Reasoning about GADT Pattern Matching in Haskell

George Karachalias

UNIVERSITEIT GENT

Literals

Handling Literals

Cannot be treated as guards

- Must be matched eagerly

```
f1 :: Int -> Bool -> Int
f1 5 True = 1
```

Handling Literals

Cannot be treated as guards

- Must be matched eagerly

```
f1 :: Int -> Bool -> Int
f1 5 True = 1

ghci> f1 (error “1st”) (error “2nd”)
```

Handling Literals

Cannot be treated as guards

- Must be matched eagerly

```
f1 :: Int -> Bool -> Int
f1 5 True = 1
```

```
ghci> f1 (error “1st”) (error “2nd”)
*** Exception: 1st
```

Handling Literals

Cannot be treated as guards

- Must be matched eagerly

```
f2 :: Int -> Bool -> Int
f2 x True | x==5 = 1
```

Handling Literals

Cannot be treated as guards

- Must be matched eagerly

```
f2 :: Int -> Bool -> Int
f2 x True | x==5 = 1

ghci> f2 (error “1st”) (error “2nd”)
```

Handling Literals

Cannot be treated as guards

- Must be matched eagerly

```
f2 :: Int -> Bool -> Int
f2 x True | x==5 = 1
```

```
ghci> f2 (error “1st”) (error “2nd”)
*** Exception: 2nd
```

Handling Literals

Cannot be treated as nullary constructors

- Exceedingly large set (or infinite)

```
f1 :: Int -> Bool -> Int
f1 5 True = 1
```

Handling Literals

Cannot be treated as nullary constructors

- Exceedingly large set (or infinite)

```
f1 :: Int -> Bool -> Int
f1 5 True = 1
```

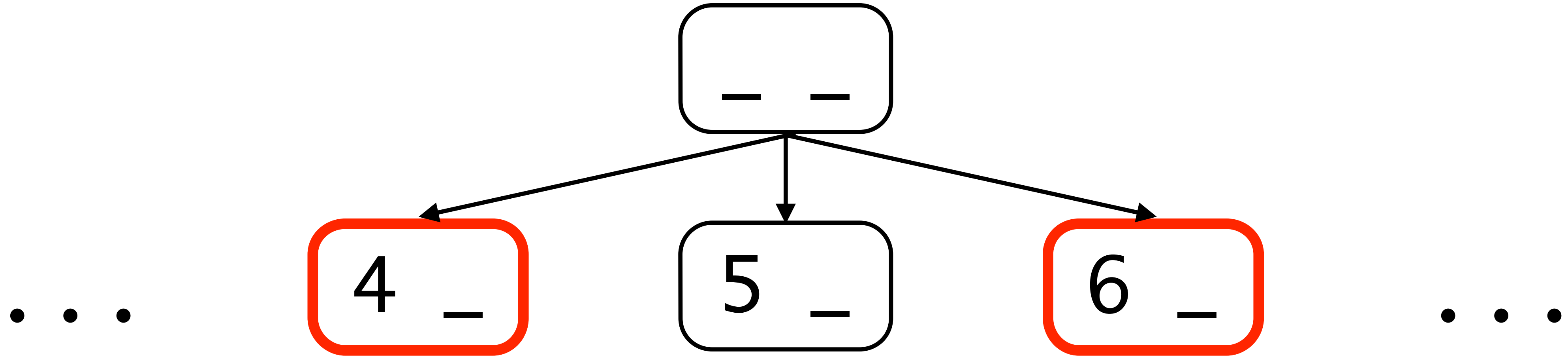
```
_ _
```

Handling Literals

Cannot be treated as nullary constructors

- Exceedingly large set (or infinite)

```
f1 :: Int -> Bool -> Int
f1 5 True = 1
```



_ _

. . .

4 _

5 _

6 _

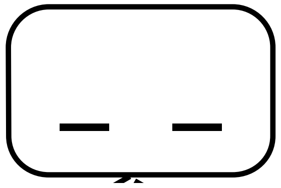
. . .

Example

```haskell
f :: Int -> Int -> Int
f 1 5 = 1
f 2 _ = 2
f 1 _ = 3
```

Example

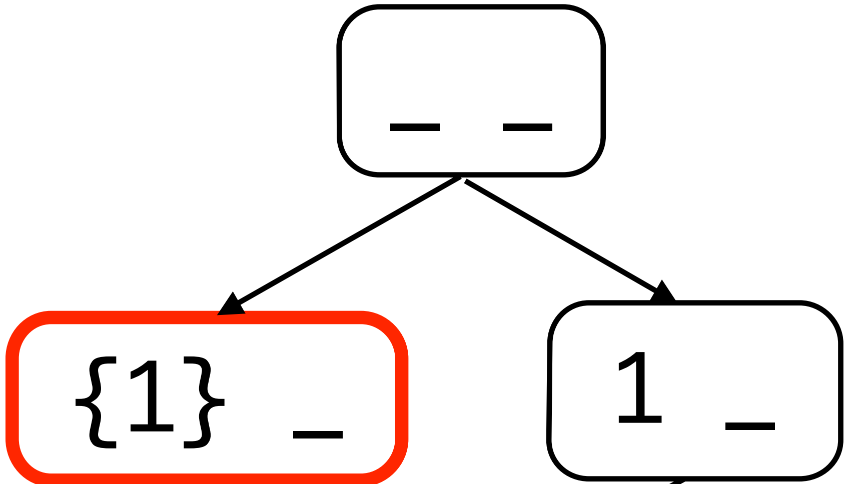
```
f :: Int -> Int -> Int
f 1 5 = 1
f 2 _ = 2
f 1 _ = 3
```



