Semi-automated Program Synthesis

Contact: [email]@grinnell.edu

Kevin Connors
Grinnell College
[connorsk]

Reilly Grant
Grinnell College
[grantrei]

Zachary Segall
Grinnell College
[segallza]

Peter-Michael Osera
Grinnell College
[oserapet]

Abstract

Program synthesizers have evolved over the past several decades as a method for generating programs from user specifications. One approach to synthesis is using a type theoretic approach and proof search; this is the approach utilized by the MYTH synthesis engine. Difficulties with this synthesis model come from an exponential blow up in the search space. We present the SCOUT synthesis engine, designed for *semi-automated synthesis*, to circumvent these difficulties. SCOUT takes user input as guidance during the synthesis process. This has advantages over previous systems that only allow specifications *before* synthesis begins. Our study also reveals some interesting realizations about synthesis in general and some potential future paths for improving the synthesis process.

1. Introduction

Program synthesis is the process of generating code from specifications given by the user. Synthesis has existed for almost 50 years, though modern synthesizers are much more advanced (Manna and Waldinger 1979). They can increase the speed at which programming can occur, remove errors or make challenging code easier to read. Program synthesis has been approached in a number of dimensions: the type of language, functional versus imperative, and the type of inputs, formal specifications versus input/output examples (Leino and Milicevic 2012; Polikarpova and Solar-Lezama 2015; Kitzelmann 2010; Gulwani 2010).

Most synthesizers use SAT and SMT solvers. SAT solvers determine whether or not a boolean expression is satisfiable, and if it is, they find a solution (Malik and Zhang 2009). SMT solvers are SAT solvers built to handle more complexity: they are capable of finding solutions to systems of equations involving different theories, such as arithmetic or arrays (de Moura and Bjrner 2011). This has lead to a large amount of success with specific synthesis techniques, and this method has gained popularity among researchers. But, it is not the only viable approach towards program synthesis. Other techniques include genetic programming and other machine learning modalities that can take samples of programs and use the information to create estimations for how a program should look (Cochran et al. 2015).

The objective of our project was to build upon the MYTH synthesis engine. MYTH, unlike the synthesis engines previously discussed, implements a type theoretic model with proof search to synthesize functions from input/output examples. With our tool, SCOUT, we sought to increase the efficiency of the MYTH synthesis engine, as well as the size of the programs that it can generate. By allowing for direct user interaction during the synthesis process, we hope to synthesize larger programs than with a purely automated approach. For example, a user can see a program develop incrementally, which helps with understanding and allows the user to help the synthesizer. In particular, SCOUT allows partial or total completion of specific parts of functions prior to synthesis, while MYTH attempts to synthesize automatically from the input/output examples.

2. Background

MYTH is founded in type theory, which deals with assigning types to terms in a logical system as a means of ensuring correctness. MYTH takes advantage of the fact that the types of terms restrict the possible expressions that can be used to produce a given function in order to guide the synthesis process. To get a sense for the underlying mechanics, we will briefly discuss the basics of type theory.

A type system is an application of type theory to categorize terms in a programming language. It is a collection of rules for assigning types to terms and checking that the types for each expression remain consistent with other expressions. For example, we can see evidence of an underlying type system in the primitive types `bool, double,` and `char` in languages like C and Java. The type system used in MYTH is built on top of a lambda calculus. A lambda calculus is "A formal system in which all computation is reduced to the basic operations of function definition and application" (Pierce 2002). Essentially, a pure lambda calculus takes every construct we are familiar with in functional programming—conditionals, repetition, variables, constants—and reduces them all to function definitions and application. If we take the pure lambda calculus one step further and introduce some typing, we wind up with a simply-typed lambda calculus. In addition to terms, the simply typed lambda calculus adds types, which classify terms, and the type system, which govern which types that result from applying functions and includes a set of inference rules.

MYTH has its own inference rules that are built from the type system of the language. The typing rules take an expression and a context, then apply the variables in the context to the expression to determine the type of the expression. The synthesis rules the system uses effectively reverse this process: given a type and a context, they produce a range of expressions of the given type. Terms fall into two categories: introduction forms and elimination forms. Introduction terms introduce values of a given type and

elimination forms remove values. An example of an introduction form is a let binding, which introduces a function value, or pairs, which introduce two terms. An example of an elimination form is a function application.

When an introduction form is needed, MYTH uses the synthesis rules to produce a range of potential expressions that introduce the appropriate type. The generation of expressions through this process is called I-Refinement. I-Refinement is completely defined by types and examples - the synthesizer can only generate expressions that satisfy the current context and type, since the process is determined by the inference rules. In contrast to the deterministic I-Refinement, there is no method available for deriving code from elimination forms and the synthesizer must guess and check. Accordingly, this process is called E-Guessing. Together, I-Refinement and E-Guessing narrow the search space of terms enough to make synthesis viable. However, the search space still increases exponentially as each step of I-Refinement or E-Guessing often has a branching factor larger than one. The SCOUT system allows the user to select which I-Refinement steps they want the synthesizer to pursue. This takes some of the I-Refinement steps out of the equation entirely, because they are filled in by the user, and reduces the search space by several orders of magnitude.

MYTH stores the range of possible I-Refinements in a tree structure called a refinement tree. Each node of the tree contains the goal type that the user is trying to generate, the names in the scope at the node, the examples to which the node conforms, a list of possible further refinements for the node, and various other fields used for bookkeeping, such as maximum match scrutinee size (the size of the expression the match statement can match on), and whether or not the node is slated for E-Guessing. Possible refinements include match statements, abstractions (i.e., functions), constructors, tuples, records, and units. Match statements in particular add another layer of complexity: for any other refinement, there is only one possible expression that can yield the desired type. Match statements, however, can match on any expression as long as all the branches return the desired type. This flexibility results in an explosion of possible match statements used for refinements. Another important detail is the concept of an I-Refinement skeleton: an I-Refinement skeleton is a series of refinements that yields a partially completed function, which can be filled in by adding further refinements. Each path through the refinement tree represents a different possible I-Refinement skeleton. Our primary development goal was to give the user the ability to choose between and manipulate these skeletons.

