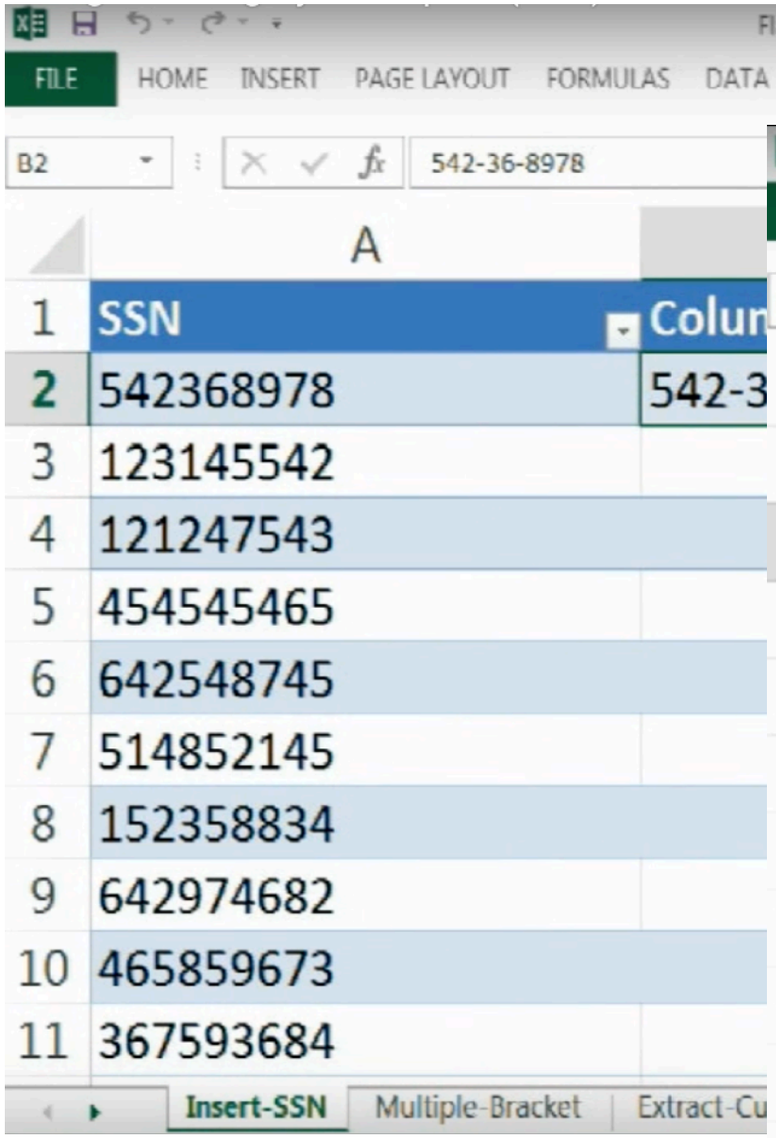
“Information technology has been praised as a labor saver and cursed as a destroyer of obsolete jobs. But the entire edifice of modern computing rests on a fundamental irony: **the software that makes it all possible is, in a very real sense, handmade.** Every miraculous thing computers can accomplish begins with a human programmer entering lines of code by hand, character by character.”

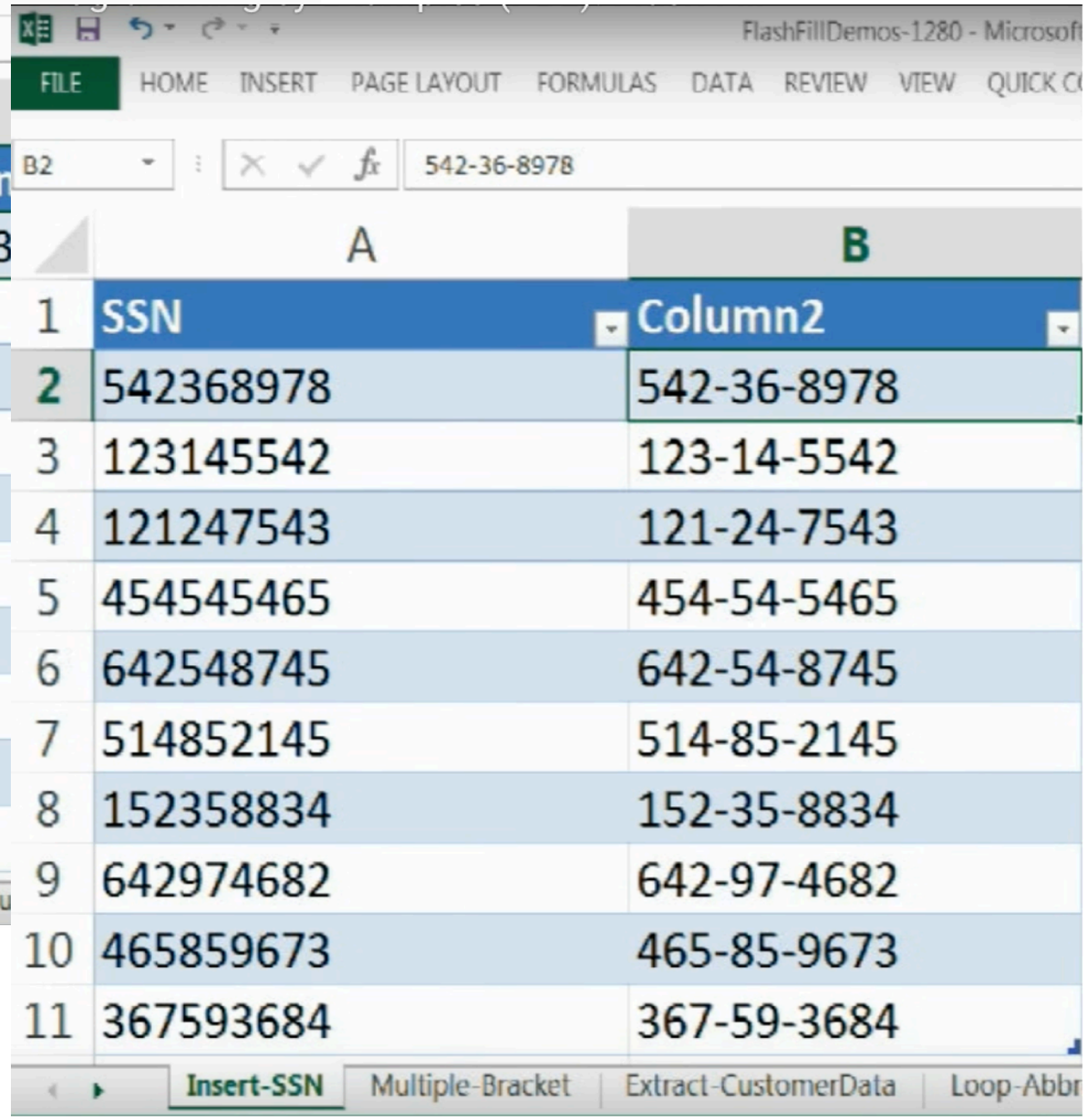
— Moshe Y. Vardi

An Introduction to Program Synthesis

魏家一
wjydzh1@gmail.com

An example: Flash Fill in Excel 2013





Flash Fill

An example: Flash Fill in Excel 2013

	A	B
1	Names	Initia
2	Neil Lieber	N L
3	Mathew Prisco	
4	Althea Bertin	
5	Kelly Gamblin	
6	Chandra Valenzula	
7	Cody Castillon	
8	Tyrone Brazier	
9	Althea Buhl	
10	Dollie Munsey	
11	Allyson Phou	

	A	B
1	Names	Initials
2	Neil Lieber	N L
3	Mathew Prisco	M P
4	Althea Bertin	A B
5	Kelly Gamblin	K G
6	Chandra Valenzula	C V
7	Cody Castillon	C C
8	Tyrone Brazier	T B
9	Althea Buhl	A B
10	Dollie Munsey	D M
11	Allyson Phou	A P

Flash Fill

An example: Flash Fill in Excel 2013

	Column1	Column2
2	[CPT-00350	[CPT-00350]
3	[CPT-00340	
4	[CPT-11536	
5	[CPT-11222	
6	[CPT-115]	
7	[CPT-1153622]	
8	[CPT-00340]	
9	[CPT-11536]	
10	[CPT-11222]	
11	[CPT-115	
12	[CPT-1163	

Add a ‘]’

Some ‘]’ are missing

An example: Flash Fill in Excel 2013

	Column1	Column2
2	[CPT-00350	[CPT-00350]
3	[CPT-00340	[CPT-00340]
4	[CPT-11536	[CPT-11536]
5	[CPT-11222	[CPT-11222]
6	[CPT-115]	[CPT-115]]
7	[CPT-1153622]	[CPT-1153622]]
8	[CPT-00340]	[CPT-00340]]
9	[CPT-11536]	[CPT-11536]]
10	[CPT-11222]	[CPT-11222]]
11	[CPT-115	[CPT-115]
12	[CPT-1163	[CPT-1163]

Too many!