_ _

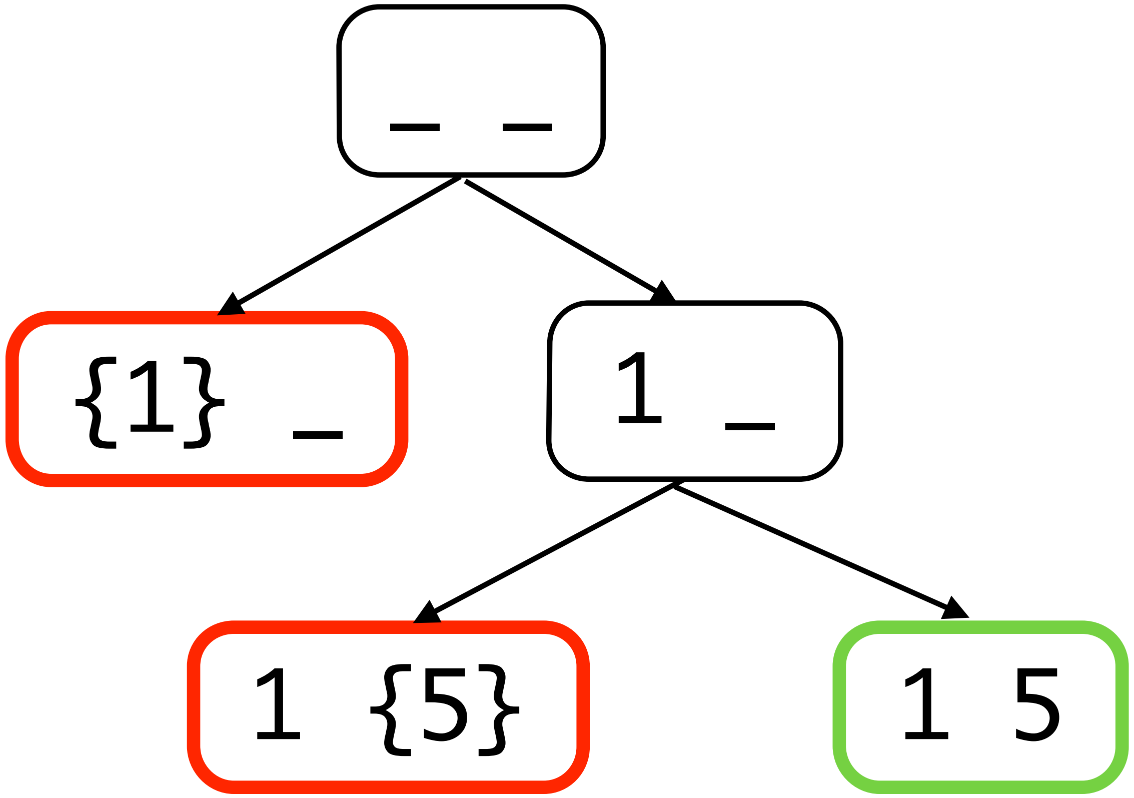
Example

```
f :: Int -> Int -> Int
f 1 5 = 1
f 2 _ = 2
f 1 _ = 3
```