3. Overview

The core questions that we sought to answer with this inquiry were:

1. With user interaction during the synthesis process, could a program be synthesized with fewer input/output examples?
2. Could larger and more complex programs be created than were previously possible without user interaction?
3. Is the code created by the tool more efficient in terms of size of the program, or its time or space complexity, than would have previously been possible?
4. Is the tool easier or more helpful to use than its predecessor, the MYTH synthesis engine?
5. Is the code created by Scout more readable to human programmers than the MYTH variant?
6. Is the process of interacting with the synthesizer in this way productive towards understanding how a synthesizer functions and general algorithmic design?

The original MYTH synthesis system is a fully automated program synthesizer that reads in a file containing definitions and examples, and outputs a function satisfying the examples given. Over the course of this project, we incorporated MYTH into the SCOUT system. SCOUT is a semi-automated synthesis system which allows the user to guide the process along. The primary user interaction is deciding which I-Refinement steps the synthesizer uses. The front-end of SCOUT is implemented in Emacs, but in theory it could be implemented in any IDE. In Emacs, the user can run the synthesizer from within an OCaml file to fit naturally with the flow of writing code. The synthesizer uses the definitions in the file from which it is called in the synthesis process. Scout is also capable of reading examples from a file, in case the user needs to store the same set of examples between sessions. We will walk through an example showing synthesis of the Fibonacci sequence.

```
type nat =
  | O
  | S of nat

type bool =
  | True
  | False

let rec sum (n1:nat) (n2:nat) : nat =
  match n1 with
  | O -> n2
  | S n3 -> S (sum n3 n2)
;;
```

Figure 1. Definitions necessary for synthesis

SCOUT can be called while the user is writing normal OCaml code (**Fig. 1**). The parser for SCOUT does not give detailed error messages: it returns the same feedback (Parser Error) regardless of the location of nature of a typo in code, so such typos must be located manually. Moreover, only functions recognized by MYTH can be used in the file.

```
let fib: nat -> nat |>
{ 0 => 1
| 1 => 1
| 2 => 2
| 3 => 3
| 4 => 5
| 5 =>

J:**-  *examples*     All L8
enter result :
```

Figure 2. Entering examples with the minibuffer

After starting SCOUT, the user is prompted in the mini-buffer to add examples to a buffer. The examples are automatically formatted (**Fig. 2**, right), but if the user makes a mistake, they can go back and edit the buffer manually after they finish entering the examples. Alternatively, the user can read in examples from a file. Once the user is satisfied with their example set, they can begin the synthesis process.

```
Current Skeleton:

let fib : nat -> nat =
  [!]
;;

Potential Refined Skeleton (1 of 2):

let fib : nat -> nat =
  fun (n1:nat) -> [?]
;;
```

Figure 3. Initial synthesis state, no I-Refinements chosen

Synthesis begins with no I-Refinements chosen (**Fig. 3**). The display presents the user with a program skeleton featuring all the refinements they have chosen so far as well as one of the next possible refinements the user can choose from.

```
Current Skeleton:

let fib : nat -> nat =
  fun (n1:nat) ->
    match n1 with
      | O -> 1
      | S n2 -> (match n2 with
                   | O -> S [!]
                   | S n3 -> [?])
;;

Potential Refined Skeleton (1 of 2):

let fib : nat -> nat =
  fun (n1:nat) ->
    match n1 with
      | O -> 1
      | S n2 -> (match n2 with
                   | O -> S (O [?])
                   | S n3 -> [?])
;;
```

Figure 4. Some I-Refinements have been selected

The user can select refinements they think are correct to further develop the skeleton, switch between possible refinements, undo previous refinements, and switch which branch of a match statement they are editing (**Fig. 4**).

The synthesizer tries successively larger terms for the E-Guess node each time the Boots synthesize command is run. If the synthesizer finds a solution, it will display it on the screen in functional OCaml code (**Fig. 5**). At this point, the user can copy the resulting function into their function and continue working.

4. Technical Details and System

In order to change MYTH from a fully-automated system to a semi-automated system, we expand the back-end to process and track user input, add a front-end interface, and create functions to intermediate between the two. On the back end, we add an intermediary, dubbed BOOTS, which takes commands from Emacs and passes them to the MYTH system. We also augment the refinement tree structure and accompanying functions. For the interface, we use GNU Emacs. We use Emacs primarily because it is a popular editor for OCaml that provides a good environment for rapid prototyping.

```
let fib : nat -> nat =
  let rec f1 (n1:nat) : nat =
    match n1 with
      | O -> S n1
      | S n2 -> (match n2 with
                   | O -> n1
                   | S n3 -> sum (f1 n3) (f1 n2))
  in
    f1
;;
```

Figure 5. Synthesis is complete

4.1 Back-end

SCOUT's back-end system tracks the state of the I-Refinement tree and the user's position in the tree, progresses the refinement, and returns feedback to the user. The back-end also contains an interpreter that receives commands from the Emacs side of SCOUT, but it is little more than a shell for the data structure and corresponding functions that track the state of the refinement tree and the user's changes.

Branch Complexity Match statements are the source of complexity and refinement tree branching. They are complicated due to the fact that each branch of the match statement corresponds to a new path in the refinement tree. The scrutinee of a match statement can be any expression, though the expressions are limited by size, to ensure that there are a finite number of options. In contrast, for any individual I-Refinements, there is only one constructor that could satisfy it. Thus, the majority of the branching comes from match statements. It is important to mention that all I-Refinement branches in the refinement tree must ultimately end in an E-Guess, since the entire function cannot be I-Refined.

