Assignment 1: Induction
Model Solution

15-312 Foundations of Programming Languages
Kevin Watkins ⟨kw@cmu.edu⟩

February 3, 2005

Please refer to the assignment itself for the full description and statement of each problem.

§ 1. Higher-order abstract syntax (5 points). There wasn't much of a trick to the first part of this problem. The main mistakes were not including a production for variables x, or including $x.e$ by itself as a production. Remember that $x.e$ doesn't mean anything by itself; we use it in combination with the $op(arg, \ldots)$ notation to denote places where variables are bound.

A1.1. The grammar is

$$e ::= \texttt{plus}(e_1, e_2) \mid \texttt{times}(e_1, e_2) \mid \texttt{integ}(e_1, e_2, x.e) \mid \texttt{sum}(e_1, e_2, x.e) \mid x,$$

and $\sum_{i=a}^{b} e$ is represented by $\texttt{sum}(a, b, i.e)$. ∎

The second (extra credit) part was tricky. The catch is that

$$¿ \quad \frac{d}{dx}(x \cdot x) = \frac{d}{dy}(y \cdot y) \quad ?$$

is *not* a valid α-conversion. (To see this: the left-hand side is mathematically equal to $2 \cdot x$, while the right-hand side is $2 \cdot y$, which are certainly not α-equivalent.) The underlying reason is that the derivative (d/dx) really turns a *function* into a *function*. The way to get good higher-order abstract syntax is to add a second argument to the derivative operator specifying at which point the derivative is to be evaluated. This is analogous to the explicit limits we have on the integral and sum signs.

A1.2. The grammar is

$$e ::= \ldots \mid \texttt{deriv}(e_1, x.e_2),$$

and $(d/dx)(e_2)|_{x=e_1}$ is represented by $\texttt{deriv}(e_1, x.e_2)$. ∎

Now we have

$$\frac{d}{dx}(x \cdot x) = \frac{d}{dx}(x \cdot x)|_{x=x} = \frac{d}{dy}(y \cdot y)|_{y=x} \neq \frac{d}{dy}(y \cdot y)|_{y=y} = \frac{d}{dy}(y \cdot y)$$

so the problem with α-conversion is fixed.

§ 2. Substitution theorem (25 points). This problem was rather difficult for a number of reasons. First, why were the notions of free variable and substitution given as inference rules rather than as functions, like we saw in lecture? Because the best way to prove the theorem turns out to be by induction over the derivation $\{e_1/x\}e_2 = e_3$ of substitution. It's possible to induct over the derivation $x : \texttt{int}, \Gamma \vdash e_2 : \texttt{int}$, but the proof turns out to be less direct.

If you're trying to prove a theorem, and there are several possible derivations you could induct on, how can you decide which one to use? One rule of thumb is to see which derivation *gives you the most useful information*. In this problem, inducting on $\{e_1/x\}e_2 = e_3$ gives us information about both e_2 and e_3, which turns out to be extremely useful in writing the cases. If we induct on $x : \texttt{int}, \Gamma \vdash e_2 : \texttt{int}$, we only get information about e_2.

Another issue that seemed to cause trouble was understanding the difference between using an inference rule in the *forward* direction, from knowing the premises to knowing the conclusion, and using inversion to move *backwards* from the conclusion to the premises. If you've already written the premises of a rule as lines in your proof, you can write the conclusion as the next line of the proof. That's just ordinary forward reasoning.

But if you have the conclusion of a rule as a line (say it's line n) in your proof, you *can't* just write a premise as line $n + 1$ and call it an application of the rule. If you want to move backwards this way, you need to call it *inversion* and you need to make sure that the rule you're inverting on is the *only* rule that has a conclusion matching line n. If more than one rule has a conclusion matching line n, then you need to do case analysis, with one case for each rule that could possibly be the bottom-most rule in the derivation represented by line n.

A2.1. Proof. By rule induction on the derivation of $\{e_1/x\}e_2 = e_3$.

Case [$\dfrac{}{\{e_1/x\}x = e_1}$ SVar=]:
where $e_2 = x$ and $e_3 = e_1$
wts $\Gamma \vdash e_1 : \texttt{int}$
1. $\Gamma \vdash e_1 : \texttt{int}$ (given)

Case [$\dfrac{x \neq y}{\{e_1/x\}y = y}$ SVar$\neq$]:
where $e_2 = y$ and $e_3 = y$
wts $\Gamma \vdash y : \texttt{int}$
1. $x : \texttt{int}, \Gamma \vdash y : \texttt{int}$ (given)
2. $\Gamma(y) = \texttt{int}$ ($x \neq y$, inversion on 1)
3. $\Gamma \vdash y : \texttt{int}$ (Var on 2)

