University of Science and Technology of China

A dissertation for bachelor's degree

Oracle-free Synthesis of Recursive Programs

Author's Name:	Jiayi Wei
Speciality:	Applied Physics
Supervisor:	Prof. Xinyu Feng
Finished Time:	May, 2017

Acknowledgements

First of all, I would like to thank my advisor Professor Xinyu Feng for his patient guidance and help on this thesis. I have learned a lot from him through our weekly discussions. His rigorous approach to research taught me how to think and discuss in a more mathematically thorough manner, and the insightful questions he posed have pushed me to study the problems from many new perspectives. I am very grateful for the special lecture opportunity he offered me, which allowed me to share my recent work with many other undergraduate students.

I am also very grateful to Professor Işıl Dillig for the consistent encouragement and support she gave. Professor Dillig was the person that led me into this research area, and who kindly offered me an internship at her research group. Her positive feedback about this thesis meant a lot to me.

I would also like to thank Yuepeng Wang, Zhifeng Wang, Zhen Zhang and Yu Feng for providing useful feedback and help on this thesis.

Finally, I thank all my family for their unconditional support and encouragement throughout my undergraduate study. Having a dream of becoming a physicist ever since I was 13, it was a tough decision for me to switching from Physics to Computer Science, but my family have always been supportive on my every major decision along the path, and I know they are always there for me whenever I need their help.

Abstract

Input-output examples are a simple and accessible way to describe program behavior. In example-driven program synthesis, the user only needs to provide some input-output examples, and a synthesizer can then algorithmically synthesize target programs that are consistent with these examples. ESCHER is a famous example-driven algorithm for synthesizing recursive programs, and it can efficiently perform a large class of synthesis tasks. However, ESCHER needs using a reference implementation (called oracle) for each task to ask for additional input-output examples during synthesis; In common scenario, the user plays the role of this oracle. Because of this behavior, in some synthesis tasks, ESCHER may need to ask for hundreds or even thousands of additional examples, and as a result, this can greatly limit the algorithm's application. To address this problem, in this thesis, we present a new algorithm called ASCENDREC which is based on ESCHER but does not need to use an oracle. Like ESCHER, ASCENDREC can efficiently synthesize elegant recursive programs from input-output examples, but because it is oracle-free, it only requires much fewer examples compared to ESCHER. Moreover, although ASCENDREC puts some extra restrictions on the form of programs it can efficiently search for, we demonstrate, both trough algorithmic analysis and experimental results, that by cleverly constructing trace-complete examples, a large class of synthesis tasks can still be synthesized using ASCENDREC, even if they do not satisfy those requirements.

Keywords: Program synthesis; programming by example; automated programming

Contents

1 Introduction

Automated program synthesis aims to simplifying programming by allowing users to express their intent as formal or informal specifications. An algorithmic program synthesizer is then used to discover executable implementations of these specifications. Program synthesis can be seen as an ultimate abstraction that leaves the implementation details completely handled by synthesis algorithms.

Example-driven program synthesis is a particularly important form of program synthesis because it can potentially help many users with little or no programming knowledge perform complex computational tasks. For instance, recent work on synthesizing string manipulation programs from examples in spreadsheets [4] has already helped millions of Microsoft Excel users in their everyday work [6].

ESCHER is an example-driven algorithm for synthesizing recursive programs [1]. By harnessing the power of recursion, it achieves expressiveness using only a simplistic target programming language (A language with only three constructs: *input variables*, *function calls*, and *if-then-else expressions*). ESCHER is able to perform a large class of synthesis tasks in very short time (usually within several seconds), and it uses oracles (reference implementations) to obtain additional input-output examples during synthesis. But for some synthesis tasks, ESCHER may require hundreds or even thousands of additional input-output examples in order to enumerate through different recursive expressions (see Table 2), and this can greatly limit the algorithm's application since in most scenarios, it is not practical for users to provide the synthesizer with so many examples.

To overcome this problem, we decided to abandon the use of oracles and developed a new algorithm called ASCENDREC. Since the source code of ESCHER is not publicly available, we also implemented our own version of ESCHER. We named our implementation TYPEDESCHER to reflect the addition of a static type system to the original algorithm's target language, and such a polymorphic type system can often boost searching efficiency drastically [2, 3]. Both TYPEDESCHER and ASCENDREC will be introduced in detail in the following sections of this thesis.