Tracking Position Representing the user's position in the I-Refinement tree is also complex. At each node, there are a variety of potential refinements that the user could take, and at match statements, there are multiple branches the user could edit. Paths through the tree correspond to a series of refinement choices and match branches. To track the user's path, we print the current refinement skeleton and potential future refinement skeletons. The user can select refinements at the ending refinement node and switch between different paths when there is a match statement. To achieve these goals, we considered representing the user's position as a field in the refinement tree data structure. However, in order to switch between paths, we needed some way to access information about other paths. We also considered creating a separate data structure to track user paths through the tree, but decided that this would be inelegant, as it would involve recreating some of the information naturally stored in the refinement tree structure. Ultimately, we settled on a compromise between the two.

We store the user's chosen refinements within the refinement tree structure by adding a chosen refinement field to the refinement tree node, and create external lists to track which branch of the match statement the user is currently editing. Each path is represented by a list. A list of lists, stored in the synthesis problem module, represents all the possible paths through the various match

statements. The user can switch between these paths. We store an additional value to track which potential I-Refinement the user is currently viewing.

Updating the Refinement Tree To progress the refinement, the program adds the current refinement to the chosen node field of the current refinement tree node, changes the match statement path list if necessary, generates the next set of possible refinements, and updates the copy of the refinement tree stored in the problem module. Compared with MYTH, using a choice node field helps cut down dramatically on search space. Rather than having to search several possible I-Refinements, the synthesizer only has to search the E-Guesses that branch off from the path. Each chosen node reduces the search space by roughly an order of magnitude, varying based on the number of I-Refinements available. In order to give feedback to the user, we convert the user's current path into a string of a program skeleton that the user can study. This is done by traversing the current path and converting it into an expression. From there, we use existing pretty printing functions for expressions to display a readable string. At its core, SCOUT stores information about the tree and refinement paths, allows selection of further refinements, and displays the results. However, we present other features, and SCOUT can be extended to allow for more user interaction.

Heuristics Since there are a range of possible I-Refinements that the user can chose from, SCOUT uses heuristics to sort the potential I-Refinements so that useful refinements are more likely to be useful to the user appear first. There will be at least one E-Guess ending a path in every function, since the entire tree cannot be I-Refined. Thus, the E-Guess choice always appears as the last I-Refinement, so that the user can easily access it by using the `back` command.

The heuristic for sorting the I-Refinement is a function that takes an I-Refinement and assigns it a number. Sorting the potential I-Refinements is simply a matter of mapping the heuristic onto the list of refinements and sorting by their numerical value. Currently, SCOUT uses two heuristics: Whenever a constructor is an available heuristic, it places that refinement at the beginning of the list. Afterwards, it sorts refinements by the size of the expression, or in the case of a match statement, the size of the match scrutinee.

Implementing Undo Early test runs of the semi-automated synthesizer indicated that users often make mistakes. Due to the design of the system, a complete restart was required every time the user chose an incorrect refinement. Therefore, an `undo` command was necessary. To support this command, we needed a way to reset the tree and the corresponding paths to earlier states. One option was to simply create a stack of trees and a stack of lists of paths: with this data structure, we could push a tree and its paths onto the stack each time we make a selection and pop the top values off the path each time we use the undo command. We used this approach for resetting paths, but decided that refinement trees were large enough structures that this would unnecessarily increase the burden on memory. Instead, we simply follow the last used path from the stack, move one node up in the refinement tree, then clear the chosen node and its corresponding refinements.

Another approaches we considered was building all the information necessary for the undo function into the tree itself. However, doing this would have been complicated to integrate with the path system, which we rely on for managing SCOUT's synthesis process. This ultimately reflects the general advantages and disadvantages of the path system. One of the struggles with this particular approach is that it is inelegant; handling of paths is not initially easily integrated into the style of the larger program, and ensuring that refinement trees, and the paths are synchronized is not a trivial problem. Many of these difficulties stem from the fact that the path system is much more of an imperative type of programming compared to the much more functional style of rest of the system. These difficulties are inherent to working with the path, and although less elegant, they are easier to work with, and more reliable and efficient than any of the other methods we could devise.

4.2 Design

The design of SCOUT involves an Emacs plug-in that displays refinement trees and allows user interaction. Here we discuss the design of architecture, the refinement tree display, and commands.

Architecture The MYTH synthesizer runs as a process in Emacs and displays the synthesizers output in its own buffer. In addition to running the synthesizer itself, Emacs handles formatting user input and file management, ensuring that the interactive functions behave consistently. By default, BOOTS reads definitions from the users current buffer. BOOTS can either take examples as user input or file input. With user input, the examples are formatted to fit MYTH's specifications. We decided to make user input the default under the premise that the user will want to start the synthesis in the middle of working on a project and have all the project's definitions available to the synthesizer. Reading examples from a file allows the user to store example sets, rather than retyping or copying in a set if they need to exit the synthesis session. We also allow the user to restart the synthesis process. After the initial definitions and examples are entered, regardless of their original format, the restart command brings the user back to a buffer where they can edit the example set. This feature allows users to add examples if they realize that SCOUT isn't synthesizing the intended function, or simply to correct mistakes. With this feature, changing the I/O examples is quick and easy. With MYTH, changing the examples requires editing a separate file.

Refinement Tree Display We display the user's current path through the refinement tree in the form of a partially completed function skeleton as well as the skeleton for the next possible refinement, all displayed in the same buffer. Unless the match statements have a lot of branches, both refinements can fit in this buffer with no scrolling required. Another approach to this interface would be to create multiple buffers, one for the current I-Refinement skeleton and others for the potential future skeletons. However, we felt this would clutter the interface. The simplified, one-buffer approach lets the user see which code they already had in place and cycle through the possible next steps they could take. We chose this approach because it seems unrealistic to expect the user to keep a mental picture of the current skeleton while looking at potential future skeletons. The current tree provides a point of comparison for potential next steps; it is easy to parse the difference between two largely similar blocks of code. The choice also allows us to maintain a relatively straightforward design in terms of the supporting back-end code, as we only have to manage one future option in addition to the user's current path.