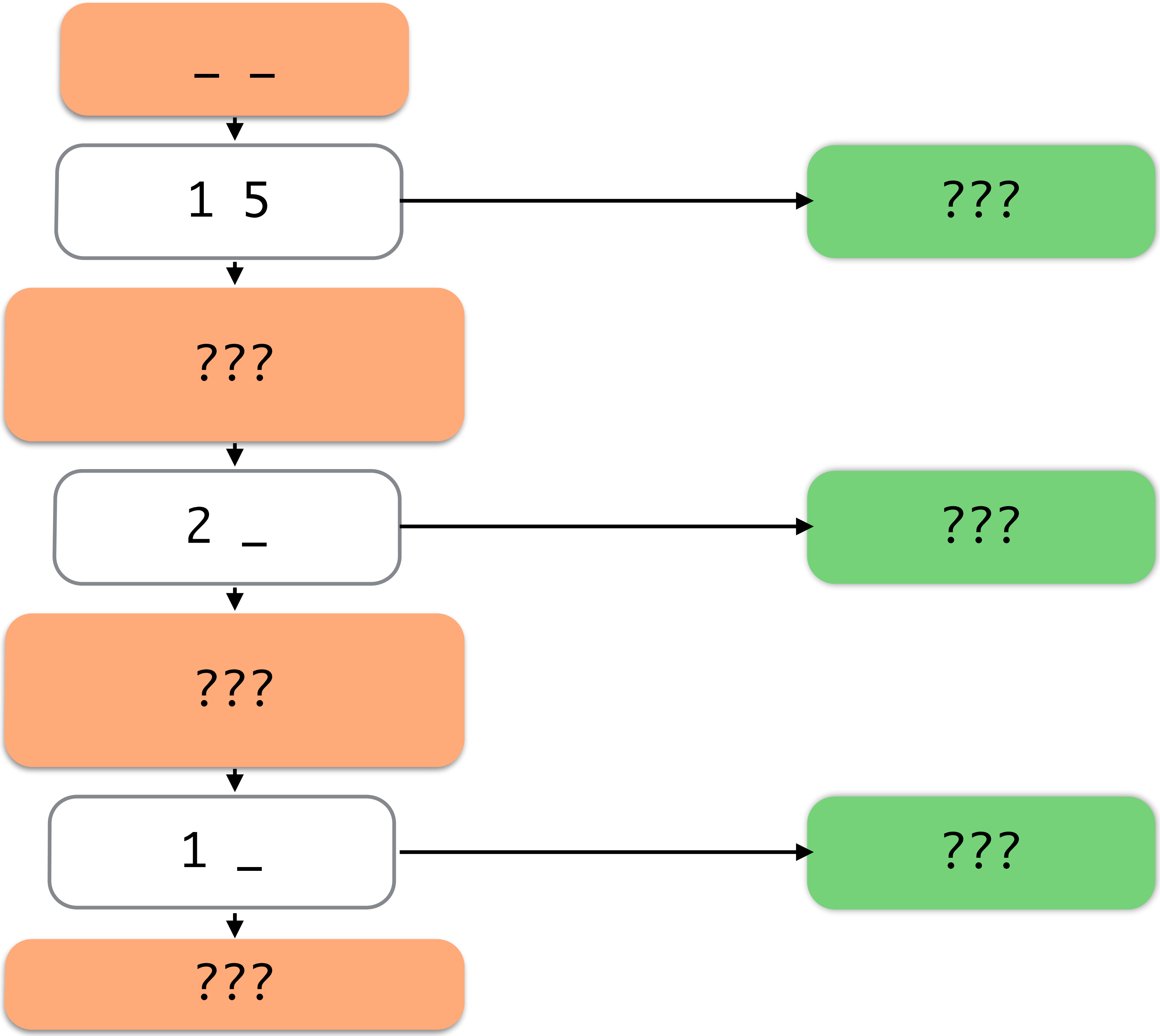


Example

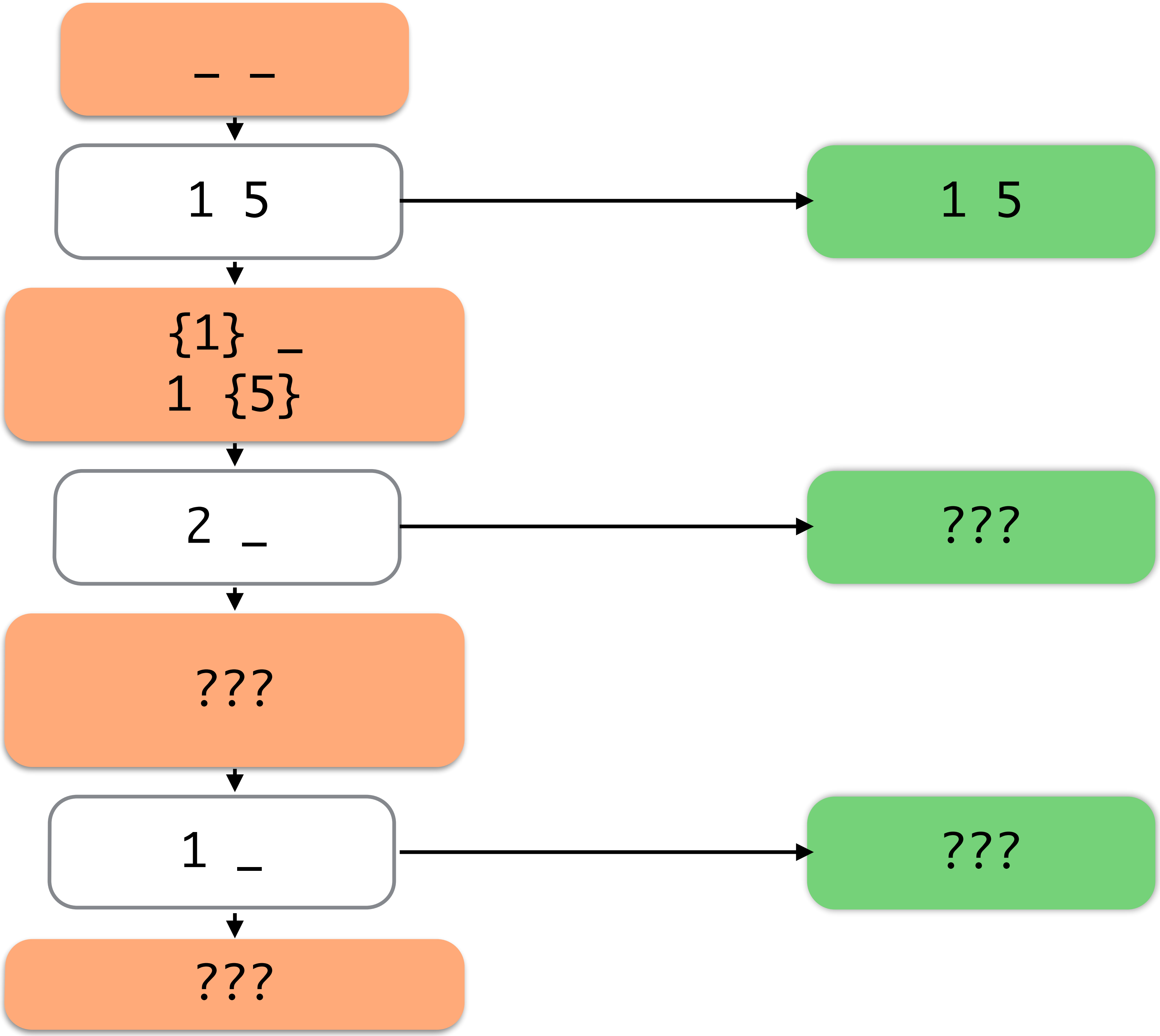
```
f :: Int -> Int -> Int
f 1 5 = 1
f 2 _ = 2
f 1 _ = 3
```



Example



Example



_ _

1 5

1 5

{1} _
1 {5}

2 _

???

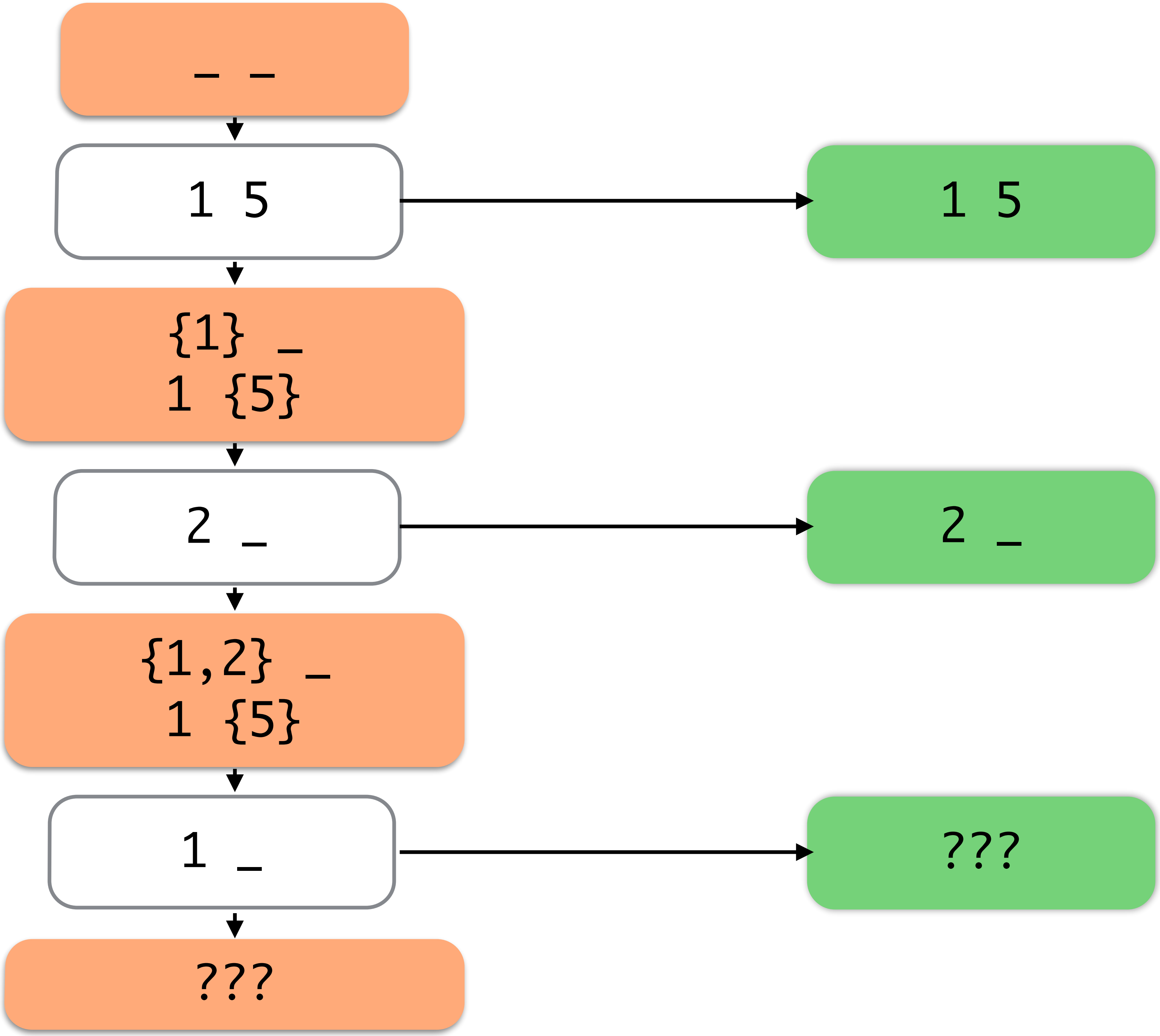
???