Contributions We summarize our contribution as follows:

1. TypedEscher: an extension to the original ESCHER algorithm. Unlike the original ESCHER algorithm, which uses an untyped target programming language, TYPEDESCHER uses a polymorphic, statically typed language to increase the searching efficiency through the space of programs
2. ASCENDREC: a novel algorithm based on TYPEDESCHER but dose not need to ask additional inputs from oracles
3. An implementation of both TYPEDESCHER and ASCENDREC, as well as an evaluation of them on a set of benchmarks that demonstrates their effectiveness at synthesizing a wide range of programs. Moreover, our comparison highlights the advantage of using an oracle-free algorithm.

2 TypedEscher: An Example

In this section, we demonstrate the operation of TYPEDESCHER through a simple example: Suppose we would like to synthesize a list `length` function, that is, a program that gives the number of elements in a list.

Type Signature Because TYPEDESCHER uses a polymorphic typed system, our first step on this synthesis task is to give a type signature of the function we want to synthesize.

```
length(@xs: List['0]): Int
```

This says that our desired function is called `length` and takes one single argument `@xs`, which is a list of some element type `'0`, and that it returns an integer.

Input-Output Examples The next step is to provide TypedEscher with several input-output examples. Here we use "`[...]`" to denote a list.

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

Component Set TypedEscher always works under a *Component Set* — it only uses *components* (library functions) from this Component Set to synthesize target programs. The Component Set is not fixed, so it can be customized for different synthesis tasks. To write down the `length` function, we will need the following three components:

Name	Input Types	Return Type	Component Description
zero	`()`	`Int`	the constant 0
isNil	`(List['0])`	`Bool`	checks wether a list is empty
inc	`(Int)`	`Int`	increment by 1
tail	`(List['0])`	`List['0]`	gives a list's tail

Since it often requires non-trivial thinking on the part of the user to choose the right components for different tasks, we provide a base set of components (same as in [1], shown in Table 1), which includes the above four components needed for `length`. Note that all the following synthesis tasks were tested under this base component set.

Component Type	Supplied Components
Boolean	T, F, and, or, not, equal, isNil, isNonNeg, isZero
List	head, tail, cons, concat, nil
Integer	zero, inc, dec, neg, plus, div2
Binary Tree	createLeaf, createNode, isLeaf, treeValue, treeLeft, treeRight, treeTag

Tab. 1: Base component set used in experiments

The Oracle During the synthesis, TypedEscher will ask the oracle for corresponding results of some additional input cases. In this simple example, TypedEscher only requires 2 additional examples:

```
  [4] -> 1
[3,4] -> 2
```

Synthesized Program After setting up the aforementioned environment parameters, TypedEscher can now successfully synthesize a correct `length` function, as shown below, in about 20 milliseconds. Note that this program describes the exact functionality as we want and can correctly generalize to new examples that are not present in the provided example set.

```
length(@xs: List['0]): Int =
  if isNil(@xs)
  then zero()
  else inc(length(tail(@xs)))
```

3 TypedEscher: Algorithm Overview

In this section, we introduce how TypedEscher works in more detail, and we use the `length` task described in the previous section as a concrete example to illustrate new concepts and definitions.

3.1 Definitions

Value Vector Given the input-output examples on the preceding page, we can represent the three inputs as a single input vector `{([]),([1]),([2,3,4])}`, since we only have one argument in each input, we can omit the parentheses and write `{[],[1],[2,3,4]}`. We also represent our desired outputs as an *output vector* `{0,1,3}`

Term 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.

For example, term `tail(@xs)` has output vector `{Err,[],[3,4]}`, where the special value `Err` indicates that apply tail on the first input `[]` results in an error. A constant term like `nil()` has an output vector in which all elements are the same (`{[],[],[]}` in this case).

Since TypedEscher can synthesize recursive programs, we also allow the use of recursive calls in terms. In our `length` example, term `length(tail(@xs))` contains a recursive call, and its output vector is `{Err,0,2}`. The first output `Err` corresponds to `length(tail([]))`, because `tail([])` causes an `Err`, the whole term results in an `Err`. The second output corresponds to `length(tail[1])`, which reduces to `length([])`, and TypedEscher knows this evaluates to `0` because this example is presented in the example set. However, for the last input, it requires evaluating `reverse([3,4])`, which is not in our example set; hence, in such cases, the algorithm needs to ask the oracle for additional results.