User Commands The user commands are intentionally designed to minimize user error and synthesis time, and for ease of use. To understand this design, refer to the image below:

Figure 6. Depiction of Emacs keybindings

The most frequently used commands, `select-skeleton`, `next-skeleton`, and `prev-skeleton`, are shown in green. All the commands have semantic meaning, e.g. select as "s," next as "f" for forward, "b" for back. But, most commands are also designed for ease of use. All the interactive Emacs commands begin with `M-b`, so it is helpful that the most common commands are in proximity to the "B" key.

The commands in yellow are less frequently used. Since the number of paths to focus on is generally very small, `next-path` and `prev-path` are effectively redundant. `next-path` is placed in proximity of the "B" key, and `prev-path` is placed far away. `undo-selection` is also infrequently used, so it's simply `M-b M-u`.

The commands depicted in red are commands the user rarely needs. Typing `M-b M-q` will quit the process, with no confirmation, so it is far away from the other commands. Similarly, the restart command is relatively far away. Both of these commands are memorable. The only command without semantic meaning is "synthesize," placed at "K." It is placed here arbitrarily, but with the intention that synthesizing only happens at the end of the tool-assisted synthesis process.

Future versions of BOOTS could allow users to edit these key-bindings; a keyboard user using a DVORAK or similar setup would not benefit much from these bindings.

4.3 BOOTS Commands

The full set of commands for running BOOTS in Emacs are:

- `M-x start-examples`: Opens a new buffer in which the user can enter examples for Scout to use in the synthesis process. Automatically copies definitions from whichever buffer the user called the command from.
- `M-x read-from-file`: Reads in examples from a file that the user provides. This function assumes that the buffer that the user runs the command from has the type definitions already in it.
- `M-x read-myth-file`: Reads in a file that the user provides. This function makes no assumptions about the buffer the user runs the command from, but makes the assumption that the file given is formatted for myth, and thus includes type definitions
- `M-x end-examples`: When you are done entering your examples, navigate to the `*examples*` buffer, and enter the end-examples command. This will start boots, and give the definitions and examples as input to `init`. You should then see the `*boots*` buffer appear on your screen, with the initial function representation.

The commands for the BOOTS process and their associated key bindings are:

- Select-skeleton (`M-b M-s`): Selects the current program skeleton to refine further. Use Next and Prev to swap between refinements.
- Next-skeleton (`M-b M-f`): Moves the problem onto the next program skeleton. (this is a skeleton who varies by a refinement in just one branch).
- Prev-skeleton (`M-b M-b`): Moves the problem onto the next program skeleton (this is a skeleton who varies by a refinement in just one branch).
- Next-path (`M-b M-n`): Moves the focus of the synthesis to the next path in the skeleton indicated by the [?] symbol.
- Prev-path (`M-b M-p`): Moves the focus of the synthesis to the previous path in the skeleton indicated by the [?] symbol.
- Undo-selection (`M-b M-u`): Undoes one selection made by the user. Switches to the branch the user was on when the previous selection was made.
- Restart (`M-b M-r`): Restarts the synthesis process, and brings the user back to a buffer containing all examples
- Synthesize (`M-b M-k`): Attempts to synthesize a solution from the current skeleton. Each call of the function increases the size of the expression that can be E-Guessed by one.
- Quit-boots (`M-b M-q`): Exits the process.

On the back-end, SCOUT has a separate series of commands to run the appropriate processes. The Emacs file calls combinations of these commands whenever the Emacs commands are used. The SCOUT commands include:

- `init`: Initializes the synthesizer with declarations and a synthesis problem. All subsequent commands require that `init` has been successfully called.
- `fetch`: Returns the current synthesis skeleton in a printable form, the next possible skeleton currently being considered, and the I/O examples associated with the current branch in a string.
- `prev`: Moves to the previous possible synthesis skeleton at the current refinement tree node.
- `next`: Moves to the next possible synthesis skeleton at the current refinement tree node.
- `select`: Adds the skeleton being considered to a new `rnode`, expands the refinement tree, and takes a snapshot of the state of the paths for the undo function.
- `synthesize`: Attempts to synthesize a final solution path the user has selected in the refinement tree.
- `forward`: Moves to the next path to edit.
- `backward`: Moves to the previous path to edit.
- `undo`: Undoes the previous selection.
- `quit`: Terminates this Boots process.

5. Results

We tested SCOUT's usability, the efficacy of I-Refinement skeleton ordering heuristics, and capacity to synthesize novel functions by performing a case study of functions that draw from propositional logic.

Propositional logic is a simple logical system that combines symbols, which can have a true or false value, and logical connectives, such as `NOT`, `OR`, `AND`, or `IMPLIES`, to form logical sentences. We focused on developing functions and helper functions for logical entailment, which is testing to see whether one logical sentence implies another, and conversion of a proposition sentence to conjunctive normal form, which is a sentence which only uses `OR` connectives separated by `AND` connectives and has all `NOT` connectives in front of symbols.

For logical entailment, we synthesized two functions. The first takes a logical sentence as an input and produces a list of all the symbols in the sentence as an output. The second applies a model (a mapping between symbols and true/false values) to a sentence, substituting the true/false values for the symbols in a sentence, and returning whether or not the resulting sentence evaluated to true or false. Both synthesized functions used only a single match statement, with recursive calls in every branch but the base case. The input/output sets for these functions were produced with the example generator. The functions were relatively easy to synthesize with SCOUT, and we were able to synthesize them on the first attempt.

```
Bool     ::=  True | False
Sym      ::=  x
Sen      ::=  Sym | ¬Sen | ∧ Sen Sen | ∨ Sen Sen | ⇒ Sen Sen | ⇔ Sen Sen
Model    ::=  MNil | (Sym * Bool) * Model
SymList  ::=  Nil | Sym * SymList
World    ::=  Model | World * World
```