1 _

???

???

Example



_ _

1 5

1 5

{1} _
1 {5}

2 _

2 _

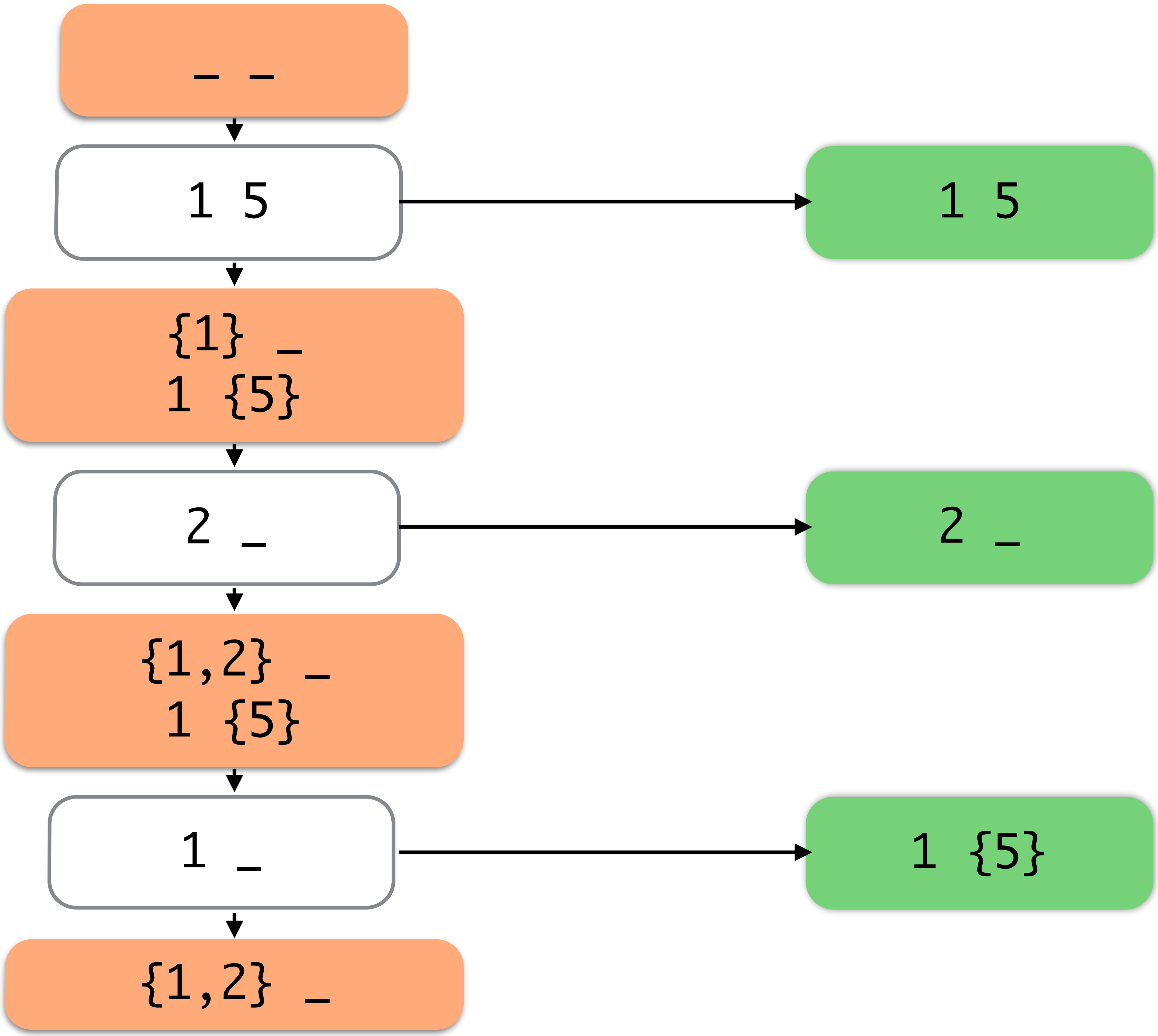
{1,2} _
1 {5}

1 _

???

???

Example



_ _

1 5

1 5

{1} _
1 {5}

2 _

2 _

{1,2} _
1 {5}

1 _

1 {5}

{1,2} _

Guards

Simple cases

```
isZero :: Int -> Bool
isZero x | x == 0 = True
isZero x | x /= 0 = False
```

Simple cases

```
isZero :: Int -> Bool
isZero x | x == 0 = True
isZero x | x /= 0 = False

total?
```

Simple cases

```
isZero :: Int -> Bool
isZero x | x == 0 = True
isZero x | x /= 0 = False
```

```
total?
```

```
instance Eq Int where
  _ == _ = False
  _ /= _ = False
```

NO!