An example: Flash Fill in Excel 2013

	Column1	Column2
2	[CPT-00350	[CPT-00350]
3	[CPT-00340	[CPT-00340]
4	[CPT-11536	[CPT-11536]
5	[CPT-11222	[CPT-11222]
6	[CPT-115]	[CPT-115]
7	[CPT-1153622]	[CPT-1153622]]
8	[CPT-00340]	[CPT-00340]]
9	[CPT-11536]	[CPT-11536]]
10	[CPT-11222]	[CPT-11222]]
11	[CPT-115	[CPT-115]
12	[CPT-1163	[CPT-1163]

Fix it

Then rerun Flash Fill

An example: Flash Fill in Excel 2013

	Column1	Column2
2	[CPT-00350	[CPT-00350]
3	[CPT-00340	[CPT-00340]
4	[CPT-11536	[CPT-11536]
5	[CPT-11222	[CPT-11222]
6	[CPT-115]	[CPT-115]
7	[CPT-1153622]	[CPT-1153622]
8	[CPT-00340]	[CPT-00340]
9	[CPT-11536]	[CPT-11536]
10	[CPT-11222]	[CPT-11222]
11	[CPT-115	[CPT-115]
12	[CPT-1163	[CPT-1163]

Correct!

What is Program Synthesis?

- Program Synthesis aims to automate (tedious part of) programming.

- A programmer only tells the computer what he wants (through formal or informal specifications), and leaves the details of ***how*** to the computer to figure out.

- Flash Fill is an instance of Example-Driven Program Synthesis (also called Inductive Program Synthesis)

- There is also another kind of program synthesis: Deductive Program Synthesis (usually using formal specifications)

The Leon Framework

- Leon is an automated system for **synthesizing** and **verifying** functional **Scala** programs, developed by EPFL

- Leon uses Scala constructs ***require*** and ***ensuring*** to document preconditions and postconditions of functions

```
def max(x: Int, y: Int): Int = {
    val d = x - y
    if (d > 0) x
    else y
  } ensuring(res =>
    x <= res && y <= res && (res == x || res == y))
```

The Leon Framework - Verification

- The code seems to work correctly. However, Leon automatically finds that it is not correct, showing us a counter-example inputs, such as

```
x -> -1639624704
y -> 1879048192
```

- The problem is due to overflow of 32-bit integers!

```
def max(x: Int, y: Int): Int = {
    val d = x - y
    if (d > 0) x
    else y
  } ensuring(res =>
    x <= res && y <= res && (res == x || res == y))
```

The Leon Framework - Verification

- To use unbounded integers, we simply change the types to *BigInt*, obtaining a program that verifies (and, as expected, passes all the test cases)

```
def max(x: BigInt, y: BigInt): BigInt = {
    val d = x - y
    if (d > 0) x
    else y
  } ensuring(res =>
    x <= res && y <= res && (res == x || res == y))
```

The Leon Framework - Synthesis

- Leon can also **synthesize** programs from given specifications

- We omit the implementation of *max*, using only a placeholder `???[BigInt]` indicating we are looking for an unknown implementation of an integer type

```
def max(x: BigInt, y: BigInt): BigInt = {
  ???[BigInt]
} ensuring(res =>
  x <= res && y <= res && (res == x || res == y))
```

- Leon can then automatically generate an implementation that satisfies this specification, such as

```
if (x <= y) x
else y
```

The Escher Algorithm

Synthesizing recursive programs

The *Escher* Algorithm

- An example-driven synthesis algorithm aims to synthesize recursive programs
- Uses a very simple target language (only argument variable, if-then-else statement and function application)
- Initially described in the CAV'13 paper *Recursive Program Synthesis*
- I'm going to introduce a **modified version** of this algorithm called *Escher-Scala* (the project I'm currently working on)

An *Escher-Scala* Example

- Suppose we want to synthesize a list reverse function
- First, we need to provide a function signature

```
reverse(@xs: List['0]): List['0]
```

- Haskell equivalent:

```
reverse :: List t0 -> List t0
reverse xs = ...
```

An *Escher-Scala* Example

- Then we provide some input-output examples

```
Initial Examples (3):
([]) -> []
([1, 2]) -> [2, 1]
([1, 2, 3]) -> [3, 2, 1]
```

- During the synthesis, the algorithm also asks results for some **additional input cases**, we say this algorithm uses an Oracle.

```
Additional examples provided (9):
([2, 3]) -> [3, 2]
([2]) -> [2]
([1, 1]) -> [1, 1]
([3, 3]) -> [3, 3]
([1]) -> [1]
([3]) -> [3]
([3, 2]) -> [2, 3]
([1, 3]) -> [3, 1]
 ...(1 more not shown)...
```

An *Escher-Scala* Example

- The program Escher-Scala found:

```
Program found:

reverse(@xs: List['0]): List['0] =
  if isNil(@xs)
  then @xs
  else concat(reverse(tail(@xs)), cons(head(@xs), nil()))
```

- Haskell equivalent:

```
reverse xs =
  if xs == []
  then xs
  else reverse (tail xs) ++ (head xs : [])
```

An *Escher-Scala* Example

- The program Escher-Scala found:

```
Program found:

reverse(@xs: List['0]): List['0] =
  if isNil(@xs)
  then @xs
  else concat(reverse(tail(@xs)), cons(head(@xs), nil()))
```

- It took the algorithm only 219ms to find this example

An *Escher-Scala* Example

- The program Escher-Scala found:

```
Program found:

reverse(@xs: List['0]): List['0] =
  if isNil(@xs)
  then @xs
  else concat(reverse(tail(@xs)), cons(head(@xs), nil()))
```

- It took the algorithm only 219ms to find this example

- Actually, for easy testing, instead of manually type in those Oracle inputs, I used reference implementations

The Component Set

- You probably have noticed that Escher used some library functions (Components) in the previous example:

  isNil, head, tail, concat …

- Escher always works under a Component Set — it only uses components (library functions) from this Component Set to synthesize programs.

- The Component Set is not fixed, they can be customized for different synthesis tasks

The Standard Component Set

- All the following synthesis tasks were tested under a standard Component Set (26 components), possibly with additional components.

```
// boolean
T, F, and, or, not, equal, isNil, isNonNeg,
// list
head, tail, cons, concat, nil,
// integer
zero, inc, dec, neg, length, plus, div2
```

```
//binary tree
createLeaf, createNode, isLeaf, treeValue, treeLeft, treeRight
```

The Standard Component Set

- It is also quite easy to implement new components and data types in Escher-Scala

```scala
case object TPair extends TypeConstructor {
  val name: String = "Pair"

  val arity: Int = 2
}
```

```scala
case class ValueTree(value: BinaryTree[TermValue]) extends TermValue{
  override def matchTypeAux(ty: Type, freeId: () => Int): Option[TypeSubst] = {...}

  override def show: String = {...}

  val sizeCompare: PartialFunction[TermValue, Boolean] = {...}
}
```

More than reverse

- *Escher-Scala* can also synthesize many other interesting programs!

- fib: Fibonacci numbers

```
fib(@n: Int): Int =
  if isNonNeg(inc(neg(@n)))
  then inc(zero())
  else plus(fib(dec(@n)), fib(dec(dec(@n))))
```

```
fib n =
  if (-n+1) >= 0        -- n <= 1
  then 1
  else fib (n-1) + fib (n-2)
```

```
Initial Examp
(0) -> 1
(1) -> 1
(2) -> 2
(-3) -> 1
(3) -> 3
(4) -> 5
(5) -> 8
(6) -> 13
Additional ex
(-2) -> 1
(-5) -> 1
(-1) -> 1
(-4) -> 1
```

More than reverse

- *Escher-Scala* can also synthesize many other interesting programs!

- nodesAtLevel: All nodes of a binary tree at some depth

```
nodesAtLevel(@tree: Tree['0], @level: Int): List['0] =
  if or(isLeaf(@tree), not(isNonNeg(@level)))
  then nil()
  else if isNonNeg(neg(@level))
       then cons(treeValue(@tree), nil())
       else concat(nodesAtLevel(treeLeft(@tree), dec(@level)),
  nodesAtLevel(treeRight(@tree), dec(@level)))
```

Contains and Dedup

- contains: Whether an element is in a list

```
contains(@xs: List['0], @x: '0): Bool =
  if isNil(@xs)
  then F()
  else if equal(@x, head(@xs))
       then T()
       else contains(tail(@xs), @x)
```

- dedup: Remove all duplicate elements of a list

```
dedup(@xs: List['0]): List['0] =
  if isNil(@xs)
  then @xs
  else if contains(tail(@xs), head(@xs))
       then dedup(tail(@xs))
       else cons(head(@xs), dedup(tail(@xs)))
```

Cartesian Product

- cartesian: Cartesian Product of two lists
- Example Set:

```
Initial Examples (4):
([], [2, 3, 4]) -> []
([5], []) -> []
([5], [7, 8, 9]) -> [(5, 7), (5, 8), (5, 9)]
([2, 3], [4, 5]) -> [(2, 4), (2, 5), (3, 4), (3, 5)]
```

Cartesian Product in detail

- The program Escher-Scala found:

```
cartesian(@xs: List['0], @ys: List['1]): List[Pair['0,'1]] =
  if or(isNil(@ys), isNil(@xs))
  then nil()
  else if isNil(tail(@xs))
       then cons(createPair(head(@xs), head(@ys)), cartesian(@xs, tail(@ys)))
       else concat(cartesian(cons(head(@xs), nil()), @ys), cartesian(tail(@xs), @ys))
```