Figure 7. Propositional Logic Grammar

For reduction to conjunctive normal form, we synthesized functions that eliminated biconditionals and implications from a sentence, and attempted to synthesize functions that moved NOT, AND, and OR statements inward to their appropriate places for CNF. The experiments with reduction to conjunctive normal form used a slightly different grammar than those for the logical entailment functions: The only difference is that a sentence can also be a boolean. The biconditional and implication elimination both used only a single match statement with recursive calls in each case except the base cases. They were straightforward to synthesize, only requiring one attempt. See figures 8 and 9.

```
let remove_bicond : sentence -> sentence =
 let rec f1 (s1:sentence) : sentence =
   match s1 with
     | Bool b1 -> s1
     | Sym s2 -> s1
     | Not s2 -> Not (f1 s2)
     | And (s2, s3) -> And (f1 s2, f1 s3)
     | Or (s2, s3) -> Or (f1 s2, f1 s3)
     | Impl (s2, s3) -> Impl (f1 s2, f1 s3)
     | BImpl (s2, s3) ->
             And (Impl (f1 s2, f1 s3),
                  Impl (f1 s3, f1 s2))
 in
   f1
 ;;
```

Figure 8. Synthesized Remove Biconditional (28 examples)

```
let remove_impl : sentence -> sentence =
 let rec f1 (s1:sentence) : sentence =
   match s1 with
     | Bool b1 -> s1
     | Sym s2 -> s1
     | Not s2 -> Not (f1 s2)
     | And (s2, s3) -> And (f1 s2, f1 s3)
     | Or (s2, s3) -> Or (f1 s2, f1 s3)
     | Impl (s2, s3) -> Or (Not (f1 s2), f1 s3)
     | BImpl (s2, s3) -> BImpl (f1 s2, f1 s3)
 in
   f1
;;
```

Figure 9. Synthesized Remove Implication (23 examples)

The elimination of NOT, AND, and OR forms proved more difficult. We were only able to synthesize functions that eliminated a single NOT, AND, or OR sentence. The resulting functions had match statements that went two or three match statements deep. For the AND and OR elimination functions, we built helper functions to reduce the number of match statements needed: Since the AND and OR logical operators are commutative, we rearranged some of the arguments so we could always predict which connective would be in the first or second spot in the AND or OR constructors (see figures 10 through 12).

```
let neg_elim : sentence -> sentence =
  fun (s1:sentence) ->
  match s1 with
   | Bool b1 -> s1
   | Sym s2 -> s1
   | Not s2 ->
     (match s2 with
      | Bool b1 -> Bool (match b1 with
                         | True -> False
                         | False -> True)
      | Sym s3 -> s1
      | Not s3 -> s3
      | And (s3, s4) -> Or (Not s3, Not s4)
      | Or (s3, s4) -> And (Not s3, Not s4)
      | Impl (s3, s4) -> s1
      | BImpl (s3, s4) -> s1)
   | And (s2, s3) -> s1
   | Or (s2, s3) -> s1
   | Impl (s2, s3) -> s1
   | BImpl (s2, s3) -> s1
;;
```

Figure 10. Synthesized Negation Elimination (17 examples)

From the more complex AND, OR, and NOT elimination functions, we found that most of the usability issues center around the difficulty in finding a complete example set. Since the MYTH synthesizer cannot evaluate recursive function calls before a function is completed, it requires that every simpler case that occurs when evaluating an input is provided as a distinct example (for example, with natural numbers, 0 is required as an example before 1 can be given). When insufficient examples are provided, MYTH will return an incomplete example set error as synthesis is attempted. Moreover, the match statements involved with propositional logic often used the five logical connectives as cases. Since MYTH currently needs an example for each branch, having match statements with several cases also results in a blowup of the number of examples necessary. These difficulties with the number of examples occur in both Scout and MYTH, and from our limited testing, it does not seem that Scout significantly reduced the number of required examples. Together, these issues made creating a thorough match set almost prohibitively time consuming, and although SCOUT allows for our goal of semi-automated program synthesis, we currently have no evidence to support our initial theory that it would reduce the required number of examples. This is particularly challenging, as the majority of time developing the functions was spent creating example sets, rather than actually running synthesis or thinking about the structure of functions, and reducing the number of required examples seems crucial for improving SCOUT's future usability.

Again, with the AND, OR, and NOT elimination functions, the need for precise example sets made it difficult to study SCOUT as

```
let and_elim : sentence -> sentence =
  fun (s1:sentence) ->
  match swap s1 with
  | Bool b1 -> s1
  | Sym s2 -> s1
  | Not s2 -> s1
  | And (s2, s3) ->
     (match s3 with
      | Bool b1 -> (match b1 with
                    | True -> s2
                    | False -> s3)
      | Sym s4 -> s1
      | Not s4 -> s1
      | And (s4, s5) -> s1
      | Or (s4, s5) ->
         And (Or (s2, s4), Or (s2, s5))
      | Impl (s4, s5) -> s1
      | BImpl (s4, s5) -> s1)
  | Or (s2, s3) -> s1
  | Impl (s2, s3) -> s1
  | BImpl (s2, s3) -> s1
;;
```

Figure 11. Synthesized And Elimination (15 examples)

```
let or_elim : sentence -> sentence =
  fun (s1:sentence) ->
    match swap s1 with
    | Bool b1 -> s1
    | Sym s2 -> s1
    | Not s2 -> s1
    | And (s2, s3) -> s1
    | Or (s2, s3) ->
       (match s3 with
        | Bool b1 -> (match b1 with
                      | True -> s3
                      | False -> s2)
        | Sym s4 -> s1
        | Not s4 -> s1
        | And (s4, s5) ->
           Or (And (s2, s4), And (s2, s5))
        | Or (s4, s5) -> s1
        | Impl (s4, s5) -> s1
        | BImpl (s4, s5) -> s1)
    | Impl (s2, s3) -> s1
    | BImpl (s2, s3) -> s1
;;
```

Figure 12. Synthesized Or Elimination (15 examples)

a tool for synthesizing novel functions. The example set constantly needed tweaking, and as a result we usually figured out how to write the function before SCOUT could synthesize it. Admittedly, the functions we chose to synthesis were limited in complexity and usually didnt go more than a few match statements deep. After observing the limitation of example generation, we shifted gears to find ways around this limitation.

