Assignment #3:
Meta Theory and Implementation:
Language **ETSP**, extended with generic lists **L**

Fundamentals of Programming Languages

Out: Thursday, Oct 13th, 2016
Due: Thursday, Oct 27th, 2016 11:59pm EST

The tasks in this homework ask you to prove ("meta theoretical") properties about the language **ETSPL**, which combines languages **E**, **T** with sums and products; in homework #2, we considered a large subset of this language (everything except language **E**). Building on these language fragments, we will add generic lists, which like natural numbers, consist of a recursive structure. We call the resulting language **ETSPL**. Unlike natural numbers, these generic list structures carry data of an arbitrary (generic) *element type*.

This homework also asks you to program an implementation of this combined language in OCaml. *We did this together in class for fragments of this language, but did not complete the implementation in OCaml; use this code and these videos as a guide.*

Grading criteria for proofs: To receive full credit for any proof below, you must *at least* do the following:

- At the beginning of your proof, specify over what structure or derivation you are performing induction (i.e., which structure's inductive principle are you using?)
- In the inductive cases of the proof, specify how you are applying the inductive hypothesis, and what result it gives you.

If you omit these steps and/or do not make them explicit, you will receive zero credit for your proof. If you attempt to do these steps, but you make a mistake, you may still receive some partial credit, depending on your proof.

Hint: If you are unsure about how to structure these proofs to receive full credit, **please refer to the HW #1 and #2 Solutions** on Moodle as a reference and guide.

Note on omitting redundant proof cases: In the proofs below, some cases are very similar to other cases, e.g., the cases for `plus` and `times` in the proofs below are likely to be analogous, in that (nearly) the same proof steps are used in each. When this happens, you can omit the redundant cases as follows: If you do one case, say for `plus`, you may (optionally) write in the other case for `times` that it is "analogous to the case above, for `plus`". You must make this omission explicit, to show that you have thought about it. Further, this shortcut is only applicable when the cases really are analogous, and (nearly) the same steps apply in the proof. **When in doubt, do not omit the proof case.**

Tasks

Syntax. The syntax of generic lists consists of several additional forms:

$$
\begin{array}{llll}
\textit{Types} & \tau & ::= & \cdots \\
 & & | & \mathsf{List}(\tau) \\
\textit{Expressions} & e & ::= & \cdots \\
 & & | & \mathsf{Nil} \\
 & & | & \mathsf{Cons}(e_1, e_2) \\
 & & | & \mathsf{match}(e_l, e_n, x.y.e_c) \\
 & & | & \mathsf{fold}(e_l, e_n, x.y.e_c) \\
 & & | & \mathsf{map}(e_l, x.e_h)
\end{array}
$$

Statics There is a new typing rule for each new syntactic form:

$$
\textsc{nil}\ \frac{}{\Gamma \vdash \mathsf{Nil} : \mathsf{List}(\tau)}
\qquad
\textsc{cons}\ \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \mathsf{List}(\tau)}{\Gamma \vdash \mathsf{Cons}(e_1, e_2) : \mathsf{List}(\tau)}
$$

$$
\textsc{match}\ \frac{\Gamma \vdash e_l : \mathsf{List}(\tau_1) \quad \Gamma \vdash e_n : \tau \quad \Gamma, x : \tau_1, y : \mathsf{List}(\tau_1) \vdash e_c : \tau}{\Gamma \vdash \mathsf{match}(e_l, e_n, x.y.e_c) : \tau}
$$

$$
\textsc{fold}\ \frac{\Gamma \vdash e_l : \mathsf{List}(\tau_1) \quad \Gamma \vdash e_n : \tau \quad \Gamma, x : \mathsf{List}(\tau_1), y : \tau \vdash e_c : \tau}{\Gamma \vdash \mathsf{fold}(e_l, e_n, x.y.e_c) : \tau}
\qquad
\textsc{map}\ \frac{\Gamma \vdash e_l : \mathsf{List}(\tau_1) \quad \Gamma, x : \tau_1 \vdash e_h : \tau_2}{\Gamma \vdash \mathsf{map}(e_l, x.e_h) : \mathsf{List}(\tau_2)}
$$

Task 1 (20 pts). Dynamics for Language **ETSPL**.

1. Add rules for the e val judgement.
2. Add dynamics rules for stepping these forms.

Task 2 (20 pts). Meta theory for Language **ETSPL**.