- Rewrite it into a more readable form:

```
cartesian :: List t0 -> List t1 -> List (t0, t1)
cartesian xs ys =
  if xs == [] || ys == []
  then []
  else if tail xs == []
    then (head xs, head ys) : cartesian xs (tail ys)
    else cartesian [head xs] ys ++ cartesian (tail xs) ys
```

- We will come back to this example in later discussions

How it works

- Now, let me introduce the algorithm in detail
- Remember our synthesis is example-driven
- I’m going to use the list length example
- Function signature: `length(@xs: List[‘0]): Int`
- Input-output examples:

```
([]) -> 0
([1]) -> 1
([2,3,4]) -> 3
```

What we want

- Basic Objective: Give the 3 input-output examples, we want to find a recursive program that passes all of them

- We can represent the 3 inputs as an input vector

```
{([]),([1]),([2,3,4])}
```

- Since we have only one value for each input:

```
{[],[1],[2,3,4]}
```

- We also represent our desired outputs as output vector

```
{0, 1, 3}
```

Terms

- Function signature: `length(@xs: List[‘0]): Int`
- Using only components and input variables (`@xs` in this example), we can write down a series of terms (expressions with valid types), each term has a corresponding output vector

```
zero() ~> {0,0,0}
@xs ~> {[],[1],[2,3,4]}
inc(zero()) ~> {1,1,1}
tail(@xs) ~> {Err,[],[3,4]}
isNil(@xs) ~> {T,F,F}
length(tail(@xs)) ~> {Err,0,2}
.. ..
```

- Note that we also treat length as a special component, to calculate its output vector, the algorithm needs to ask Oracle for additional inputs.

Target Program

- If we can find a term with the desired output vector, then we are done!

- Otherwise, we may need to combine those terms into programs

- Programs are either terms or if-then-else expressions with the following shape:

```
if term1 then term2
else if term3 then term4
else if term5 then term6
.. ..
else termN
```

- Or inductively:

```
P := term
   | if term then term else P
```

What we really want

- We can always find some “trivial programs” that pass all examples, for our length example:

```
if isNil(@xs) then zero()
else if isNil(tail(@xs)) then inc(zero())
else inc(inc(inc(zero())))
```

- But the above program is not general enough! It will fail on other inputs like `[1,2]`

- What we really want is programs like this:

```
if isNil(@xs) then zero()
else inc(length(tail(@xs)))
```

- How to avoid finding those “trivial” programs?

- Observation: The correct program is much simpler!

Guiding Heuristic

Occam's Razor

Among competing hypothesis, the one with the fewest assumptions should be selected.

Guiding Heuristic

- Occam's Razor: Among competing hypothesis, the one with the fewest assumptions should be selected.
- If there are multiple programs that pass the example set, the algorithm should return the "simplest" one.
- How to define the simpleness of a program? There is not any mathematically rigorous answer to this question yet.
- But experiment results demonstrate that the size (number of nodes in AST) of target programs is a good choice.

```
if isNil(@xs) then zero()
else inc(length(tail(@xs)))
```

Size = 8

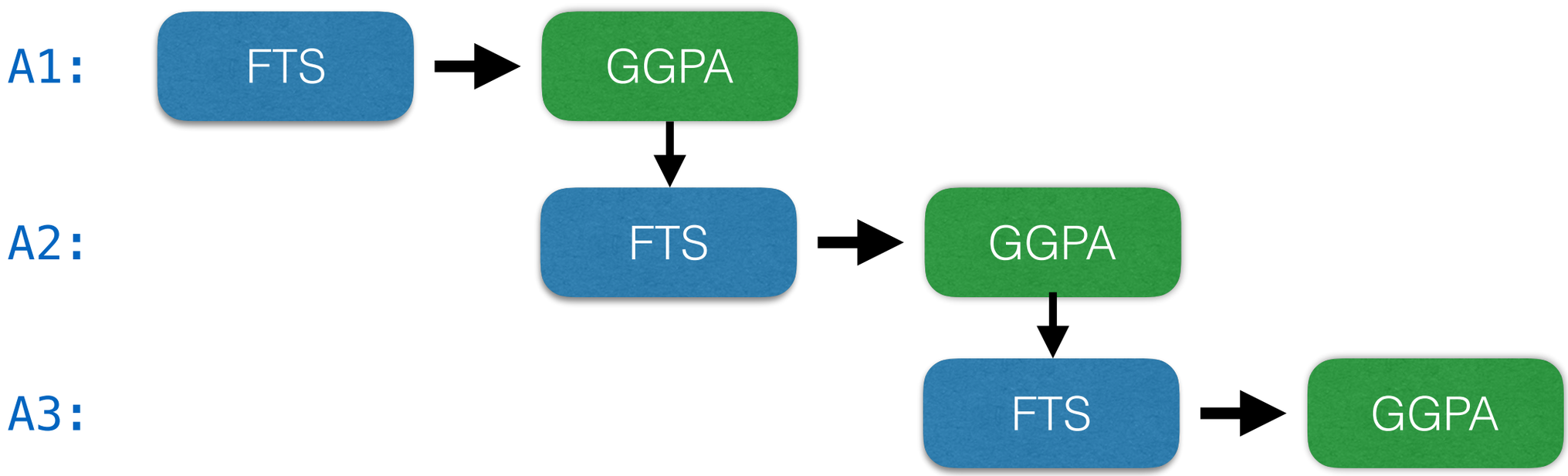
```
if isNil(@xs) then zero()
else if isNil(tail(@xs)) then inc(zero())
else inc(inc(inc(zero())))
```

Size = 14

Algorithm Overview

- The Escher-Scala algorithm consists of two alternating stages: Forward Term Search (FTS) and Goal-Guided Program Assembly (GGPA)

- In the n th alternation, the algorithm first find out all terms of size n (FTS), and then tries to assemble a correct target program using GGPA.



Forward Term Search

- If we want to synthesize term of size S, given a component that takes A arguments (arity = A), FTS searches for A terms whose sizes sum to S-1

- In our length example, at alternation 4, FTS can find all terms we need in order to synthesize a correct target program, along with many other terms:

```
zero() ~> {0,0,0}                          size=1
isNil(@xs) ~> {T,F,F}                      size=2
inc(length(tail(@xs))) ~> {Err,1,3}  size=4
```

Observational Equivalence

- We can't just naively enumerate all possible terms; otherwise there will be too many of them!
- Many syntactically different terms are actually equivalent in semantics (have same meaning), **we only need the simplest one**.
- We can use Observational Equivalence as an approximation to semantical equivalence.

```
{0,0,0}
zero()
dec(inc(zero))
plus(zero(),zero())
.. ..
```

```
{T,F,F}
isNil(@xs)
not(not(isNil(@xs)))
equals(T(),isNil(@xs))
.. ..
```

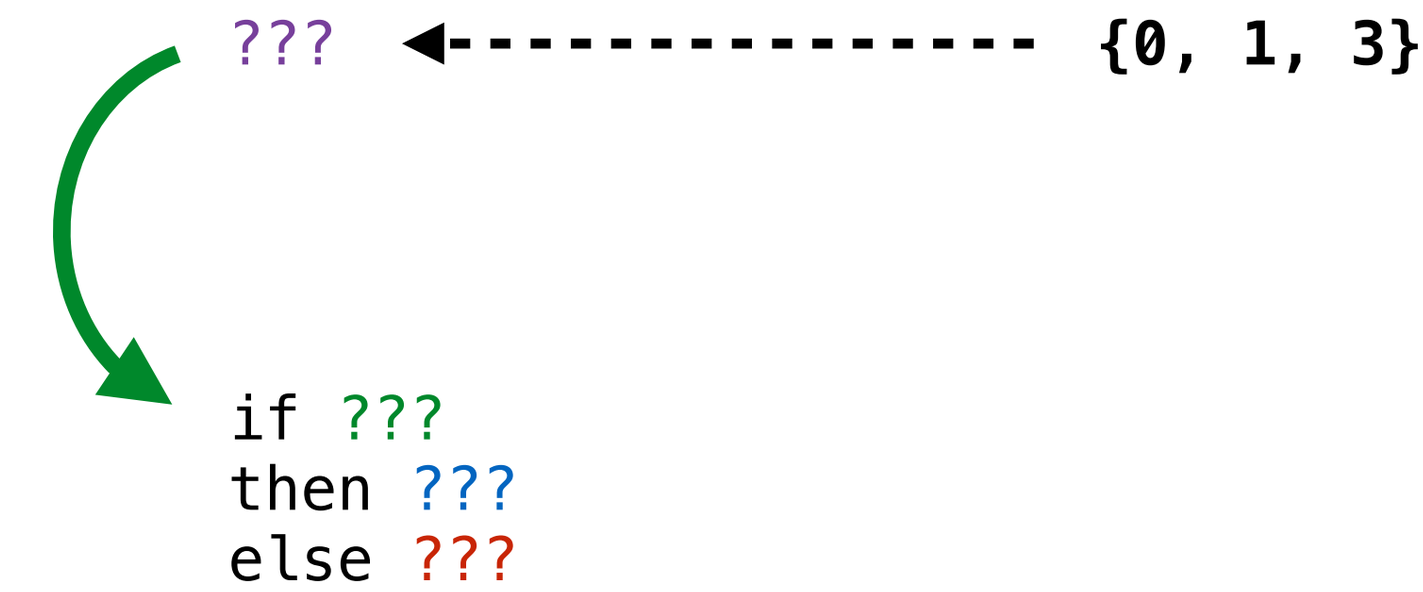
Goal-Guided Program Assembly

- Main idea: If we can't find a term to satisfy a goal, we can split the goal using if-then-else expressions

```
???  <---------------  {0, 1, 3}
```