Example Generation After struggling with creating sufficient example sets, we developed a simple example generator assistant program. It takes a function, inputs, and a formatting helper function, and writes a formatted set of examples to a file for the synthesizer to use. One key part of the generator was a set of helper functions, tailored to the data types being studied, that took an input example and recursively generated a list of all the constructors that were used in the constructor for the input. Thus, given a complex input example, these helper functions could unpack all of the relevant recursive cases to the original input example.

```
let extract_symbols : sentence -> symbol_list |>
    { Impl(And(Not(Sym(P)), Not(Sym(Q))), Impl(Not(Sym(R)), Not(Sym(S)
    | And(And(Not(Sym(P)), Not(Sym(Q))), Impl(Not(Sym(R)), Not(Sym(S))
    | Not(Sym(S)) => Cons (S, Nil)
    | Not(Sym(R)) => Cons (R, Nil)
    | Impl(Not(Sym(R)), Not(Sym(S))) => Cons (S, Cons (R, Nil))
    | Not(Sym(P)) => Cons (P, Nil)
    | And(Not(Sym(P)), Not(Sym(Q))) => Cons (Q, Cons (P, Nil))
    | Or(And(Not(Sym(P)), Not(Sym(Q))), Impl(Not(Sym(R)), Not(Sym(S)))
    | Sym(T) => Cons (T, Nil)
    | Sym(S) => Cons (S, Nil)
    | And(Sym(S), Sym(T)) => Cons (T, Cons (S, Nil))
    | Not(And(Sym(S), Sym(T))) => Cons (T, Cons (S, Nil))
    | Sym(R) => Cons (R, Nil)
    | Or(Sym(R), Not(And(Sym(S), Sym(T)))) => Cons (T, Cons (S, Cons (
    | Sym(Q) => Cons (Q, Nil)
    | Not(Sym(Q)) => Cons (Q, Nil)
    | Sym(P) => Cons (P, Nil)
    | Impl(Sym(P), Not(Sym(Q))) => Cons (Q, Cons (P, Nil))
    | Bicond(Impl(Sym(P), Not(Sym(Q))), Or(Sym(R), Not(And(Sym(S), Sym
    } = ?
```

Figure 13. Example generator output (clipped to fit)

For a single example, these functions create precisely the set of input examples required to satisfy all simpler recursive cases within the example. When these unpacking functions are applied to multiple input examples, the set of examples created might contain some redundancies, as simple constructor cases from one input example might be the same as the simple cases from another. In order to remove these redundancies from a list of examples used for the synthesizer, we would need some way to check if two expressions were the same. We did not implement this feature, since it did not seem to have a direct bearing on synthesis. Thus, the unpacking functions can led to a superset of the examples necessary to account for recursive cases.

Additionally, it was not clear whether or not the set of input examples generated was precisely the set needed to generate a given function; the generator took the a few complex examples, unpacked them into lists containing all simpler recursive cases along with the original examples, then fed the resulting set of inputs through the desired function and a formatter. It's entirely possible that the initial complex examples were more complex than necessary, leading to a superset of examples necessary to synthesize the function. We were unable to synthesize precisely the set needed to synthesize the function because it's unclear how complex of a case we need to account for. The unpacking functions simply account for all recursive subcases; the example generator does nothing explicitly to give more information on the minimum number of examples needed to synthesize the function.

Another major disadvantage of using the example synthesizer is that it needs the function we are trying to synthesize to process the example set. Thus, we cannot use it to test the synthesis of novel functions. However, the generator does make the generation of large, complete example sets much easier. We used the example generator to help study heuristics, usability, and compare SCOUTs capabilities to MYTHs with tests involving the previous propositional logic systems and rational number functions.

Testing Heuristics Throughout testing, we observed that our heuristics are effective. From the functions observed, our basic smallest skeleton first and "constructors first" heuristics work in the majority of cases: when an E-Guess is not necessary, the desired skeleton can be found in the first ten suggestions for over 90% of tests we ran. However, our capacity to test was limited by the

fact that most of the time running tests was spent tweaking example sets, rather than running synthesis and studying where the desired I-Refinements were. We also observed that adding functions that the synthesizer can use greatly increases the number of match statements the synthesizer presents as potential I-Refinement, so further heuristic development will be necessary as SCOUT is integrated with established libraries.

Summary Overall, the tests found SCOUT's capabilities to be roughly comparable to MYTH's; for all the functions tested, MYTH was able to synthesize the function in a relatively short period of time, no more than a few seconds. SCOUT was also able to synthesize the functions tested with standard guidance from the user. An example set with around 5,000 examples demonstrated one notable exception between the two systems. At this point, MYTH took over 20 minutes and failed to synthesize the function, while SCOUT remained functional, albeit with several second delays between each selection step. MYTH had to handle hundreds or thousands of examples at each step, which drastically slowed its speed. While SCOUT had to manipulate the same quantity of information, it was able to stop at each I-Refinement step and wait for user input, resulting in less time spent searching overall. This suggests that SCOUT is much more usable if a function needs to be synthesized based on an example set with thousands of input/output example pairs. Further tests are needed to determine how number of examples affect the speed of the synthesizers.

For all the functions we tested with SCOUT, MYTH was also capable of producing the same function with the same example set. There was one example where, when given a file containing more helper functions than necessary, MYTH used a redundant function call where SCOUT did not. However, for the most part, the functions produced by both were identical. We found that SCOUT was more helpful than MYTH in that we were able to see the terms the MYTH synthesizer was considering for I-Refinement. These observation were helper in terms of understanding the possibilities the synthesizer was sorting through during the I-Refinement steps. We were able to examine first hand how possibilities for I-Refinement increase as more function are introduced to the file and the size of the expression that can be used in match statements is increased. Whether or not these observations are helpful for understanding general algorithmic design is unclear, as we were only able to conduct tests on ourselves and we already understood the design of all the functions we tried to create. In terms of usability, the same difficulties in choosing example sets remained consistent for both in the examples we tried. Since the code both synthesizers created was nearly identical, we can't say that the code created by SCOUT was more human readable.