Definition 1. Term

A `term` is an input variable `x`, or a component `c` applied to zero or more arguments which are themselves `terms`, or a recursive call shown here as a special component `self`.

```
term := x
     | c(term, ..., term)
     | self(term, ..., term)
```

Target Program For our synthesis task, if TypedEscher can find a term with the desired output vector, then, its task is completed. Otherwise, it may need to combine those terms into *Target Programs* (sometimes just called programs).

Definition 2. Target Program

A target program is either a `term` or an if-then-else expression whose condition and then branches are required to be terms.

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

Cost Function We want to find a recursive program that is consistent with our example set. However, we do not want our synthesis program to overfit on our example set and not be able to generalize to new examples.

In order to achieve this, here we use Occam's Razor as the guiding heuristic and try to find the "simplest" program that is consistent with the example set. By "simplest", we mean using some predefined metric function that maps each possible target programs to corresponding cost value and searching for a program with the lowest cost.

Definition 3. Cost function

In our current implementation, we choose the AST size, i.e., the number of nodes in target programs' abstract syntax trees as our cost function. When TypedEscher is instantiated with this heuristic function, the search is biased towards smaller programs.

$$\begin{aligned}\text{size}(c(t_1,\dots,t_n)) &= \text{size}(t_1)+\cdots+\text{size}(t_n)+1\\ \text{size}(\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } p) &= \text{size}(t_1)+\text{size}(t_2)+\text{size}(p)+1\end{aligned}$$

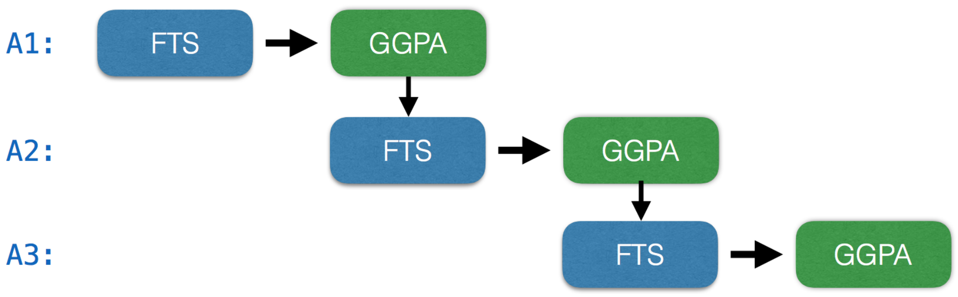


Fig. 3.1: AscendRec stages

3.2 Algorithm Description

The TypedEscher algorithm consists of two alternating stages: *Forward Term Search* (FTS) and *Goal-Guided Program Assembly* (GGPA). In the n th alternation, the algorithm first searches out all terms of size n using FTS and then tries to assemble a correct target program using GGPA. An illustration of this process is shown in Figure 3.1

3.2.1 Forward Term Search

Suppose we want to synthesize a 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, along with many other terms, FTS can find all terms we need in order to synthesize a correct target program:

Term	Output Vector	Size
`zero()`	`{0,0,0}`	1
`isNil(@xs)`	`{T,F,F}`	2
`inc(length(tail(@xs)))`	`{Err,1,3}`	4

3.2.2 Observational Equivalence

If we just naively enumerate all possible terms in FTS, there would be exponentially many of them. But many syntactically different terms are actually equivalent in semantics, and we only need to keep simplest ones in FTS. So we can use *Observational Equivalence* as an approximation to semantical equivalence.