Goal-Guided Program Assembly

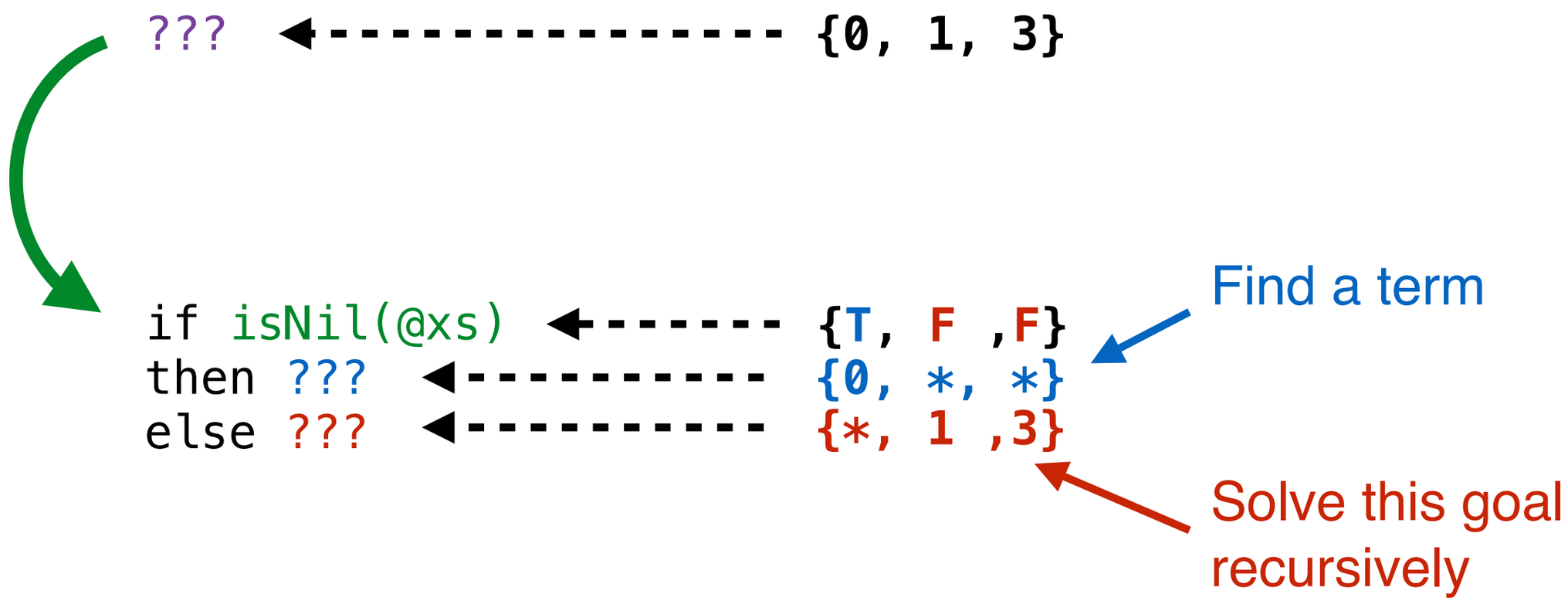
- Main idea: If we can't find a term to satisfy a goal, we can split the goal using if-then-else expressions



Now try to fill these 3 holes

Goal-Guided Program Assembly

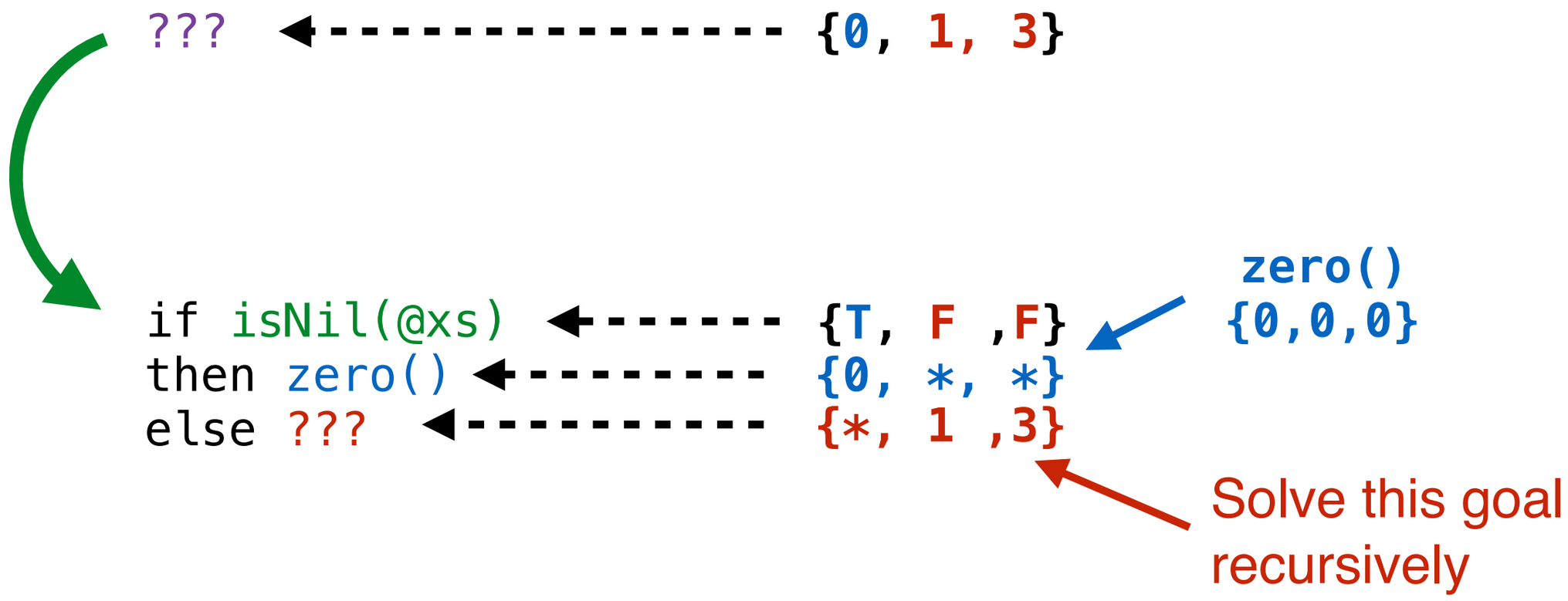
- Main idea: If we can't find a term to satisfy a goal, we can split the goal using if-then-else expressions



Try with bool expressions that can split the goal

Goal-Guided Program Assembly

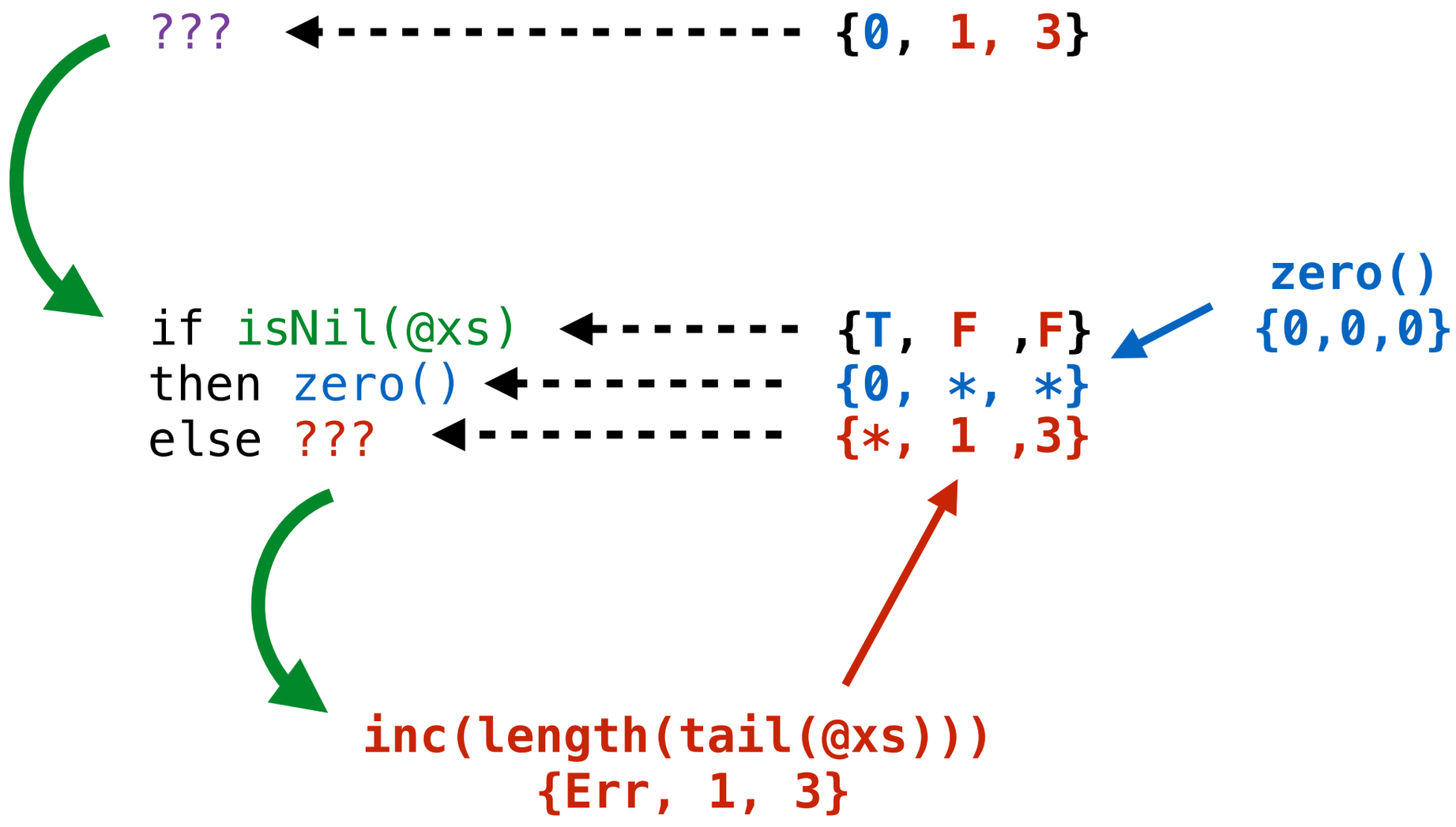
- Main idea: If we can't find a term to satisfy a goal, we can split the goal using if-then-else expressions



