Type checking Swift, in reasonable time

Slava Pestov

Agenda

- How does the type checker work?
- Improvements in Swift 6.3
- Improvements in main branch

First example

Swift 6.2

```swift
func test() {
    let x = (0 * 0) + (0 * 0) + (0 * 0) + (0 * 0) + (0 * 0)
    print(x)
}
```

- Swift 6.2:
  - ~150 milliseconds total (invocation overhead, codegen, optimizer, linker, ...)
  - ~1 millisecond to type check ✅

First example

Swift 6.2

```swift
func test() {
    let x: Int = (0 * 0) + (0 * 0) + (0 * 0) + (0 * 0) + (0 * 0)
    print(x)
}
```

- Swift 6.2: ~1 millisecond to type check ✅

First example

Swift 6.2

```
func test() {
    let x: UInt = (0 * 0) + (0 * 0) + (0 * 0) + (0 * 0) + (0 * 0)
    print(x)
}
```

- Swift 6.2: the compiler is unable to type-check this expression in reasonable time; try breaking up the expression into distinct sub-expressions ❌

Why is Swift type checking hard?

Expressivity + **static type safety** =

more work at **compile time**

Why is Swift type checking hard?

- Overloading
  - Operators: `x + y` can add two `Int`s, or two `Double`s, etc
  - Overloaded functions more generally
  - (**Note:** overloading on argument label or arity is easy)
- Literals
  - `0` might be an `Int`, `Int8`, `UInt32`, `Double`, ...

Why is Swift type checking hard?

- Closures with inferred types

  -
    ```
    func foo(_: Int) {}
    { x in foo(x) }
    ```
- Implicit conversions

  -
    ```
    protocol Shape {...}
    struct Rectangle: Shape {...}
    func draw(_: any Shape) {...}

    draw(Rectangle(...))
    ```

How do we type check Swift?

- One expression at a time (except for multi-statement closures)
- **Type variables** and **constraints**
- We **create a new type variable** for each sub-expression with unknown type
- We **generate constraints** from the syntactic structure of the expression
- We **search** for an assignment of fixed concrete types to type variables that **satisfies** all of the constraints

Why is Swift type checking hard?

- Classic result in computer science: this sort of problem is *NP-hard*
- Informally speaking: in pathological instances, the search might take "unreasonable time"
- Might have to stop and give up

The + operator

- `(Int, Int) -> Int`
- `(UInt, UInt) -> UInt`
- `(Int8, Int8) -> Int8`
  `(UInt8, UInt8) -> UInt8`
- `(Int16, Int16) -> Int16`
  `(UInt16, UInt16) -> UInt16`
- `...`

The + operator

-
  ```
  (Float16, Float16) -> Float16
  (Float, Float) -> Float
  (Double, Double) -> Double
  ```

-
  ```
  (String, String) -> String
  ```

-
  ```
  (Duration, Duration) -> Duration
  ```

-
  ```
  <Element> (Array<Element>, Array<Element>) -> Array<Element>
  ```

- Generic overloads in protocol extensions...

The + operator

+

- Create a new **type variable** to stand in for the type of the overload: $\$T_n$
- Generate a **disjunction constraint** for all possible choices
- $\$T_n$ bind (Int, Int) -> Int OR $\$T_n$ bind (UInt, UInt) -> UInt OR ... 35 more ...

Literals

-
  ```swift
  let x: Float = ...
  let y = x + 0
  ```

-
  ```swift
  let u: UInt32 = ...
  let v = u + 0
  ```

-
  ```swift
  struct MyCustomType: ExpressibleByIntegerLiteral { ... }
  let w: MyCustomType = 0
  ```

Literals

```
0
```

- Create a new **type variable** to stand in for the type of the literal: $\$T_n$
- Generate **conformance constraint** $\$T_n$: ExpressibleByIntegerLiteral
- Attempt to solve remaining constraints, see if they assign a fixed type to $\$T_n$
- If this fails, we assign the **default type** to $\$T_n$ (eg, for 0 it's `Int`)

What is "reasonable time" exactly?

- *Not* real time
- Scope limit: 2^{20} (approximately one million) scopes
  - Roughly: a "scope" ~ "a combination of choices"
  - (`-Xfrontend -solver-scope-threshold=1048576`)
- Memory limit: 512Mb
  - (`-Xfrontend -solver-memory-threshold=536870912`)

Scalability

Swift 6.2

-
  ```
  func test() {
    let _: Int = (0 * 0)  // 8 scopes ✅
    let _: Int = (0 * 0) + (0 * 0)  // 35 scopes ✅
    let _: Int = (0 * 0) + (0 * 0) + (0 * 0)  // 61 scopes ✅
    let _: Int = (0 * 0) + (0 * 0) + (0 * 0) + (0 * 0)  // 87 scopes ✅
    let _: Int = (0 * 0) + (0 * 0) + (0 * 0) + (0 * 0) + (0 * 0)  // 113 scopes ✅
  }
  ```
- If n is the number of + signs, the number of scopes is O(n)
- (`-stats-output-dir` dumps statistics from the compiler)

Scalability

Swift 6.2

-
  ```
  func test() {
    let _: UInt = (0 * 0)  // 23 scopes ✅
    let _: UInt = (0 * 0) + (0 * 0)  // 11767 scopes ✅
    let _: UInt = (0 * 0) + (0 * 0) + (0 * 0)  // 858277 scopes ✅
    let _: UInt = (0 * 0) + (0 * 0) + (0 * 0) + (0 * 0)  // >1048576 scopes ❌
    let _: UInt = (0 * 0) + (0 * 0) + (0 * 0) + (0 * 0) + (0 * 0)  // >1048576 scopes ❌
  }
  ```
- $O(k^n)$ where k is a little bit less than the number of overloads

First example, again

Swift 6.3 developer snapshot from swift.org

-
  ```
  func test() {
    let x: UInt = (0 * 0) + (0 * 0) + (0 * 0) + (0 * 0) + (0 * 0)
    print(x)
  }
  ```
- Swift 6.3: ~1 millisecond to type check ✅

Scalability

Swift 6.3 developer snapshot from swift.org

-
  ```
  func test() {
    let _: UInt = (0 * 0)  // 8 scopes ✅
    let _: UInt = (0 * 0) + (0 * 0)  // 20 scopes ✅
    let _: UInt = (0 * 0) + (0 * 0) + (0 * 0)  // 32 scopes ✅
    let _: UInt = (0 * 0) + (0 * 0) + (0 * 0) + (0 * 0)  // 44 scopes ✅
    let _: UInt = (0 * 0) + (0 * 0) + (0 * 0) + (0 * 0) + (0 * 0)  // 56 scopes ✅
  }
  ```

- O(n)

Three questions

- Why was the `Int` version fast in Swift 6.2?
    - Pre-processing pass to "shrink" the constraint system before solving
- Why is the `UInt` version slow in Swift 6.2?
    - This pre-processing was insufficiently general
- Why are both fast in Swift 6.3?
    - New **disjunction selection algorithm**

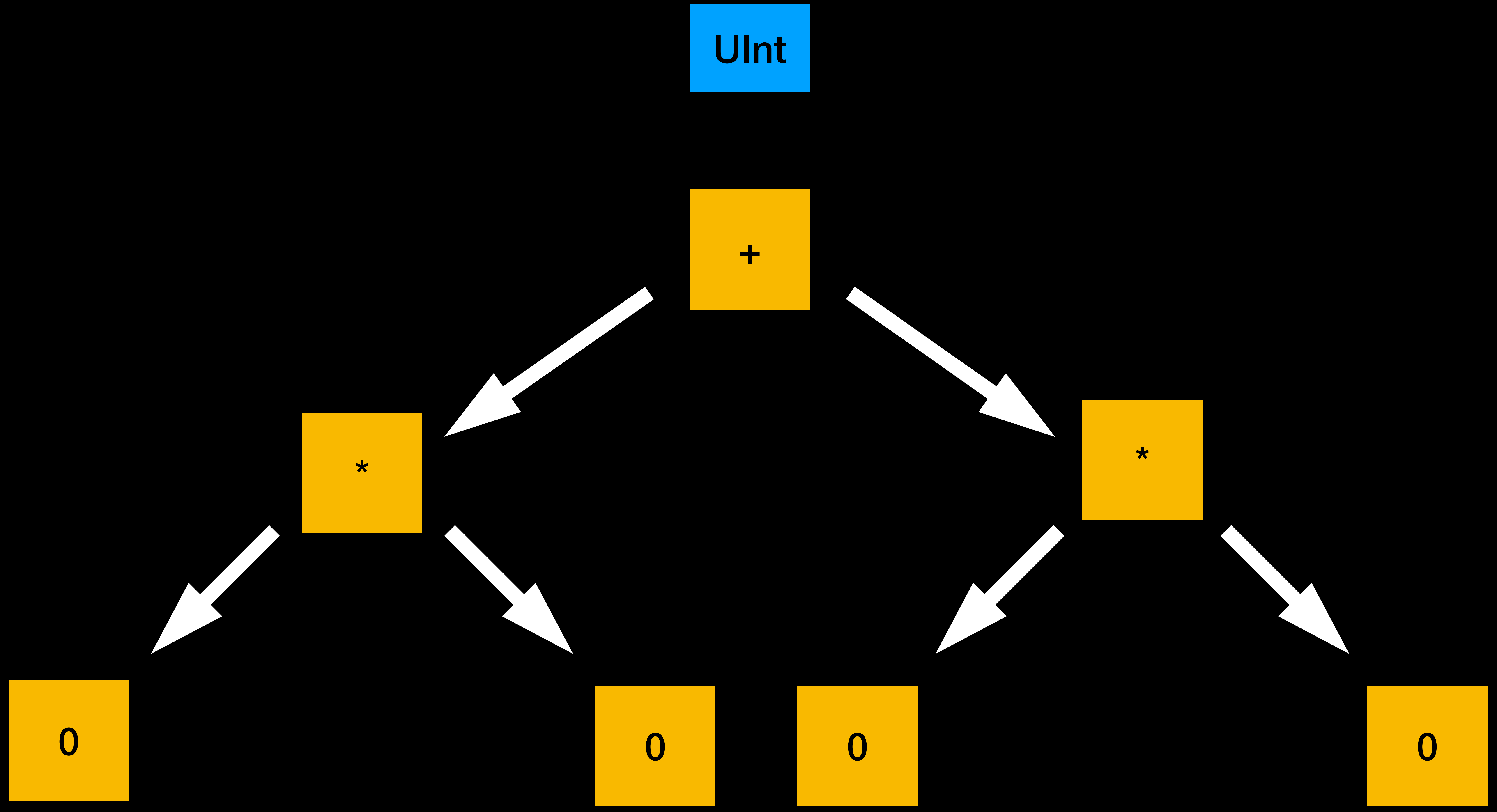
Simplifying assumptions

- Let's simplify the expression a bit:
  - `let x: UInt = (0 * 0) + (0 * 0)`
- Let's pretend we only have two overloads of +:
  - `(Int, Int) -> Int`
  - `(UInt, UInt) -> UInt`

Disjunction selection: "bad" order

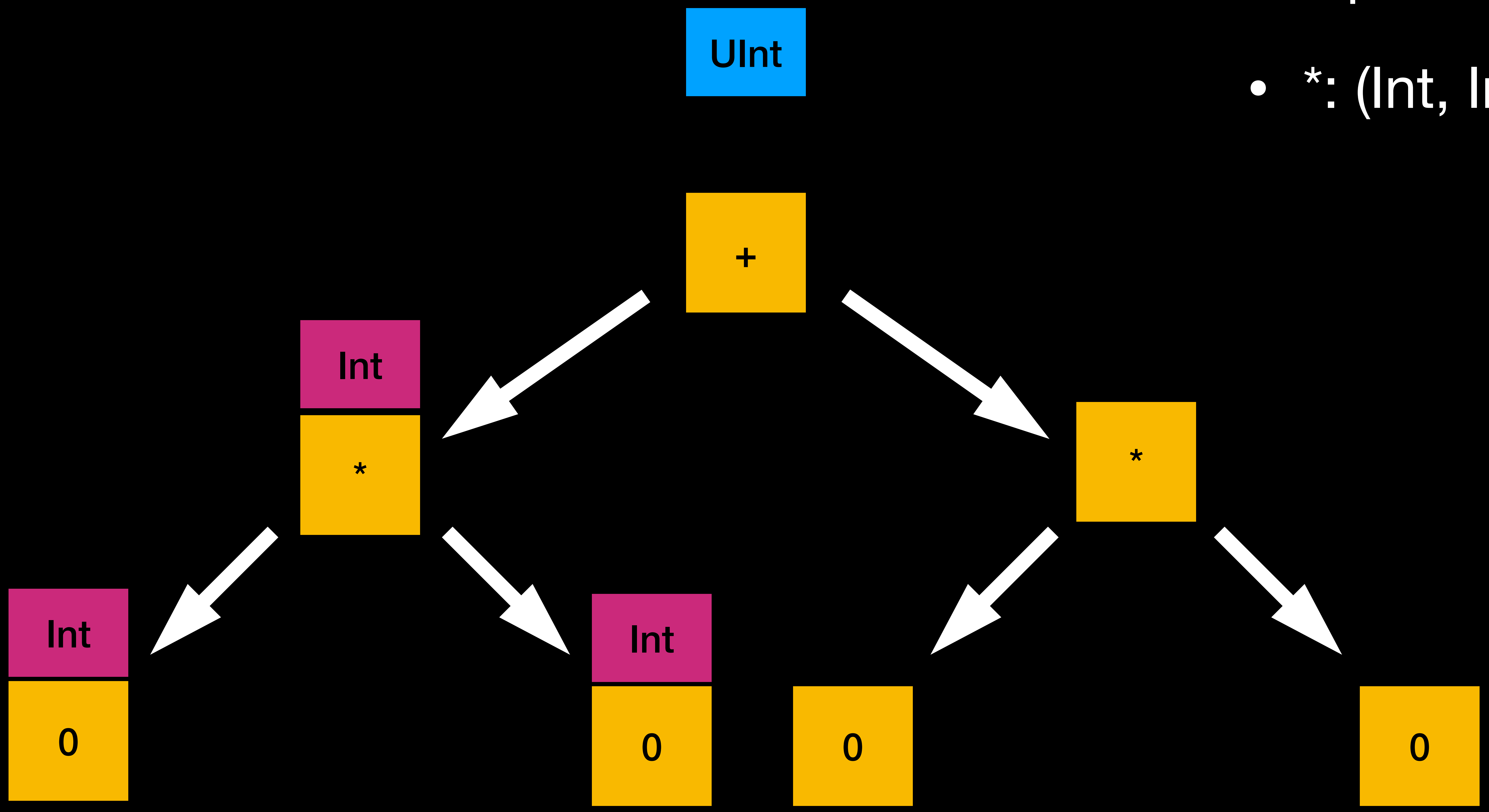
Scopes:

0



Disjunction selection: "bad" order

1

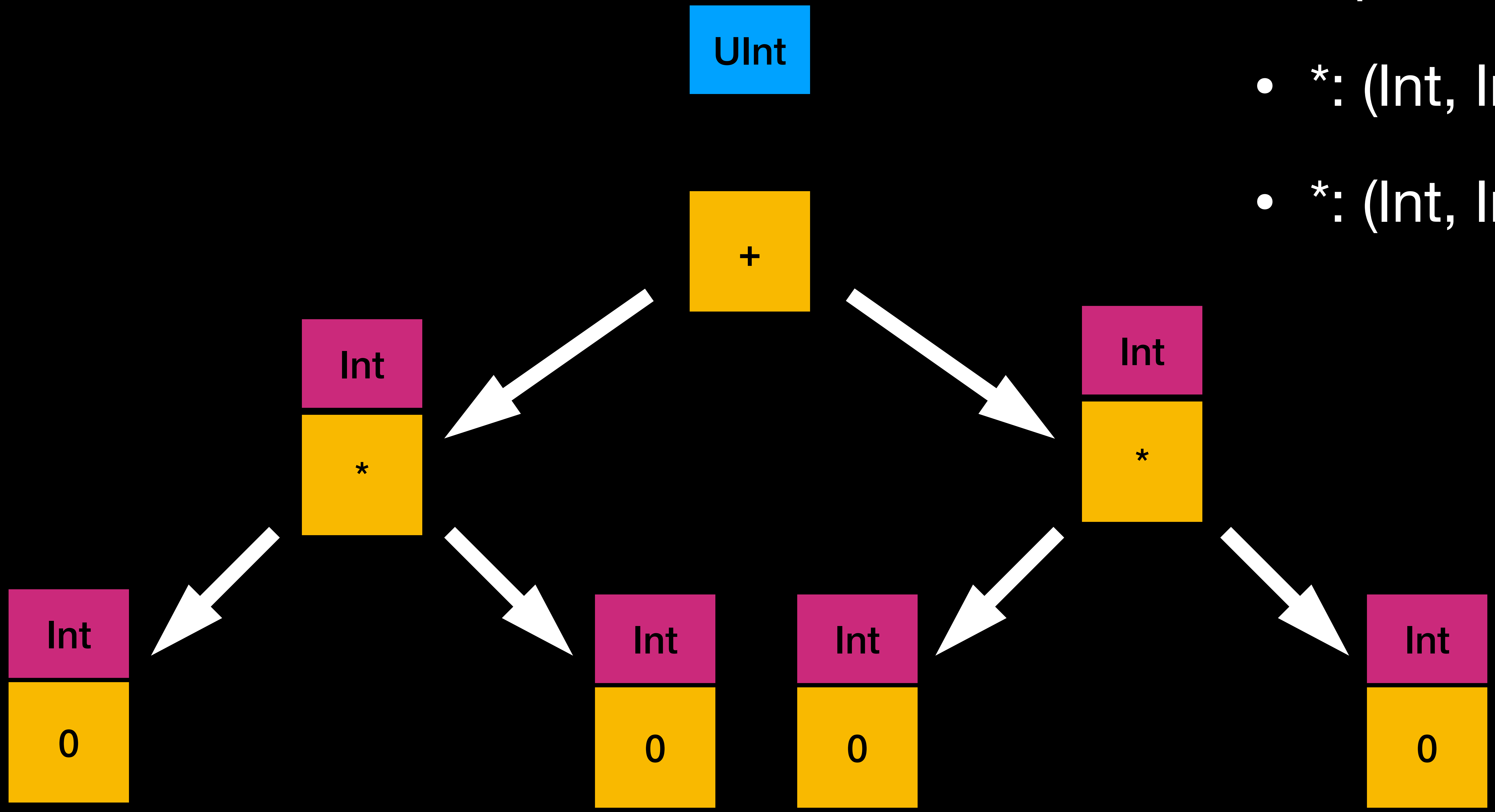


Scopes:

