COMP703 Project Proposal

Travis Pence and Lao Chang

November 2023

1 Background

Program synthesis is having a machine write a program to do a task for you. The user must specify what they want the machine to generate, which can be done in several different ways. One is providing inputs and corresponding outputs, which is how Microsoft Excel fills in cells based on examples.

This project implements program synthesis by constructive logic, which one can learn more about on wikipedia or through course materials from Carnegie Mellon. To understand our project, one must be familiar with constructive logic as our description of what the machine should do is formulated as a theorem and then proved using said logic. Proofs in constructive logic are in the form of "trees," meaning they start at a root node and branch out to the leaves.

One must also understand the concept of code extraction from a proof. Starting at the leaves of a proof, code is generated for each step of the proof tree corresponding to the rule in constructive logic used, working down until the root node, where the final program is given. A more detailed explanation is given in Dr. Pfenning's course.

2 Problem Statement

We aim to build a program synthesis tool which, when given a theorem in constructive logic, proves the theorem using said logic and subsequently extracts a program from the generated proof.

3 Rationale

We chose to explore this problem because of our interest and fascination with program synthesis. The ability for a program to write itself does truly seem like the "holy grail" of computer science. As Pfenning's course does not cover proof generation, we wanted to complete his approach and provide an automated theorem prover along with code to extract the proof. Implementing a programming synthesis technique from the ground up will also help cement our understanding of the underlying techniques used.

4 Objectives and Scope

Our project will be broken down into two pieces: theorem proving and program extraction. None of our project will come from open-source sites as we expect to implement both ourselves, although this may change if we find that an outside tool is useful. On the theorem proving side, we will express the logic system in code, and prove the theorem using brute force, trying each possible rule and going down that path if the rule applies.

For program extraction, we must implement the given rules in code in Pfenning's course for every rule which could be used in theorem proving. The actual program extraction will most likely be done as a recursive function starting at the bottom of the proof tree, working its way up to the top leaves, and adding the corresponding code on its way back down. We expect this approach to work on any valid proof generated, as it is "turning the crank". In Pfenning's course, after the program synthesis he adds the ability to generate code dealing with basic data types such as integers and lists. An ambitious goal would be to implement the given proof rules as well as code extraction rules for these data types.

We will measure the success of program synthesis by first determining if the theorem prover can prove relatively small theorems without running out of memory or taking too long. By relatively small, we mean examples such as those given in the slides (swap) or other proofs of similar height and width. We aim and believe that the theorem prover will be able to find bigger proofs for "harder" theorems, but do not know the scope of provable theorems using the brute force approach. If our theorem prover cannot find a proof in a reasonable amount of time or memory, we will consider that theorem out of scope for this project.

Our second measure of success is determining if the extracted program is correct, i.e., gives the expected output for certain input. If time allows, we would like to verify the generated program is correct using an outside checker, however, we will consider it adequate to write our own version of the correct program and test for equality of output for a large number of random inputs.

We will obtain the test cases, i.e. the theorems, by writing them ourselves. We do not know of another way to generate theorems which are (i) true, (ii) provable in a reasonable amount of time and memory, and (iii) give programs which we can easily know what they do. We expect to write at least 10 - 15 theorems.

See below for the breakdown of the project into milestones.
Milestone 1 (10/10): Have several theorems written, proved, and code extracted by hand. These will look similar to the example of "swap" given in the slides, but in the language of Pfenning's course. We would also like to have the theorem prover done by milestone 1 as we expect it to be the easier part of the project, but this depends on the progress and difficulties we run into while developing the theorem prover.