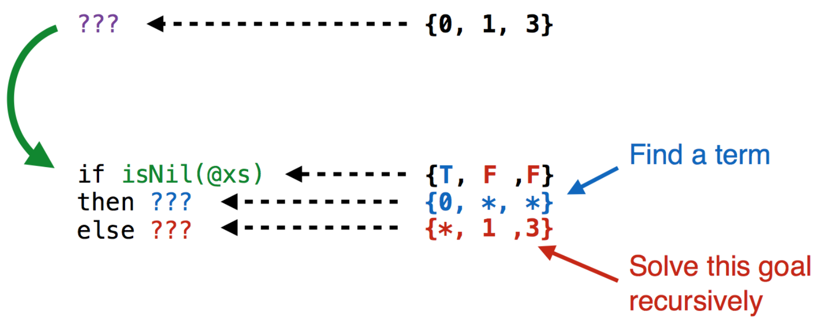
Definition 4. Observational Equivalence
Given an example set S and two *known* terms t_1 t_2, we say t_1 is observational equivalent to t_2 with respect to S if they have the same output vector under S.

For example, both `zero()` and `dec(inc(zero))` has the output vector `{0,0,0}`; hence, they are observational equivalent w.r.t. our example set.

3.2.3 Goal-Guided Program Assembly

If we cannot find a term to satisfy a goal, we can split the goal using if-then-else expressions.

In our example, when synthesizing `length` at alternation 4, there is no term that has our desired output vector {0,1,3}, but since the boolean term `isNil(@xs)` has output vector `{T,F,F}`, we can try to use it as the condition of an if-then-else expression and distribute our goal into the then and else branches and solve these smaller goals separately. After the distribution, the then branch now corresponds to an output vector `{0,*,*}`, in which a “*” matches any value; similarly, the else branch corresponds to the output vector `{*,1,3}`. This process is shown in the following figure.



Algorithm 1 shows the pseudo code of a simplified version of the GGPA algorithm. Here, `goal` is the target output vector with which GGPA is trying to synthesize a program to match. At line 2-3, GGPA first tries to find a single term to match the goal, if this succeeds, it returns this term immediately. Otherwise, GGPA tries to split the current goal using all possible boolean terms available, and this is achieved by a *for loop* at line 4. In the body of the for loop, the algorithm splits the goal into two smaller ones (`thenGoal` and `elseGoal`, each corresponds to a branch of the if-then-else expression), and if it can find a term and a target program to match these two subgoals separately, it assembles these two expressions into one if-then-else expression and returns it. Note that in our actual implementation of GGPA, the implementation also meets other requirements like only returning the smallest matching term and avoiding repeated calculations using dynamic programing techniques.

Algorithm 1 GGPA: TypedEscher Version

```
1  def GGPA(goal) = {
2      let term = find_term_match_goal(goal);
3      if (term != None) return term;
4      for(cond from terms_that_splits(goal)){
5          let (thenGoal, elseGoal) = splitGoal(cond, goal);
6          for(thenTerm from terms_that_matches(thenGoal)){
7              let elseExpr = GGPA(elseGoal);
8              if(elseExpr != None)
9                  return (IF cond THEN thenTerm ELSE elseExpr);
10         }
11     }
12     return None;
13 }
```

3.2.4 Avoiding Nonterminating Programs

There is one important problem our synthesis program needs to address: Nontermination. For example, what if GGPA returns the term `length(@xs)`? This is problematic because this term has the exact output vector `{0,1,3}` we want, yet using it would synthesize this program: `length(@xs) = length(@xs)`, which is clearly nonterminating.

Also, there always exists some nonterminating programs that are simpler than the correct, terminating ones! To avoid problems like this, Escher requires the argument list of recursive calls to be "smaller" than the previous argument list. Instead of enforcing this requirement using some static checking techniques, like the approaches taken by many proof assistants [7], Escher performs a kind of run-time check. More specifically, we first defines a well-founded binary relation $\prec$ on each data type such that $v_1 \prec v_2$ means value v_1 has a smaller size than v_2, and we require there to be a smallest value for each data type. Then, during the FTS stage, we can calculate the output vector of a recursive call as follows: if the argument list of this recursive call is smaller than the current inputs, i.e. the function we are trying to synthesize gets called on a smaller argument list (we can easily define a $\prec$ relation for argument lists using the normal *alphabetic ordering*), then the corresponding element in the output vector is calculated same as before; otherwise, we set the corresponding element to `Err`. Using this new calculation rule, the algorithm can efficiently rule out nonterminating terms during FTS, and all programs found by GGPA are thus guaranteed to terminate. The previous term, `length(@xs)`, now has `{Err,Err,Err}` as its output vector because `length` is called directly on the input variable `@xs`, hence every recursive call has an argument list same as the one in the current input vector.