Example

```
f :: List Int -> Int
f (Cons x xs) | x < 0  = 1
f (Cons y ys) | y == 1 = 2
f _                    = 3
```

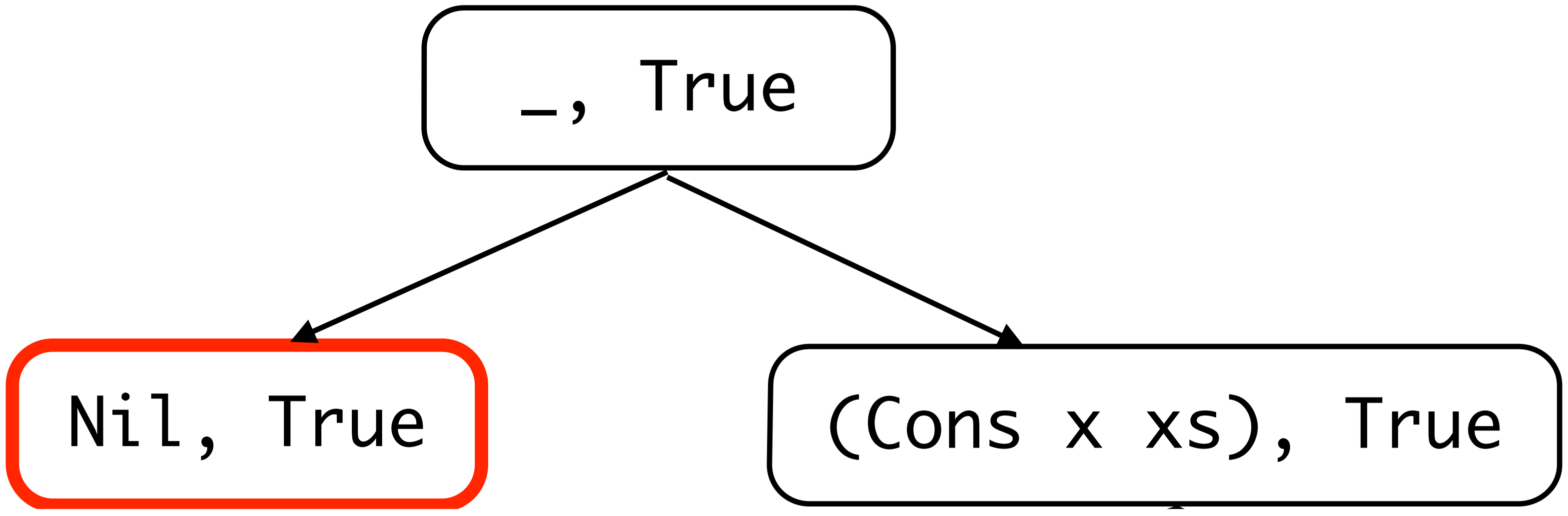
Example

```
f :: List Int -> Int
f (Cons x xs) | x < 0  = 1
f (Cons y ys) | y == 1 = 2
f _                    = 3
```

```
_, True
```

Example

```
f :: List Int -> Int
f (Cons x xs) | x < 0  = 1
f (Cons y ys) | y == 1 = 2
f _                    = 3
```



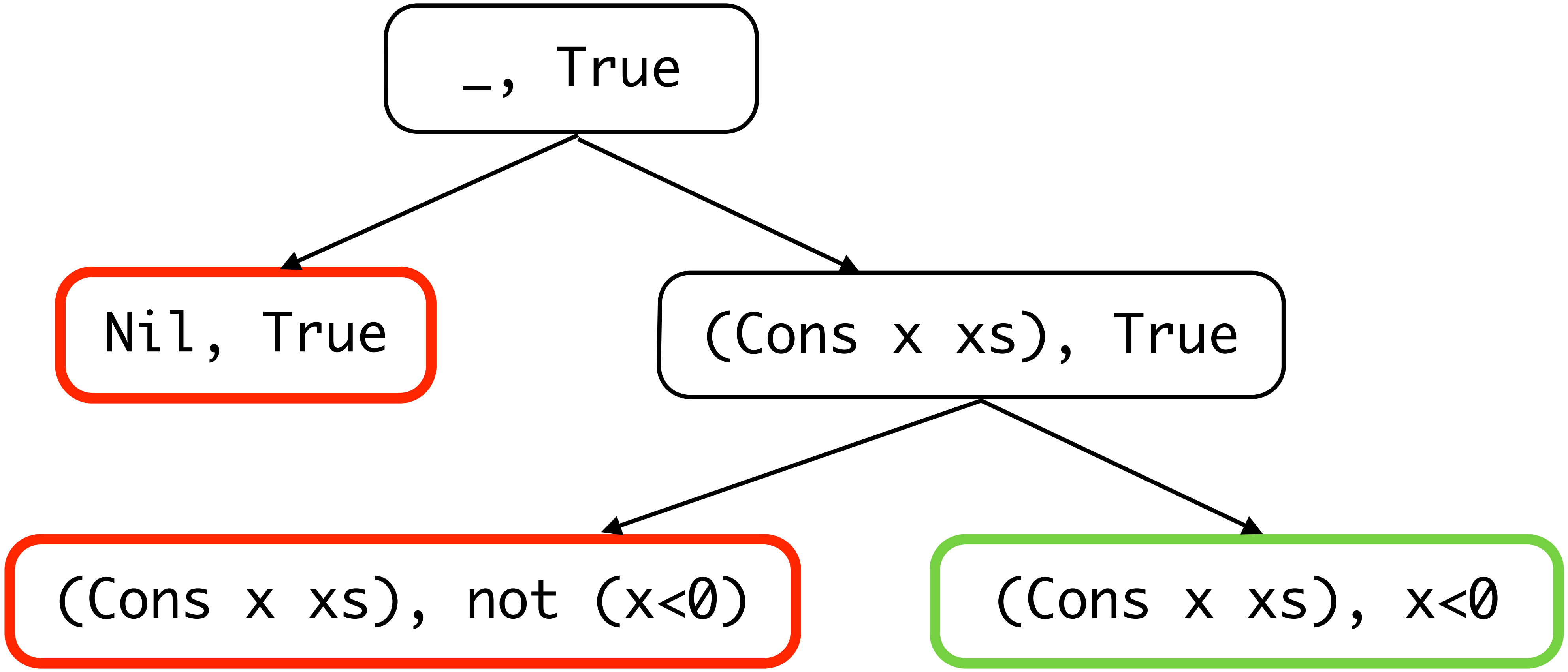
_, True

Nil, True

(Cons x xs), True

Example

```
f :: List Int -> Int
f (Cons x xs) | x < 0  = 1
f (Cons y ys) | y == 1 = 2
f _                    = 3
```