1. State the substitution lemma for Language **ETSPL**.
2. State the canonical forms lemma for Language **ETSPL**.
3. State and prove progress for Language **ETSPL**.
4. State and prove preservation for Language **ETSPL**.

You only need to do the *new* cases of the proofs, for generic lists.

Hint (repeated again, for emphasis): If you are unsure about how to structure these proofs to receive full credit, **please refer to the HW #1 and #2 Solutions** on Moodle as a reference and guide.

Task 3 (20 pts). Implement the theory of Language **ETSPL** in OCaml.

Task 4 (20 pts). Implement tests for each construct in the language (at least one test per construct; some tests can test multiple constructs). In particular, implement a function `test_pap` that tests *progress and preservation*. Given an expression, this function computes a type for the expression, then steps that expression until it is a value. After each step, it computes a new type for the expression and asserts that the new and old type are equal. The function returns the final expression (a value) when it terminates. For each of your tests, use `test_pap` to test that progress and preservation indeed hold on your initial expression, in addition to testing that the final value matches the one that you expect.

How to implement a Language Theory: When we say "implement Language X in OCaml", we mean precisely the following. For a concrete example, see our implementation of Language **E** as a guide, where we did this together in class. (There is a lecture video and OCaml code from September 29 available online).

1. Define syntax forms as OCaml datatypes
   (a) Define variables `var` as OCaml strings (type `string`)
   (b) Define the syntax of expressions as a new OCaml datatype named `exp`
   (c) Define the syntax of types as a new OCaml datatype named `typ`
   (d) Define type contexts `gamma` as OCaml lists of variable-type pairs.
   (e) Implement pretty-printing functions for expressions, types and contexts:
      `exp_string` : `exp -> string`
      `typ_string` : `typ -> string`
      `gam_string` : `gamma -> string`
2. Implement a function `is_val : exp -> bool` that implements a check for the e val judgment
3. Implement a substitution function `subst : exp -> var -> exp -> exp`. Make sure that you implement *shadowing* correctly, and you do not allow *variable capture*. See our implemention of the `Let` case in language **E** as a reference; notice how we compare the `Let`-bound variable against the one being substituted, and do not substitute further if they are the same.
4. Implement a type-checking function `exp_typ : gamma -> exp -> typ option`.

   Hint: Note that the Nil form has a similar problem to lambda and some other forms in the prior homework: The list type is not clear from the term Nil, and thus could be anything. To give programs enough information to type them, add a type annotation to the Nil form, and other forms, but only *if they require it.*
5. Implement a steps-to function `step : exp -> exp`. It should take exactly one small step, or raise an exception if the expression is a value.

6. Implement a multiple-steps function `test_pap : exp -> exp`. It should take as many steps as possible, and it should test progress and preservation as it steps. This is precisely what you proved in your *meta theory* proofs about well-typed programs in the language.

   Use the pretty-printing functions above to print the type, and to print the expression as it steps. In OCaml, the function `print_string` is a simple way to print strings.

Task 5 (10 pts).

Continue reading the papers that you chose in Homework #2. For each of the five papers, and for each question below, write two concise sentences:

1. Why did *you* select this paper?
2. What is the "main idea" of the paper?
3. How well is this main idea communicated to you when you read the *first two sections and conclusion* of paper, and skimmed the rest? In particular, explain what aspects seem important, are which are clear versus unclear. You may want to read deeper into the details of the paper body if these beginning and ending sections do not make the main ideas clear; make a note if this is required.

Task 6 (10 pts).

Continue thinking about your class project. Write an updated 250 word explanation of your plan, and what you hope to accomplish with your project by the end of the semester.

That is, on what artifact do you want to be graded? Recall that you may choose to write a survey paper or implement something, but even implementation projects require a short report. By writing your plan now, you are also generating a draft of part of this report.

Here are the same suggestions as on the prior homework:

- **Functional implementation:** Consider an algorithm or system that seems interesting to you. Can you write this algorithm in a purely-functional style in OCaml?
- **Functional reactive implementation:** Learn a functional reactive programming language like Elm. (See `http://elm-lang.org/`). Write a game, simulation or productivity application in this new language.
- **Survey project:** Choose a theme and six to eight papers from POPL, PLDI, ICFP and OOPSLA (or other ACM SIGPLAN Conferences in PL). Write a survey paper about these papers, trying to tell a cohesive story about how they relate.