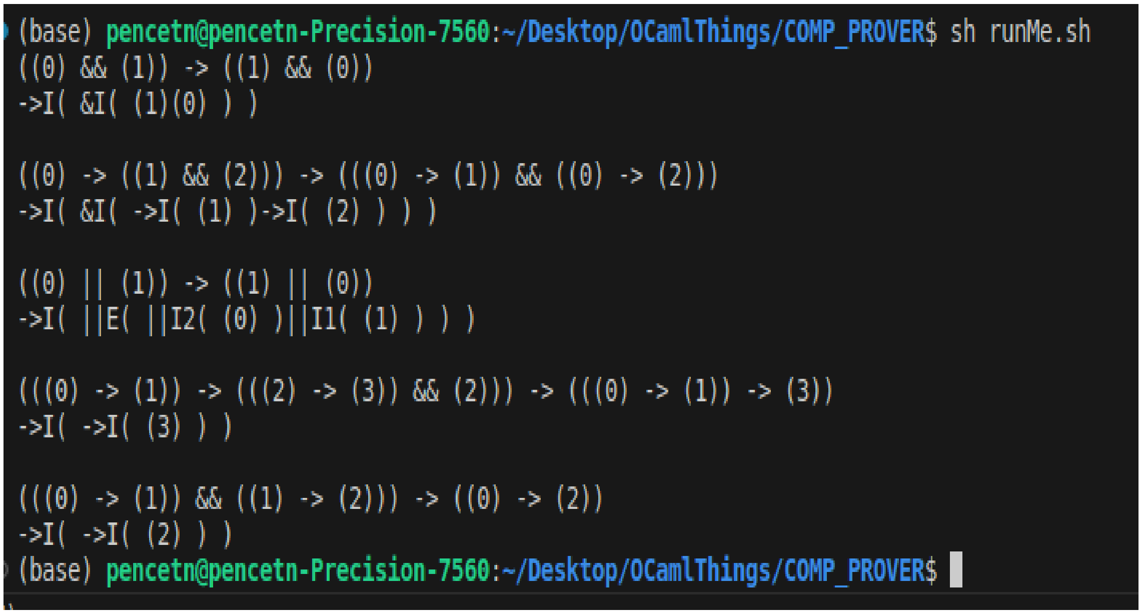
Milestone 2 (10/22): We will have the theorem prover done by this milestone if not completed by milestone 1. We aim to have part of the program extractor written as well. We will implement the "easier" rules first, and hope to have it working for small programs, such as swap. We plan to finish half of the program extractor in this milestone.

Final Turn-in (TBD): The program extractor will be done by the end of the semester, and the theorem prover and program extractor tested on at least 10 - 15 theorems. We are confident this is achievable. If we have time, we would also like to add the notion of integers, and then lists to the possible programs we can generate, but this depends on our progress throughout the semester.

5 Appendix A: Milestone 1

We have the theorem prover working for the introduction and elimination rules for implication, conjunction, and disjunction. Pfenning's course represents A as $A \rightarrow Falsehood$, which we are still working on implementing. You can see our progress on github **here**. We are using the latest version of OCaml, 5.1.0. Note that the latest version of OCaml for windows (Diskuv OCaml) is 4.1.4, which will NOT work as some API methods have changed name. To run the file with our code "COMP_PROVER/bin/main.ml", run the shell script "runMe.sh". The code has obvious naming and is commented for readability.

At the end of the file are five examples, most from Pfenning's course and one that we came up with to test that certain functionality was working. Each theorem is first printed, then proven resulting in a proof, and then the proof printed. The results are shown below (which can also be seen by running the file), and are the expected results.



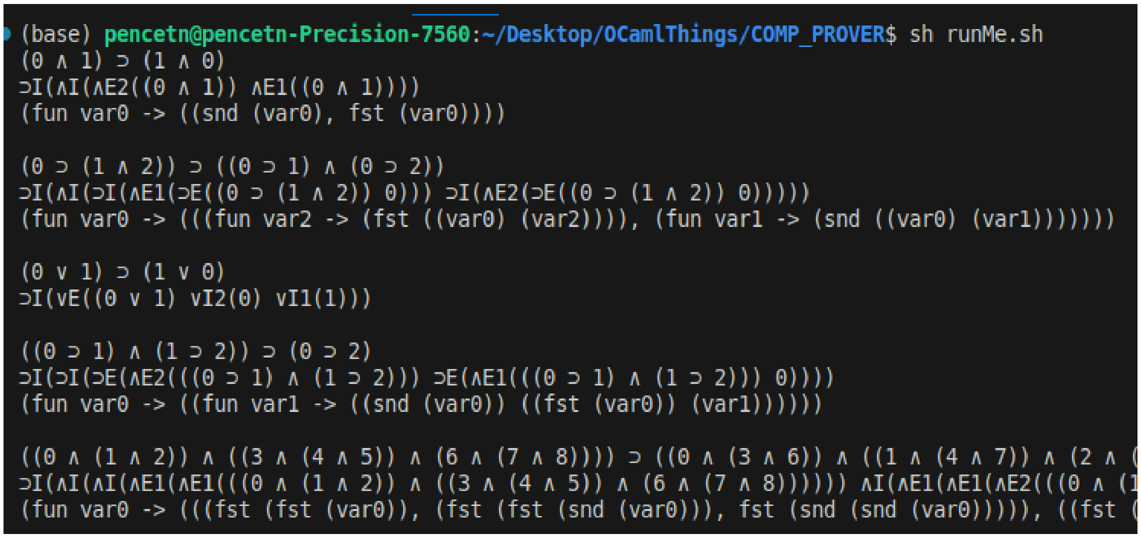
```
(base) pencetn@pencetn-Precision-7560:~/Desktop/OCamlThings/COMP_PROVER$ sh runMe.sh
((0) && (1)) -> ((1) && (0))
->I( &I( (1)(0) ) )

((0) -> ((1) && (2))) -> (((0) -> (1)) && ((0) -> (2)))
->I( &I( ->I( (1) )->I( (2) ) ) )

((0) || (1)) -> ((1) || (0))
->I( ||E( ||I2( (0) )||I1( (1) ) ) )

(((0) -> (1)) -> (((2) -> (3)) && (2))) -> (((0) -> (1)) -> (3))
->I( ->I( (3) ) )

(((0) -> (1)) && ((1) -> (2))) -> ((0) -> (2))
->I( ->I( (2) ) )
(base) pencetn@pencetn-Precision-7560:~/Desktop/OCamlThings/COMP_PROVER$ 
```