_, True

Nil, True

(Cons x xs), True

(Cons x xs), not (x<0)

(Cons x xs), x<0

Example

```
f :: List Int -> Int
f (Cons x xs) | x < 0  = 1
f (Cons y ys) | y == 1 = 2
f _                    = 3
```

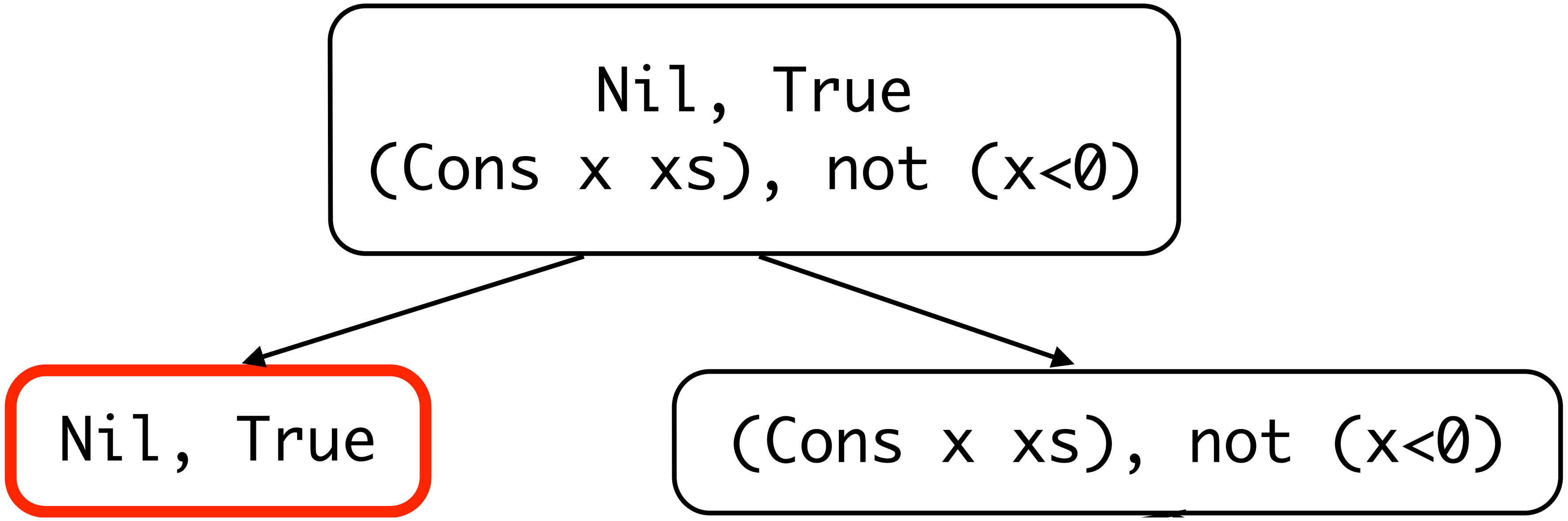
Example

```
f :: List Int -> Int
f (Cons x xs) | x < 0  = 1
f (Cons y ys) | y == 1 = 2
f _                    = 3
```

```
Nil, True
(Cons x xs), not (x<0)
```

Example

```
f :: List Int -> Int
f (Cons x xs) | x < 0  = 1
f (Cons y ys) | y == 1 = 2
f _                    = 3
```



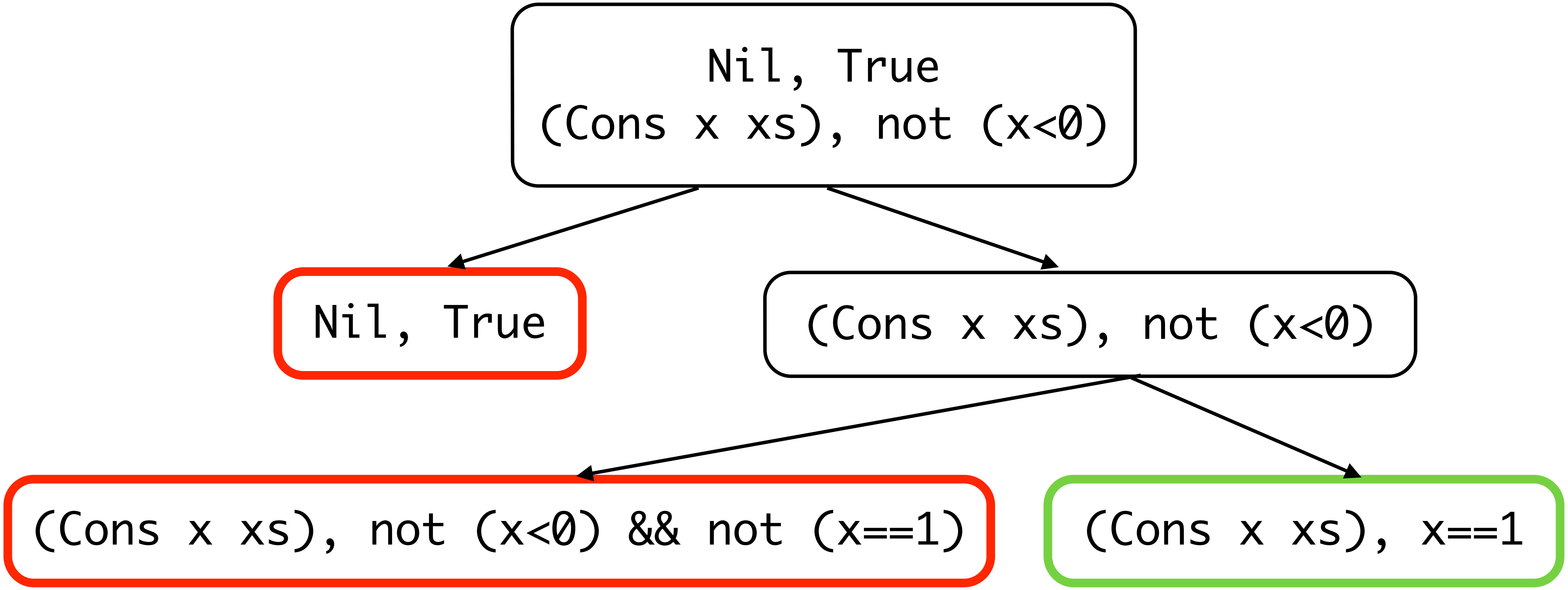
Nil, True
(Cons x xs), not (x<0)