Found zero()

Goal-Guided Program Assembly

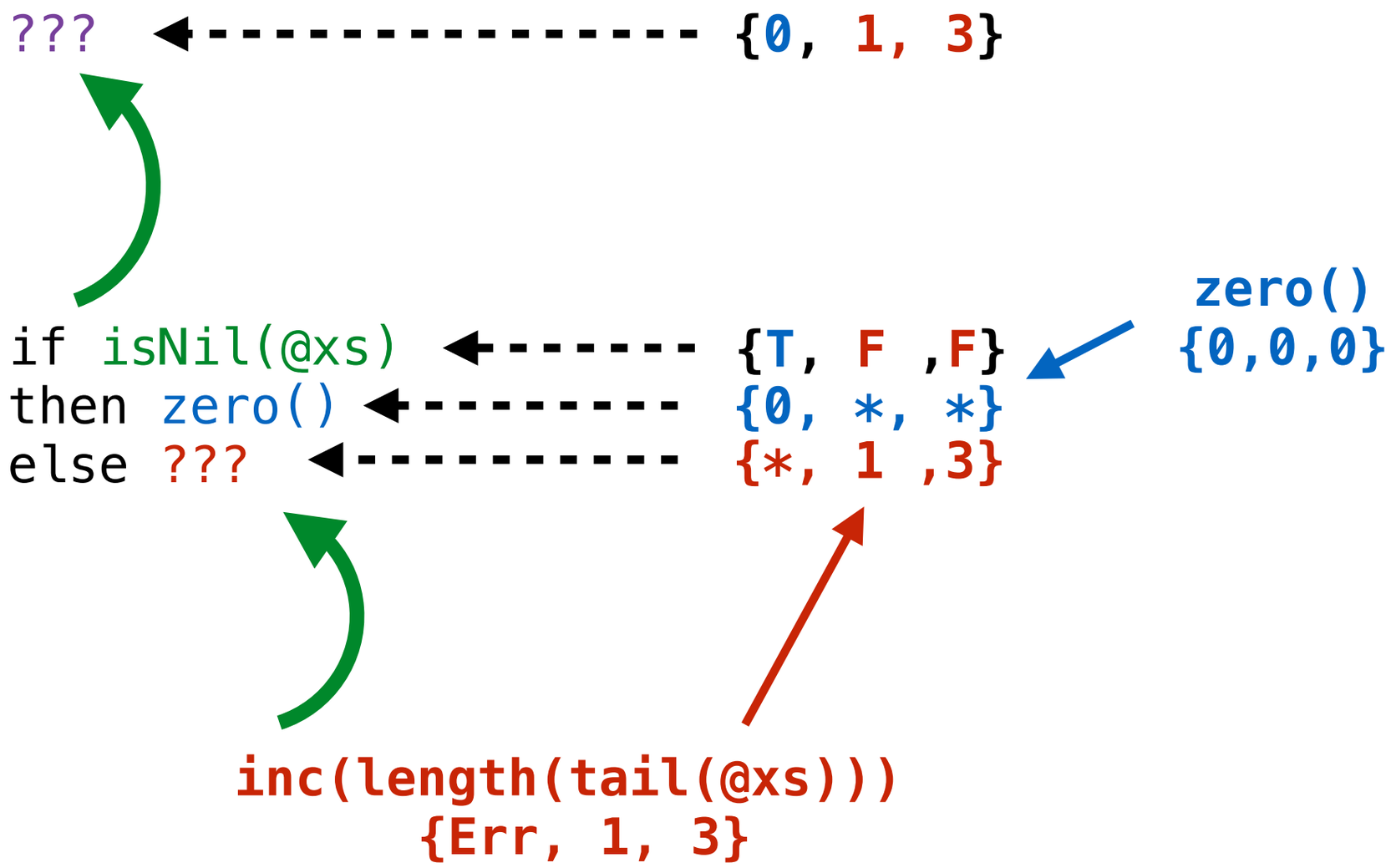
- Main idea: If we can't find a term to satisfy a goal, we can split the goal using if-then-else expressions



Recursively solved

Goal-Guided Program Assembly

- Main idea: If we can't find a term to satisfy a goal, we can split the goal using if-then-else expressions



Program Found!

Goal-Guided Program Assembly

- Main idea: If we can't find a term to satisfy a goal, we can split the goal using if-then-else expressions

- Result:

```
length(@xs)=
  if isNil(@xs)
  then zero()
  else inc(length(tail(@xs)))
```

Avoid Non-terminating programs

- What if the algorithm find terms like this:

```
length(@xs) ~> {0, 1, 3}
```

- Note that this term has the exact output vector we want. But the resulting program **will not terminate**!

```
length(@xs)=
  length(@xs)
```

- There always exists some non-terminating programs that are simpler than the correct, terminating ones!

- To avoid problems like this, Escher requires the argument list of recursive calls must be “smaller” than the previous argument list.

Termination Guarantee

- Intuition: If every time the program calls its self, the size of its argument list becomes smaller, then eventually, its argument list cannot be smaller anymore, thus the program terminates.

```
length([2,3,4])

inc(length([3,4]))

inc(inc(length([4])))

inc(inc(inc(length([]))))

inc(inc(inc(0)))

3
```

Termination Guarantee

- Intuition: If every time the program calls itself, the size of its argument list becomes smaller, then eventually, its argument list cannot be smaller anymore, thus the program terminates.

- To achieve this, we should be able to compare the size of two values of the same data type. And for each data type, there must exists some smallest element.

- For integer, we can define the size as its absolute value.

- For lists, we define the size as its length.

- Then we can define the usual alphabetic order on argument lists.

Check recursive calls

- In our length example, remember we have input vector:

```
@xs ~> {[],[1],[2,3,4]}
```

- For term `@xs`, it we apply length on it, because those three argument lists are the same as the input vector (thus not smaller than it), this results in 3 **Err**s:

```
length(@xs) ~> {Err,Err,Err}
```

- For term `tail(@xs)`

```
tail(@xs) ~> {Err,[],[3,4]}
```

Check recursive calls

- In our length example, remember we have input vector:

```
@xs ~> {[],[1],[2,3,4]}
```

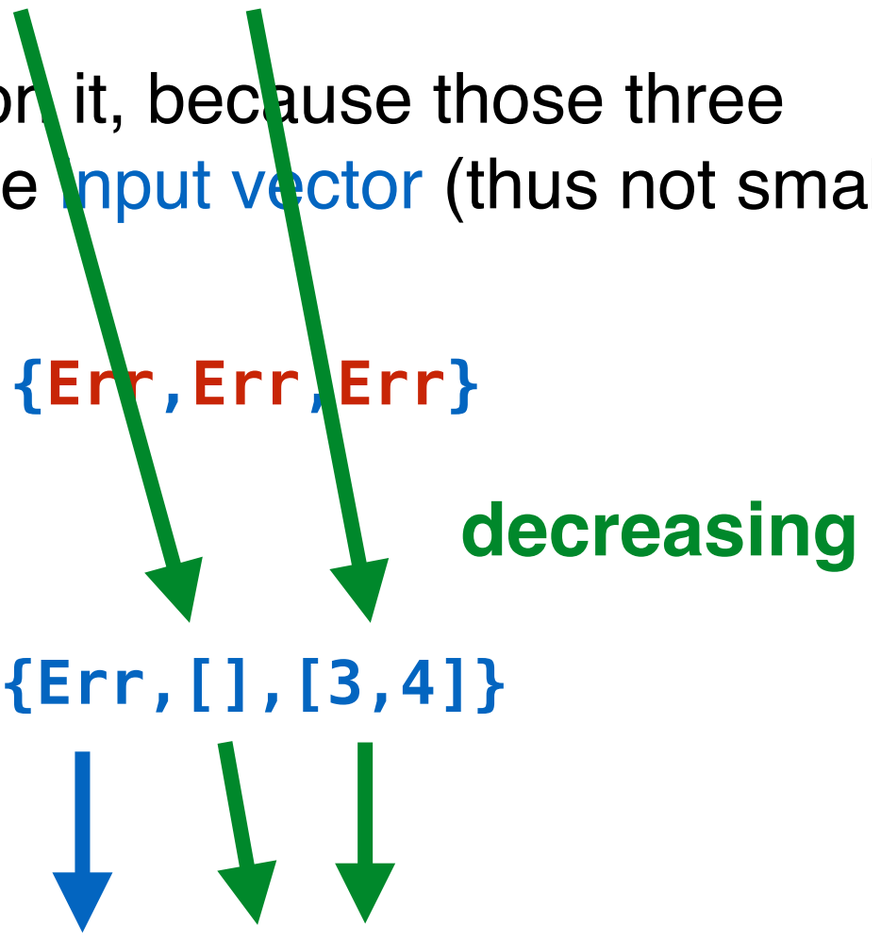
- For term `@xs`, it we apply length on it, because those three argument lists are the same as the input vector (thus not smaller than it), this results in 3 **Err**s:

```
length(@xs) ~> {Err,Err,Err}
```