We decided to represent single propositions as integers. For example, the first theorem is swap "$(A \wedge B) \to (B \wedge A)$", with the corresponding proof of using the introduction implication rule " $\to I$, then the introduction conjunction rule $\wedge I$, which then is proven using the derived axioms B and A. Note that this proof does not show the elimination conjunction rule. Our prover currently does not keep track of using the elimination rules for conjunction or implication, which we need to add for milestone 2. our prover does keep track of all introduction rules as well as the elimination rule for disjunction (example in the third theorem).

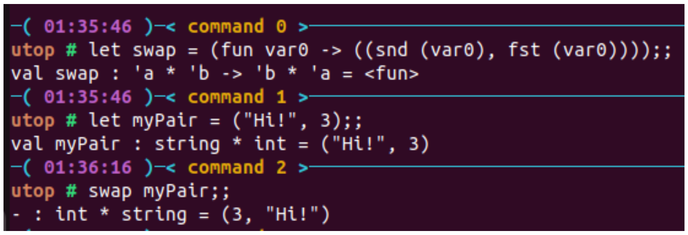
At the end of this document are the worked out examples by hand proven in the picture above.

6 Appendix B: Milestone 2

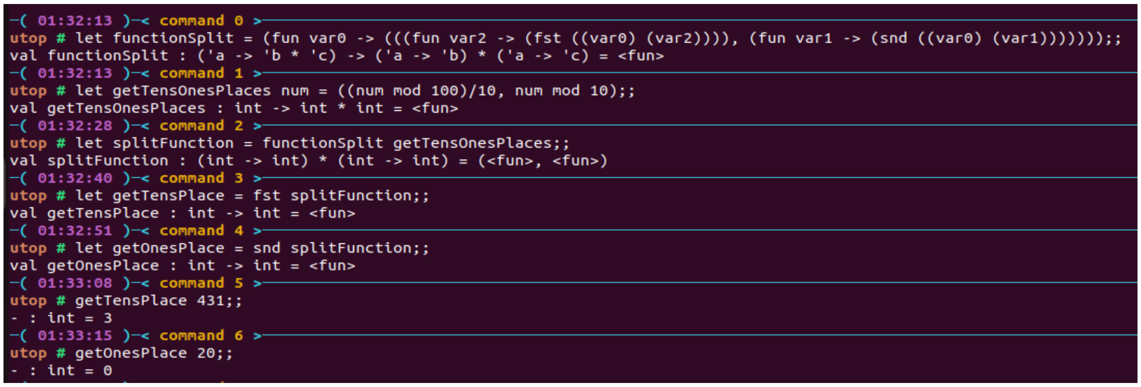
Progress: The theorem prover is almost fully done. The NOT ELIM rule has not been implemented, but it's use corresponds to aborting the program, so we are not sure if we even want to give that option. If we do not implement the NOT ELIM rule, then the theorem prover is considered working. The program extractor is finished. We are working on taking the abstract program form to OCaml code, but the use of OR in the theorem is giving us trouble. Below are pictures of swap, function splitting, function composition, and the third theorem proven above in milestone 1. Note that for the third theorem, since it uses an OR, we show the abstract program form and not the OCaml form as the OCaml converted is not working for OR yet. We also provide a new theorem that was not shown in milestone 1, which represents taking the transpose of a 3x3 matrix.