Nil, True

(Cons x xs), not (x<0)

Example

```
f :: List Int -> Int
f (Cons x xs) | x < 0  = 1
f (Cons y ys) | y == 1 = 2
f _                    = 3
```



Nil, True
(Cons x xs), not (x<0)

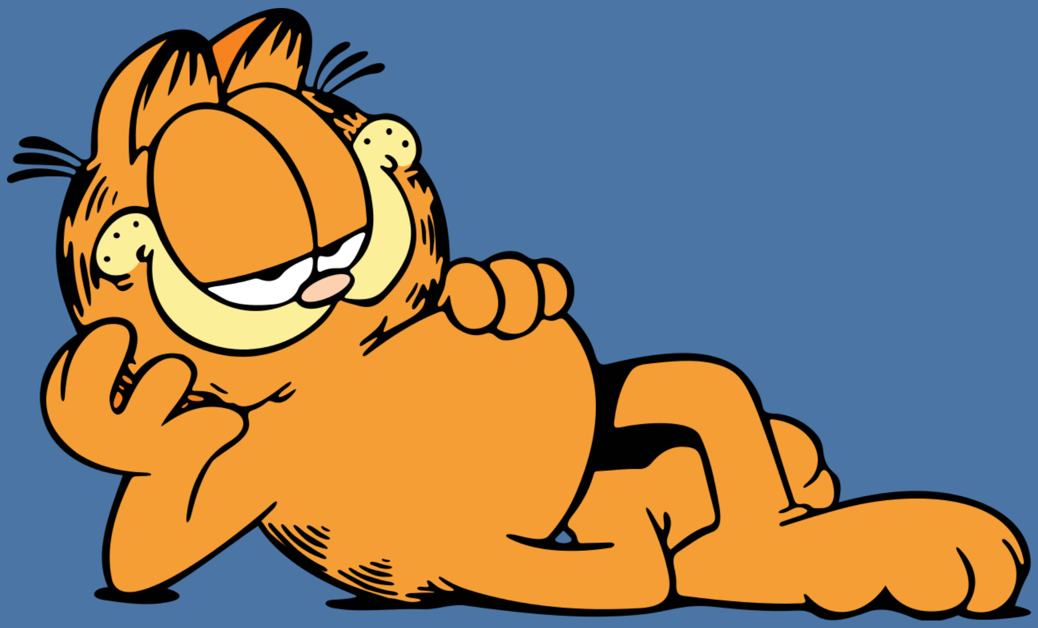
Nil, True

(Cons x xs), not (x<0)

(Cons x xs), not (x<0) && not (x==1)

(Cons x xs), x==1

Laziness



Laziness

```
f1 :: Bool -> Bool -> Int
f1 _    True = 1
f1 True True = 2
f1 _    _    = 3
```

Laziness

```
f1 :: Bool -> Bool -> Int
f1 _    True = 1
f1 True True = 2
f1 _    _    = 3
```

Laziness

```
f1 :: Bool -> Bool -> Int
f1 _    True = 1
f1 True True = 2
f1 _    _    = 3

ghci> f1 (error "1st”) False
```

Laziness

```
f1 :: Bool -> Bool -> Int
f1 _    True = 1
f1 True True = 2
f1 _    _    = 3

ghci> f1 (error "1st”) False
*** Exception: 1st
```

Laziness

```
f2 :: Bool -> Bool -> Int
f2 _    True = 1
f2 _    _    = 3
```

Laziness

```
f2 :: Bool -> Bool -> Int
f2 _    True = 1
f2 _    _    = 3

ghci> f2 (error "1st”) False
```

Laziness

```
f2 :: Bool -> Bool -> Int
f2 _    True = 1
f2 _    _    = 3
```

```
ghci> f2 (error "1st”) False
3
```

Laziness meets GADTs

```
data F :: -> * -> * where
  F1 :: F Int
  F2 :: F Char

data G :: -> * -> * where
  G1 :: G Int
  G2 :: G Bool
```

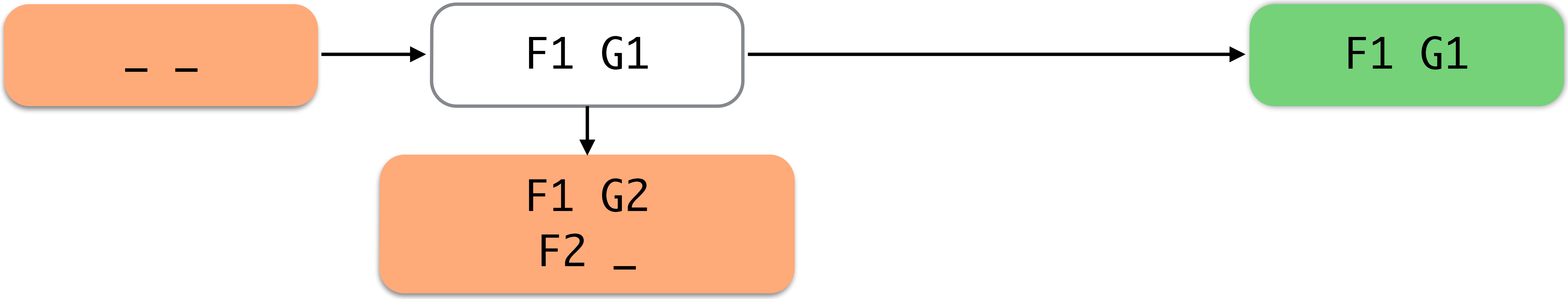
Laziness meets GADTs

```
f :: F a -> G a -> Int
f F1 G1 = 1
f _  G1 = 2
```