- For term `tail(@xs)`

```
tail(@xs) ~> {Err,[],[3,4]}
```

```
length(tail(@xs)) ~> {Err, 0, 2}
```



Rebooting

- After our algorithm has found a recursive program, we are not done yet. We must check whether it works correctly on those additional recursive calls we encountered during the synthesis.

- Why? Because during the Goal-Guided Program Assembly stage, we may have used the output vector of some recursive terms

```
length({Err, [], [3,4]}) ~> {Err, 0, 2}
```

- But GGPA only guarantee to find a program that satisfies the original input-output examples

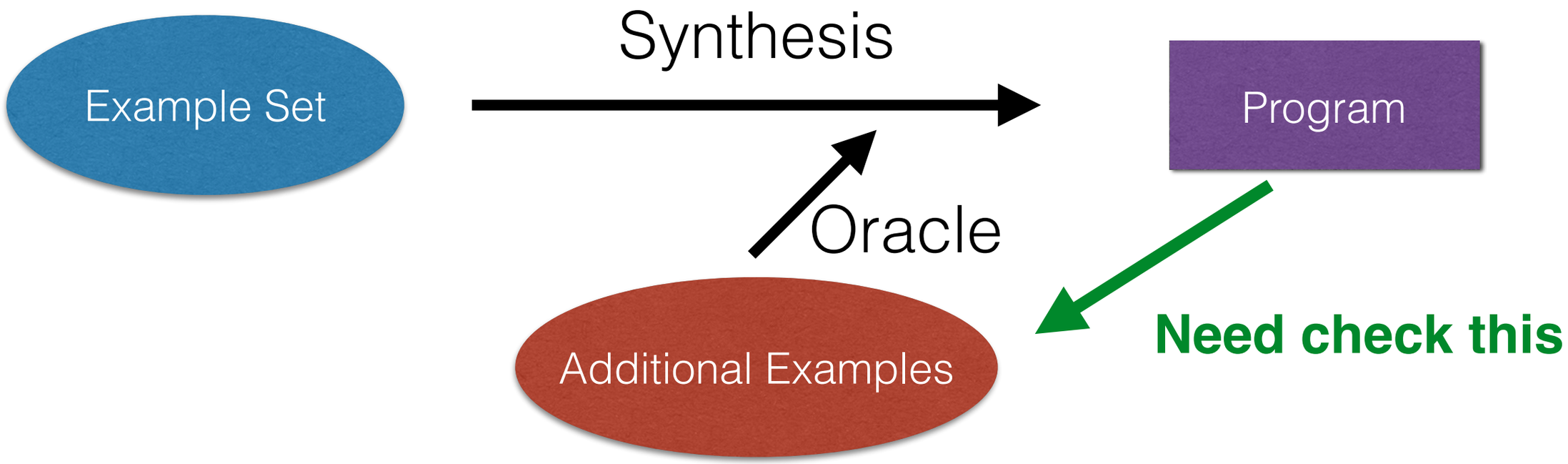
```
length({[], [1], [2,3,4]}) ~> {0, 1, 3}
```

- The resulting program may not satisfy those additional examples:

```
length([3,4]) =??= 2
```

Rebooting, continued

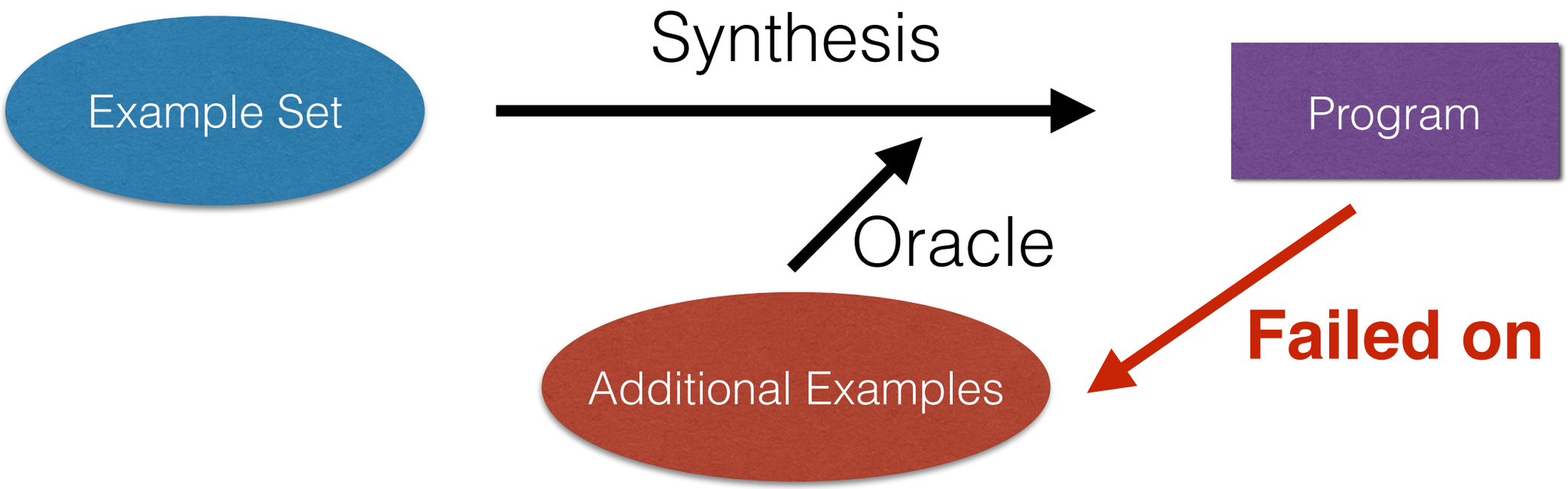
- But the correctness of GGPA relies on the output vector of these recursive terms. So we must check that the algorithm we get is actually correct on these additional examples.



Rebooting, continued

- But the correctness of GGPA relies on the output vector of these recursive terms. So we must check that the algorithm we get is actually correct on these additional examples.

- What if the found program failed on some additional examples?

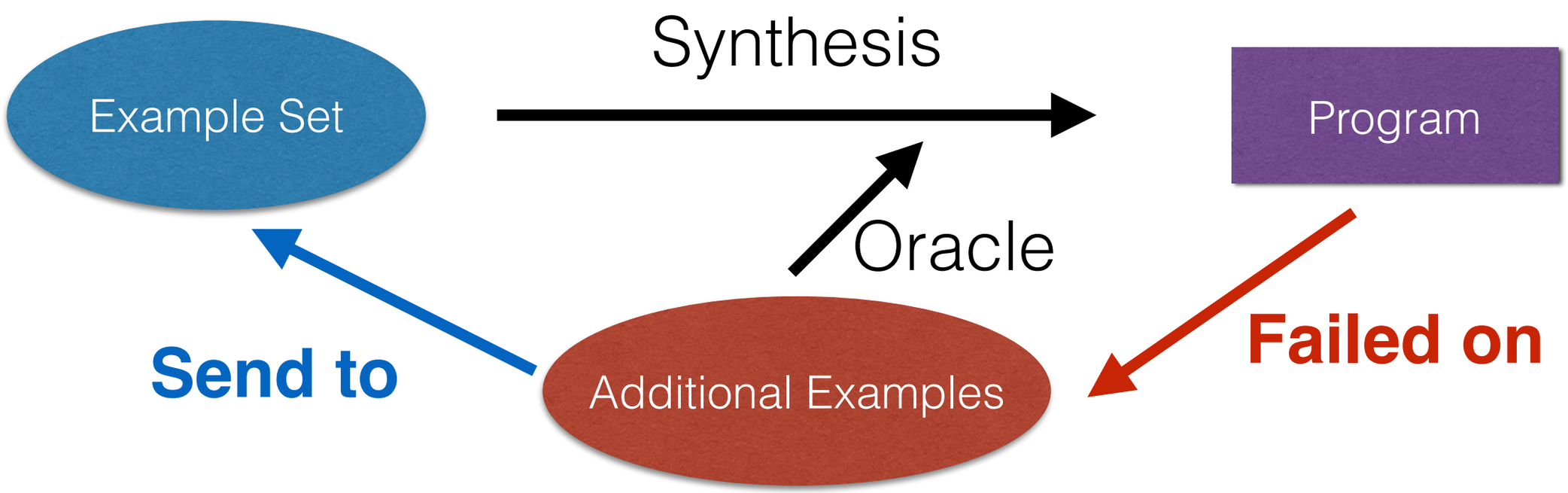


Rebooting, continued

- But the correctness of GGPA relies on the output vector of these recursive terms. So we must check that the algorithm we get is actually correct on these additional examples.