Case [$\dfrac{\{e_1/x\}e_2' = e_3' \quad \{e_1/x\}e_2'' = e_3''}{\{e_1/x\}(e_2' + e_2'') = (e_3' + e_3'')}$ SPlus]:
where $e_2 = e_2' + e_2''$ and $e_3 = e_3' + e_3''$
wts $\Gamma \vdash e_3' + e_3'' : \texttt{int}$

1. $x : \texttt{int}, \Gamma \vdash e_2' + e_2'' : \texttt{int}$ (given)
2a. $x : \texttt{int}, \Gamma \vdash e_2' : \texttt{int}$ (inversion on 1)
2b. $x : \texttt{int}, \Gamma \vdash e_2'' : \texttt{int}$ (inversion on 1)
3. $\Gamma \vdash e_1 : \texttt{int}$ (given)
4. $\Gamma \vdash e_3' : \texttt{int}$ (i.h. on $\{e_1/x\}e_2' = e_3'$, 3, and 2a)
5. $\Gamma \vdash e_3'' : \texttt{int}$ (i.h. on $\{e_1/x\}e_2'' = e_3''$, 3, and 2b)
6. $\Gamma \vdash e_3' + e_3'' : \texttt{int}$ (Plus on 4 and 5)

Case [$\dfrac{}{\{e_1/x\}k = k}$ SNum]:
where $e_2 = k$ and $e_3 = k$
wts $\Gamma \vdash k : \texttt{int}$
1. $\Gamma \vdash k : \texttt{int}$ (Num)

Case [$\dfrac{\texttt{FV}(e_1) = S \quad y \notin S \cup \{x\} \quad \{e_1/x\}e_2' = e_3' \quad \{e_1/x\}e_2'' = e_3''}{\{e_1/x\}(\texttt{let } y = e_2' \texttt{ in } e_2'' \texttt{ end}) = (\texttt{let } y = e_3' \texttt{ in } e_3'' \texttt{ end})}$ SLet]:
where $e_2 = (\texttt{let } y = e_2' \texttt{ in } e_2'' \texttt{ end})$ and $e_3 = (\texttt{let } y = e_3' \texttt{ in } e_3'' \texttt{ end})$
wts $\Gamma \vdash (\texttt{let } y = e_3' \texttt{ in } e_3'' \texttt{ end}) : \texttt{int}$
1. $x : \texttt{int}, \Gamma \vdash (\texttt{let } y = e_2' \texttt{ in } e_2'' \texttt{ end}) : \texttt{int}$ (given)
2a. $x : \texttt{int}, \Gamma \vdash e_2' : \texttt{int}$ (inversion on 1)
2b. $x : \texttt{int}, \Gamma, y : \texttt{int} \vdash e_2'' : \texttt{int}$ (inversion on 1)
3. $\Gamma \vdash e_1 : \texttt{int}$ (given)
4. $y \notin S$ ($y \notin S \cup \{x\}$)
5. $\Gamma, y : \texttt{int} \vdash e_1 : \texttt{int}$ (Weakening Lemma on 3, $\texttt{FV}(e_1) = S$, and 4)
6. $\Gamma \vdash e_3' : \texttt{int}$ (i.h. on 3 and 2a)
7. $\Gamma, y : \texttt{int} \vdash e_3'' : \texttt{int}$ (i.h. on 5 and 2b)
8. $\Gamma \vdash (\texttt{let } y = e_3' \texttt{ in } e_3'' \texttt{ end}) : \texttt{int}$ (Let on 6 and 7)

And that's the whole proof. ∎

One thing to note about this proof is that each rule in the rule induction is copied at the start of its case, with variables renamed in order to make sure they don't collide with the variable names in the statement of the theorem. This is really important. Several people got themselves tangled up doing this proof because of a collision between a name like Γ in the rule for the case they were working on and a different Γ in the statement of the theorem.

How many steps do I have to show? There aren't really any hard and fast rules, but there are certainly some steps you can *never* omit:

- Stating which derivation you are inducting on, or case analyzing.
- Applying an inference rule in the forward direction.
- Using inversion to analyze an inference rule in the backward direction.
- Appealing to the induction hypothesis.

- Using the definition of some operator. (Example: using the definition of collapsing for C-machine stacks to get $(k \rhd \Box + e_2) @ e_1 = k @ (e_1 + e_2)$.)

Any time you do any of these things it needs to be a separate step in your proof.

§ 3. Propositional logic (20 points). The first question was straightforward.

A3.1.

$$\frac{}{P_i \texttt{ prop}} \qquad \frac{A \texttt{ prop} \quad B \texttt{ prop}}{A \Rightarrow B \texttt{ prop}}$$

∎

The proof for the second question was somewhat easier because there was only one possible derivation to induct on.

A3.2. Proof. By rule induction on the derivation of A `thm`.

