Homework 1: Reasoning about Languages
15-814: Types and Programming Languages
Fall 2016
Instructor: Karl Crary
TA: Ankush Das

Out: Sep 14, 2016 (02 pm)
Due: Sep 28, 2016 (11 pm)

Notes:
• Welcome to 15-814’s first homework assignment!
• Please email your work as a PDF file to ankushd@cs.cmu.edu titled “15-814 Homework 1”. Your
  PDF should be named “<your-name>-hw1-sol.pdf”.
• We will be using Piazza to answer questions as well as make clarifications, corrections and
  announcements. Please sign up for the class on Piazza to make sure you don’t miss anything:
  http://piazza.com/cmu/fall2016/15814
• Although office hours haven’t been posted yet, feel free to contact the instructor or the TA to
  set up a meeting for questions.

1 Arithmetic

We’ll start with a simple expression language. There are two types, integers int and booleans bool.
The language consists of literals, a few arithmetic and comparison operators, and a conditional expression.

$$
\begin{align*}
\tau & ::= \text{int} | \text{bool} \\
 e & ::= x | n | \text{tt} | \text{ff} | e + e | e * e | e \leq e | \text{if}(e, e, e)
\end{align*}
$$

We define the typing judgment $$\Gamma \vdash e : \tau$$ and operational semantics judgments $$\Delta \vdash e \text{ val}$$ and $$\Delta \vdash e \mapsto e'$$ in Appendix A (N.B. $$n$$ denotes the term representation of the numeral $$n$$ in our simple language). Type contexts $$\Gamma$$ and evaluation contexts $$\Delta$$ are defined with the following grammar:

$$
\begin{align*}
\Gamma & ::= \cdot | \Gamma, x : \tau \\
\Delta & ::= \cdot
\end{align*}
$$

For the moment, the only evaluation context is the empty context $$\cdot$$, but we will change this as we build the language.

Before we add anything new, however, let’s check some properties to make sure you’re comfortable
with rule induction. For the proofs in this assignment, you may condense similar cases in arguments
by induction, but be clear and rigorous in your reasoning.

Task 1  Prove the following inversion lemma:

(If Inversion) If $$\Gamma \vdash \text{if}(e_1, e_2, e_3) : \tau$$, then $$\Gamma \vdash e_1 : \text{bool}$$, $$\Gamma \vdash e_2 : \tau$$, and $$\Gamma \vdash e_3 : \tau$$.

This seems immediate, but really follows from the induction principle for the typing judgment. (Tip:
Prove that for all $$\Gamma$$, $$e$$, $$\tau$$ such that $$\Gamma \vdash e : \tau$$, if $$e = \text{if}(e_1, e_2, e_3)$$ for some $$e_1$$, $$e_2$$, $$e_3$$, then $$\Gamma \vdash e_1 : \text{bool}$$,
$$\Gamma \vdash e_2 : \tau$$, and $$\Gamma \vdash e_3 : \tau$$.)

In general, an inversion lemma is one which recovers the premises of a rule from its conclusion. In
the rest of the assignment, you may use these without proof, but be sure to note explicitly when you
apply them and check for yourself that they hold.
Task 2  Prove unicity of typing for this language.

(Unicity of Typing) For any $\Gamma, e, \tau, \tau'$ such that $\Gamma \vdash e : \tau$ and $\Gamma \vdash e : \tau'$, we have $\tau = \tau'$.
You may assume that any variable appears at most once in a given context.

2 Functions

On top of the expression language, we now add a syntax for programs $p$, which consist of a series of one-argument function definitions and a final expression computation. (In such a small language, it would be useful to have multi-argument functions, but we will restrict ourselves for the sake of simplicity.) In order to make use of these definitions, we add syntax for function calls to the expression language. The sort of functions consists only of function variables, which we denote here with the letter $u$.

\[
\begin{align*}
f & ::= u \\
e & ::= \cdots | f(e) \\
p & ::= \text{fun } u (x : \tau) \{ e \}; p | \text{result } e
\end{align*}
\]

We allow calls to previously defined functions to appear in the definitions of later functions as well as in the result expression. As an example, the program

\[
\begin{align*}
\text{fun not } (b : \text{bool}) \{ \text{if}(b, \overline{f}, \overline{f}) \}; \\
\text{fun abs } (x : \text{int}) \{ \text{if}(\text{not}(0 \leq x), -1 \times x, x) \}; \\
\text{fun square } (z : \text{int}) \{ z \times z \}; \\
\text{fun cube } (z : \text{int}) \{ \text{square}(z) \times z \}; \\
\text{result } \text{cube}(\text{abs}(3)) + \text{cube}(\text{abs}(-2))
\end{align*}
\]

should compute to 35.