- What if the found program failed on some additional examples?

- In this case, the program must be wrong and we add those failed examples into the example set and reboot the algorithm. This is called rebooting.



Rebooting, continued

- But the correctness of GGPA relies on the output vector of these recursive terms. So we must check that the algorithm we get is actually correct on these additional examples.

- What if the found program failed on some additional examples?

- In this case, the program must be wrong and we add those failed examples into the example set and reboot the algorithm. This is called rebooting.

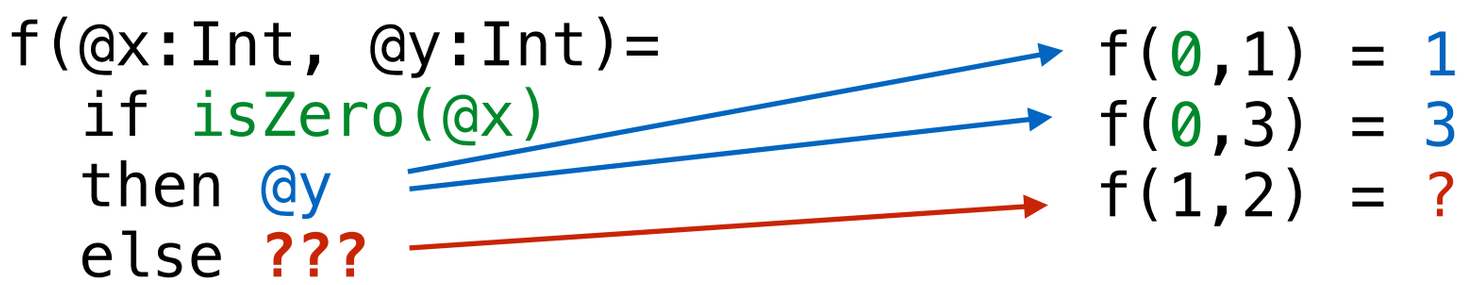
Benchmark

- Problem: **Too many** examples required!

```
Summery:
  name           cost  depth  examples  reboots          time
  reverse          12      8      2/10     None      279.58ms
  stutter          13      9      3/10     None      129.90ms
  cartesian        32     12      4/78     None      502.89ms
  squareList       14      9       6/2     None      538.35ms
  fib              15      8       8/4     None      344.94ms
  insert           19      9     8/292     None      534.71ms
  compress         25      9      8/22     None       36.98ms
  nodesAtLevel     27     11    11/427     None    3,472.47ms
  contains         14      5      7/22     None        4.58ms
  dropLast         15      6      5/10     None        9.10ms
  evens            16      7      5/20     None       10.68ms
  dedup            46     16    16/149        7  103,932.40ms
  tConcat          15     11    6/2213     None   12,057.56ms
Total time: 121.854s
```

A New Algorithm

- I'm currently working on a new algorithm to address this problem
- The new algorithm is called AcsendRec and it **does not require an Oracle** to work
- Basic idea: Instead of asking the Oracle, we try to partially evaluate the output vector of a recursive term during GGPA



f({(0,1),(0,3),(1,2)}) ~> {1, 3, ?}

The Unknown Value

- In addition to Err, we need another special value, called Unknown, written as '**?**'

- Moreover, during Forward Term Search, we must distinguish recursive from non-recursive terms. An recursive term must have at least one '**?**'s in its output vector, **otherwise we treated it as if it's non-recursive**

```
@xs ~> {[],[1],[2,3,4]}
tail(@xs) ~> {Err,[],[3,4]}
isNil(@xs) ~> {T,F,F}
length(nil())
            ~> {0,0,0}
.. ..
```

Non-recursive

```
length(tail(@xs)) ~> {Err,0,?}
inc(length(tail(@xs)))
                  ~> {Err,1,?}
.. ..
```

Recursive

Ascending-Recursive Form

- This new algorithm can only efficiently find programs of a special form — The target program should be ascending-recursive

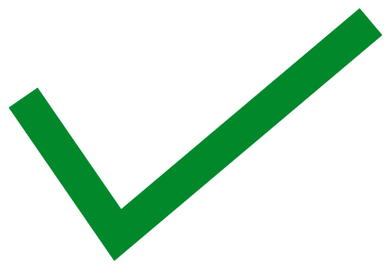
- Ascending-recursive Form (ARF): A program is said to be in ARF if:

  1. Its first branch and condition expressions are non-recursive

(suppose length is an recursive call)

```
if not(isNil(@xs))
then inc(length(tail(@xs)))
else zero()
```

```
if isNil(@xs)
then zero()
else inc(length(tail(@xs)))
```

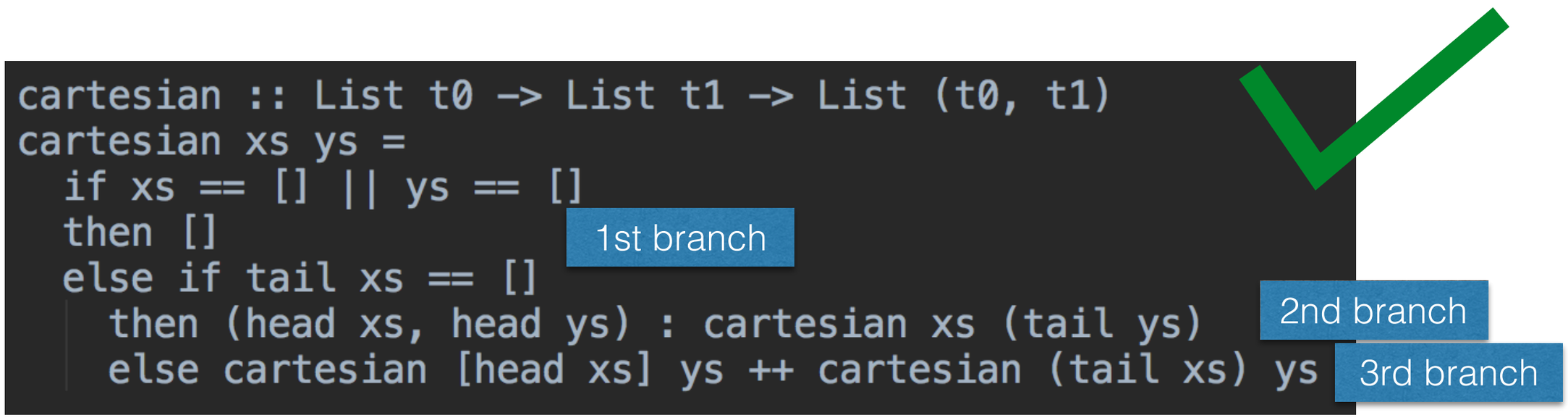


Ascending-Recursive Form

- This new algorithm can only efficiently find programs of a special form — The target program should be ascending-recursive

- Ascending-recursive Form (ARF): A program is said to be in ARF if:

  1. Its first branch and condition expressions are non-recursive
  2. If the program recursively calls itself in the **i** th branch, then in the following recursive evaluation, it can only access the **j** th branches where 1≤ **j** ≤ **i**



Ascending-Recursive Form

- This new algorithm can only efficiently find programs of a special form — The target program should be ascending-recursive

- Ascending-recursive Form (ARF): A program is said to be in ARF if:

  1. Its first branch and condition expressions are non-recursive
  2. If the program recursively calls itself in the **i** th branch, then in the following recursive evaluation, it can only access the **j** th branches where 1≤ **j** ≤ **i**
  3. Nested recursive calls are not allowed

(suppose fib is an recursive call)

```
fib(fib(n-1))
```

```
fib(n-1)+fib(n-2)
```

Partial Evaluation

- We are going to use the cartesian example to demonstrate how to efficiently search for ascend-recursive programs without using an Oracle.