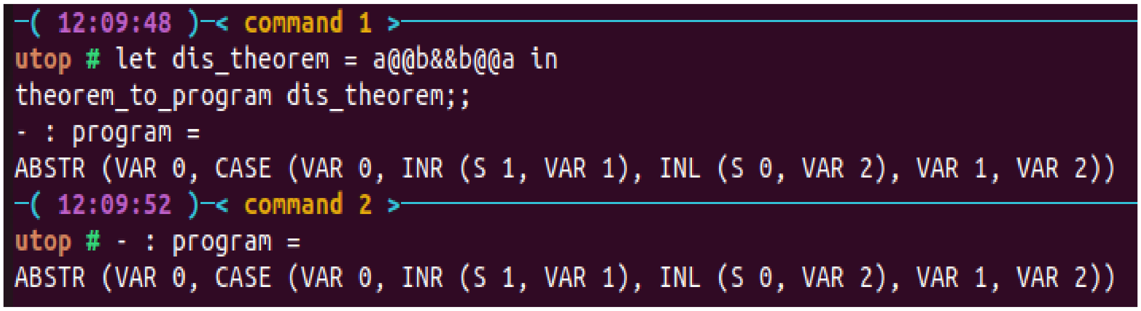
```
(base) pencetn@pencetn-Precision-7560:~/Desktop/OCamlThings/COMP_PROVER$ sh runMe.sh
(0 ∧ 1) ⊃ (1 ∧ 0)
⊃I(∧I(∧E2((0 ∧ 1)) ∧E1((0 ∧ 1))))
(fun var0 -> ((snd (var0), fst (var0))))

(0 ⊃ (1 ∧ 2)) ⊃ ((0 ⊃ 1) ∧ (0 ⊃ 2))
⊃I(∧I(⊃I(∧E1(⊃E((0 ⊃ (1 ∧ 2)) 0))) ⊃I(∧E2(⊃E((0 ⊃ (1 ∧ 2)) 0)))))
(fun var0 -> (((fun var2 -> (fst ((var0) (var2)))), (fun var1 -> (snd ((var0) (var1)))))))

(0 ∨ 1) ⊃ (1 ∨ 0)
⊃I(∨E((0 ∨ 1) ∨I2(0) ∨I1(1)))

((0 ⊃ 1) ∧ (1 ⊃ 2)) ⊃ (0 ⊃ 2)
⊃I(⊃I(⊃E(∧E2(((0 ⊃ 1) ∧ (1 ⊃ 2))) ⊃E(∧E1(((0 ⊃ 1) ∧ (1 ⊃ 2))) 0))))
(fun var0 -> ((fun var1 -> ((snd (var0)) ((fst (var0)) (var1))))))

((0 ∧ (1 ∧ 2)) ∧ ((3 ∧ (4 ∧ 5)) ∧ (6 ∧ (7 ∧ 8)))) ⊃ ((0 ∧ (3 ∧ 6)) ∧ ((1 ∧ (4 ∧ 7)) ∧ (2 ∧ (
⊃I(∧I(∧I(∧E1(∧E1(((0 ∧ (1 ∧ 2)) ∧ ((3 ∧ (4 ∧ 5)) ∧ (6 ∧ (7 ∧ 8)))))) ∧I(∧E1(∧E1(∧E2(((0 ∧ (1
(fun var0 -> (((fst (fst (var0)), (fst (fst (snd (var0))), fst (snd (snd (var0))))), ((fst (
```



