Assignment #2: Meta Theory and Implementation: Language **T**, with Finite Data Types

Fundamentals of Programming Languages

Out: Thursday, Sept 30th, 2016
Due: Thursday, Oct 13th, 2016 11:59pm EST

The tasks in this homework ask you to prove ("meta theoretical") properties about the language **T** and about finite data types, defined in Ch. 8–10 of *PFPL*. These properties are "meta theoretical" in that they give a theory about the theory of the language, and are true about *all* well-typed programs, not specific, individual programs. This homework also asks you to program an implementation of these languages in OCaml. We did this together in class for the simpler Language **E**; use this code and the video as a guide.

Grading criteria: 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 Solution** 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

Task 1 (20 pts). Meta theory for Language **T** (See PFPL Chapter 9).

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

Task 2 (20 pts). Meta theory for finite data types (See PFPL Chapters 10 and 11).

1. State the substitution lemma for Language **T** extended with sum and product types.
2. State the canonical forms lemma for Language **T** extended with sum and product types.
3. State and prove progress for Language **T** extended with sum and product types.
4. State and prove preservation for Language **T** extended with sum and product types.

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 Solution** as a reference and guide.

Task 3 (30 pts). Implement the theory of Language **T** in OCaml, along with tests.

Task 4 (30 pts). Extend the theory of Language **T** with Pairs and Sums. Include additional tests.

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 the syntax of expressions as a new OCaml datatype named `exp`
   (b) Define the syntax of types as a new OCaml datatype named `typ`
   (c) Define variables `var` as OCaml strings (type `string`)
   (d) Define type contexts `gamma` as OCaml lists of variable-type pairs.
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: Notice that to type-check lambda expressions *without type annotations*, an implementation of type-checking must "guess" a type for the bound variable. After all, this variable stands in for a parameter that we may not have locally; how would we know its type?

This situation is different from the `Let` form in language **`E`**, where we have the sub-expression to which the variable is bound, and hence, we can get the type for the bound variable by processing this sub-expression.

To avoid this guessing problem, define your syntax for lambda expressions to include a type for the argument variable.

5. Implement a suite of five **interesting** tests of type-checking. ("Interesting" means that you attempt to cover different execution paths of your implementation with each test).

6. 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.

7. Implement a multiple-steps function `steps : exp -> exp`. It should take as many steps as possible. For programs that type-check, it should produce a value of the same type. This is precisely what you proved in your *meta theory* proofs about the language.

8. Implement a suite of five **interesting** tests of stepping once and multiple times. (Do your expressions always have the same type as they step?) ("Interesting" means that you attempt to cover different execution paths of your implementation with each test).