6. Related Work

The prior work in this area is numerous and varied. The key features of SCOUT, are the way in which it uses type theory as a central part of its synthesis, and the close user interactions with the synthesis process. To compare prior work to SCOUT, we can categorize the most relevant prior work as follows: work in which type theory is part of the primary synthesis process, and work which user interaction is very ingrained into the synthesis process work. If the work does not rely on type theory (e.g., an SAT/SMT solver) to ensure correctness, we will consider its interactibility instead.

Type Theory SCOUT is not the only tool which uses types as central to its processing. A recent paper implemented a synthesizer called INSYNTH that relied on the use of types as a Scala plugin for eclipse (Gvero et al. 2013). Although this work is similar to our own with regards to relying on types, it is unlike our own in its goals, and focus. SCOUT is capable of synthesizing functions that use conditionals, while INSYNTH isn't. INSYNTH is also much more focused on handling large amount of external libraries, which is currently unimplemented in our tool. SCOUT has a much more active role in the synthesis process, directly guiding the synthesizer while it is synthesizing, as opposed to selecting the final result from a list of possibilities.

SAT/SMT Solvers The synthesizer and programming language SKETCH also takes advantage of user interaction to guide a synthesis process (Solar-Lezama 2008). It allows users to specify high level descriptions of programs, and then leave holes for a synthesizer to complete. This has some conceptual similarities to our work, but differs in that the underlying synthesizer used by Sketch is a SAT solver as opposed to a system which relies on type theory. In addition, The user in SKETCH gives specifications before the synthesis process, but doesn't directly guide the synthesizer as the user does with SCOUT.

LEON is the tool most similar to SCOUT (Blanc et al. 2013) (Koukoutos et al. 2016) (Antognini et al. 2015). LEON is a program verification tool written for Scala that is capable, among other things of synthesizing and executing functions from input and output cases. The primary difference between our approaches is that LEON uses an SMT solver as opposed to using type theory and proof search (Blanc et al. 2013). SCOUT also has the user provide input during the synthesis process to help guide it, while in LEON and all other synthesis tools we have observed, the user has only provided input before or after synthesis.

Other Solvers Programming with Angelic nondeterminism is one way in which a user can interact with a synthesis process (Bodik et al. 2010). An Angelic operator is an operator which if possible returns the value necessary to terminate the program. An implementation of this operator can be used to allow users to write part of a program, and be assured that it will terminate. This approach takes advantage of user and synthesizer interaction in a manner similar to SKETCH, and similarly does not rely on active guidance by the user during the synthesis process, and does not have type systems as an integral part of the system.

7. Future Work

Future work can be divided into roughly three categories: improving the capabilities of the Myth synthesis system, improving the usability of SCOUT, and further testing.

Improving* MYTH *Capabilities Currently, MYTH only has access to a limited number of language features, which somewhat restricts the complexity of the functions that it can produce. We would like to eventually expand the language features available to MYTH, such as adding polymorphism or pattern matching wildcards. Polymorphism would allow us to design more general functions with list and wildcards, making synthesizing complex match statements considerably more concise. Such expansions would require both an expansion of the type theory basis of the synthesizer and developing corresponding implementations to allow the synthesizer to include these functions in its term generation. Along the same lines, we would like to eventually develop a way for MYTH to have access to third party libraries. This will be essential if SCOUT is ever to become a useful development tool. However, giving MYTH access to third party libraries will quite likely cause an explosion in the number of possible I-Refinements and E-Guesses generated, which threatens the efficient run-time of synthesis. Developing better I-Refinement and E-Guess heuristics are a potential way to circumvent this problem.

Improving* SCOUT *Usability There are a variety of potential features that could be added to make the SCOUT system more user-friendly and effective. One of the more effective methods would

to install asynchronous processing capabilities. This improvement would allow Scout to continuously try to synthesize the desired function with whatever information it has while the user thinks about and chooses further I-Refinements. Asynchronous processing would allow the Scout system to have the best of both worlds - if the function is relatively simple, the system could finish quickly on its own and return a function. If the function is more complex, the system would still be able to wait for the user to enter more I-Refinements and search ahead of where the user currently is.

Another method for streamlining the synthesis process would be to improve the heuristics used to order the potential refinements, which would reduce the number of skeletons the user has to look through. Since most potential refinements currently involve match statements, the difficulty is in figuring out which match statement scrutinees yield valuable information. We would need a method that distinguishes the functions it doesn't make sense to call in a match statement, such as a concatenation function in a sorting algorithm, from the functions that split the cases well, such as a comparison function in the same sorting algorithm. Another approach is to "throw out" redundant expressions that can be expressed in a simpler way (such as `append lst []`).

A potential improvement during the synthesis process would be to give the user the ability to fill in variables and functions in the E-Guess spots in the I-Refinement skeleton. Doing so would require commands to allow switching from choosing I-Refinements to entering values for E-Guessing, a way to pass what the user writes to the synthesizer during the synthesis process, and typechecking to make sure what the user has entered is an expression of the correct type. Additionally, We could experiment with different types of displays and methods for interfacing with the synthesizer. We chose the current display for its simplicity and easiness while prototyping, but it's possible that a more developed display could be more useful to the user.

Stepping outside of the synthesis process, there are more ways to streamline the user experience. Supporting more features of OCaml will allow us to integrate the system more closely into a real development environment, with standard libraries. One improvement would be to remove the restriction of the synthesizer needing one example for each branch. This possibly has drawbacks, though. If the synthesizer doesn't need a complete example set, the refinement skeletons may grow too large to make this a useful change. Alternatively, SCOUT could check if an insufficient example set has been provided before the synthesis process begins, and prompt the user to fill in the remaining examples.