- Remember GGPA is a top-down process — We start from an initial goal (output vector), and gradually split it using if statements

Partial Evaluation

```
inputs: {([], [2, 3, 4]),([5], []),([5], [7, 8, 9]),([2, 3], [4, 5])}
goal:   {[], [], [(5,7),(5,8),(5,9)], [(2,4),(2,5),(3,4),(3,5)]}
```

```
cartesian(@xs, @ys) =
  ???
```

- Now try to split the goal. Because the first branch must be non-recursive, we split it just like before

Partial Evaluation

```
inputs: {([], [2, 3, 4]),([5], []),([5], [7, 8, 9]),([2, 3], [4, 5])}

goal:   {* , * , [(5,7),(5,8),(5,9)], [(2,4),(2,5),(3,4),(3,5)]}
```

```
cartesian(@xs, @ys) =
  if or(isNil(@ys), isNil(@xs))
  then nil()
  else ???
```

- The first two cases in the goal vector are solved
- Try to split again

Partial Evaluation

```
inputs: {([], [2, 3, 4]),([5], []),([5], [7, 8, 9]),([2, 3], [4, 5])}

goal:   {* , * , [(5,7),(5,8),(5,9)], [(2,4),(2,5),(3,4),(3,5)]}
```

```
cartesian(@xs, @ys) =
  if or(isNil(@ys), isNil(@xs))
  then nil()
  else if isNil(tail(@xs))
       then ???
       else ???
```

- Now we need to decide which term to put into the red hole
- Suppose we have tried all non-recursive terms and none of them can lead to a successful assembly
- Then the algorithm tries to find a recursive term

Partial Evaluation

```
inputs: {([], [2, 3, 4]),([5], []),([5], [7, 8, 9]),([2, 3], [4, 5])}
goal:   {* , * , [(5,7),(5,8),(5,9)], [(2,4),(2,5),(3,4),(3,5)]}
```

```
cartesian(@xs, @ys) =
  if or(isNil(@ys), isNil(@xs))
  then nil()
  else if isNil(tail(@xs))
       then ???
       else ???
```

- What's the output vector of the term below?

```
cons(createPair(head(@xs), head(@ys)), cartesian(@xs, tail(@ys)))
```

- Before GGPA, it is:

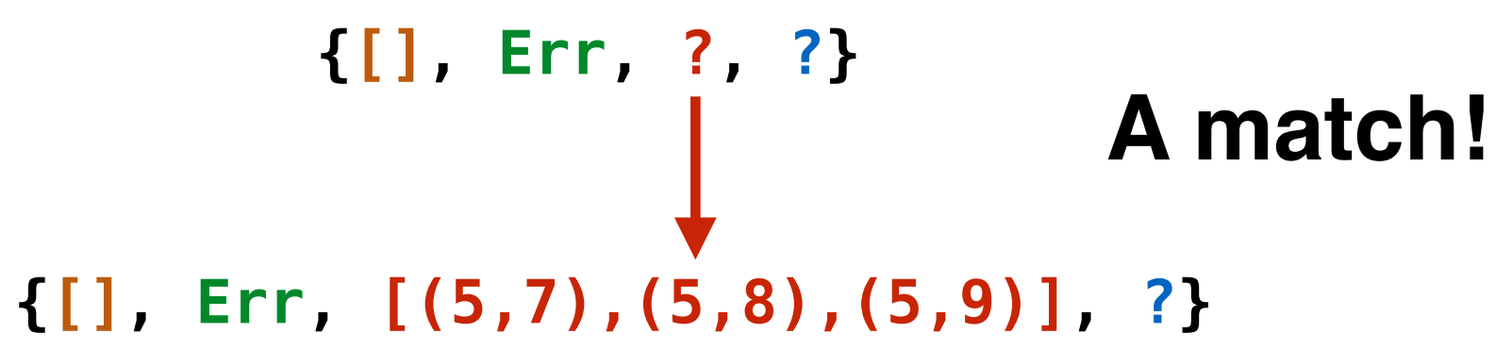
```
{?, Err, ?, ?}
```

Partial Evaluation

```
inputs: {([], [2, 3, 4]),([5], []),([5], [7, 8, 9]),([2, 3], [4, 5])}
goal:   {* , * , [(5,7),(5,8),(5,9)], [(2,4),(2,5),(3,4),(3,5)]}
```

```
cartesian(@xs, @ys) =
  if or(isNil(@ys), isNil(@xs))
  then nil()
  else if isNil(tail(@xs))
       then cons(createPair(head(@xs), head(@ys)),
                            cartesian(@xs, tail(@ys)))
  else ???
```

- But now, we can plug this term into the red hole and calculate its new output vector, and **we only need to evaluate its value on the third input**



Partial Evaluation

```
inputs: {([], [2, 3, 4]),([5], []),([5], [7, 8, 9]),([2, 3], [4, 5])}
goal:   {* , * , [(5,7),(5,8),(5,9)], [(2,4),(2,5),(3,4),(3,5)]}
```

```
cartesian(@xs, @ys) =
  if or(isNil(@ys), isNil(@xs))
  then nil()
  else if isNil(tail(@xs))
       then cons(createPair(head(@xs), head(@ys)),
                 cartesian(@xs, tail(@ys)))
       else ???
```

- But now, we can plug this term into the red hole and calculate its new output vector, and **we only need to evaluate its value on the third input**

- Note that because we are trying to find an ascending-recursive program, a correct term for the red hole can always successfully evaluate its value like we just saw!

Partial Evaluation

```
inputs: {([], [2, 3, 4]),([5], []),([5], [7, 8, 9]),([2, 3], [4, 5])}

goal:   {* , * , [(5,7),(5,8),(5,9)], [(2,4),(2,5),(3,4),(3,5)]}
```

```
cartesian(@xs, @ys) =
  if or(isNil(@ys), isNil(@xs))
  then nil()
  else if isNil(tail(@xs))
       then cons(createPair(head(@xs), head(@ys)),
                 cartesian(@xs, tail(@ys)))
  else ???
```

- Now we are only left with this blue hole and we can solve it recursively, thus finish the GGPA stage.

Benchmark (AscendRec)

Summery:

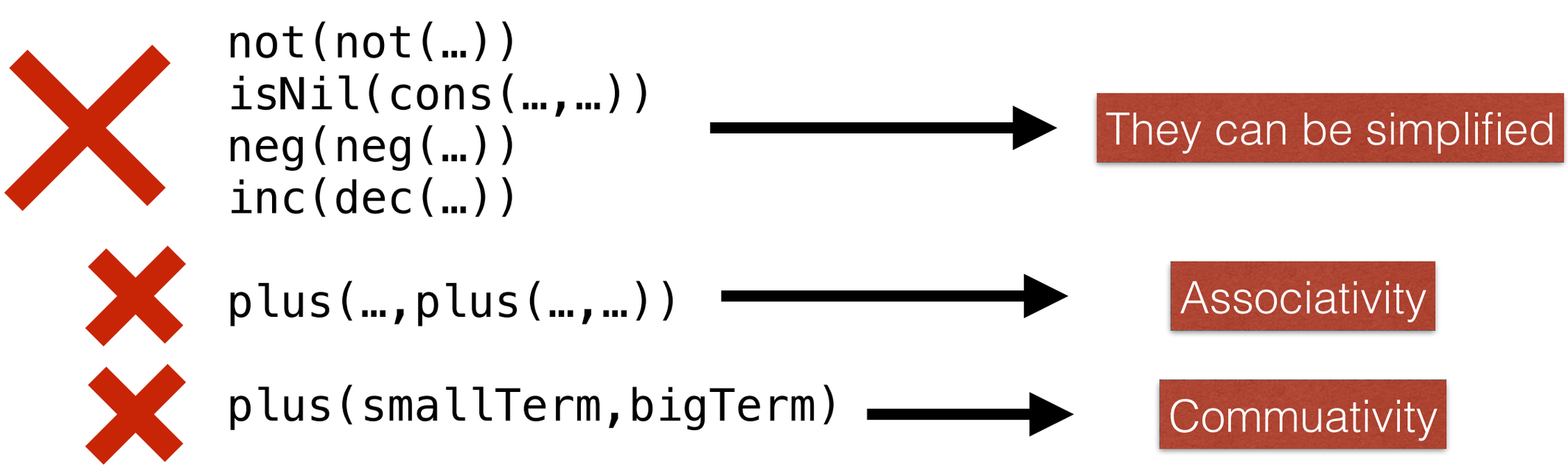
name	cost	depth	examples	time
reverse	12	8	5	538.26ms
stutter	13	9	3	235.57ms
squareList	14	9	6	3,946.44ms
fib	15	8	8	2,575.98ms
insert	19	9	8	5,977.26ms
lastInList	14	5	5	7.77ms
shiftLeft	12	8	5	32.81ms
maxInList	20	10	7	12,844.99ms
flattenTree	14	10	3	79.12ms
sumUnder	10	5	5	7.83ms
times	11	6	6	319.75ms
contains	14	5	7	7.33ms
dropLast	15	6	5	11.93ms
evens	16	7	5	17.06ms
tConcat	15	11	6	48,338.31ms
cartesian	32	11	4	3,419.98ms
nodesAtLevel	27	11	12	241,406.70ms
Total	273	138	100	319.767s

We need much less examples now!

Limitations

- Because of the presence of '**?**'s, we can't use observational equivalence for recursive terms. This can quickly lead to too many terms.

- To alleviate this problem, we can introduce some simple term-rewriting check to reject terms which can be rewritten into simpler ones:



Limitations

- For those programs that are not ascending-recursive, if we provide trace-complete examples (i.e. all recursive calls are included in the example set), the algorithm can synthesize it as if it was in ascending-recursive form.

```
dedup(@xs: List['0]): List['0] =
  if isNil(@xs)
  then @xs
  else if contains(tail(@xs), head(@xs))
       then dedup(tail(@xs))
       else cons(head(@xs), dedup(tail(@xs)))
```

- I’m still working on this algorithm. There might be more improvements in the future!

References

- Recursive Program Synthesis: https://www.microsoft.com/en-us/research/publication/recursive-program-synthesis/
- Escher-Scala on Github: https://github.com/MrVPlusOne/Escher-Scala
- Leon Online: https://leon.epfl.ch
- Flash Fill: https://www.microsoft.com/en-us/research/video/programming-examples-pbe-flash-fill/

That's all,
Thank You!

Any Questions?