Task 3  Give a definition in uniform syntax for the language constructs described above. Include the arities (with sorts) of the operators, per Section 1.2 of PFPL.

Before giving the precise operational semantics of this language, we have an ambiguity to settle. Consider the following program.

\[
\begin{align*}
\text{fun } u (x : \text{int}) \{ x \}; \\
\text{fun } v (y : \text{int}) \{ u(y) \}; \\
\text{fun } u (z : \text{int}) \{ z + 1 \}; \\
\text{result } v(1)
\end{align*}
\]

The result expression $v(1)$ will be computed in an environment of function definitions. Executing the call $v(1)$ looks up the definition of $v$, which then leads to a call to $u$. But which of the two functions named $u$ does this call refer to? Depending on how the environment behaves, it may be the first or the second, and so this program might compute either 1 or 2. The first evaluation strategy, in which the $u$ that $v$ calls remains the $u$ available when $v$ was defined, is called static scoping (sometimes lexical scoping). The second, wherein the call to $u$ in $v$ refers to whichever $u$ was most recently added to the environment, is called dynamic scoping.

Task 4  We want to respect the identification convention:

Abstract binding trees are always identified up to $\alpha$-equivalence.

With this in mind, which of the two strategies is correct? Explain.
Now, let’s move on to defining the statics and dynamics of this language precisely. We will need two typing judgments to determine if programs are well-typed. Besides the judgment $\Gamma \vdash e : \tau$ for typing expressions we have already mentioned, there should be a judgment $\Gamma \vdash p : \tau$ which states that a program has result of type $\tau$. We also need to add a new kind of hypothesis to our type contexts:

$$\Gamma ::= \cdots \mid u : \tau_1 \Rightarrow \tau_2$$

The assumption $u : \tau_1 \Rightarrow \tau_2$ is used to denote that $u$ is a function variable in scope which takes an argument of type $\tau_1$ and returns a result of type $\tau_2$.

**Task 5** Starting with the rules in Appendix A, complete the definition of the typing judgment $\Gamma \vdash e : \tau$, and define the judgement $\Gamma \vdash p : \tau$.

**Task 6** Give a structural operational semantics for this language by completing the definition of $\Delta \vdash e \mapsto e'$ and defining a judgment $\Delta \vdash p \mapsto p'$. Assume that the value judgment $\Delta \vdash p \text{ val}$ is defined by the single rule

$$\frac{\Delta \vdash e \text{ val}}{\Delta \vdash \text{result } e \text{ val}} \quad (\text{RES-V})$$

In order to track the environment of function definitions, we will add a new kind of assumption to our evaluation context:

$$\Delta ::= \cdots \mid u \text{ def } x.e$$

The assumption $u \text{ def } x.e$ expresses that the function variable $u$ is currently in scope and that it takes an argument $x$ and computes the expression $e$.

Your definition of the semantics should satisfy the properties of progress and preservation.

* (Progress) If $\Gamma \vdash p : \tau$, then either $\Gamma \vdash p \text{ val}$ or there exists $p'$ such that $\Gamma \vdash p \mapsto p'$.

* (Preservation) If $\Gamma \vdash p : \tau$ and $\Gamma \vdash p \mapsto p'$, then $\Gamma \vdash p' : \tau$.

**Task 7** Prove progress for the rules you have specified. You will want to state and prove an analogous theorem for expressions first. For programs, it may be helpful to prove the following more general theorem:

If $\Delta :: \Gamma$ and $\Gamma \vdash p : \tau$, then either $\Delta \vdash p \text{ val}$ or there exists $p'$ such that $\Delta \vdash p \mapsto p'$.