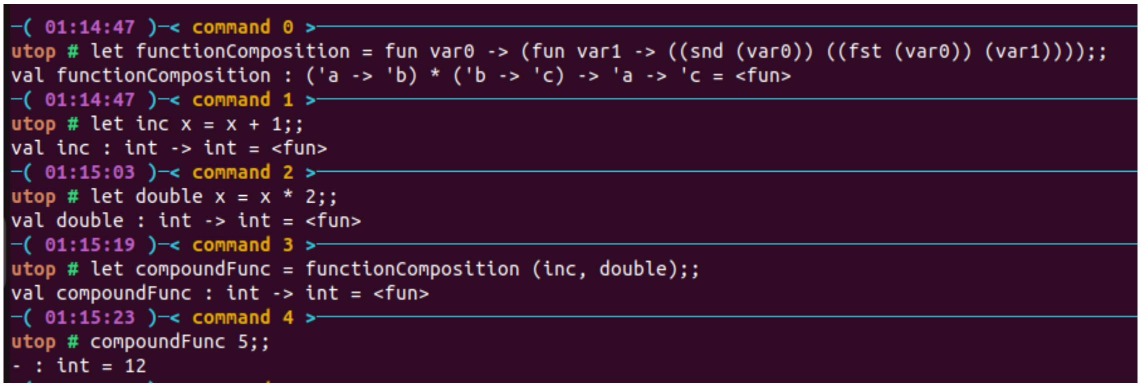
```
─( 01:35:46 )─< command 0 >──────────────────────────────
utop # let swap = (fun var0 -> ((snd (var0), fst (var0))));;
val swap : 'a * 'b -> 'b * 'a = <fun>
─( 01:35:46 )─< command 1 >──────────────────────────────
utop # let myPair = ("Hi!", 3);;
val myPair : string * int = ("Hi!", 3)
─( 01:36:16 )─< command 2 >──────────────────────────────
utop # swap myPair;;
- : int * string = (3, "Hi!")
```



