Prose tutorial

Edit New Page

Sumit Gulwani edited this page 9 minutes ago · 60 revisions

Welcome to the prose-tutorial wiki.

In this tutorial, we will learn how to synthesize programs for learning string transformations.

Set up (Windows).

If you get stuck at any of the steps in this section, please send an email to Gustavo Soares at gsoares@microsoft.com and Sumit Gulwani at sumitg@microsoft.com.

1. Get Visual Studio.

- Option 1: Download Visual Studio 2017 Community: https://www.visualstudio.com/downloads/ (Be sure to select the ".Net Desktop Development" box in the installation wizard).
- Option 2: Download Visual Studio 2015. Visual Studio 2017 requires .Net Framework 4.6, which may require Windows 8 or above. In case you cannot use VS 2017, you can still use VS 2015. However, the latest version of PROSE is currently not compatible with VS 2015. Therefore, you will need to work on different branches of this repository. These branches use an earlier version of PROSE and are prefixed with "v15-". For example: branch "v15-substring-part1a" uses PROSE 1.1.0 and works on VS2015 and VS2017, and branch "substring-part1a" uses PROSE 4.0.0 and only works on VS2017.
- Option 3: Install the windows VM from this Official Microsoft link: https://developer.microsoft.com/en-us/windows/downloads/virtual-machines. This VM comes with VS installed. When you open VS 2017, it may say that your subscription is expired. Just sign in with your account (or sign up for free).

2. Clone this repository:

```
git clone https://github.com/gustavoasoares/prose-tutorial.git
cd prose-tutorial
```

3. Open the project solution (ProseTutorial.sln) in Visual Studio.

4. Build the project: Solution Explorer -> Right-Click on Solution 'ProseTutorial' -> Build

Set up (Linux and Mac).

If you get stuck at any of the steps in this section, please send an email to Gustavo Soares at gsoares@microsoft.com and Sumit Gulwani at sumitg@microsoft.com.

- Download and install VS Code (https://code.visualstudio.com/Download)
- Install .Net core SDK (https://www.microsoft.com/net/download/linux)
- Install VS Code extension for C#. (https://marketplace.visualstudio.com/items?itemName=ms-vscode.csharp)

2. Clone this repository:

```
git clone https://github.com/gustavoasoares/prose-tutorial.git
cd prose-tutorial
```

Pages 1

Prose tutorial

+ Add a custom sidebar

Clone this wiki locally

https://github.com/gustavoa

Clone in Desktop

```
git checkout vscode-substring-part2b-complete
```

- Use the prefix "vscode-" to access the branches that are compatible with VS Code.

3. Open the prose-tutorial folder in VS Code:

- Open VS Code
- Choose "Open Folder" and select the "prose-tutorial" folder.
- VS Code may show a warning at the top of the screen about missing dependences. Click on "yes" to install them.

4. Building the project:

- Open the VS Code terminal: View -> Integrated Terminal (or Ctrl+`)

```
dotnet build
```

5. Running the tests

- Open the VS Code terminal: View -> Integrated Terminal (or Ctrl+`)

```
dotnet test
```

6. Running the console application

- Open the VS Code terminal: View -> Integrated Terminal (or Ctrl+`)

```
cd ProseTutorial
dotnet run
```

Set up (Ubuntu Virtual Machine)

- Download this VirtualBox Ubuntu VM
- Open VirtualBox and import the VM file.
- VM username: prose
- VM password: PROSEtutorial
- Open Visual Studio Code by clicking on its icon in the left bar
- Open the folder /home/prose/git/prose-tutorial in VS Code
- Follow the steps 4-6 of the previous section to build, test, and run the solution.
- use the prefix "vscode-" to access the branches that are compatible with VS code.
- see this video tutorial showing how to work on this virtual machine.

Part 1a: Writing your first witness function.

1. Checkout the "substring-part1a" ("v15-substring-part1a" if you use VS2015) branch.

```
git checkout substring-part1a
```

The solution 'ProseTutorial' contains three projects. The first project, ProseTutorial, contains the components that you should implement to perform inductive synthesis using PROSE:

- ProseTutorial/grammar/substring.grammar - The grammar of our DSL.
- ProseTutorial/Semantics.cs - The semantics of our DSL.
- ProseTutorial/WitnessFunctions.cs - Witness functions used by Prose to learn programs using examples.

The second project, ProseTutorial.Tests, contains the test cases that we will use to guide the tutorial (see SubstringTest.cs). Finally, the ProseTutorialApp project contains a console application where you can try out the synthesis system that you have created on new tasks (by providing input-output examples to synthesize a program for a task and then checking the behavior of the synthesized program on new test inputs).