3.2.5 Rebooting

After TypedEscher has found a recursive program, it must check whether the target program works correctly on those additional recursive calls encountered during synthesis. While GGPA only guarantees to find a program that satisfies the original input-output examples, its correctness also relies on the output vectors of those recursive terms, so it must perform this additional check. If the found program fails on some additional examples, the target program must be wrong and we should add those failed examples into the example set and reboot the synthesis algorithm. This is called *rebooting*, as shown in Figure 3.2

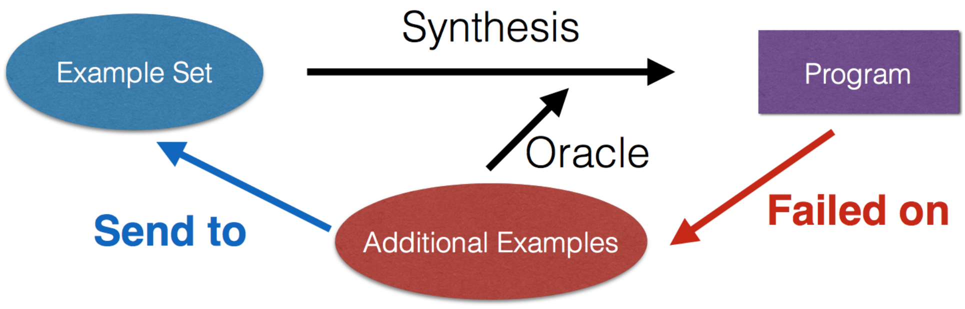


Fig. 3.2: Rebooting

3.3 Benchmarks & A Problem

To evaluate our synthesis algorithm, we used a benchmark suite consists of a number of recursive integer, list, and tree manipulating programs, many of them are taken from [1, 2], and we also added more challenging tasks like squareList and sortList. The results are summarize in Table 2 on the next page.

Too many examples From the "examples" column we can see that although many synthesis tasks only ask oracles for several examples, some tasks can ask for hundreds, even thousands of additional examples! This behavior is due to the enumerative nature of this algorithm – the more complex the task is, the more recursive terms will be enumerated, and for each recursive call on new examples, an additional example is required. As a result, this can greatly limit the application scenario of TypedEscher since it is often impractical for the user to enter so many examples during synthesis. To address this problem, we developed a new algorithm called AscendRec, which does not need to use an oracle.

4 The AscendRec Algorithm

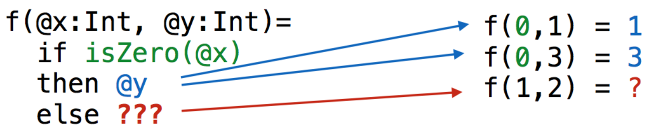
AscendRec is a new algorithm based on TypedEscher, hence they share many similarities, and most concepts and ideas introduced on previous sections also apply to AscendRec. The main difference is that AscendRec does not use an oracle to calculate the output vector of recursive terms, hence it does not require the user to provide additional inputs during synthesis, and this means the total amount of examples needed can be much fewer than in TypedEscher, and we shall see this in later comparisons.

The Basic Idea Instead of asking an oracle, we can postpone the evaluation of a recursive term in FTS and try to partially, progressively evaluate the output vector later in GGPA. In the example shown below, `f` is an unknown function we are trying to synthesize suing GGPA, but we have not decided which expression to put into its else branch yet. The insight is that, even this

	Name	Cost	Depth	Exs	Add. Ex	Reboots	Runtime
Lists	reverse	12	8	3	9	None	500ms
	length	8	4	3	2	None	19ms
	compress	25	9	8	22	None	415ms
	stutter	13	9	3	10	None	79ms
	squareList*	14	9	6	2	None	799ms
	insert	19	9	8	292	None	554ms
	contains	14	5	7	22	None	10ms
	lastInList	14	5	5	9	None	6ms
	shiftLeft	12	8	5	45	None	23ms
	maxInList	24	8	8	154	1	233ms
	dropLast	15	6	5	10	None	22ms
	evens	16	7	5	20	None	24ms
	cartesian*	32	12	4	78	None	174ms
	sortList	35	13	7	3702	1	123s 855ms
	dedup*	19	7	9	16	None	35ms
	dedup	46	16	16	149	7	99s 518ms
Integers	fib	15	8	8	4	None	228ms
	sumUnder	10	5	5	3	None	9ms
	mod*	8	8	14	588	5	40s 103ms
	times	22	8	19	182	13	59s 827ms
Trees	flattenTree	14	10	3	16	None	28ms
	tConcat	15	11	6	2213	None	14s 899ms
	nodesAtLevel	27	11	11	427	None	1s 751ms
	Total	429	196	168	7975	27	343s 111ms