_ _

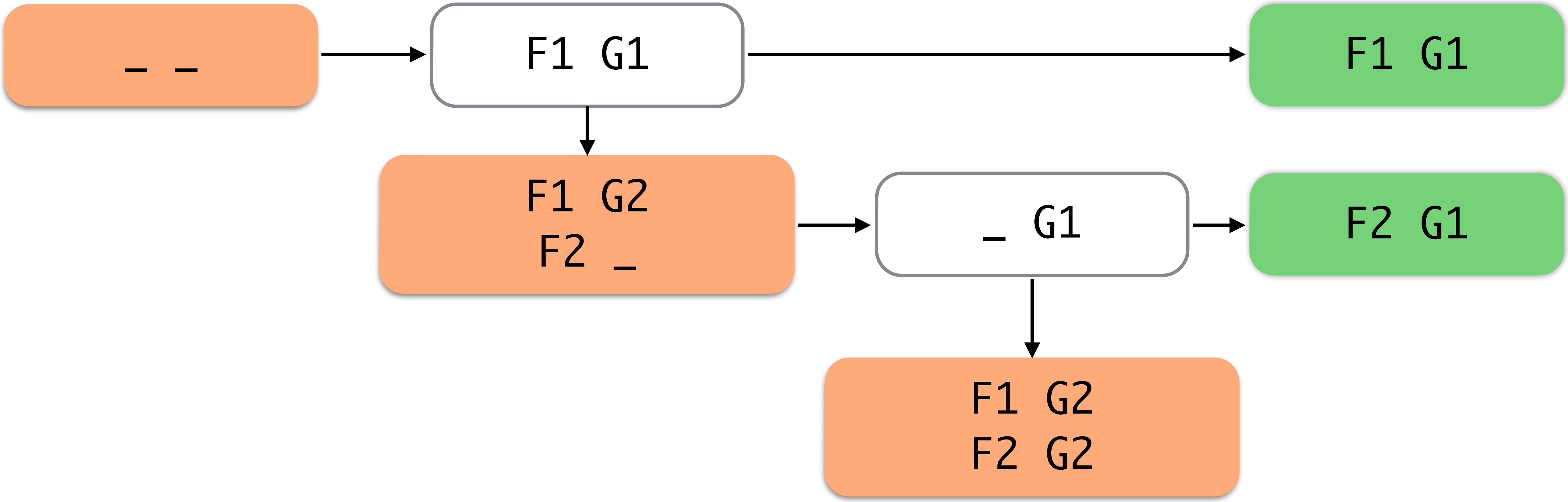
Laziness meets GADTs

```
f :: F a -> G a -> Int
f F1 G1 = 1
f _  G1 = 2
```



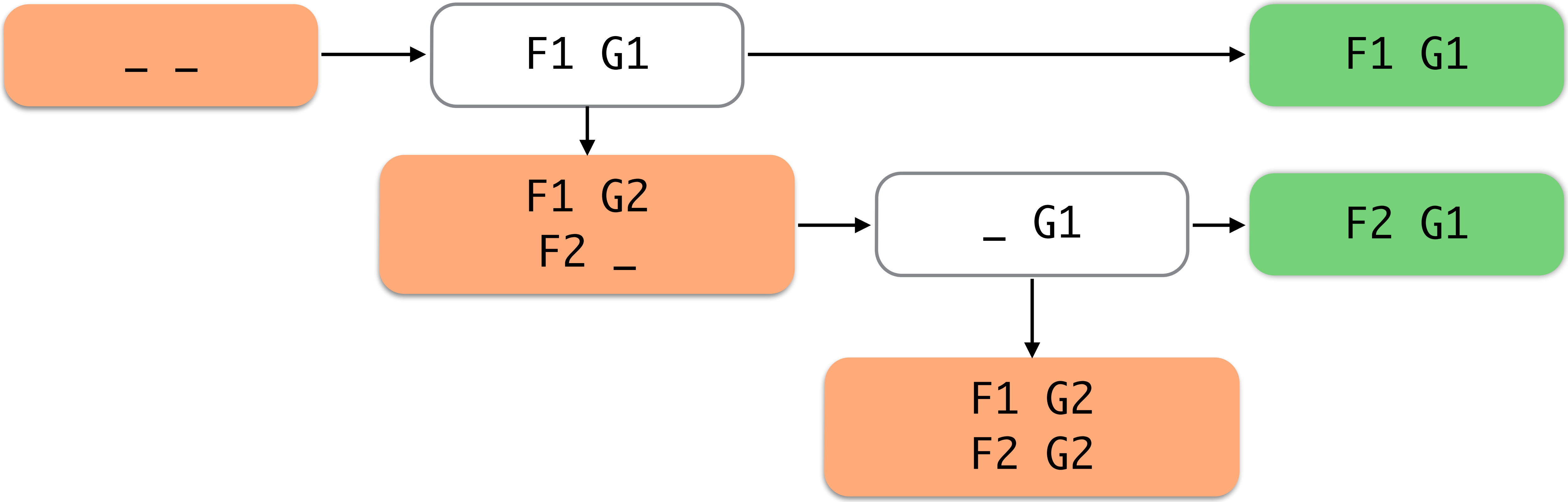
Laziness meets GADTs

```
f :: F a -> G a -> Int
f F1 G1 = 1
f _  G1 = 2
```



Laziness meets GADTs

```
f :: F a -> G a -> Int
f F1 G1 = 1
f _  G1 = 2 -- eliminates (F2 _|_)
```



Three cases

A clause may either

1. Cover some cases
2. Cover no cases
   - A. Does not force evaluation
   - B. Forces evaluation of some arguments

Three cases

A clause may either

1. Cover some cases
2. Cover no cases
   1. Does not force evaluation
   2. Forces evaluation of some arguments

Useful

Three cases

A clause may either

1. Cover some cases
2. Cover no cases
    A. Does not force evaluation
    B. Forces evaluation of some arguments

Useful

Redundant

Three cases

A clause may either

1. Cover some cases
2. Cover no cases
   - A. Does not force evaluation
   - B. Forces evaluation of some arguments

Useful

Redundant

???

Three cases

A clause may either

1. Cover some cases
2. Cover no cases
   A. Does not force evaluation
   B. Forces evaluation of some arguments

Useful

Redundant

???

Detection: Branching of the algorithm