Assignment 3
Dynamics

15-814: Types and Programming Languages
Frank Pfenning

Due Tue Sep 23, 2025
80 pts

This assignment is due on the above date and it must be submitted electronically on Gradescope. Please use the attached template to typeset your assignment and make sure to include your full name and Andrew ID. For the written problems, you may also submit handwritten answers that have been scanned and are **easily legible**.

Please carefully read the policies on collaboration and credit on the course web pages at `http://www.cs.cmu.edu/~fp/courses/15814-f25/assignments.html`.

You should hand in two files separately:

- `hw03.pdf` with the written answers to the questions.
- `hw03.poly` with the code, where the solutions to the problems are clearly marked and auxiliary code (either from lecture or your own) is included so it passes the LAMBDA checker.

As always, in your proofs you may use the theorems in the lecture notes.

1 Sequentiality

Task 1 (10 pts) Prove sequentiality: If $\cdot \vdash e : \tau$, $e \mapsto e_1$ and $e \mapsto e_2$ then $e_1 = e_2$. You only need to show the cases pertaining to functions in our call-by-value language. For equality, you should assume α-conversion, that is, equality modulo the name of bound variables.

2 Lazy Pairs

Task 2 (30 pts) *Lazy pairs*, constructed as $\langle\!| e_1, e_2 |\!\rangle$, are an alternative to the eager pairs $\langle e_1, e_2\rangle$. Lazy pairs are typically available in "lazy" languages such as Haskell. The key differences are that a lazy pair $\langle\!| e_1, e_2 |\!\rangle$ is always a value, whether its components are or not. In that way, it is like a λ-expression, since $\lambda x.\, e$ is always a value. The second difference is that its destructors are $\mathsf{fst}\ e$ and $\mathsf{snd}\ e$ rather than a new form of case expression.

We write the type of lazy pairs as $\tau_1 \mathbin{\&} \tau_2$. In this task you are asked to design the rules for lazy pairs and check their correctness.

1. Write out the new rule(s) for e *val*.
2. State the typing rules for new expressions $\langle\!| e_1, e_2 |\!\rangle$, $\mathsf{fst}\ e$, and $\mathsf{snd}\ e$.

3. Give evaluation rules for the new forms of expressions.

Instead of giving the complete set of new proof cases for the additional constructs, we only ask you to explicate a few items. Nevertheless, you need to make sure that the progress and preservation continue to hold.

4. State the new clause in the canonical forms theorem.
5. Show one case in the proof of the preservation theorem where a destructor is applied to a constructor.
6. Show the case in the proof of the progress theorem analyzing the typing rule for $\mathsf{fst}\, e$.

3 Nontermination

Task 3 (30 pts) Consider adding a new expression $\bot$ to our call-by-value language (with functions and Booleans) with the following evaluation and typing rules:

$$\frac{}{\bot \mapsto \bot}\ \mathsf{step/bot} \qquad \frac{}{\Gamma \vdash \bot : \tau}\ \mathsf{bot}$$

We do not change our notion of value, that is, $\bot$ is not a value.

1. Does preservation (Theorem L6.2) still hold? If not, provide a counterexample. If yes, show how the proof has to be modified to account for the new form of expression.
2. Does the canonical forms theorem (L6.4) still hold? If not, provide a counterexample. If yes, show how the proof has to be modified to account for the new form of expression.
3. Does progress (Theorem L6.3) still hold? If not, provide a counterexample. If yes, show how the proof has to be modified to account for the new form of expression.

Once we have nonterminating computation, we sometimes compare expressions using *Kleene equality*: e_1 and e_2 are Kleene equal ($e_1 \simeq e_2$) if they evaluate to the same value, or they both diverge (do not compute to a value). Since we assume we cannot observe functions, we can further restrict this definition: For $\cdot \vdash e_1 : \mathsf{bool}$ and $\cdot \vdash e_2 : \mathsf{bool}$ we write $e_1 \simeq e_2$ iff for all values v, $e_1 \mapsto^* v$ iff $e_2 \mapsto^* v$.

4. Give an example of two closed terms e_1 and e_2 of type bool such that $e_1 \simeq e_2$ but not $e_1 =_\beta e_2$, or indicate that no such example exists (no proof needed in either case).

Task 4 (10 pts) In our call-by-value language with functions, Booleans, and $\bot$ (see Task 3) consider the following specification of *or*, sometimes called "short-circuit or":

$$\begin{array}{rcl} \mathit{or}\ \mathsf{true}\ e & \simeq & \mathsf{true} \\ \mathit{or}\ \mathsf{false}\ e & \simeq & e \end{array}$$

where $e_1 \simeq e_2$ is Kleene equality from Task 3.

1. We cannot define a *function* $\mathit{or} : \mathsf{bool} \to (\mathsf{bool} \to \mathsf{bool})$ with this behavior. Prove that it is indeed impossible.
2. Show how to translate an expression $\mathit{or}\ e_1\ e_2$ into our language so that it satisfies the specification, and verify the given equalities by calculation. By "translation" we mean to find an suitable existing expression in the language with primitive Booleans and conditionals, presumably using e_1 and e_2.