Tab. 2: TypedEscher Performance
Column "Exs" and "Add. Ex" respectively displays the number of initial and additional examples used in each synthesis task.
All tasks were synthesized using the base component set (Table 1 on page 5). Tasks marks with "*" use some additional components.

function contains "holes" and is incomplete, we can still infer some of its behavior using partial evaluation. In the example shown below, we can infer from the code sketch that `f(0,1)=1` and `f(0,3)=3`, regardless what expression will be put into the else branch.



```
f(@x:Int, @y:Int)=
   if isZero(@x)
   then @y
   else ???
```

```
f(0,1) = 1
f(0,3) = 3
f(1,2) = ?
```

4.1 New Definitions

The Unknown Value Because AscendRec does not use an oracle to evaluate unseen (i.e., not in the example set) recursive calls in FTS, it uses a special value "Unknown", written as "?", to represent the corresponding undetermined elements in output vectors. Consequently, the recursive term `length(tail(@xs))` now has an output vector `{Err,0,?}` because `length([3,4])` is an unseen recursive call.

Definition 5. Known and Unknown terms
If a term contains no '?' in its output vector, we say this term is a *known term*; otherwise, it is an *unknown term*.

We must distinguish between known and unknown terms in FTS because we can only apply Observational Equivalence for known terms.

Term-rewriting Check For unknown terms, however, because of the presence of '?'s, we are no long able to use observational equivalence. This can quickly lead to too many terms. To

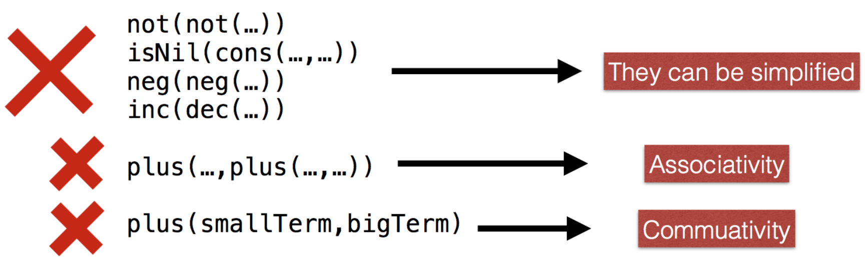


Fig. 4.1: Term Rewriting Check

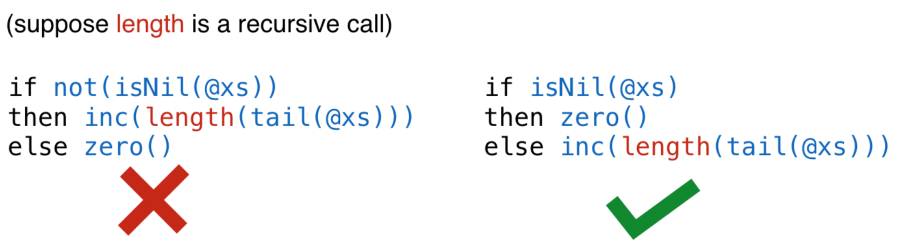
alleviate this problem, we can introduce some simple term-rewriting check to reject terms which can be rewritten into simpler ones.