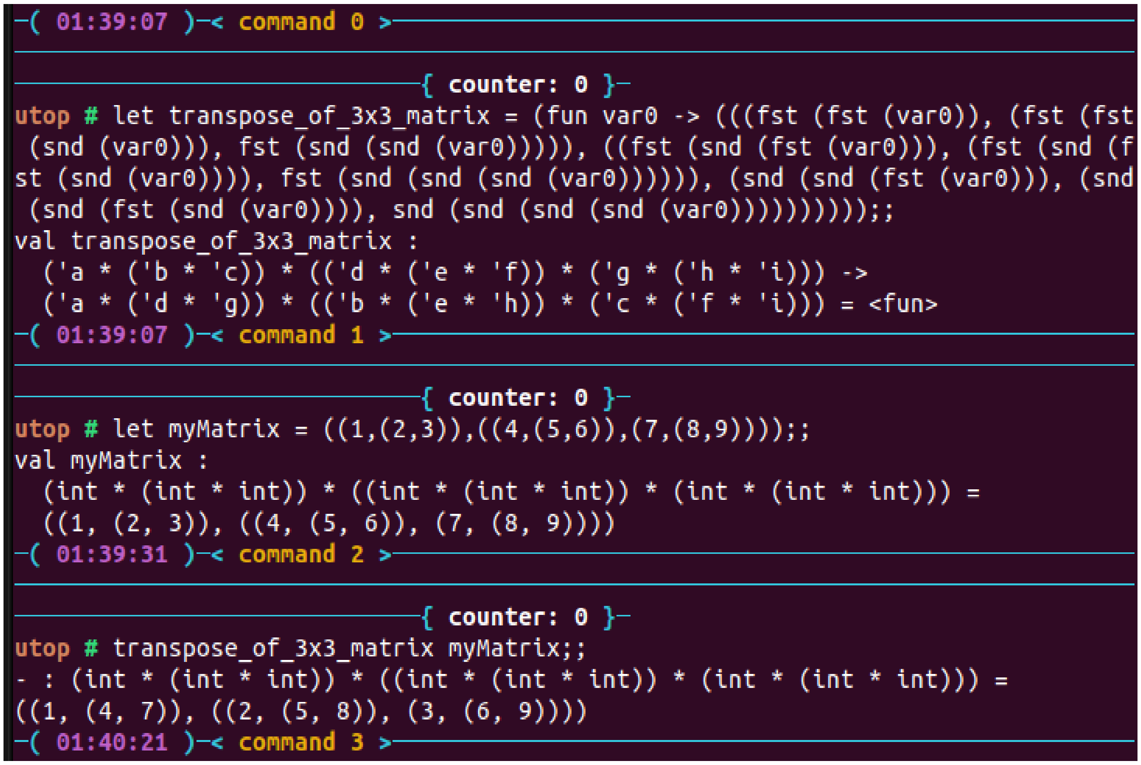
```
─( 01:32:13 )─< command 0 >──────────────────────────────────────────────────────────────
utop # let functionSplit = (fun var0 -> (((fun var2 -> (fst ((var0) (var2)))), (fun var1 -> (snd ((var0) (var1)))))));;
val functionSplit : ('a -> 'b * 'c) -> ('a -> 'b) * ('a -> 'c) = <fun>
─( 01:32:13 )─< command 1 >──────────────────────────────────────────────────────────────
utop # let getTensOnesPlaces num = ((num mod 100)/10, num mod 10);;
val getTensOnesPlaces : int -> int * int = <fun>
─( 01:32:28 )─< command 2 >──────────────────────────────────────────────────────────────
utop # let splitFunction = functionSplit getTensOnesPlaces;;
val splitFunction : (int -> int) * (int -> int) = (<fun>, <fun>)
─( 01:32:40 )─< command 3 >──────────────────────────────────────────────────────────────
utop # let getTensPlace = fst splitFunction;;
val getTensPlace : int -> int = <fun>
─( 01:32:51 )─< command 4 >──────────────────────────────────────────────────────────────
utop # let getOnesPlace = snd splitFunction;;
val getOnesPlace : int -> int = <fun>
─( 01:33:08 )─< command 5 >──────────────────────────────────────────────────────────────
utop # getTensPlace 431;;
- : int = 3
─( 01:33:15 )─< command 6 >──────────────────────────────────────────────────────────────
utop # getOnesPlace 20;;
- : int = 0
```



```
-( 12:09:48 )-< command 1 >
utop # let dis_theorem = a@@b&&b@@a in
theorem_to_program dis_theorem;;
- : program =
ABSTR (VAR 0, CASE (VAR 0, INR (S 1, VAR 1), INL (S 0, VAR 2), VAR 1, VAR 2))
-( 12:09:52 )-< command 2 >
utop # - : program =
ABSTR (VAR 0, CASE (VAR 0, INR (S 1, VAR 1), INL (S 0, VAR 2), VAR 1, VAR 2))
```



```
-( 01:14:47 )-< command 0 >
utop # let functionComposition = fun var0 -> (fun var1 -> ((snd (var0)) ((fst (var0)) (var1))));;
val functionComposition : ('a -> 'b) * ('b -> 'c) -> 'a -> 'c = <fun>
-( 01:14:47 )-< command 1 >
utop # let inc x = x + 1;;
val inc : int -> int = <fun>
-( 01:15:03 )-< command 2 >
utop # let double x = x * 2;;
val double : int -> int = <fun>
-( 01:15:19 )-< command 3 >
utop # let compoundFunc = functionComposition (inc, double);;
val compoundFunc : int -> int = <fun>
-( 01:15:23 )-< command 4 >
utop # compoundFunc 5;;
- : int = 12
```



```
─( 01:39:07 )─< command 0 >─────────────────────────────────────────────
────────────────────────────────────────────────────────────────────────
──────────────────────────────────{ counter: 0 }─
utop # let transpose_of_3x3_matrix = (fun var0 -> (((fst (fst (var0)), (fst (fst
 (snd (var0))), fst (snd (snd (var0))))), ((fst (snd (fst (var0))), (fst (snd (f
st (snd (var0)))), fst (snd (snd (snd (var0)))))), (snd (snd (fst (var0))), (snd
 (snd (fst (snd (var0)))), snd (snd (snd (snd (var0)))))))))));;
val transpose_of_3x3_matrix :
  ('a * ('b * 'c)) * (('d * ('e * 'f)) * ('g * ('h * 'i))) ->
  ('a * ('d * 'g)) * (('b * ('e * 'h)) * ('c * ('f * 'i))) = <fun>
─( 01:39:07 )─< command 1 >─────────────────────────────────────────────
────────────────────────────────────────────────────────────────────────
──────────────────────────────────{ counter: 0 }─
utop # let myMatrix = ((1,(2,3)),((4,(5,6)),(7,(8,9))));;
val myMatrix :
  (int * (int * int)) * ((int * (int * int)) * (int * (int * int))) =
  ((1, (2, 3)), ((4, (5, 6)), (7, (8, 9))))
─( 01:39:31 )─< command 2 >─────────────────────────────────────────────
────────────────────────────────────────────────────────────────────────
──────────────────────────────────{ counter: 0 }─
utop # transpose_of_3x3_matrix myMatrix;;
- : (int * (int * int)) * ((int * (int * int)) * (int * (int * int))) =
((1, (4, 7)), ((2, (5, 8)), (3, (6, 9))))
─( 01:40:21 )─< command 3 >─────────────────────────────────────────────
```

What's next: We plan to finish the abstract program to OCaml code converter, and possibly implement booleans and integers if we feel that we have time. We have hit a roadblock with the OCaml code converter, and are not sure how long it will take. It seems we may have to explore OCaml's polymorphic variants. We also plan to add more examples, up to around 10 to 15.