Case [$\dfrac{}{A' \Rightarrow B' \Rightarrow A' \texttt{ thm}}$ K]:
where $A = (A' \Rightarrow B' \Rightarrow A')$
wts $A' \Rightarrow B' \Rightarrow A'$ is a tautology
Hypothetical. Suppose we have a truth assignment for $P_0, P_1, \ldots$.

This determines the truth assignment for A' and B', and by the recursive definition of truth value, the truth assignment for A.

A'	B'	$A' \Rightarrow B'$	A
`true`	`true`	`true`	`true`
`true`	`false`	`false`	`true`
`false`	`true`	`true`	`true`
`false`	`false`	`true`	`true`

Examining the table, we see that the truth assignment for A must be `true`.

So, for any truth assignment for $P_0, P_1, \ldots$, the truth assignment for A is `true`. Then by definition A is a tautology.

Case [$\dfrac{}{(A' \Rightarrow B' \Rightarrow C') \Rightarrow (A' \Rightarrow B') \Rightarrow (A' \Rightarrow C') \texttt{ thm}}$ S]:
where $A = ((A' \Rightarrow B' \Rightarrow C') \Rightarrow (A' \Rightarrow B') \Rightarrow (A' \Rightarrow C'))$
wts $(A' \Rightarrow B' \Rightarrow C') \Rightarrow (A' \Rightarrow B') \Rightarrow (A' \Rightarrow C')$ is a tautology
Hypothetical. Suppose we have a truth assignment for $P_0, P_1, \ldots$.

This determines the truth assignment for A', B', and C', and by the

recursive definition of truth value, the truth assignment for A.

A'	B'	C'	$A' \Rightarrow B'$	$A' \Rightarrow C'$	$B' \Rightarrow C'$	$A' \Rightarrow B' \Rightarrow C'$	A
true	true	true	true	true	true	true	true
true	true	false	true	false	false	false	true
true	false	true	false	true	true	true	true
true	false	false	false	false	true	true	true
false	true	true	true	true	true	true	true
false	true	false	true	true	false	true	true
false	false	true	true	true	true	true	true
false	false	false	true	true	true	true	true

Examining the table, we see that the truth assignment for A must be `true`.

So, for any truth assignment for $P_0, P_1, \ldots$, the truth assignment for A is `true`. Then by definition A is a tautology.

Case $\left[\dfrac{B \Rightarrow A\ \texttt{thm} \quad B\ \texttt{thm}}{A\ \texttt{thm}}\ \text{App} \right]$:

wts A is a tautology

1. $B \Rightarrow A$ is a tautology (i.h. on $B \Rightarrow A$ `thm`)
2. B is a tautology (i.h. on B `thm`)

Hypothetical. Suppose we have a truth assignment for $P_0, P_1, \ldots$.

This determines the truth assignment for A, B, and $B \Rightarrow A$.

A	B	$B \Rightarrow A$
true	true	true
true	false	true
false	true	false
false	false	true

By step 1, and the definition of tautology, we know that the truth assignment for $B \Rightarrow A$ is `true`. By step 2, and the definition of tautology, we know that the truth assignment for B is `true`. Then only the first line of the table can possibly apply, so the truth assignment for A is `true`.

So, for any truth assignment for $P_0, P_1, \ldots$, the truth assignment for A is `true`. Then by definition A is a tautology. ∎

The third (extra credit) question was tricky. Both $A \Rightarrow A$ and $A \Rightarrow B \Rightarrow C \Rightarrow A$ were suggested as tautologies (correct) that are not theorems (wrong). It seems hard to believe, when you first look at the rules App, K, and S, that these could be theorems. But they both are.

Claim 1. $A \Rightarrow A$ `thm` is derivable.

Proof. By forward reasoning. Pick any B you like (say $B = P_0$).

1. $(A \Rightarrow (B \Rightarrow A) \Rightarrow A) \Rightarrow (A \Rightarrow B \Rightarrow A) \Rightarrow (A \Rightarrow A)$ `thm` (S)
2. $A \Rightarrow (B \Rightarrow A) \Rightarrow A$ `thm` (K)

3. $A \Rightarrow B \Rightarrow A$ `thm` (K)
4. $(A \Rightarrow B \Rightarrow A) \Rightarrow (A \Rightarrow A)$ `thm` (App on 1 and 2)
5. $A \Rightarrow A$ `thm` (App on 4 and 3)

We need another little lemma to work our way up to $A \Rightarrow B \Rightarrow C \Rightarrow A$ `thm`.

Claim 2. If $A \Rightarrow C$ `thm` is derivable then $A \Rightarrow B \Rightarrow C$ `thm` is derivable.
Proof. By forward reasoning.