In our current implementation, we use only three kind of rules to check each newly found term: *Associativity*, *Commutativity*, and *NoDirectChild*. If a term can be rewritten using any of these rules, it can be safely ignored and will not be added to the unknown term set. See Figure 4.1

Ascending-Recursive Form In order to search for programs efficiently in the absence of an oracle, the new algorithm restricts its searching space to only search for programs in a special form:

Definition 6. Ascending-Recursive Form (ARF)
A program is said to be in ARF if:

1. Its first branch and all branch conditions are non-recursive

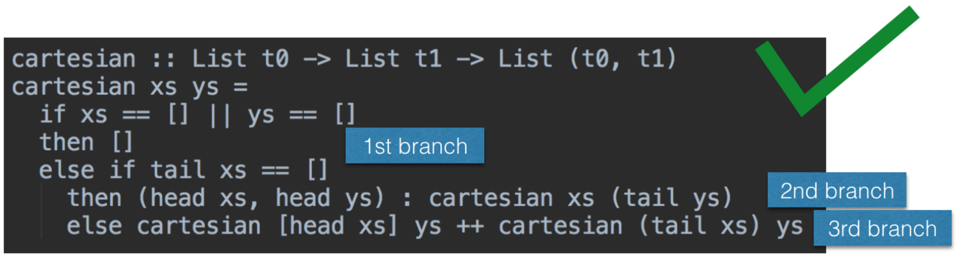


(suppose length is a recursive call)

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

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

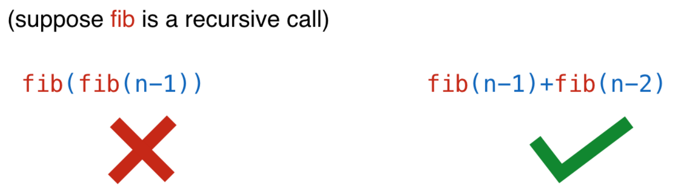
2. If this program recursively calls itself in the i th branch, then, in following recursive evaluation, it can only access the j th branches, where $j \leq i$.



```
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
```

Fig. 4.2: The synthesized cartesian product function from the previous benchmark suit. (Written in Haskell for readability)
Its three branches satisfy the above requirement. In fact, it is also ascending-recursive.

3. The program has no nested recursive calls



4.2 Algorithm Description

AscendRec also consists of the two stages, FTS and GGPA, but there are some differences from their counterparts in TypedEscher.

4.2.1 Forward Term Search

As mentioned before, AscendRec distinguishes known and unknown terms in FTS; therefore, the algorithm checks each newly enumerated term's output vector, and depends on whether there is an unknown value in the vector, either puts the term into the pool of known terms and applies observational equivalence check, or puts the term into the unknown terms pool and performs term-rewriting check.

The definition of ascending-recursive form disallows nested recursive terms; this is for helping the algorithm to perform termination check – we cannot check whether an argument is decreasing if it has an unknown value.

4.2.2 Goal-Guided Program Assembly

The pseudo code of GGPA is shown in Algorithm 2 on the following page. GGPA takes 4 parameters: 1) `goal` is the target output vector with which we are trying to search for an expression to match; 2) `sketch` is a function which represents an incomplete program that contains a hole – when applying `sketch` onto an expression, it inserts the expression into the hole and returns the completed program; 3,4) `known_terms` and `unknown_terms` are the known and unknown terms pools. They are updated before each recursive call to GGPA, since as more and more branches of our target program gets fixed, we can partially evaluate recursive terms on more and more inputs and update their output vectors correspondingly, as we mentioned in *The Basic Idea* on page 9.

At line 2-3, the algorithm first tries to find a term from the known terms pool to match the current goal. If this fails, it picks an unknown term `uTerm` out of `unknown_terms` (line 4), and insert `uTerm` into the current program sketch (line 5). Now, the assembled program is a complete (recursive) program without any holes, and it can be evaluated on any inputs where needed to replace any unknown value in its output vector with concrete values. If any of the assembled programs can match the current goal `goal`, GGPA returns the corresponding `uTerm` (line 6-7).

If GGPA could not find a term that matches `goal`, like in TypedEscher, it then searches for an if-then-else expression. After a condition term `cond` is chosen, the goal is split into `thenGoal` and `elseGoal` (line 10). Next, GGPA tries to find a term that matches `thenGoal`.

Line 11 needs a little more explanation: Remember that we are only interested in terms that, after inserting them into the then branch hole, will result in ascending-recursive programs. For each candidate term `t1,` we can perform the insertion and obtain a partial implementation of the target program we are trying to synthesize using a declaration like "`let partial_impl = sketch(IF cond THEN t1 else ?)`", where "?" denotes the special value `Unknown`. Using this partial implementation to evaluate recursive calls, we can update the output vector of recursive terms, and some previously unknown terms will now become known terms.