Scout as a Teaching Tool One unexpected finding from working with the SCOUT system was discovering that SCOUT might prove to be an effective teaching tool in a higher education setting, particularly for the subjects of unit tests and modularity. The SCOUT system can provide concrete examples for what good unit test cases might look like through the example sets that need to be written to use the synthesizer. Since a complete example set needs to cover all recursive cases and branches of the completed function, it would force students to consider most of the edge and corner cases that a function could encounter. The synthesizer also encourages brevity in writing test cases since writing out cases by hand is time-consuming and tedious, which would make most students want to write as few as possible. While the nature of example sets naturally support the writing of unit tests, the limitations and quirks of the subset of OCaml that MYTH uses promote good modularity. Since MYTH can only synthesize functions with up to two parameters, it would force computer science students to design the functions they want to synthesize in a highly modular fashion. This modularity is further encouraged by the fact that MYTH does not have access to let bindings, further breaking up how functions have to be written. While such an extreme level of modularity might not be advisable in real world coding projects, it may be useful for students to observe such modularity. Of course, the MYTH and SCOUT systems were not designed as teaching systems and would require a fair amount of development before they are usable as teaching tools.

Further Testing One major area to work on going forward is testing synthesis for a variety of different types of functions and conducting further subject case studies. As we expand MYTH and SCOUT, we will want to design and test new heuristics to ensure that the relevant I-Refinements stay near the front of the list. Additionally, with the help of the example generator, we can test more complex known functions and use them to compare MYTH and SCOUT. The example generator can also be used to test how MYTH and SCOUT handle different number of examples and where the two are limited in terms of the number of examples they can handle. Finally, we will need to run more usability tests on the learning curve for using SCOUT, as we have only tested SCOUT on ourselves. Usability tests should allow us to get a better sense of the strengths of the system and provide ideas for further improvements.

8. Conclusion

At the beginning of the project, we set out to expand the MYTH synthesizer to a semi-automated synthesizer and test whether or not this expansion makes the synthesizer more usable and capable. Our expansion of the synthesizer was successful - we added a front-end interface in the Emacs text editor and augmented MYTH's infrastructure to support interactions with the front end. As designers of the system, we were able to become quite proficient with the tool, but have yet to see how accessible it is to programmers without the same background knowledge. Additionally, the development of the SCOUT synthesizer lays the foundation for further expansion and development of the interactive synthesizer.

After creating the synthesizer, we ran a limited range of tests, including the development of various functions in the domain of propositional logic and rational numbers. These tests revealed that the most significant limitation to making SCOUT a practical tool is the difficulty of creating complete example sets. While our results do not show practical advantages of SCOUT over MYTH *right now*, the SCOUT system we have built allows for future enhancements, and there are still many more tests that can be run to get a better sense of the full capabilities of SCOUT vs. MYTH.

References

M. Antognini, R. Blanc, S. Gruetter, L. Hupel, E. Kneuss, M. Koukoutos, V. Kuncak, S. Stucki, and P. Suter. Synthesis, 2015. URL `https://leon.epfl.ch/doc/synthesis.html`.

R. W. Blanc, E. Kneuss, V. Kuncak, and P. Suter. An overview of the Leon verification system: Verification by translation to recursive functions, 2013.

R. Bodik, S. Chandra, J. Galenson, D. Kimelman, N. Tung, S. Barman, and C. Rodarmor. Programming with angelic nondeterminism. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 339–352, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-479-9. doi: 10.1145/1706299.1706339. URL `http://doi.acm.org/10.1145/1706299.1706339`.

R. A. Cochran, L. D'Antoni, B. Livshits, D. Molnar, and M. Veanes. Program boosting: Program synthesis via crowd-sourcing. *SIGPLAN Not.*, 50(1):677–688, Jan. 2015. ISSN 0362-1340. doi: 10.1145/2775051.2676973. URL `http://doi.acm.org/10.1145/2775051.2676973`.

L. M. de Moura and N. Bjrner. Satisfiability modulo theories: Introduction and applications. 2011. URL `http://www.cis.upenn.edu/~alur/CIS673/smt11.pdf`.

S. Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Prin-*

ciples and Practice of Declarative Programming, PPDP, 2010. URL http://research.microsoft.com/en-us/umm/people/sumitg/pubs/ppdp10-synthesis.pdf.

T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac. Complete completion using types and weights. In H.-J. Boehm and C. Flanagan, editors, *PLDI*, pages 27–38. ACM, 2013. ISBN 978-1-4503-2014-6. URL http://dblp.uni-trier.de/db/conf/pldi/pldi2013.html#GveroKKP13.

E. Kitzelmann. Inductive programming: A survey of program synthesis techniques. pages 50–73, 2010. URL http://www.cogsys.wiai.uni-bamberg.de/aaip09/aaip09_submissions/inductive.pdf.

M. Koukoutos, E. Kneuss, and V. Kuncak. An update on deductive synthesis and repair in the leon tool. In *5th Workshop on Synthesis*, 2016.

K. R. M. Leino and A. Milicevic. Program extrapolation with jennisys. *SIGPLAN Not.*, 47(10):411–430, Oct. 2012. ISSN 0362-1340. doi: 10.1145/2398857.2384646. URL http://doi.acm.org/10.1145/2398857.2384646.

S. Malik and L. Zhang. Boolean satisfiability from theoretical hardness to practical success. *Commun. ACM*, 52(8):76–82, Aug. 2009. ISSN 0001-0782. doi: 10.1145/1536616.1536637. URL http://doi.acm.org/10.1145/1536616.1536637.

Z. Manna and R. Waldinger. Synthesis: Dreams programs. 1979. URL https://www.computer.org/csdl/trans/ts/1979/04/01702636.pdf.

B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, Massachusetts, 2002.

N. Polikarpova and A. Solar-Lezama. Program synthesis from polymorphic refinement types. *CoRR*, abs/1510.08419, 2015. URL http://arxiv.org/abs/1510.08419.

A. Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California, Berkeley, 2008. URL https://people.csail.mit.edu/asolar/papers/thesis.pdf.