1. $(A \Rightarrow C \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow C) \Rightarrow (A \Rightarrow B \Rightarrow C)$ `thm` (S)
2. $(C \Rightarrow B \Rightarrow C) \Rightarrow A \Rightarrow (C \Rightarrow B \Rightarrow C)$ `thm` (K)
3. $C \Rightarrow B \Rightarrow C$ `thm` (K)
4. $A \Rightarrow C \Rightarrow B \Rightarrow C$ `thm` (App on 2 and 3)
5. $(A \Rightarrow C) \Rightarrow (A \Rightarrow B \Rightarrow C)$ `thm` (App on 1 and 4)
6. $A \Rightarrow C$ `thm` (given)
7. $A \Rightarrow B \Rightarrow C$ `thm` (App on 5 and 6)

Now we can finish.

Claim 3. $A \Rightarrow B \Rightarrow C \Rightarrow A$ `thm` is derivable.
Proof. By forward reasoning.

1. $A \Rightarrow A$ `thm` (Claim 1)
2. $A \Rightarrow C \Rightarrow A$ `thm` (Claim 2 on 1)
3. $A \Rightarrow B \Rightarrow C \Rightarrow A$ `thm` (Claim 2 on 2)

So these answers don't work. A couple of students came up with a correct answer:

A3.3. $((A \Rightarrow B) \Rightarrow A) \Rightarrow A$ is a tautology for any A and B, but $((P_0 \Rightarrow P_1) \Rightarrow P_0) \Rightarrow P_0$ `thm` is not derivable. ∎

The law $((A \Rightarrow B) \Rightarrow A) \Rightarrow A$ is called *Pierce's law* (no relation to Benjamin Pierce). A proof that $((P_0 \Rightarrow P_1) \Rightarrow P_0) \Rightarrow P_0$ `thm` is not derivable is rather difficult, requiring advanced techniques from logic.

One interesting connection with programming languages is that the rules K, S, and App have counterparts in type systems. Compare App with the typing rule for application:

$$\frac{A \Rightarrow B \texttt{ thm} \quad A \texttt{ thm}}{B \texttt{ thm}}\ \text{App} \qquad \frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \tau'}$$

If we erase the Γ and e parts of $\Gamma \vdash e : \tau$, they're really the same. Similarly, there are typings

$$\Gamma \vdash \texttt{lam}(x.\texttt{lam}(y.\texttt{lam}(z.x\ z\ (y\ z)))) : (\tau_1 \to \tau_2 \to \tau_3) \to (\tau_1 \to \tau_2) \to (\tau_1 \to \tau_3)$$

and

$$\Gamma \vdash \texttt{lam}(x.\texttt{lam}(y.x)) : \tau_1 \to \tau_2 \to \tau_1$$

corresponding to S and K, respectively. (Here $\texttt{lam}(x.e)$ abbreviates $\texttt{fun}\ f(x : \tau_1) : \tau_2.e$ for some unused f and some types τ_1 and τ_2 we don't care about.)

Then the proof of $A \Rightarrow A$ `thm` from above, for example, has a programming-language interpretation as a combination of these lambda terms having type $\tau_1 \to$

τ_1 for any τ_1. (A good exercise is to work out the combination, and see that its operational semantics is the same as the identity function.)

This analogy between logic and programming languages goes very deep, and is important enough to have a special name: the *Curry-Howard correspondence*. One of the key ideas of programming languages, *polymorphic types*, was first discovered by bringing a similar idea over from logic via the C.-H. correspondence.

One interesting aspect of all this is that the correspondence doesn't work between our MinML programs and classical logic (logic based on truth tables). There is no MinML program having type $((\tau_1 \to \tau_2) \to \tau_1) \to \tau_1$ for all τ_1 and τ_2 (assuming we don't use recursion). The notion of logic defined by App, K, and S, which corresponds to MinML without recursion, is called *constructive logic*. Pierce's law $((A \Rightarrow B) \Rightarrow A) \Rightarrow A$ does not hold in constructive logic.

So is there a programming language, some extension of MinML perhaps, corresponding to classical logic? In fact there is: if we add the right notion of *continuations* to MinML, then there will be a (non-recursive) program having type $((\tau_1 \to \tau_2) \to \tau_1) \to \tau_1$ for all τ_1 and τ_2. This would be a great exercise to work out, once we see what continuations look like in MinML.

Finally, it turns out that if we work in an *untyped* setting, the programs corresponding to K and S, together with application, all by themselves form a Turing-complete programming language. This is the basis for the amusing language Unlambda, which you can google at your leisure. Challenge problem: find a *single* axiom that, along with App, lets you prove exactly the same theorems you can prove with App, S, and K. The corresponding program is a *single* operator that, together with application, forms a Turing-complete programming language. So there's an untyped Turing-complete language even simpler than Unlambda.