- *: (Int, Int) -> Int

Disjunction selection: "bad" order

2

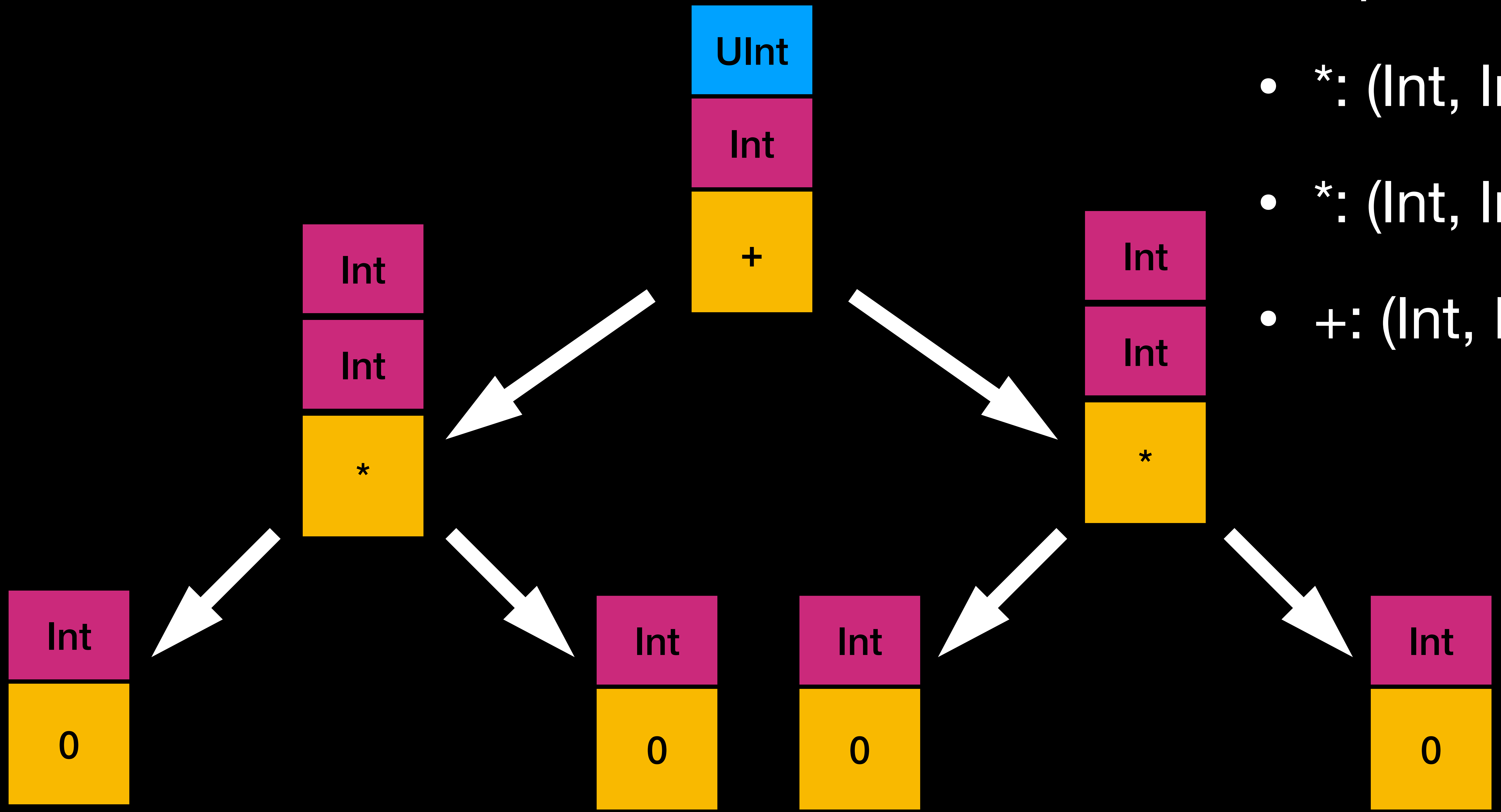


Scopes:

- *: (Int, Int) -> Int
- *: (Int, Int) -> Int

Disjunction selection: "bad" order

3

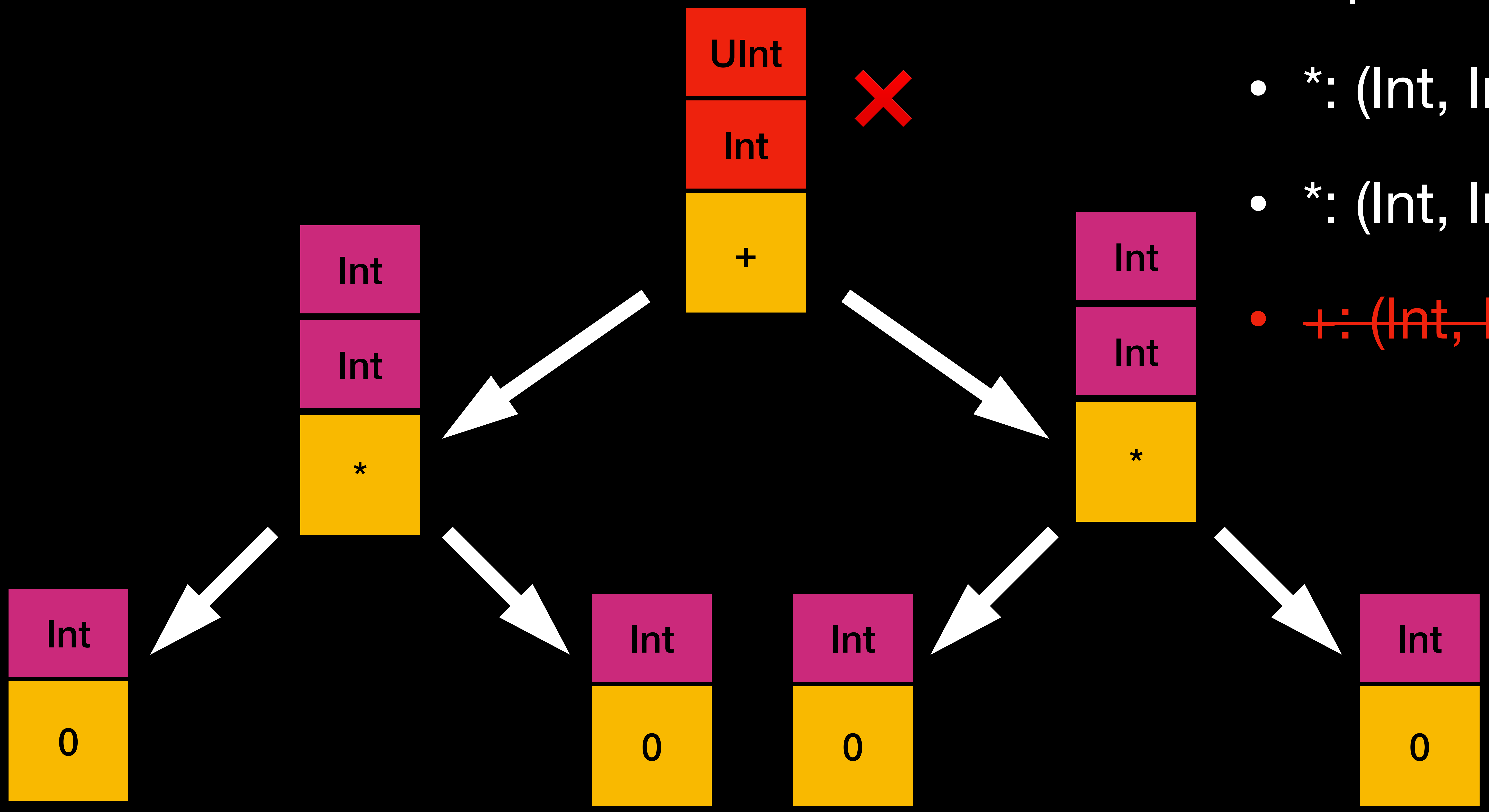


Scopes:

- *: (Int, Int) -> Int
- *: (Int, Int) -> Int
- +: (Int, Int) -> Int

Disjunction selection: "bad" order

3

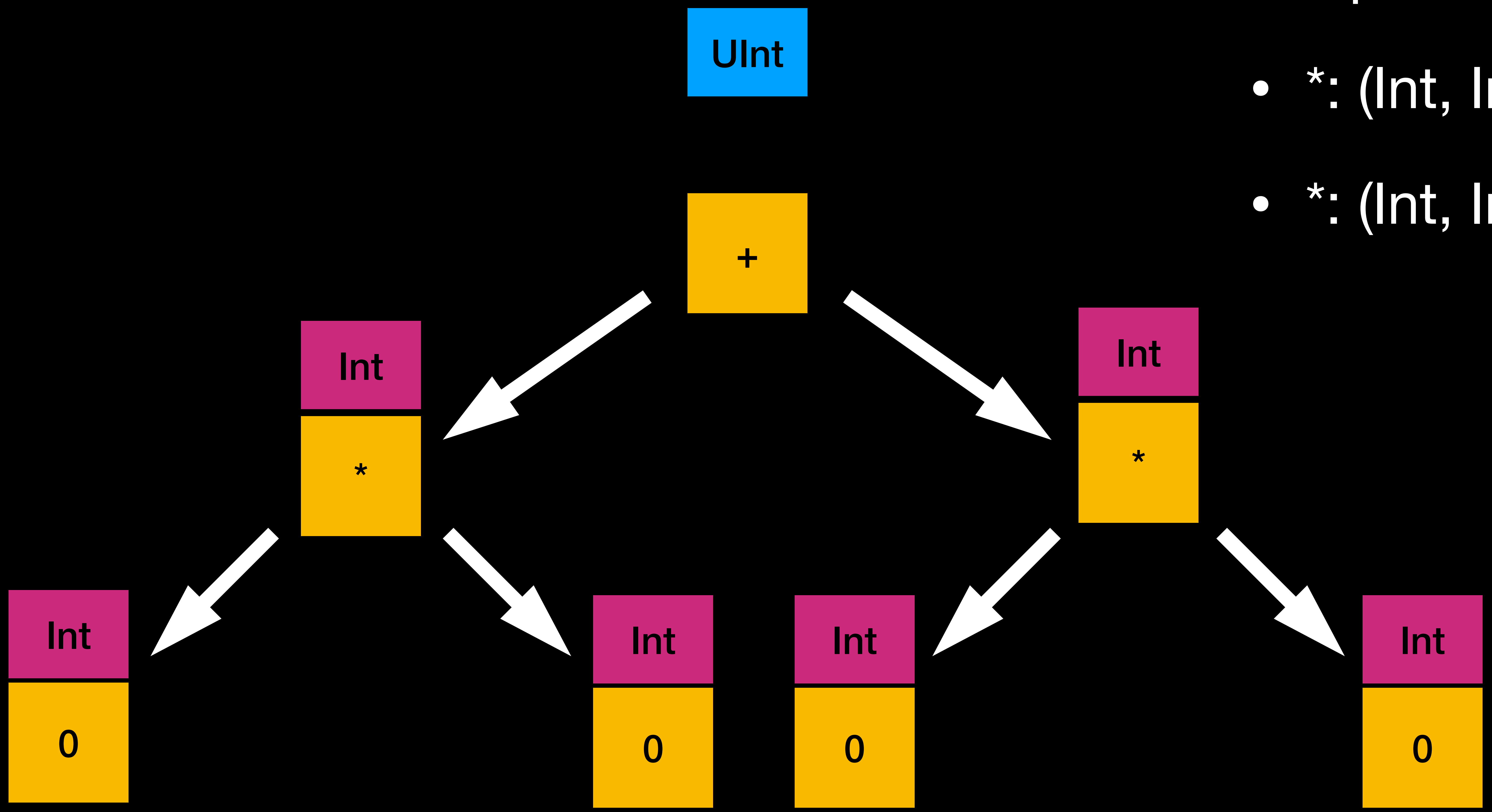


Scopes:

- *: (Int, Int) -> Int
- *: (Int, Int) -> Int
- ~~+: (Int, Int) -> Int~~

Disjunction selection: "bad" order

3

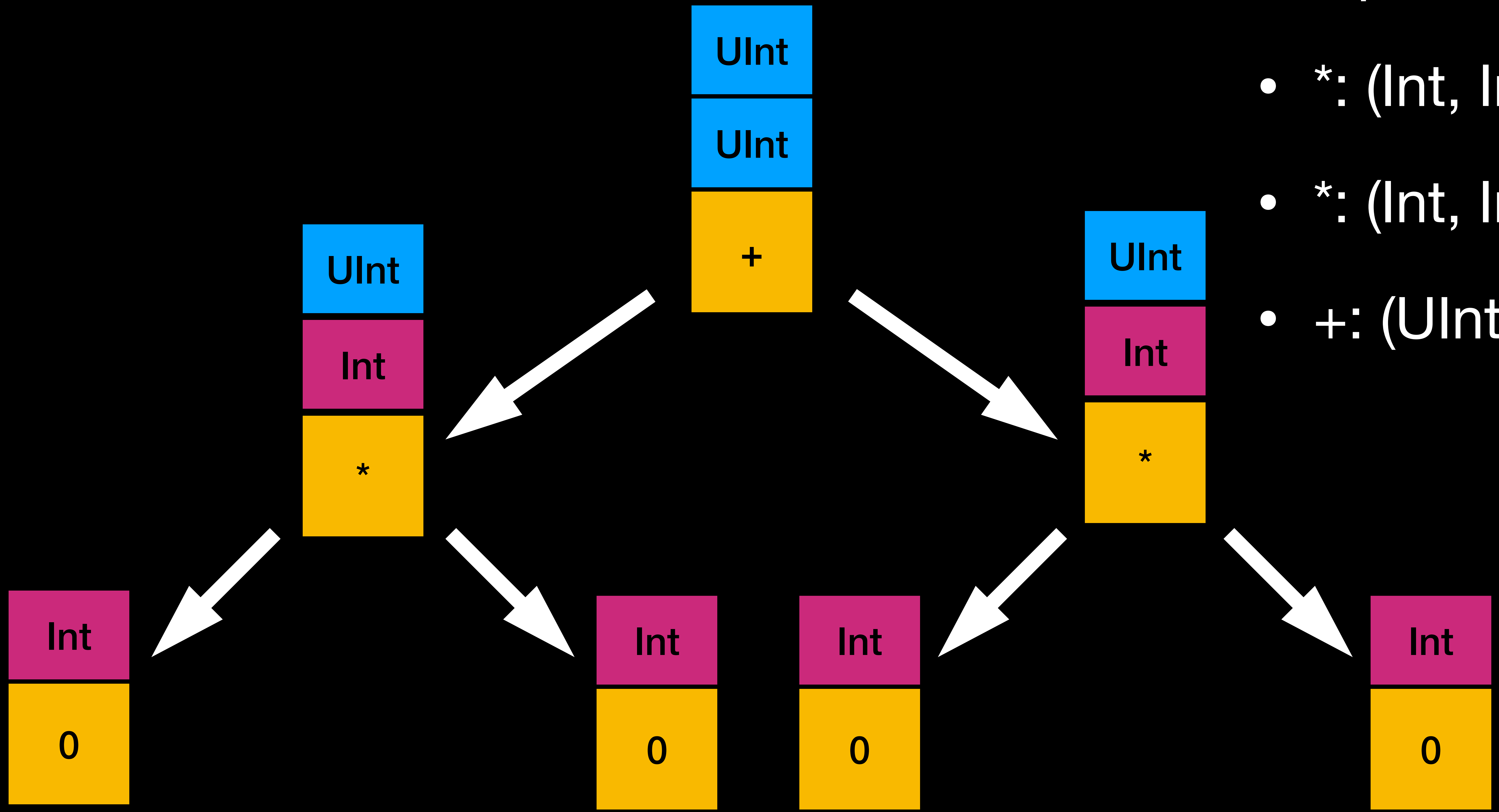


Scopes:

- *: (Int, Int) -> Int
- *: (Int, Int) -> Int

Disjunction selection: "bad" order

4

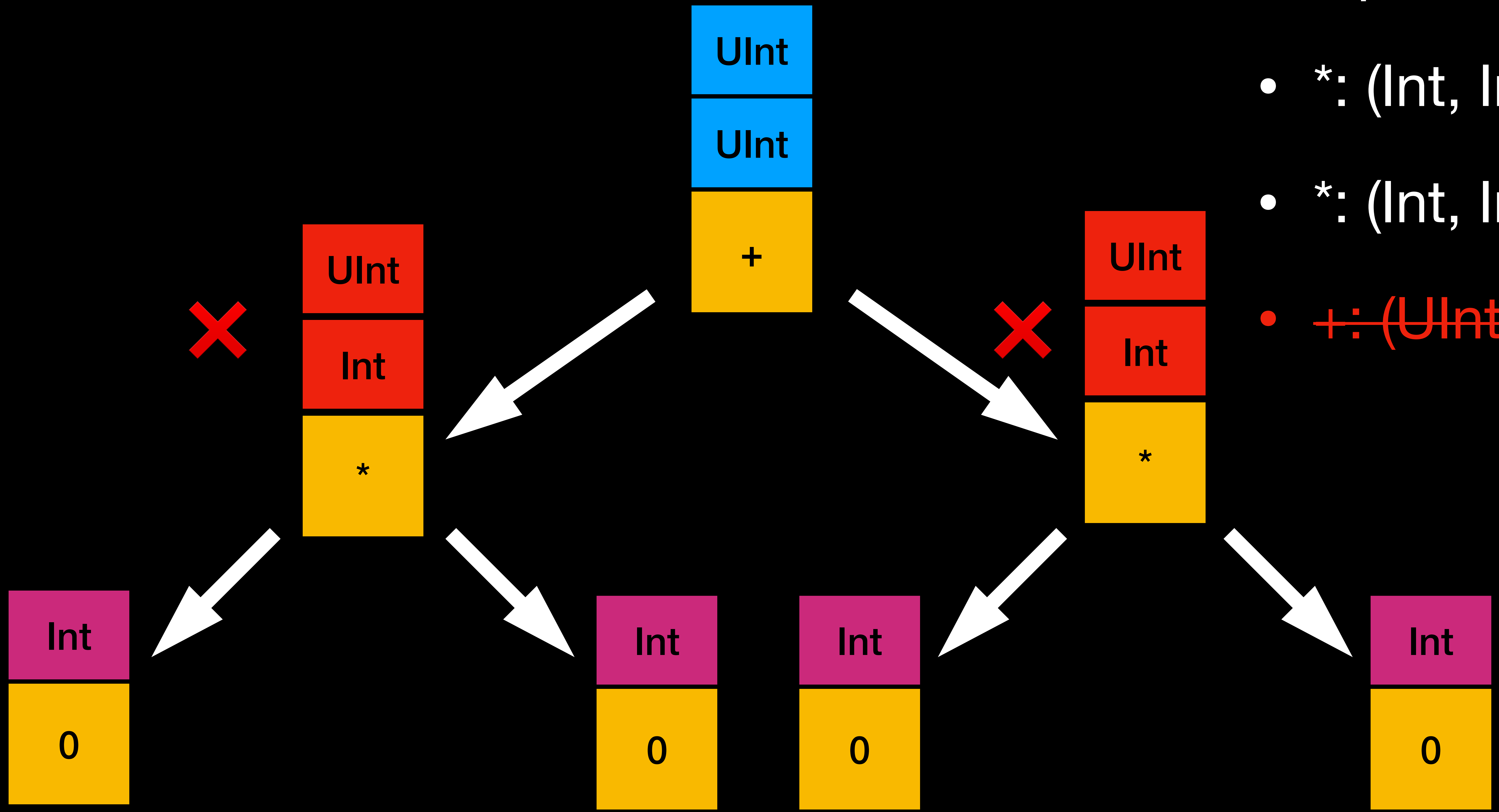


Scopes:

- *: (Int, Int) -> Int
- *: (Int, Int) -> Int
- +: (UInt, UInt) -> UInt

Disjunction selection: "bad" order

4

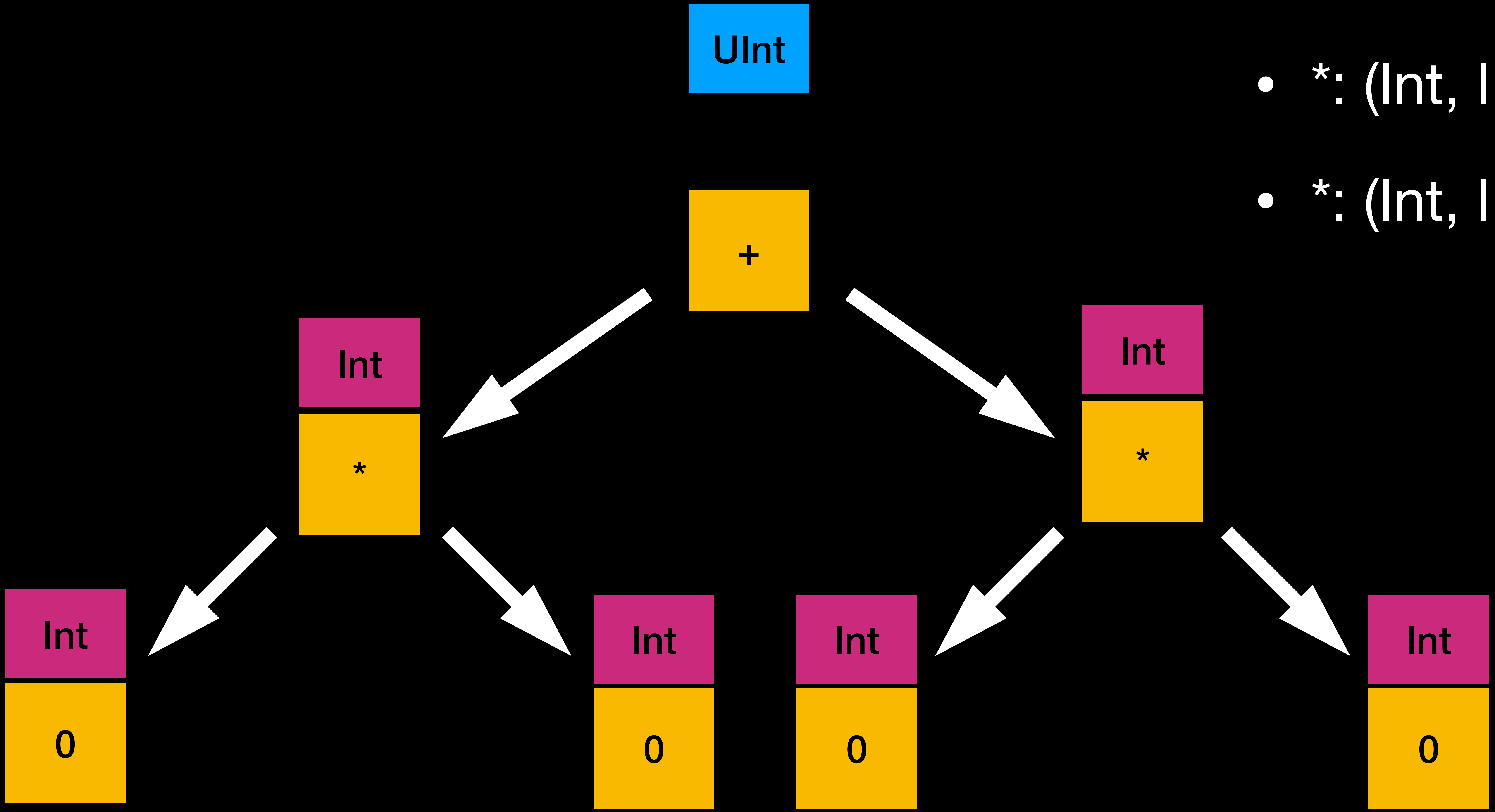


Scopes:

- *: (Int, Int) -> Int
- *: (Int, Int) -> Int
- ~~+: (UInt, UInt) -> UInt~~

Disjunction selection: "bad" order

4

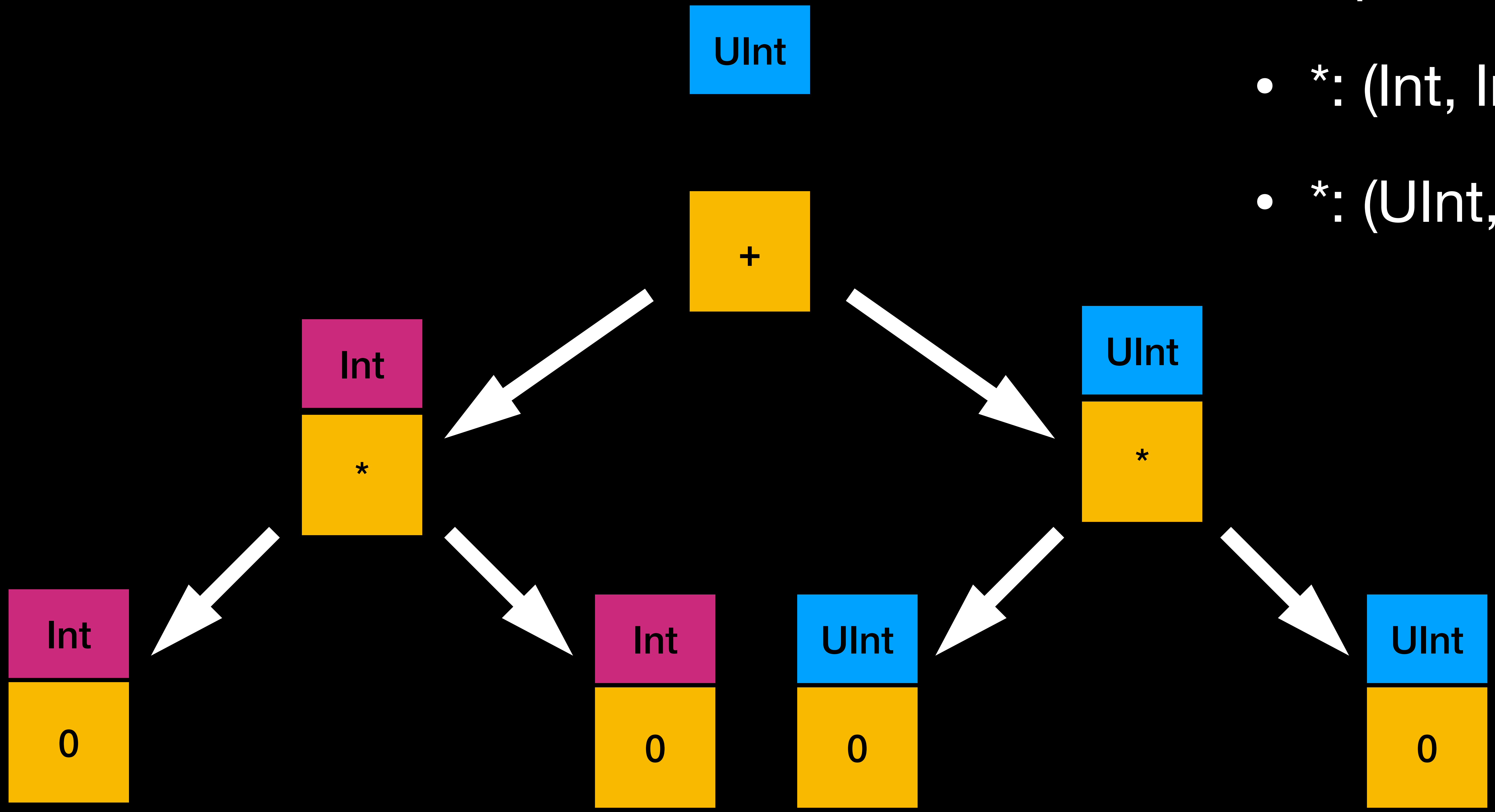


Scopes:

- *: (Int, Int) -> Int
- *: (Int, Int) -> Int

Disjunction selection: "bad" order

5

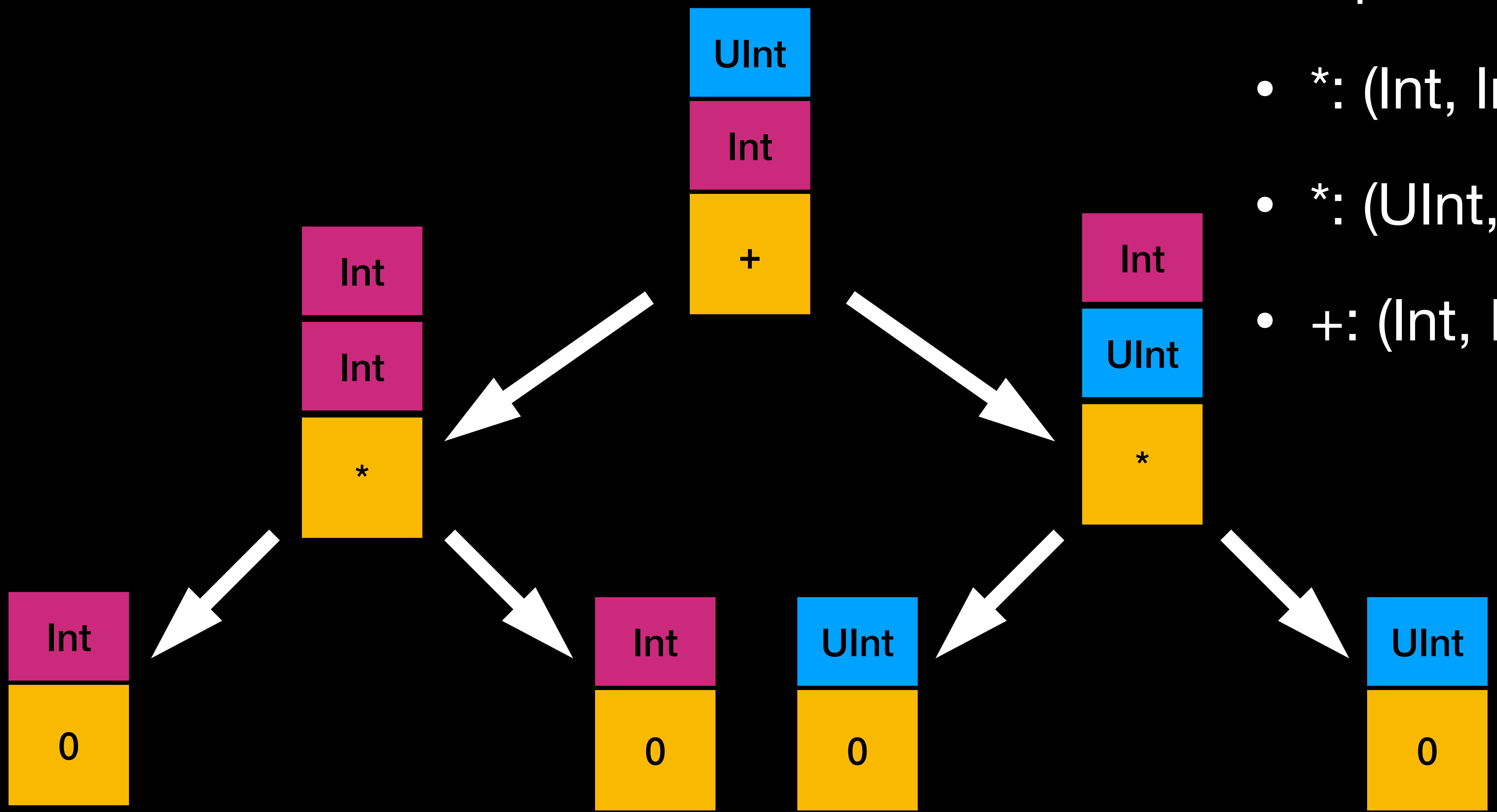


Scopes:

- *: (Int, Int) -> Int
- *: (UInt, UInt) -> UInt

Disjunction selection: "bad" order

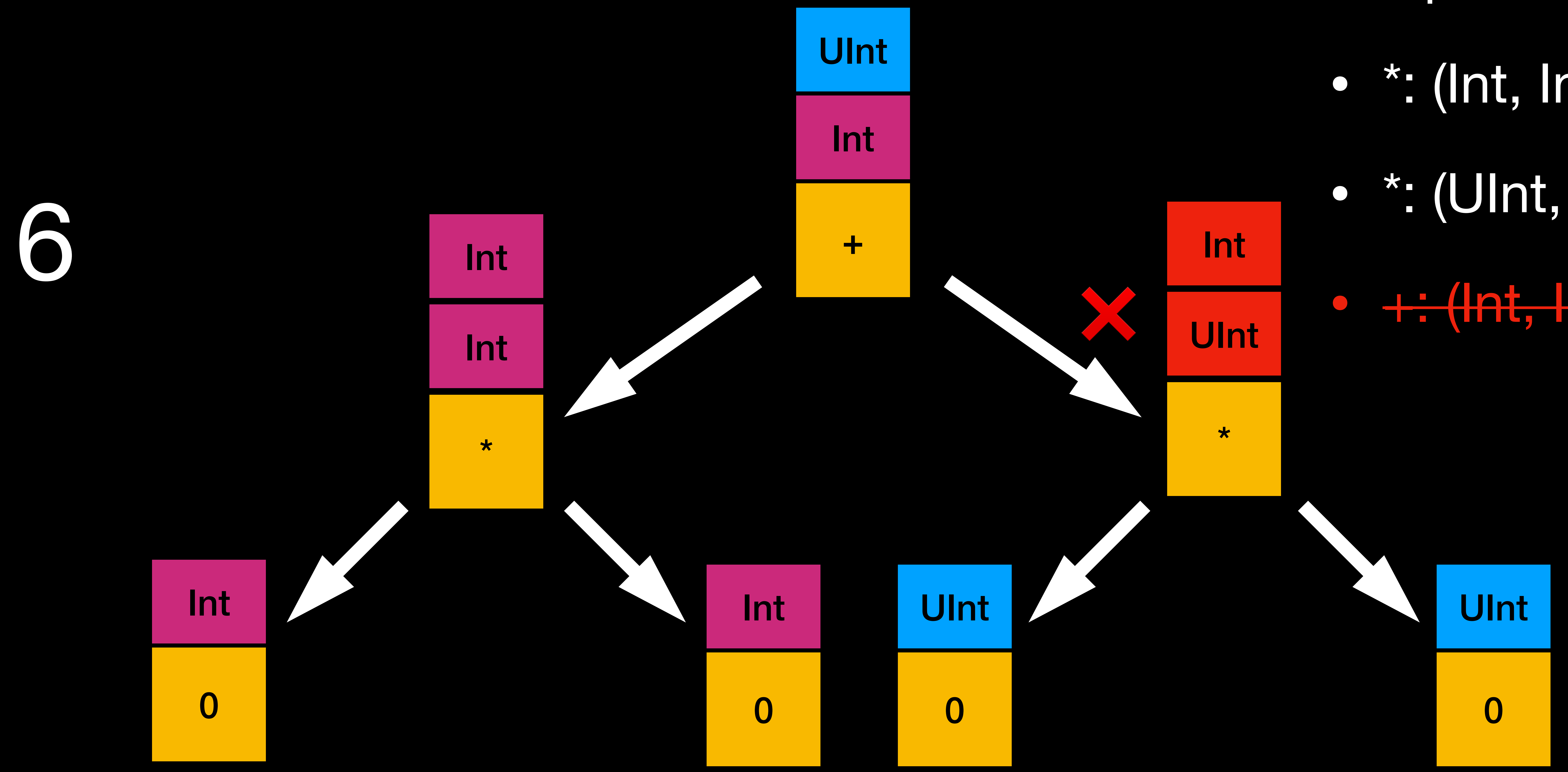
6



Scopes:

- *: (Int, Int) -> Int
- *: (UInt, UInt) -> UInt
- +: (Int, Int) -> Int

Disjunction selection: "bad" order

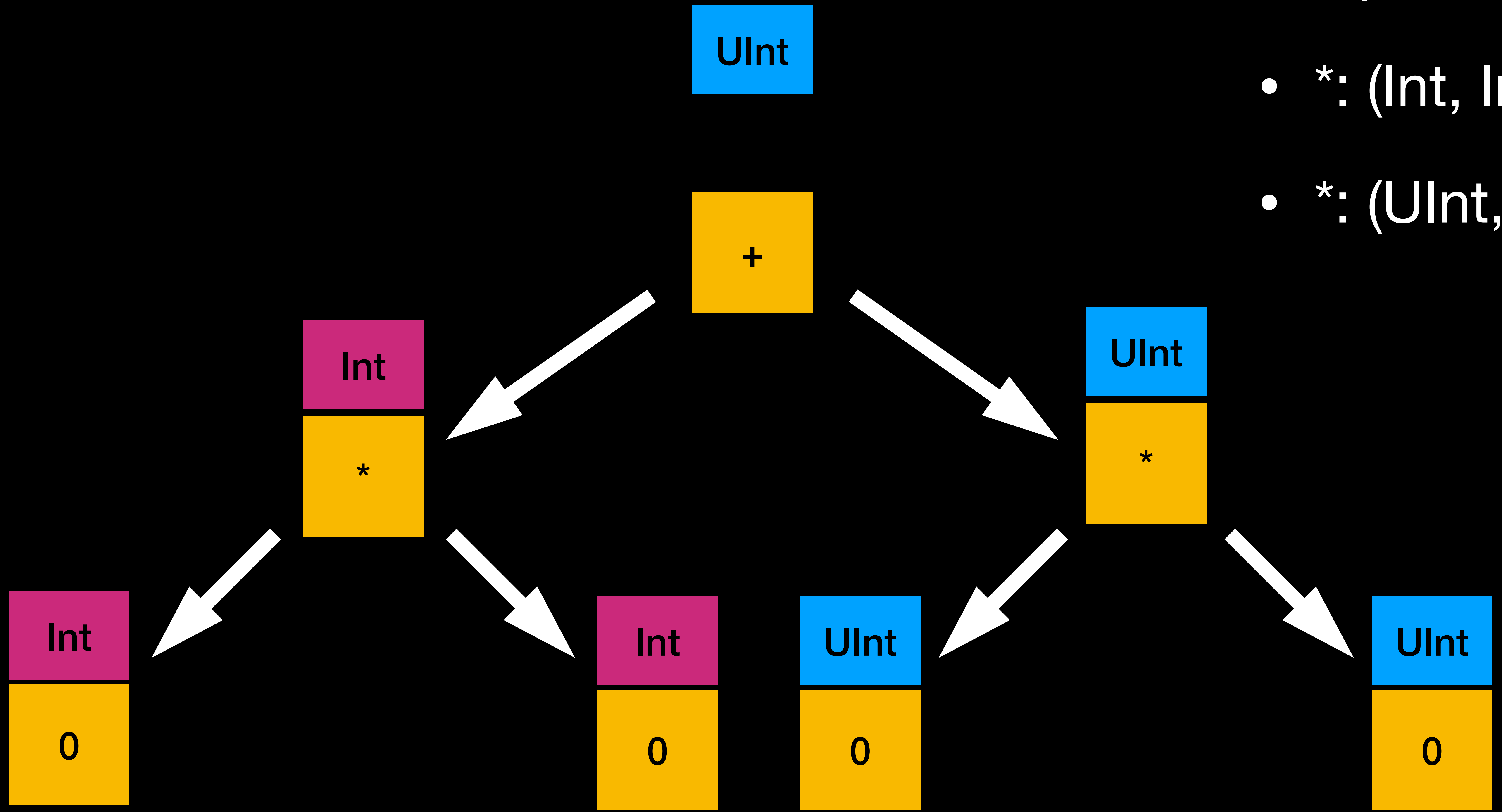


Scopes:

- *: (Int, Int) -> Int
- *: (UInt, UInt) -> UInt
- ~~+: (Int, Int) -> Int~~

Disjunction selection: "bad" order

6

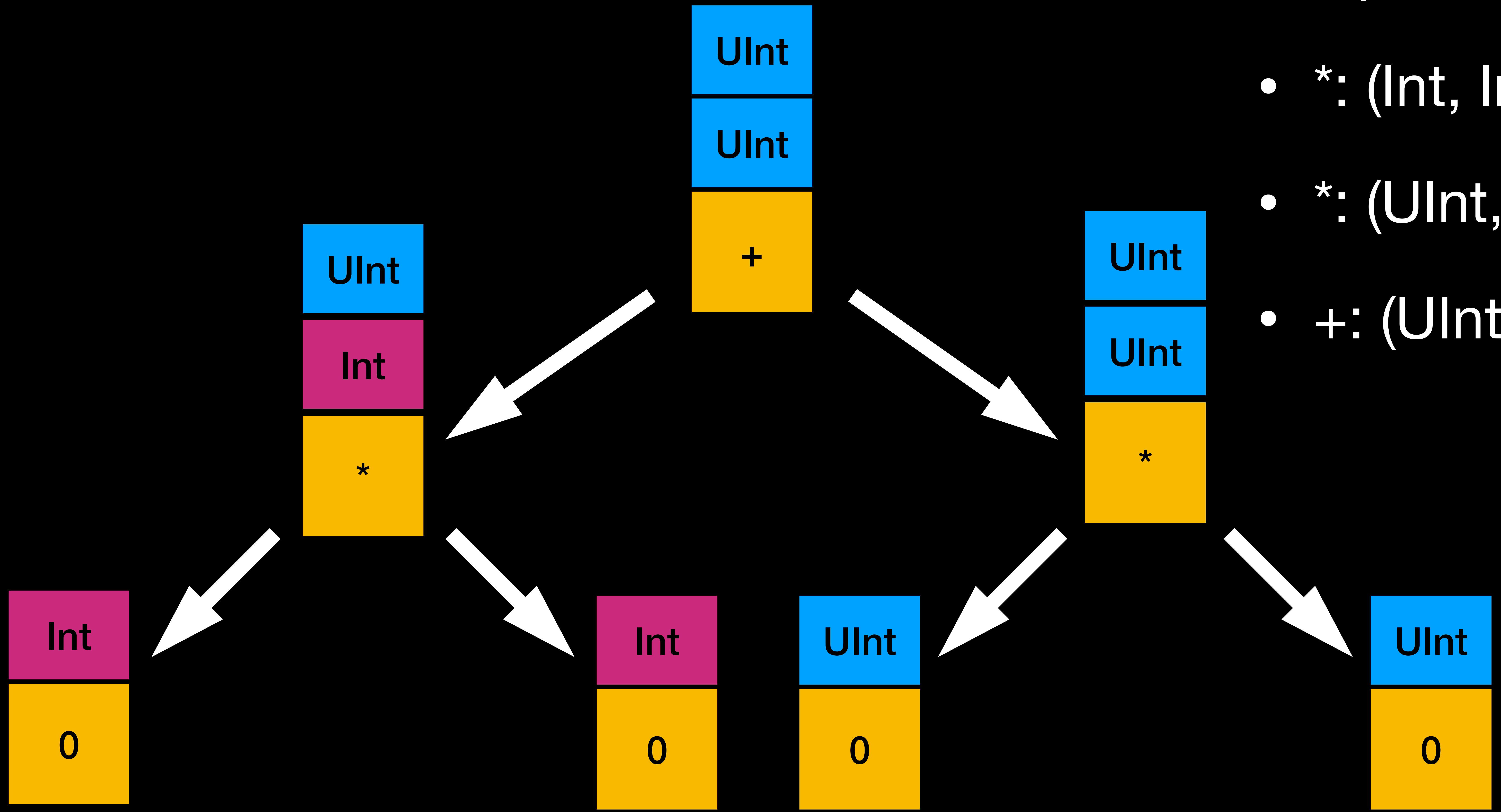


Scopes:

- *: (Int, Int) -> Int
- *: (UInt, UInt) -> UInt

Disjunction selection: "bad" order

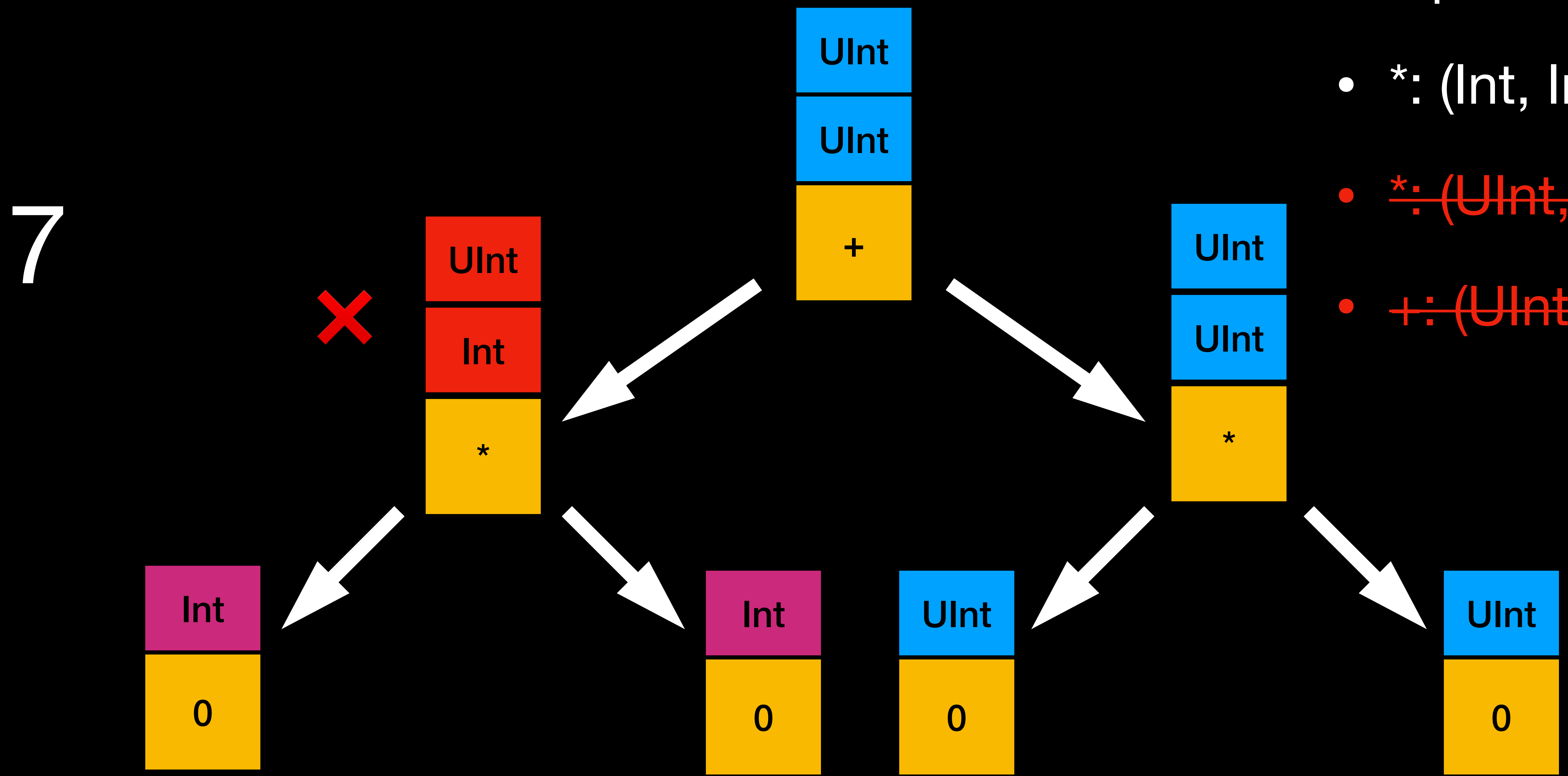
7



Scopes:

- *: (Int, Int) -> Int
- *: (UInt, UInt) -> UInt
- +: (UInt, UInt) -> UInt

Disjunction selection: "bad" order

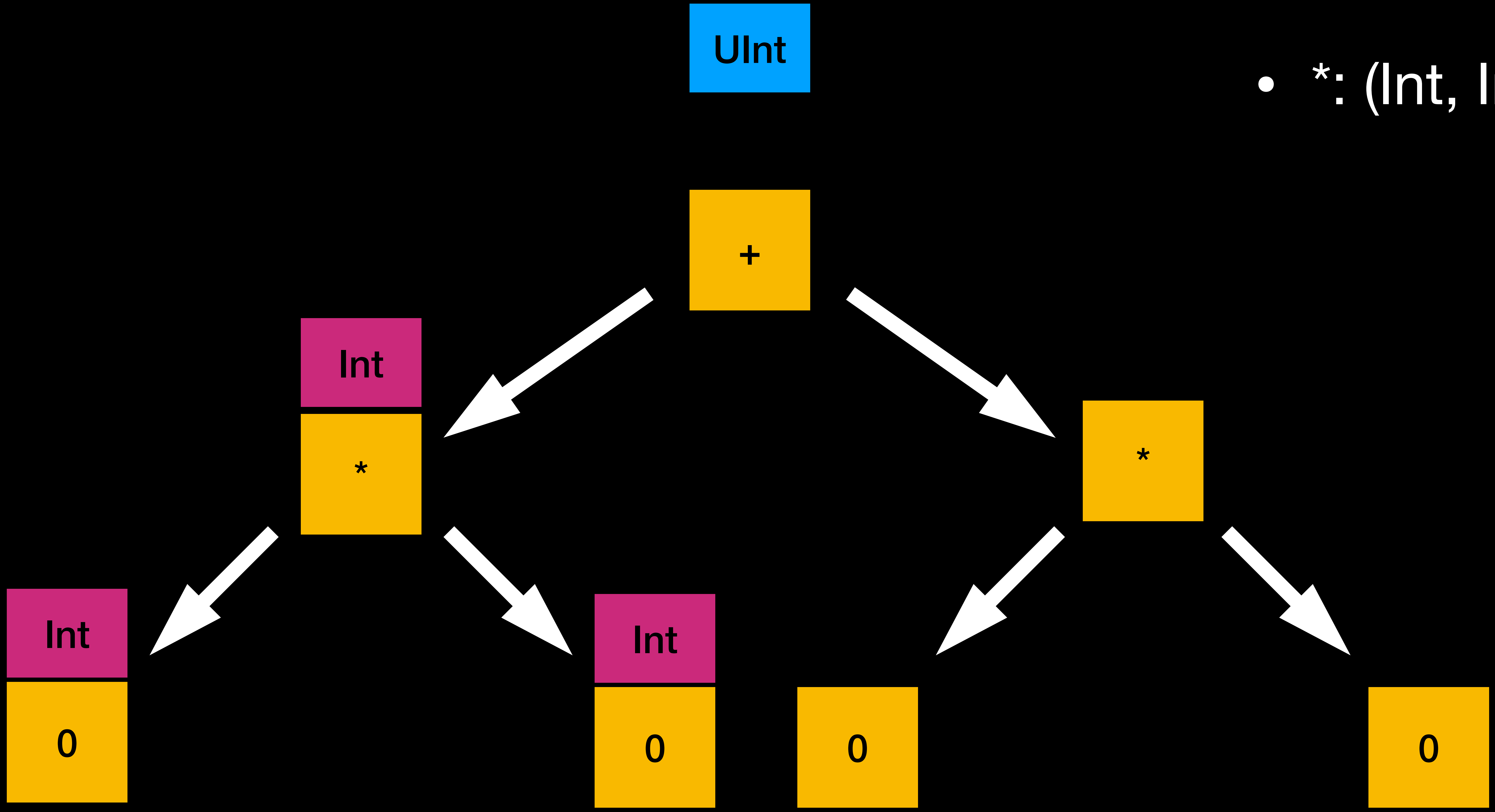


Scopes:

- *: (Int, Int) -> Int
- ~~*: (UInt, UInt) -> UInt~~
- ~~+: (UInt, UInt) -> UInt~~