But since ascending-recursive programs only access branches with smaller indices than the previous call, all correct terms for the then branch should be able to partially evaluate to concrete values on all inputs corresponding to `thenGoal`. Thus, at line 12, the algorithm only picks `thenTerm` that can concretely matches `thenGoal` from `new_known_terms` and `new_unknown_terms`.

Finally, the algorithm updates `sketch` and recursively calls itself to find an expression for the else branch (line 13-16).

Algorithm 2 GGPA: AscendRec Version

```
def GGPA(goal, sketch, unknown_terms) = {
    let term = find_term_match_goal(goal, known_terms);
    if (term != None) return term;
    for(uTerm from unknown_terms){
        let program = sketch(uTerm);
        if(* program's execution matches goal *)
            return uTerm;
    }
    for(cond from known_terms_that_splits(goal)){
        let (thenGoal, elseGoal) = splitGoal(cond, goal);
        if(thenTerm == None) continue;
        let new_sketch = (x => sketch(IF cond THEN thenTerm ELSE x));
        if(elseExpr == None) continue;
        return (IF cond THEN thenTerm ELSE elseExpr);
        let (new_known_terms, new_unknown_terms) = (* update output terms *)
    }
    return None;
}
```

Name	Cost	Depth	Examples	Escher Ex	Runtime
reverse	12	8	4	12	508ms
length	8	4	3	5	19ms
compress$^{\circledast}$	25	9	14	30	295ms
stutter	13	9	3	13	149ms
squareList	14	9	6	8	3s 279ms
insert	19	9	8	300	6s 600ms
contains	14	5	7	29	8ms
lastInList	14	5	5	14	5ms
shiftLeft	12	8	5	50	43ms
maxInList	20	10	7	162	13s 500ms
dropLast	15	6	5	15	9ms
evens	16	7	5	25	15ms
cartesian	32	11	4	82	3s 569ms
fib	15	8	8	12	2s 80ms
sumUnder	10	5	5	8	7ms
times	11	6	6	201	273ms
flattenTree	14	10	3	19	81ms
tConcat	15	11	6	2219	54s 45ms
nodesAtLevel	27	11	12	438	238s 356ms
Total	306	151	116	3642	322s 842ms

Tab. 3: AscendRec Performance
All tasks are synthesized using the base component set (Table 1 on page 5). Tasks marks with "$\circledast$" use trace-complete example sets. As a comparison, we also add a column of total examples used by TypedEscher in this table.

4.3 Benchmarks & Comparisons

As shown in Table 3, AscendRec uses much fewer examples compared to TypedEscher in many tasks, especially in those more complex ones. Furthermore, even 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), AscendRec will be able to successfully synthesize them as if they were in ascending-recursive form.

5 Conclusion

We have presented a new method for example-driven, oracle-free synthesis of functional recursive programs. Our method combines three key ideas: type-aware forward term search, goal-guided program assembly, and the use of partial evaluation to search for ascending-recursive programs. Our experimental results indicate that the proposed approach is promising.

There are many interesting directions we plan to explore in the future: We plan to study ways of exploiting dynamical programming techniques to reduce repeated calculation in GGPA, and we are also interested in extending AscendRec's searching space to a larger class of programs. Furthermore, we believe our approach to recursive synthesis is not restricted to Escher, but also has the potential to be integrated into other recursive program synthesis algorithms (e.g. [2, 3]).

References

[1] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In CAV, pages 934–950, 2013.

[2] Feser John, Chaudhuri Swarat, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In PLDI, 2015.

[3] Peter-Michael Osera and Steve Zdancewic. Type-and-Example-Directed Program Synthesis. In PLDI, 2015.

[4] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In POPL, pages 317–330, 2011.

[5] Escher-Scala on Github. (Source code and evaluation results of this thesis)
https://github.com/MrVPlusOne/Escher-Scala/

[6] Flash Fill (Microsoft Excel 2013 feature)
http://research.microsoft.com/users/sumitg/flashfill.html

[7] The Coq Proof Assistant
https://coq.inria.fr