2. Open the SubstringTest.cs file and run the testcase "TestLearnSubstringPositiveAbsPos" (Right-Click on the test case -> Run Tests). It should fail. This test case learns a substring program from an example and test whether the actual output of the program matches the expected one.
- Prose cannot learn a substring program for this test case because we didn't implement all witness functions.
- Go to the WitnessFunctions class and implement the missing ones. Follow the hints and the TODO comments to do so.
- Make sure the test case passes after the changes.
3. Now that the test case is passing, you can also check which programs were synthesized and provide more inputs to these programs.
- Set the ProseTutorialApp as StartUp project (Right-click on the project name -> Set as StartUp project)
- Open the Program.cs file in the ProseTutorialApp and run it (Click on the Start button or press Ctrl + F10).
- Try to provide a new example for your system using a JSON object. Example:

```
"(Abhishek Udupa)","Abhishek Udupa"
```

- The application will show the top 4 learned programs. Since our DSL is very small at this point, only one program is learned for this example:

```
Substring(v, AbsPos(v, 2), AbsPos(v, 16))
```

4 Run the test case "TestLearnSubstringPositiveAbsPosSecOcurrence". You will see that the test fails. Although our DSL can express a program to perform this task, our synthesis engine fails to produce this program. Notice that the first example 16-Feb-2016 -> 16 is ambiguous. 16 can come from two different locations in the input string. Our witness function considers only the first match. Next, we will see how to specify witness functions that can produce more than one output specification for a given input state.

5. The completed code for this part of the tutorial can be found in the branch "substring-part1a-complete".

Part 1b: Disjunctive conditional specification.

1. Checkout the "substring-part1b-attempt1" branch.

```
git checkout substring-part1b-attempt1
```

2. Update the witness functions for the substring positions to allow Disjunctive specification.

- Go to the WitnessFunctions class and update the Substring witness functions. Follow the hints and the TODO comments to do so.
- Make sure that the test "TestLearnSubstringPositiveAbsPosSecOcurrence" passes.
- Run the test case "TestLearnSubstringPositiveAbsPosSecOcurrenceOneExp". This test checks if PROSE learns exactly two substring programs given the example "16-Feb-2016" -> "16". The test will fail because PROSE learned four programs instead of two. The reason for this is that the sub-problems for the start position and end position are not independent any more since we added disjunctive witness function. For instance, although 1 and 10 are valid start positions, and 3 and 12 are valid end positions, Substring(v,1,12) is not a valid program. To avoid that, we need to implement conditional specifications.

3. The completed code for this part of the tutorial can be found in the branch "substring-part1b-attempt1-complete".
4. Checkout the "substring-part1b-attempt2" branch.

```
git checkout substring-part1b-attempt2
```

5. Implement the conditional witness function for the end position of the substring operator.

- Modify the WitnessFunctions.cs file.
- Make sure the test TestLearnSubstringPositiveAbsPosSecOcurrenceOneExp passes.

6. Run the test case "TestLearnSubstringNegativeAbsPos". You will see that the test fails because there is no program in our DSL that can extract the names for the two given inputs. In the next part of the tutorial, we will see how we can change the semantics of our AbsPos operator and its witness function to create programs that are expressive enough to perform this task.
7. The completed code for this part of the tutorial can be found in the branch "substring-part1b-attempt2-complete".

Part 1c: Supporting negative positions and ranking.

1. Checkout the branch "substring-part1c".
2. The AbsPos operator refers to the kth index in a given string from the left side (See Semantics.cs).

- Edit the semantics of this operator in the Semantics.cs file to make it also refer to the kth index in a given string from the right side if the integer constant k is negative.
- Update the witness functions to reflect this change.
- After making these changes, your system should be able to synthesize the following program:

```
Substring(v, AbsPos(v, 2), AbsPos(v, -2))
```

- The test case TestLearnSubstringNegativeAbsPos should pass.

3. Notice that it was necessary to provide two examples in this test case. If you try to use only one test case, your system will not produce the correct program. Try to do that by running the test case TestLearnSubstringNegativeAbsPosRanking. This test case should fail.
4. The completed code for this part of the tutorial can be found in the branch "substring-part1c-complete".

Part 1d: Ranking.

1. Checkout the branch "substring-part1d".
2. Implement a ranking function for AbsPos.

- Edit the default ranking function for AbsPos in the RankingScore.cs file. Your ranking function should prefer smaller absolute values for the constant k than larger ones.
- Make the test TestLearnSubstringNegativeAbsPosRanking pass.
- Run the test case "TestLearnSubstring". You will see that the test fails because there is no program in our DSL that can extract the last name for the two given inputs. In the next part of the tutorial, we will see how we can add an operator in our DSL to create programs that are expressive enough to perform this task.

2. The completed code for this part of the tutorial can be found in the branch "substring-part1d-complete".

Part 2a: Supporting regular expressions.

1. Checkout the branch "substring-part2a". The new version of our DSL contains the operator RelPos, which uses regular expressions to find positions in the input string.

- Run the test case "TestLearnSubstringTwoExamples". The test will fail because we did not implement the witness function for the RelPos operator. You should implement this function.
- Make sure the test case passes.
- Run the test case "TestLearnSubstringOneExample". You will see that the test will fail because we did not return the correct program given just a single example. In the next part of the tutorial, we will see how we can write RankingFunctions to rank the programs that satisfy the examples in order to select a more robust program.

2. The completed code for this part of the tutorial can be found in the branch "substring-part2a-complete".

Part 2b: Ranking for regular expressions.

1. Checkout the branch "substring-part2b".

- Modify the ranking functions in RankingScore.cs to make the test pass. The idea is to prefer RelPos over AbsPos.
- Make sure the test case "TestLearnSubstringOneExample" passes.

2. The completed code for this part of the tutorial can be found in the branch "substring-part2b-complete".