Disjunction selection: "bad" order

7

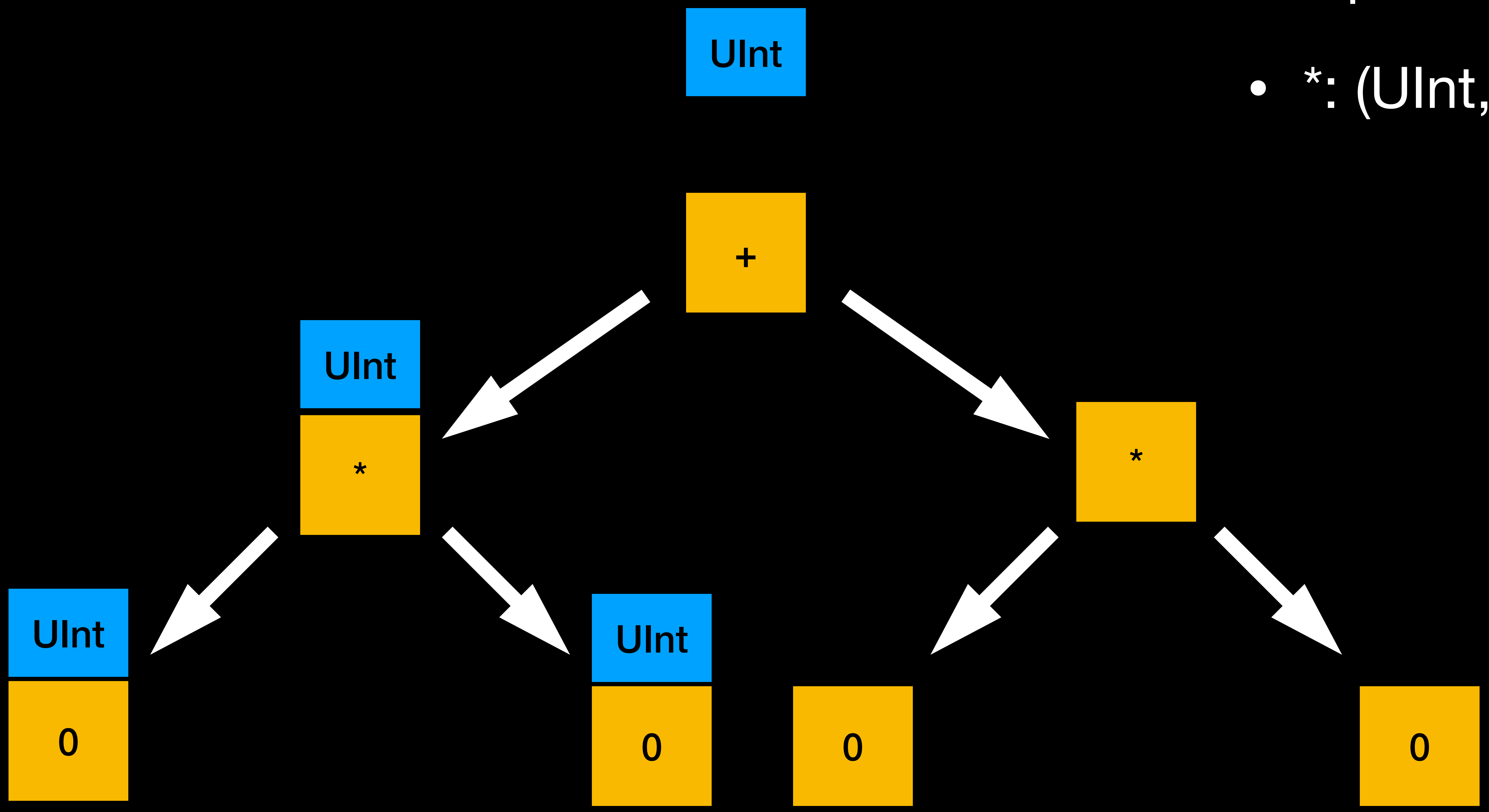


Scopes:

- *: (Int, Int) -> Int

Disjunction selection: "bad" order

8

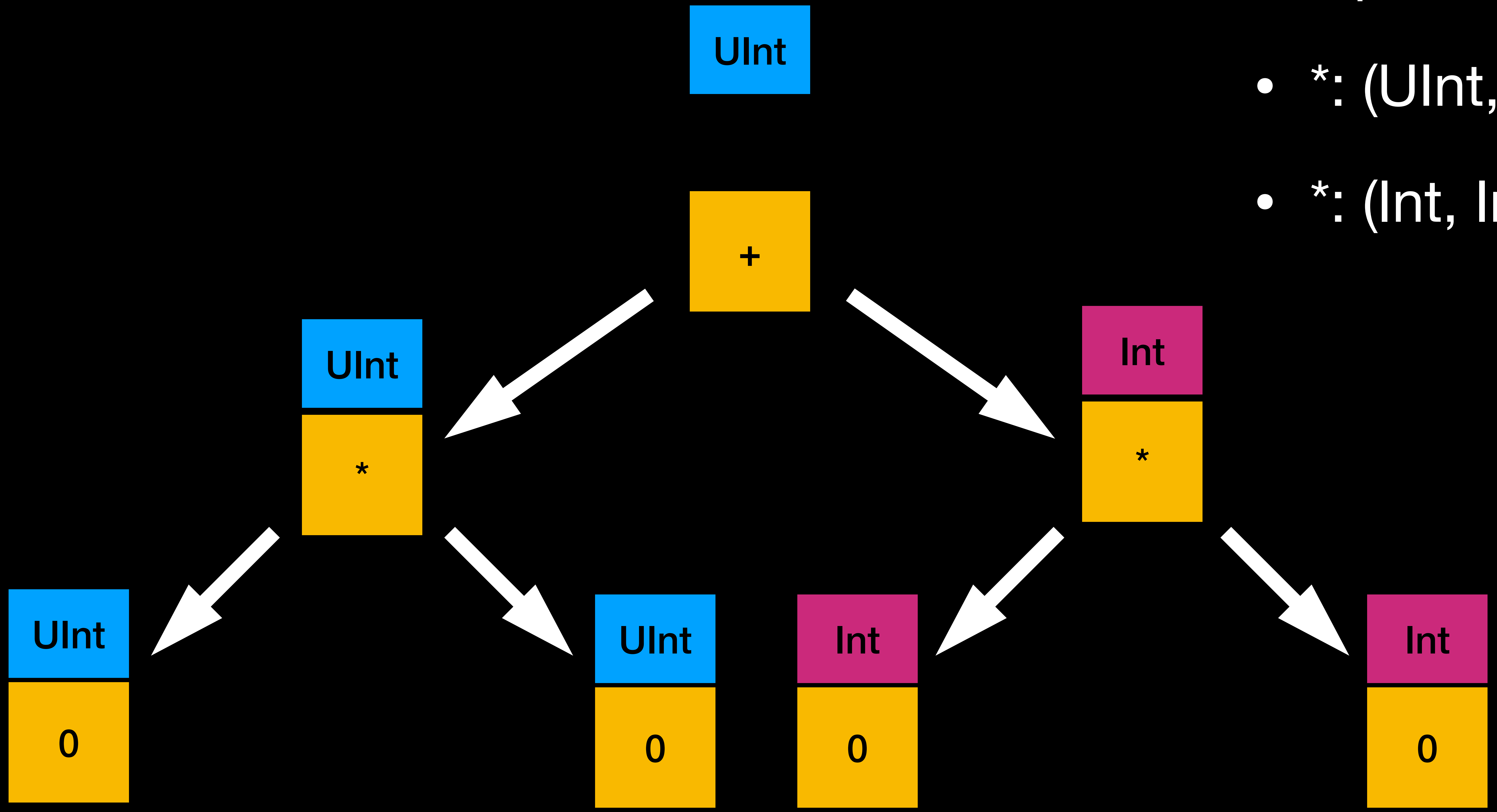


Scopes:

- *: (UInt, UInt) -> UInt

Disjunction selection: "bad" order

9

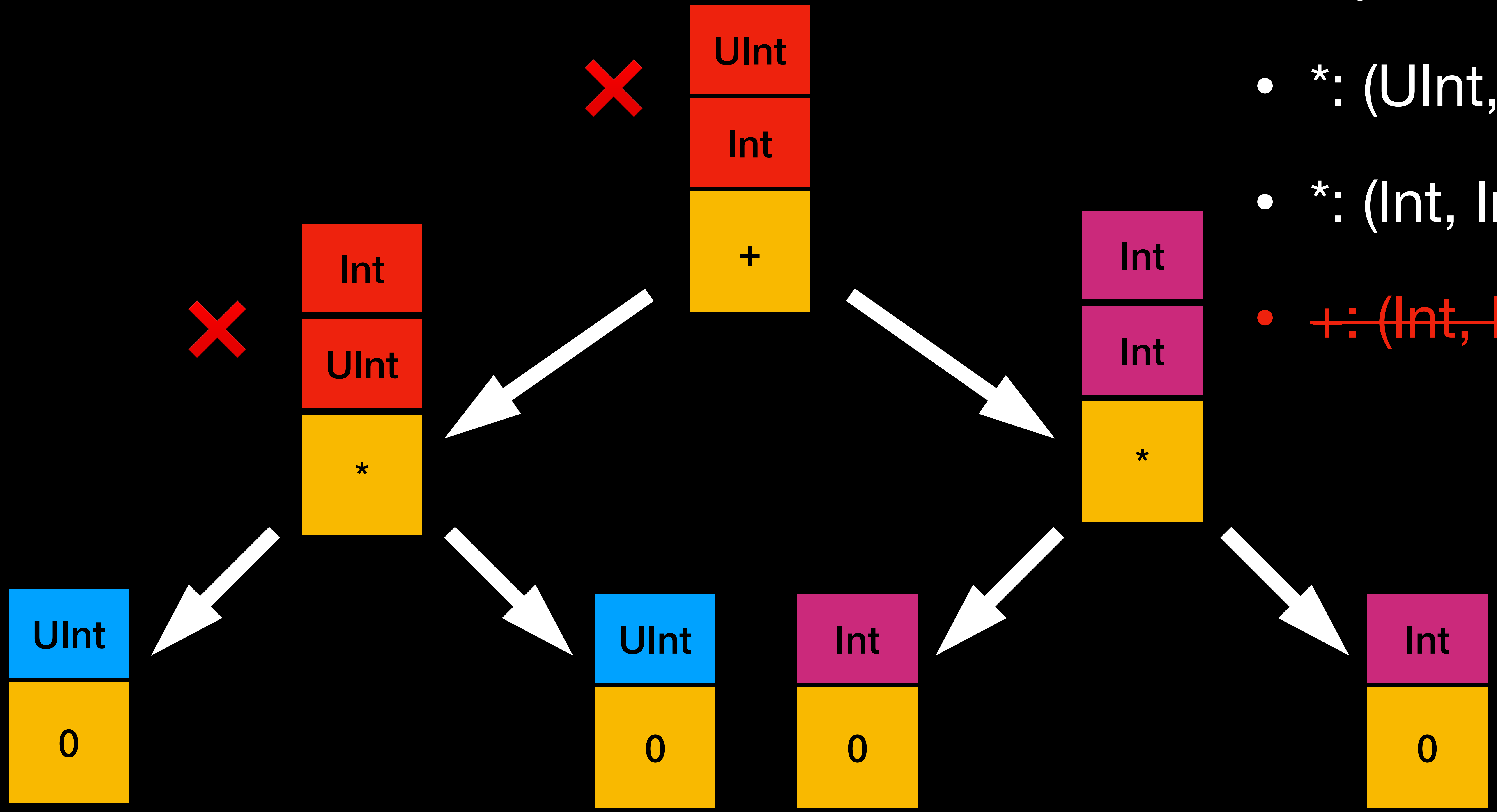


Scopes:

- *: (UInt, UInt) -> UInt
- *: (Int, Int) -> Int

Disjunction selection: "bad" order

10

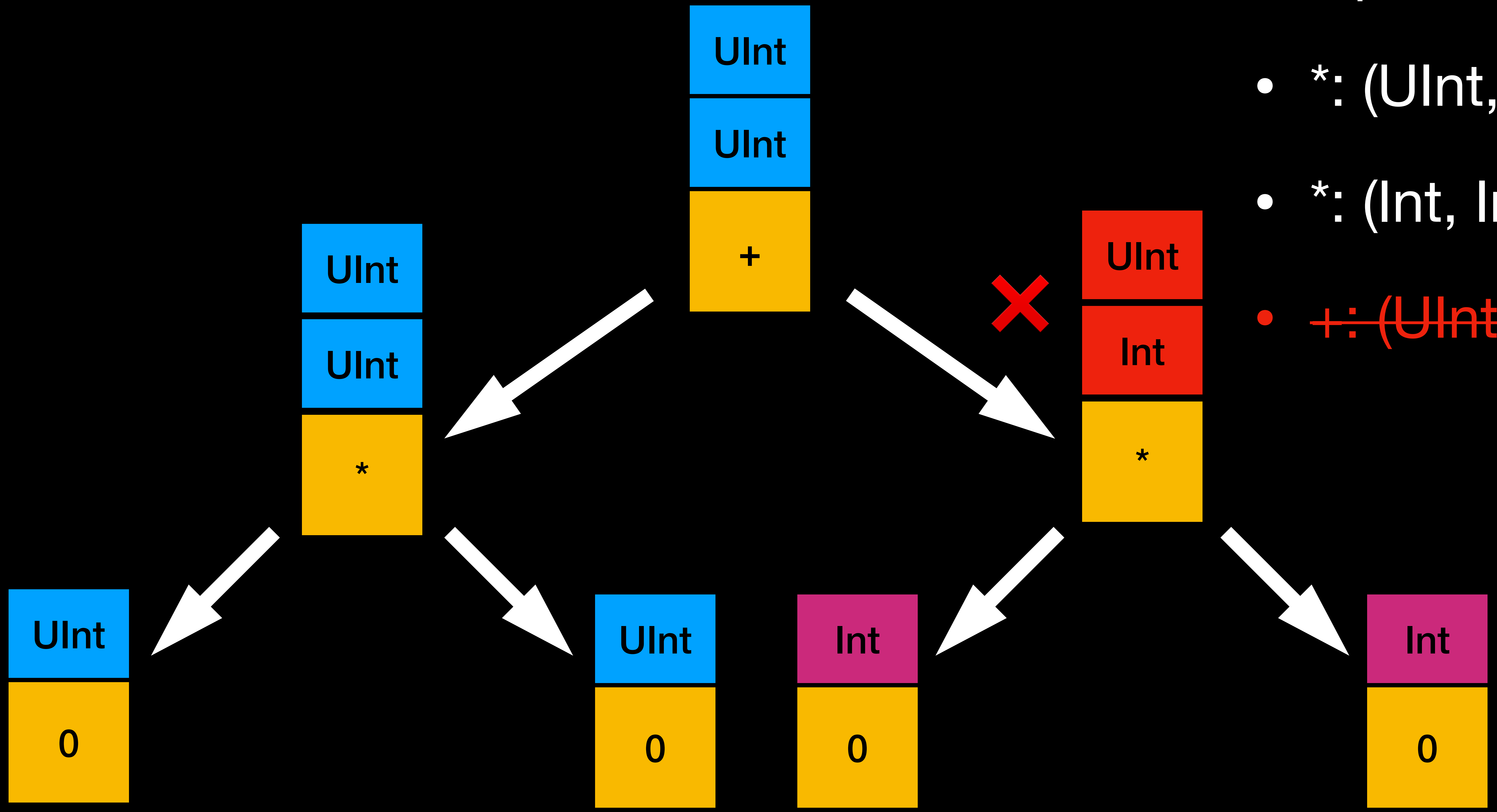


Scopes:

- *: (UInt, UInt) -> UInt
- *: (Int, Int) -> Int
- ~~+: (Int, Int) -> Int~~

Disjunction selection: "bad" order

11

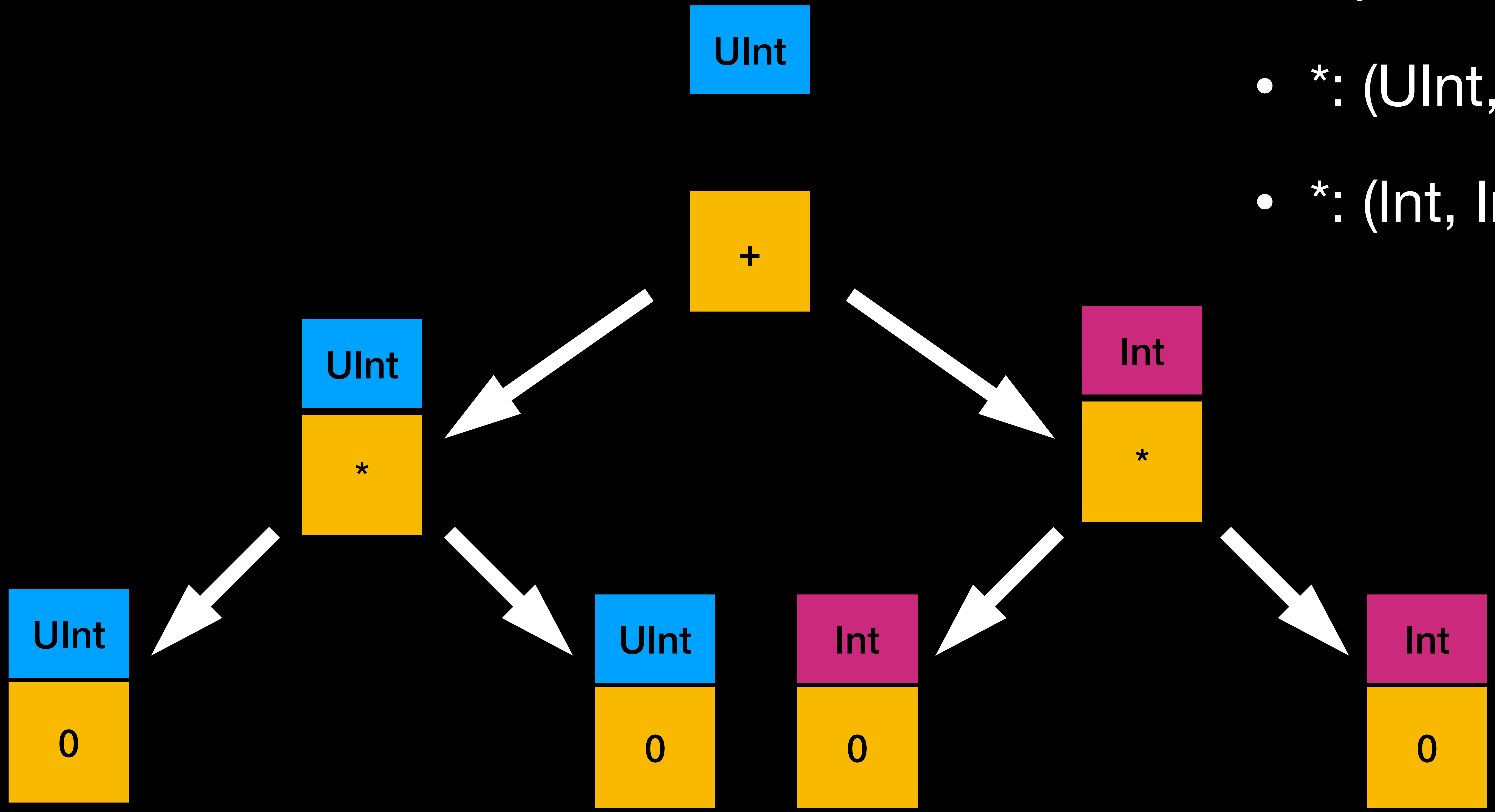


Scopes:

- *: (UInt, UInt) -> UInt
- *: (Int, Int) -> Int
- ~~+: (UInt, UInt) -> UInt~~

Disjunction selection: "bad" order

11

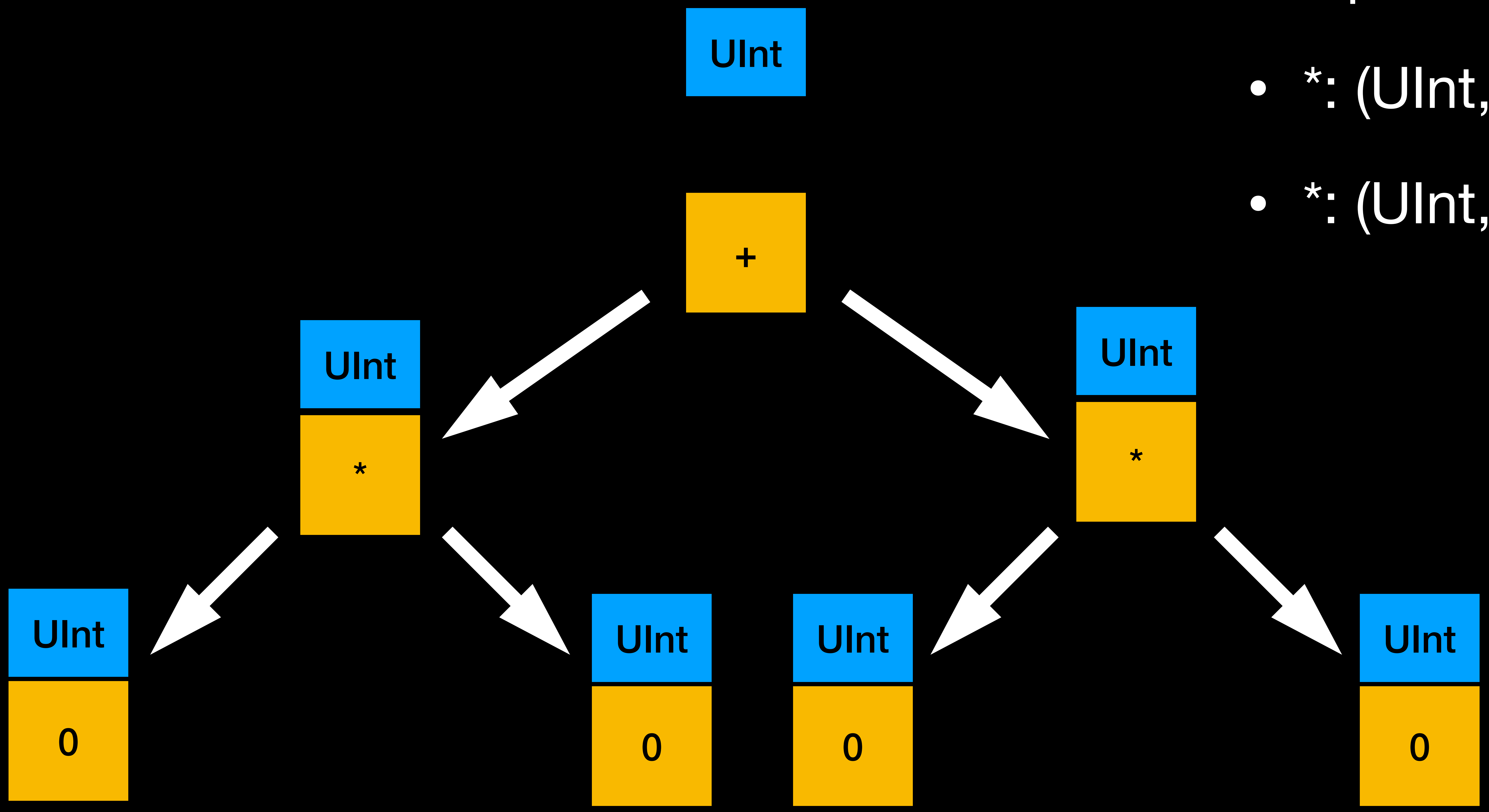


Scopes:

- *: (UInt, UInt) -> UInt
- *: (Int, Int) -> Int

Disjunction selection: "bad" order

12

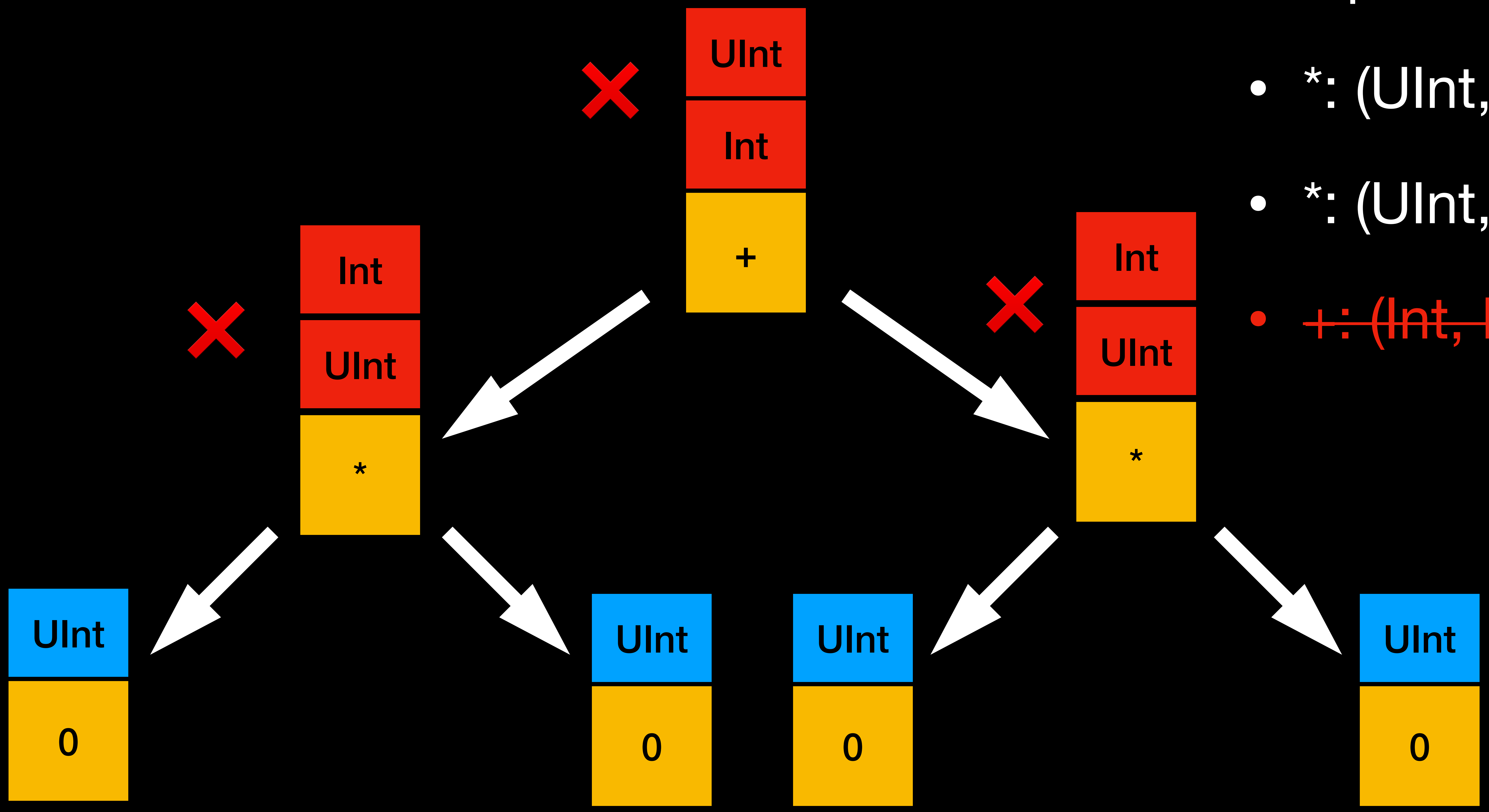


Scopes:

- *: (UInt, UInt) -> UInt
- *: (UInt, UInt) -> UInt

Disjunction selection: "bad" order

13

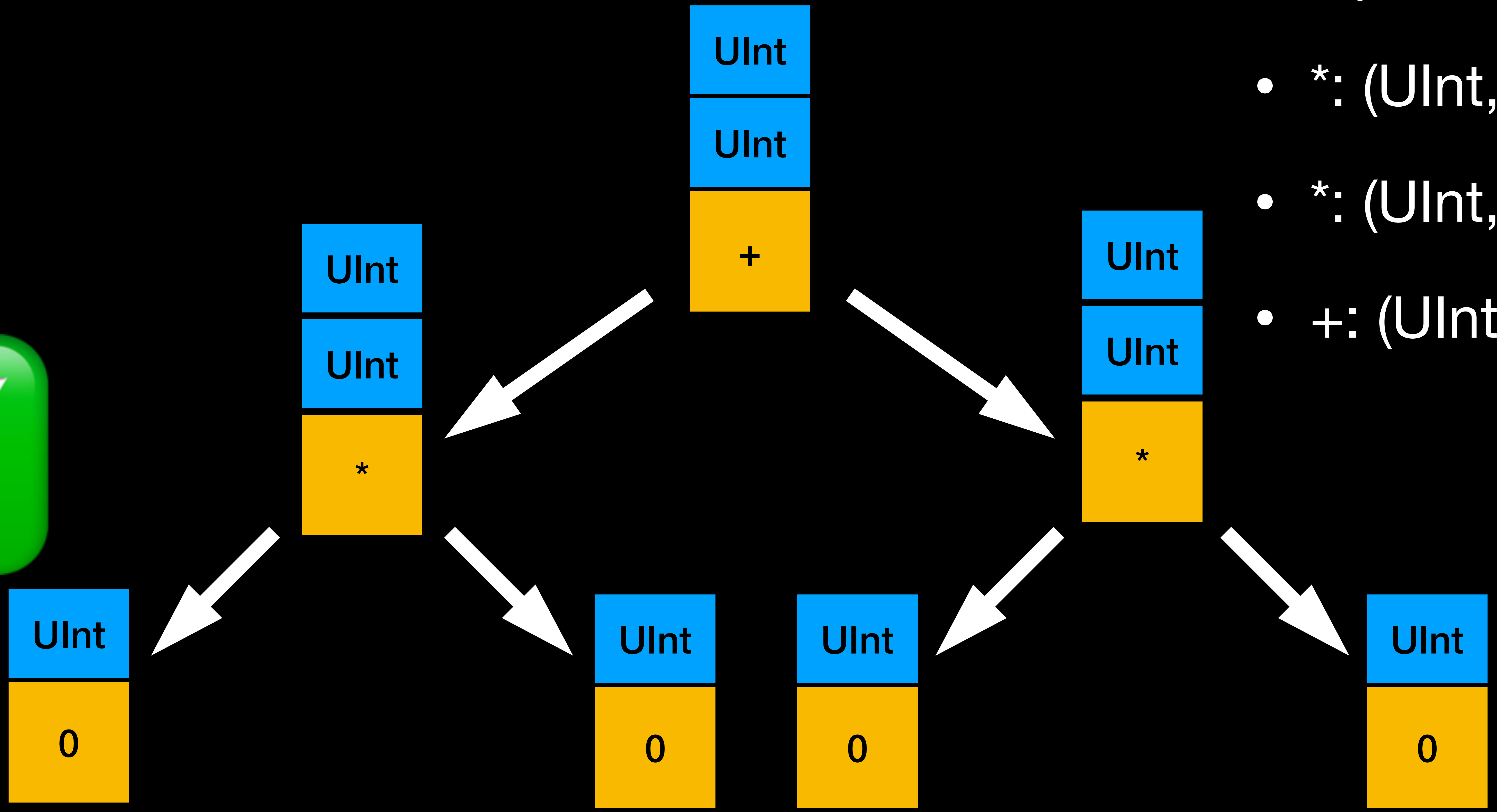


Scopes:

- *: (UInt, UInt) -> UInt
- *: (UInt, UInt) -> UInt
- ~~+: (Int, Int) -> Int~~

Disjunction selection: "bad" order

14 ✅



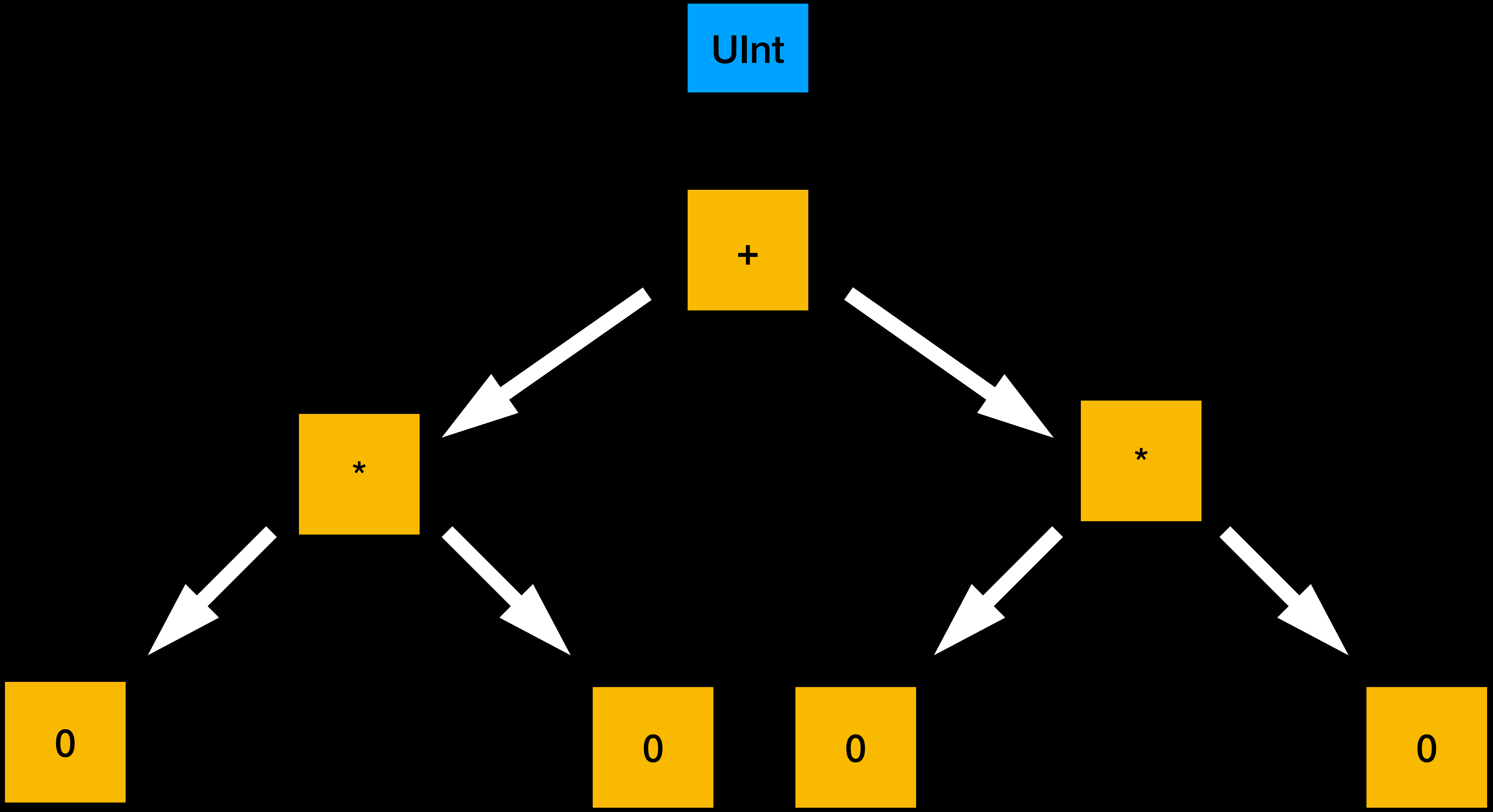
Scopes:

- *: (UInt, UInt) -> UInt
- *: (UInt, UInt) -> UInt
- +: (UInt, UInt) -> UInt

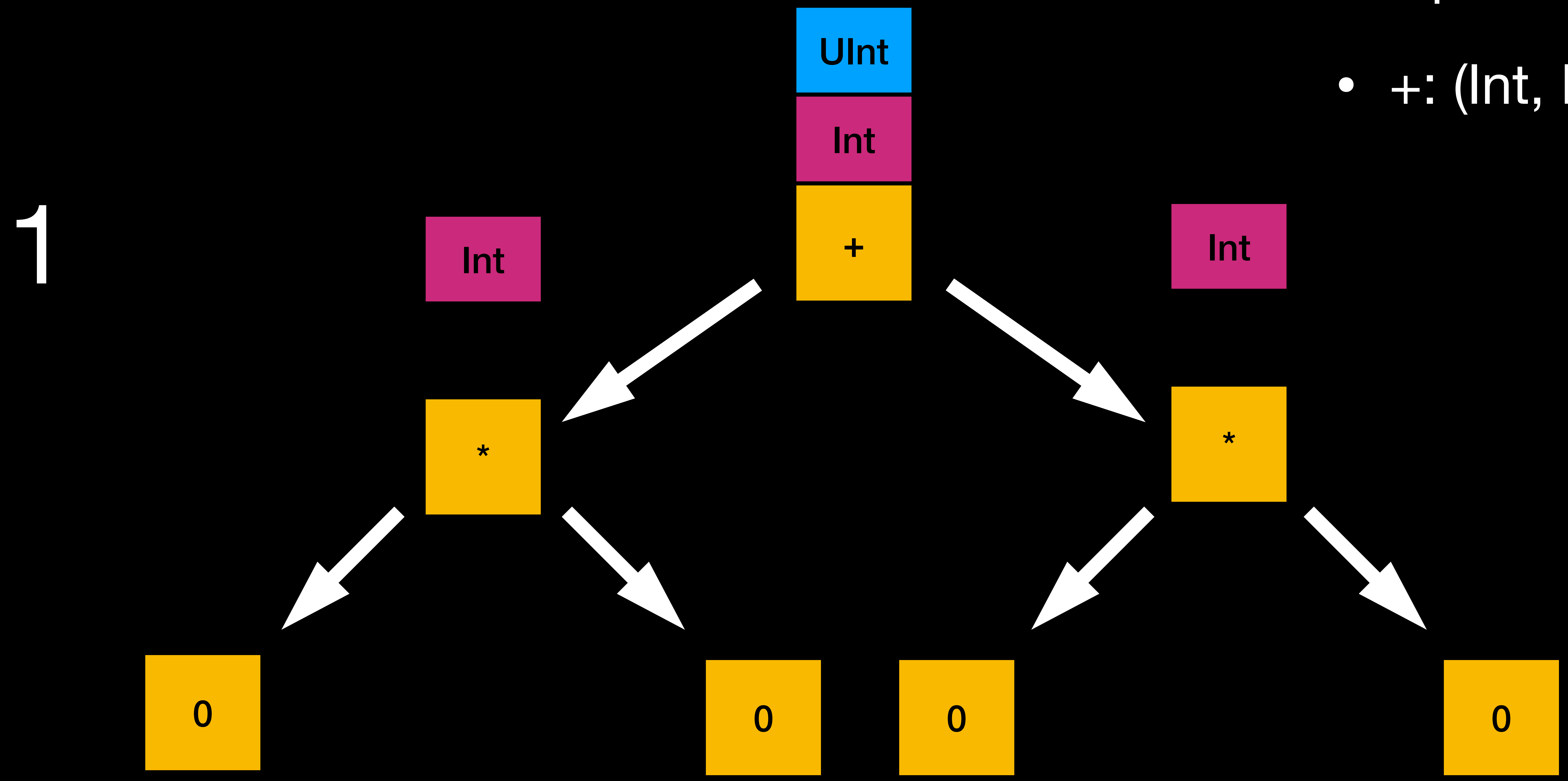
Disjunction selection: "good" order

Scopes:

0



Disjunction selection: "good" order

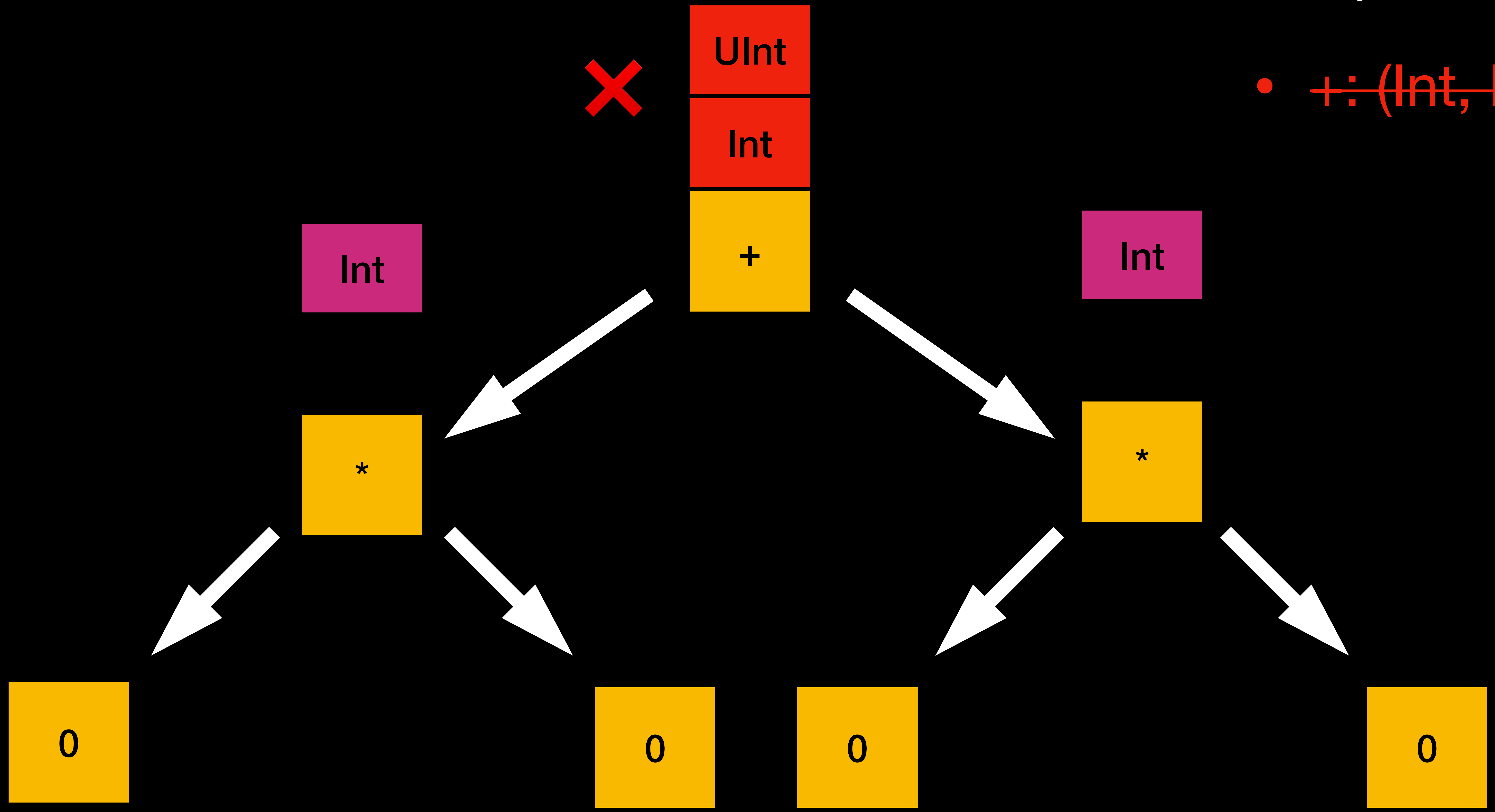


Scopes:

- +: (Int, Int) -> Int

Disjunction selection: "good" order

1

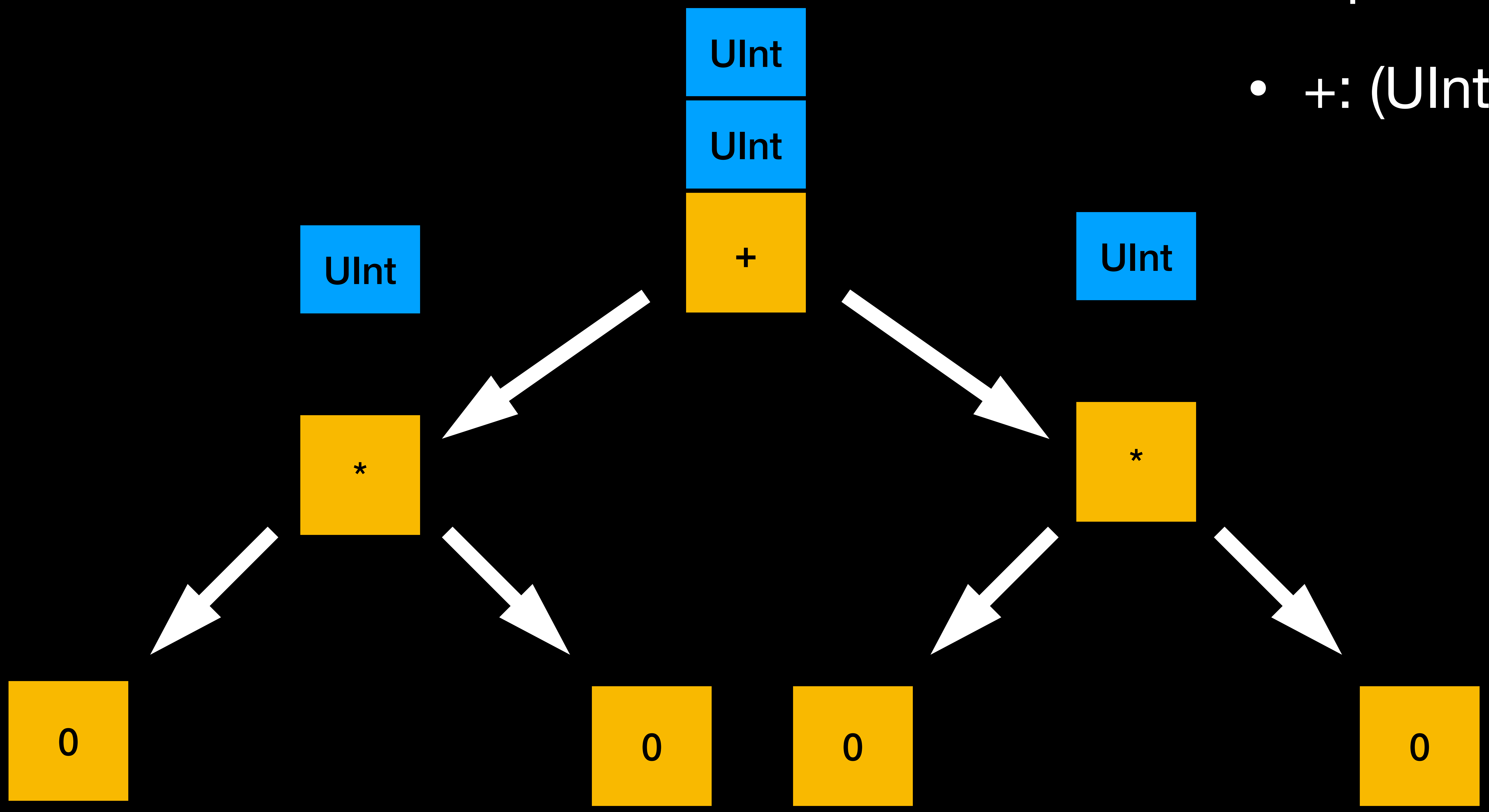


Scopes:

- ~~+: (Int, Int) -> Int~~

Disjunction selection: "good" order

2

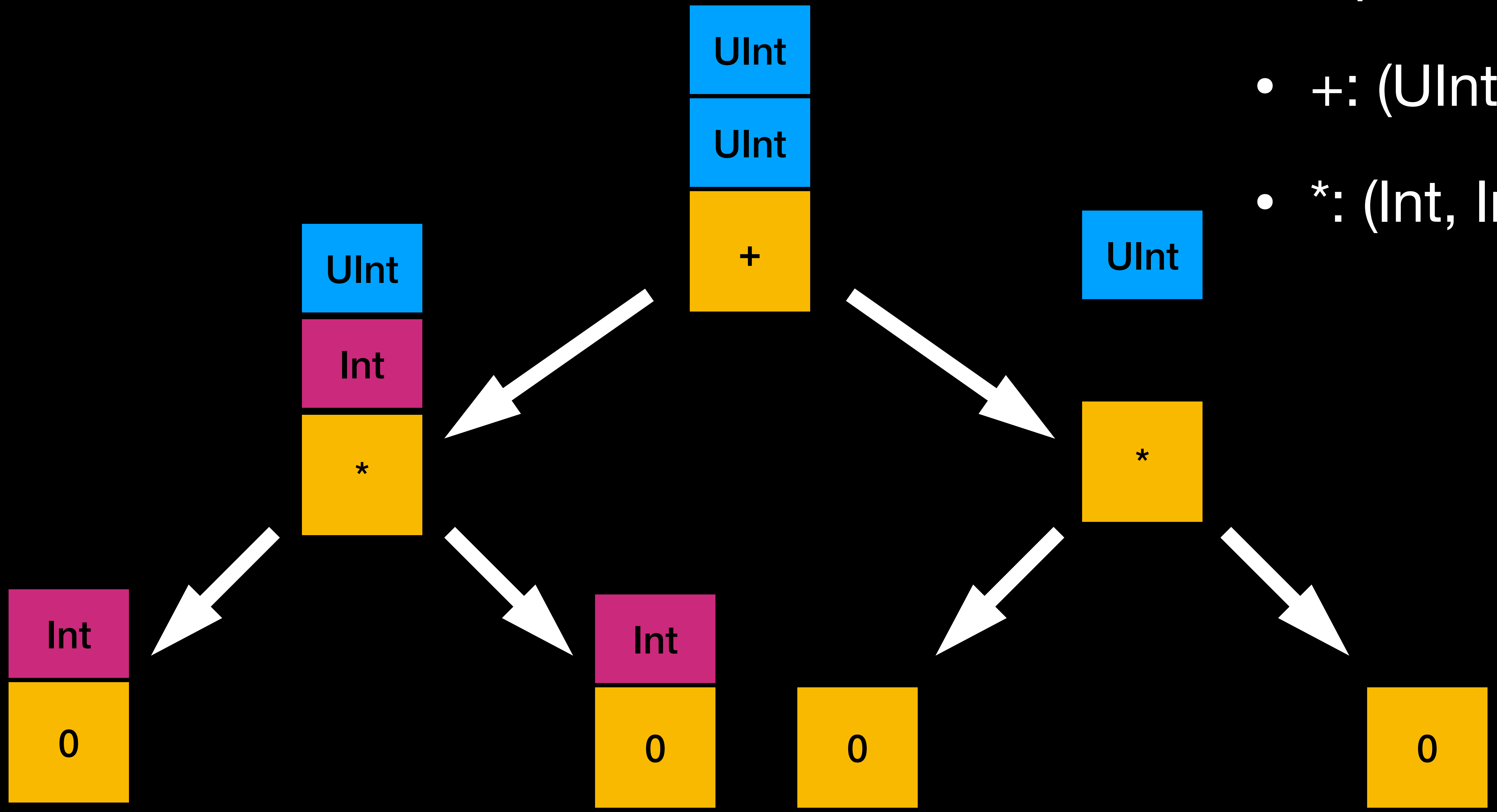


Scopes:

- +: (UInt, UInt) -> UInt

Disjunction selection: "good" order

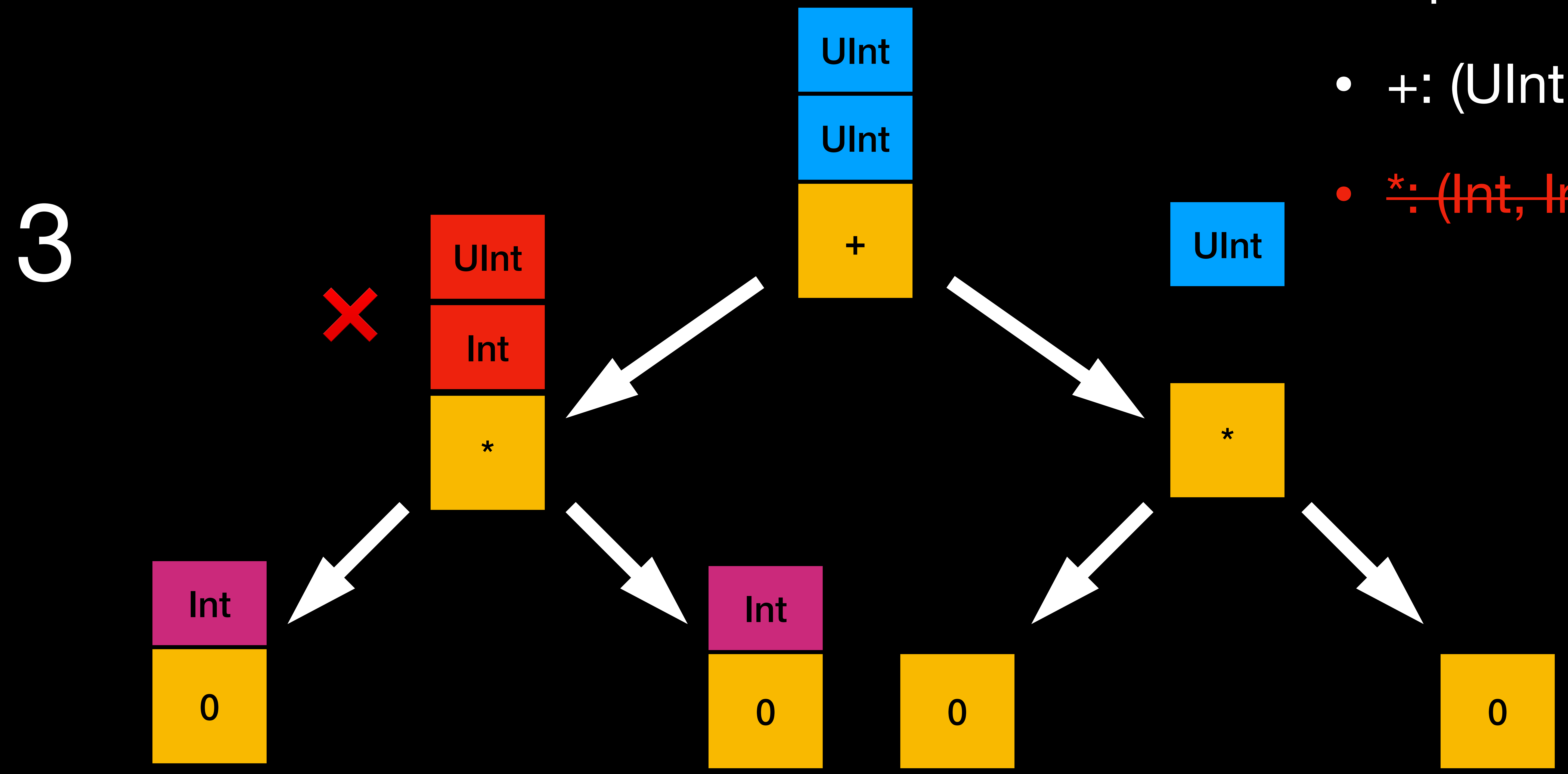
3



Scopes:

- +: (UInt, UInt) -> UInt
- *: (Int, Int) -> Int

Disjunction selection: "good" order

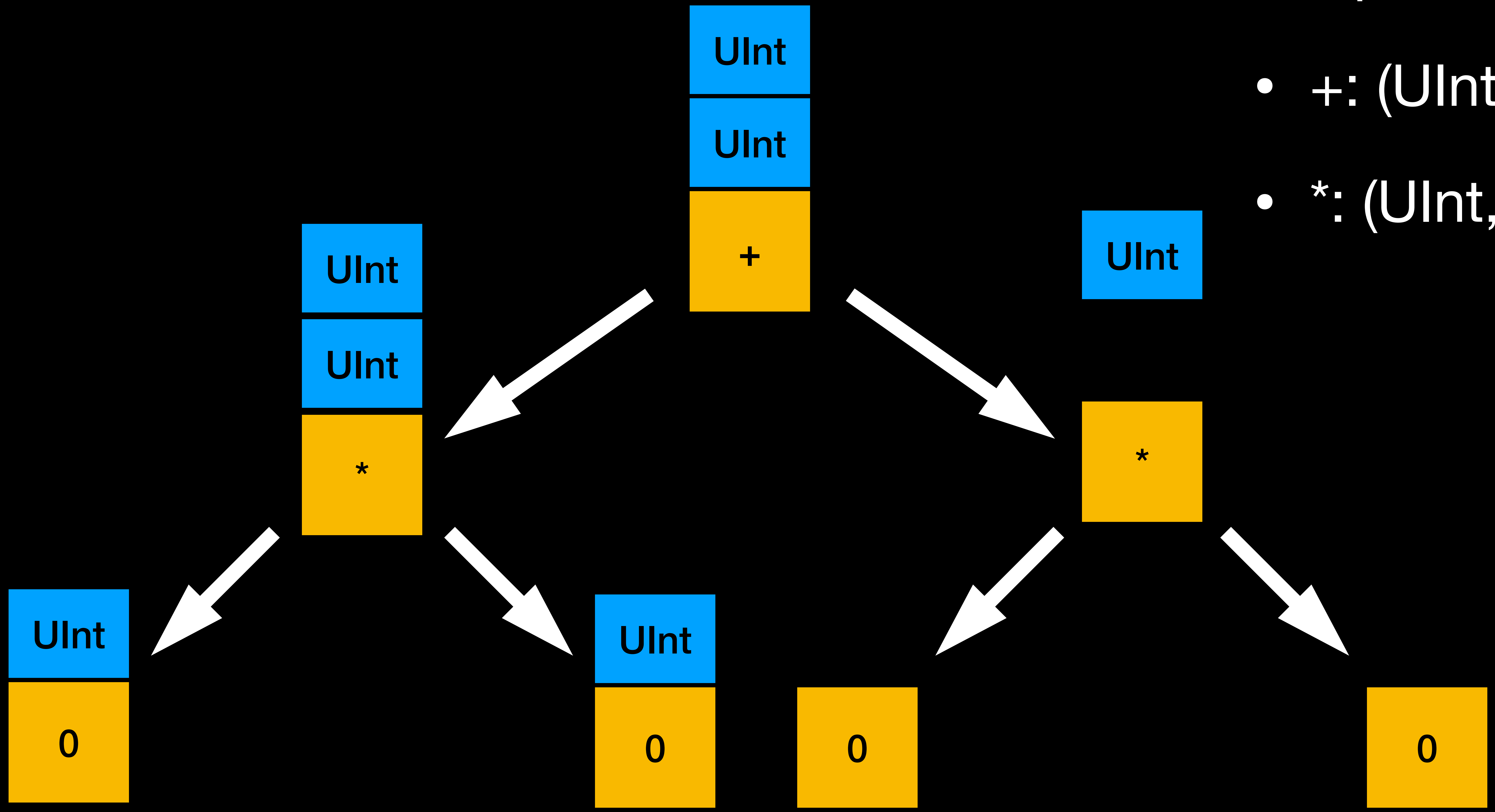


Scopes:

- +: (UInt, UInt) -> UInt
- ~~*: (Int, Int) -> Int~~

Disjunction selection: "good" order

4

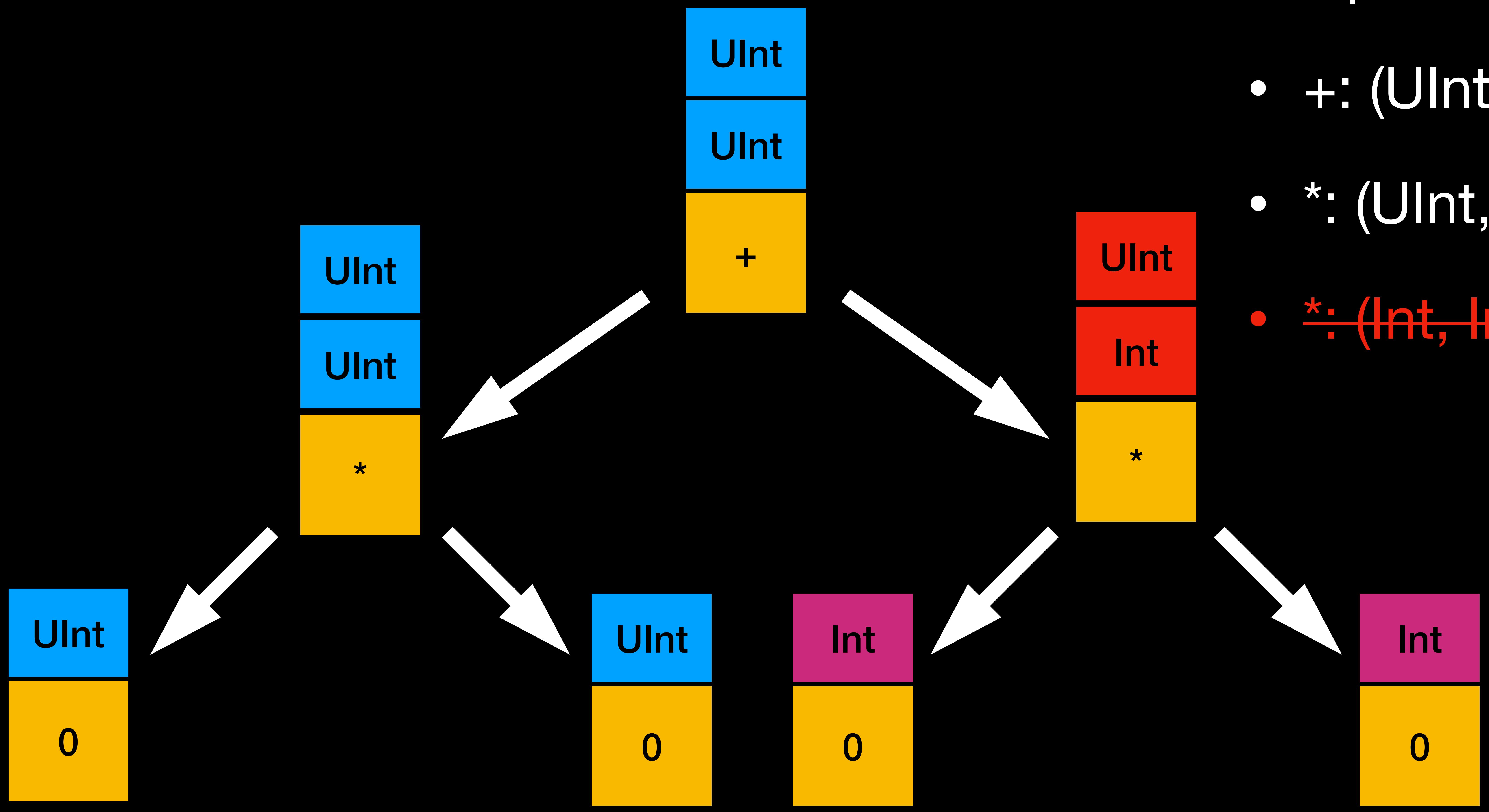


Scopes:

- +: (UInt, UInt) -> UInt
- *: (UInt, UInt) -> UInt

Disjunction selection: "good" order

4

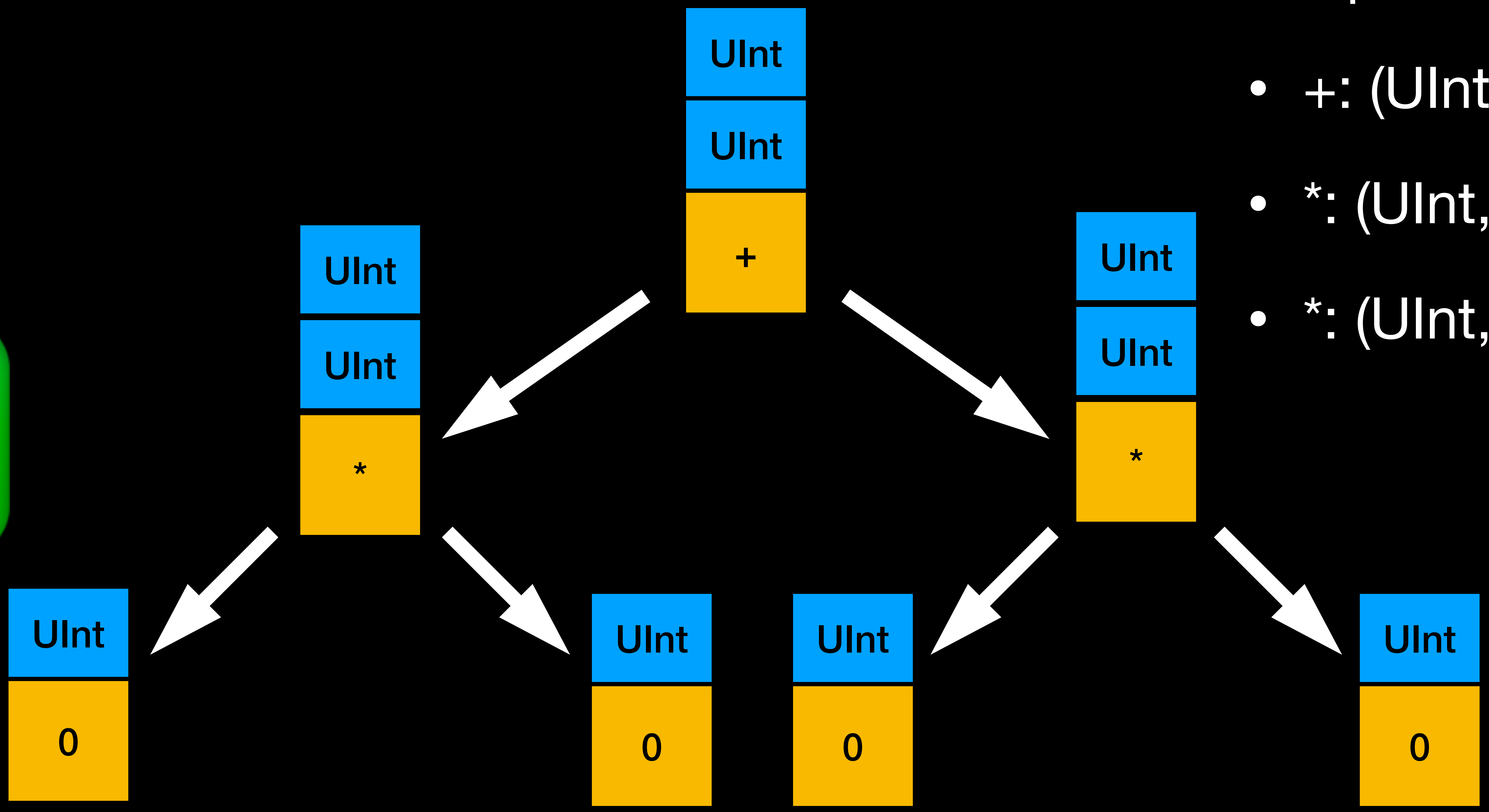


Scopes:

- +: (UInt, UInt) -> UInt
- *: (UInt, UInt) -> UInt
- ~~*: (Int, Int) -> Int~~

Disjunction selection: "good" order

5 ✅



Scopes:

- +: (UInt, UInt) -> UInt
- *: (UInt, UInt) -> UInt
- *: (UInt, UInt) -> UInt

Second example

Further improvements to disjunction selection

```
func test() -> [Double] {
  return [1/8.0, 1/4.0, 1/3.0, 1/2.0, 2/3.0, 3/4.0,
          1, 5/4.0, 4/3.0, 3/2.0, 2, 4, 8]
    .map { x in x / 8.0 }
}
```

- Swift 6.3: the compiler is unable to type-check this expression in reasonable time; try breaking up the expression into distinct sub-expressions ❌

- Now: 78 scopes, 3 milliseconds ✅

The game

- Choosing the "best" disjunction to attempt
- Skipping disjunction choices which cannot participate in a valid solution

Third example

Disabling dead-end disjunction choices

```
func test(x: UInt32, A: [[UInt32]]) -> UInt32 {
    return ((A[0][Int(x >> 24) & 0xFF] &+ A[1][Int(x >> 16) & 0xFF])
            ^ A[2][Int(x >> 8) & 0xFF]) &+ A[3][Int(x & 0xFF)]
         | ((A[0][Int(x >> 24) & 0xFF] &+ A[1][Int(x >> 16) & 0xFF])
            ^ A[2][Int(x >> 8) & 0xFF]) &+ A[3][Int(x & 0xFF)]
         | ((A[0][Int(x >> 24) & 0xFF] &+ A[1][Int(x >> 16) & 0xFF])
            ^ A[2][Int(x >> 8) & 0xFF]) &+ A[3][Int(x & 0xFF)]
         | ((A[0][Int(x >> 24) & 0xFF] &+ A[1][Int(x >> 16) & 0xFF])
            ^ A[2][Int(x >> 8) & 0xFF]) &+ A[3][Int(x & 0xFF)]
}
```

- Swift 6.3: 421593 scopes, *3 seconds* 🐢
- Now: 970 scopes, 36 milliseconds ✅

When all else fails

Swift 6.2

- Invalid expression: no + overload for Int vs. String

```
let s = ""
let n = 0

let closure = {
  let _ = 0
  let _ = "" + s + "" + s + "" + s + "" + n + ""
  let _ = 0
}
```

When all else fails

Swift 6.2

- Invalid expression: no + overload for Int vs. String

```swift
let s = ""
let n = 0

let closure = {
  let _ = 0
  let _ = "" + s + "" + s + "" + s + "" + n + ""
  let _ = 0
}
```

When all else fails

Swift 6.2

- Invalid expression: no + overload for Int vs. String

```
let s = ""
let n = 0

let closure = {
  let _ = 0
  let _ = "" + s + "" + s + "" + s + "" + n + ""
  let _ = 0
}
```

the compiler is unable to type-check this expression in reasonable time

When all else fails

Swift 6.3 developer snapshot from swift.org

- Invalid expression: no + overload for Int vs. String
- More precise source location (suggested by a user on the forums!)

```
let s = ""
let n = 0

let closure = {
  let _ = 0
  let _ = "" + s + "" + s + "" + s + "" + n + ""
  let _ = 0
}
```

the compiler is unable to type-check this expression in reasonable time

More details

- Various examples from bug reports:
  https://github.com/swiftlang/swift/tree/main/validation-test/Sema/type_checker_perf

- Roadmap for improving type checker performance:
  https://forums.swift.org/t/roadmap-for-improving-the-type-checker/82952

Thank you