Swap

pg. 29

$$\dfrac{\dfrac{\dfrac{\dfrac{}{u : A\land B}\,M}{\text{snd}\,u : B}\land E_R \qquad \dfrac{\dfrac{}{u : A\land B}\,u}{\text{fst}\,u : A}\land E_L}{\langle \text{snd}\,M, \text{fst}\,u\rangle : B\land A}\land I}{(\lambda u.\langle \text{snd}\,u, \text{fst}\,u\rangle) : (A\land B)\supset(B\land A)}\supset I^u$$

Given a function that maps A to (B, C), program gives two functions, A → B and A → C.

pg. 31

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{u : A\supset(B\land C)}\,u \qquad \dfrac{}{w : A}\,w}{u\;w : B\land C}\supset E}{\text{fst}\,(u\;w) : B}\land E_L}{\lambda w.\,\text{fst}\,(u\,w) : A\supset B}\supset I^w \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{u : A\supset(B\land C)}\,u \qquad \dfrac{}{v : A}\,v}{u\;v : B\land C}\supset E}{\text{snd}\,(u\;v) : C}\land E_R}{\lambda v.\,\text{snd}\,(u\;v) : A\supset C}\supset I^v}{\langle(\lambda w.\,\text{fst}(u\;w)), (\lambda v.\,\text{snd}(u\,v))\rangle : (A\supset B)\land(A\supset C)}\land I}{\lambda u.\,\langle(\lambda w.\,\text{fst}(u\;w)), (\lambda v.\,\text{snd}(u\,v))\rangle : (A\supset(B\land C))\supset((A\supset B)\land(A\supset C))}\supset I^u$$

pg. 13

Not sure what inl and case — of inl means yet in program extraction rules, so tree not filled out.

$$\dfrac{\dfrac{\dfrac{}{A\lor B}\,u \qquad \dfrac{\dfrac{}{A}\,v}{B\lor A}\lor I_R \qquad \dfrac{\dfrac{}{B}\,w}{B\lor A}\lor I_L}{B\lor A}\lor E^{v,w}}{(A\lor B)\supset(B\lor A)}\supset I^u$$

Comprehension of functions
pg. 33

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{m : (A\supset B)\wedge(B\supset C)}\,M}{\text{snd } m : B\supset C}\,\wedge E_R \qquad \dfrac{\dfrac{\dfrac{}{m : (A\supset B)\wedge(B\supset C)}\,M}{\text{fst } m : A\supset B}\,\wedge E_L \qquad \dfrac{}{w : A}\,w}{(\text{fst } m)\,w : B}\,\supset E}{(\text{snd } m)((\text{fst } m)\,w) : C}\,\supset E}{\lambda w.\,(\text{snd } m)((\text{fst } m)\,w) : A\supset C}\,\supset I^{w}}{(\lambda m.\,\lambda w.\,(\text{snd } m)((\text{fst } m)w)) : ((A\supset B)\wedge(B\supset C))\supset(A\supset C)}\,\supset I^{m}$$

My own example

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{m : (A\supset B)\supset((C\supset D)\wedge C)}\,M \quad \dfrac{}{v : (A\supset B)}\,v}{m\,v : (C\supset D)\wedge C}\,\supset E}{\text{fst}(m v) : C\supset D}\,\wedge E_L \qquad \dfrac{\dfrac{\dfrac{}{m : (A\supset B)\supset((C\supset D)\wedge C)}\,M \quad \dfrac{}{v : (A\supset B)}\,v}{m\,v : (C\supset D)\wedge C}}{\text{snd}(m v) : C}\,E_R}{\text{fst}(m v)\,\text{snd}(m v) : D}\,\supset E}{\lambda v.\,\text{fst}(m\,v)\,\text{snd}(m\,v) : (A\supset B)\supset D}\,\supset I^{v}}{\lambda m.\,\lambda v.\,\text{fst}(m\,v)\,\text{snd}(m\,v) : ((A\supset B)\supset((C\supset D)\wedge C))\supset((A\supset B)\supset D)}\,\supset I^{m}$$