bits



Intermission

Why do this?

1. **Static bounds checks**



Intermission

Why do this?

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.
```

Intermission

Why do this?

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
```

Intermission

Why do this?

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

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

Intermission

Why do this?

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
}
```

Intermission

Why do this?

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`