Here $\Delta :: \Gamma$ is a judgment on contexts defined by the following rules:

$$\begin{align*}
\vdots & : \text{(Env-Nil)} \\
(\Delta, f \text{ def } x.e) :: (\Gamma, f : \tau_1 \Rightarrow \tau_2) & : \text{(Env-Cons)}
\end{align*}$$

This judgment expresses that the environment $\Delta$ and type context $\Gamma$ reference the same function variables, i.e. that they agree on what is in scope. This condition is enough to guarantee a canonical forms lemma, which you will need for your proof:

* (Canonical Forms Lemma) Let contexts $\Delta :: \Gamma$ and an expression $e$ be given. Assume that $\Delta \vdash e \text{ val}$. If $\Gamma \vdash e : \text{int}$, then $e = \pi$ for some $n \in \mathbb{Z}$. If $\Gamma \vdash e : \text{bool}$, then either $e = \underline{\text{true}}$ or $e = \underline{\text{false}}$.

This lemma enumerates the forms that well-typed values can take (the eponymous canonical forms). You are not required to prove this lemma.

We can make this language a bit more useful by allowing function definitions to refer recursively to the functions they define. We can then write, for example, the factorial function:

```plaintext
fun fact (x : int) {if (x <= 0, 1, x * fact(x + (-1)))};
result fact(5)
```

**Task 8** How does this extension change your answers to Tasks 3, 5, and 6? Give the syntax and rules where different. (You should also make a change to the form of the assumption $u \text{ def } x.e$.)
A Base Expression Language

A.1 Statics

\[ \Gamma, x : \tau \vdash x : \tau \quad (\text{HYP}) \]
\[ \Gamma \vdash \pi : \text{int} \quad (\text{NUM}) \]
\[ \Gamma \vdash \text{true} : \text{bool} \quad (\text{TRUE}) \]
\[ \Gamma \vdash \text{false} : \text{bool} \quad (\text{FALSE}) \]
\[ \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \quad (\text{PLUS}) \]
\[ \Gamma \vdash e_1 \cdot e_2 : \text{int} \quad (\text{TIMES}) \]
\[ \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \quad (\text{LEQ}) \]
\[ \Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau \quad (\text{IF}) \]

A.2 Dynamics

\[ \Delta \vdash \pi \text{ val} \quad (\text{NUM-V}) \]
\[ \Delta \vdash \text{true} \text{ val} \quad (\text{TRUE-V}) \]
\[ \Delta \vdash \text{false} \text{ val} \quad (\text{FALSE-V}) \]
\[ \Delta \vdash e_1 \mapsto e'_1 \quad (\text{PLUS-S1}) \]
\[ \Delta \vdash e_1 + e_2 \mapsto e'_1 + e'_2 \quad (\text{PLUS-S2}) \]
\[ \Delta \vdash m + n \mapsto m + n \quad (\text{PLUS-I}) \]
\[ \Delta \vdash e_1 \cdot e_2 \mapsto e'_1 \cdot e'_2 \quad (\text{TIMES-S1}) \]
\[ \Delta \vdash e_1 \cdot e_2 \mapsto e'_1 \cdot e'_2 \quad (\text{TIMES-S2}) \]
\[ \Delta \vdash m \cdot n \mapsto m \cdot n \quad (\text{TIMES-I}) \]
\[ \Delta \vdash e_1 \leq e_2 \mapsto e'_1 \leq e'_2 \quad (\text{LEQ-S1}) \]
\[ \Delta \vdash e_1 \leq e_2 \mapsto e'_1 \leq e'_2 \quad (\text{LEQ-S2}) \]
\[ \Delta \vdash m \leq n \mapsto m \leq n \quad (\text{LEQ-I1}) \]
\[ \Delta \vdash m > n \mapsto m \leq n \mapsto \text{false} \quad (\text{LEQ-I2}) \]
\[ \Delta \vdash e_1 \mapsto e'_1 \quad (\text{IF-S}) \]
\[ \Delta \vdash \text{if}(e_1, e_2, e_3) \mapsto \text{if}(e'_1, e_2, e_3) \quad (\text{IF-I}) \]
\[ \Delta \vdash \text{if}(\text{true}, e_2, e_3) \mapsto e_2 \quad (\text{IF-I1}) \]
\[ \Delta \vdash \text{if}(\text{false}, e_2, e_3) \mapsto e_3 \quad (\text{IF-I2}) \]