Assignment #3:  
Dynamic Typing

15-312: Principles of Programming Languages

Out: Wednesday, February 18, 2015  
Due: Wednesday, March 4, 2013 1:29PM

Introduction

In this assignment, we will explore the relationships between “dynamically-typed” and statically-typed languages. Specifically, we will show how dynamically classified values can be understood as one among many types of values.

Submission

We will collect exactly the following files from the /afs/andrew/course/15/312/ directory:

    handin/<yourandrewid>/assn3/assn3.pdf
    handin/<yourandrewid>/assn3/dyn/dynamics.sml
    handin/<yourandrewid>/assn3/dyn/listops.dyn
    handin/<yourandrewid>/assn3/hybrid/typechecker.sml
    handin/<yourandrewid>/assn3/hybrid/dynamics.sml
    handin/<yourandrewid>/assn3/hybrid/translate.sml

Make sure that your files have the right names (especially assn3.pdf!) and are in the correct directories.

1 Dynamic PCF

In a dynamically typed language, such as Python, Ruby, Javascript, or Scheme, all implementations must internally tag values with a class when they are created, and all operations must check the class of their operands (perform a “tag check”) each time they are used. The statics are quite simple – they establish that all closed terms can be given the single type ok.
1.1 Syntax

We summarize the syntax of this language here, writing expressions as $d$ and not $e$ to distinguish it from other languages.

<table>
<thead>
<tr>
<th>Sort</th>
<th>Abstract form</th>
<th>Concrete form</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unitype</td>
<td>::= ok</td>
<td>ok</td>
</tr>
<tr>
<td>Class</td>
<td>$c ::= \text{fun}$</td>
<td>$\text{fun}$</td>
</tr>
<tr>
<td></td>
<td>num</td>
<td>num</td>
</tr>
<tr>
<td></td>
<td>nil</td>
<td>nil</td>
</tr>
<tr>
<td></td>
<td>cons</td>
<td>cons</td>
</tr>
<tr>
<td>Exp</td>
<td>$d ::= x$</td>
<td>$x$</td>
</tr>
<tr>
<td></td>
<td>$z$</td>
<td>$z$</td>
</tr>
<tr>
<td></td>
<td>$s(d)$</td>
<td>$s(d)$</td>
</tr>
<tr>
<td></td>
<td>$\text{num}[n]$</td>
<td>$\bar{n}$</td>
</tr>
<tr>
<td></td>
<td>$\text{ifz}(d; d_0; x.d_1)$</td>
<td>$\text{ifz} {z \Rightarrow d_0 \mid s(x) \Rightarrow d_1}$</td>
</tr>
<tr>
<td></td>
<td>$\text{let}(d_0, x.d_1)$</td>
<td>$\text{let} x = d_0 \text{ in } d_1$</td>
</tr>
<tr>
<td></td>
<td>$\text{lam}(x.d)$</td>
<td>$\text{fn}(x) d$</td>
</tr>
<tr>
<td></td>
<td>$\text{ap}(d_1; d_2)$</td>
<td>$d_1 (d_2)$</td>
</tr>
<tr>
<td></td>
<td>$\text{fix}(x.d)$</td>
<td>$\text{fix } x \text{ is } d$</td>
</tr>
<tr>
<td></td>
<td>$\text{triv}$</td>
<td>$\langle \rangle$</td>
</tr>
<tr>
<td></td>
<td>$\text{pair}(d_1; d_2)$</td>
<td>$\langle d_1, d_2 \rangle$</td>
</tr>
<tr>
<td></td>
<td>$\text{pr}<a href="d">l</a>$</td>
<td>$d \cdot 1$</td>
</tr>
<tr>
<td></td>
<td>$\text{pr}<a href="d">r</a>$</td>
<td>$d \cdot r$</td>
</tr>
<tr>
<td></td>
<td>$\text{instof}<a href="d">c</a>$</td>
<td>$c?d$</td>
</tr>
<tr>
<td></td>
<td>$\text{cond}(d; d_1; d_2)$</td>
<td>$\text{cond } d \text{ } d_1 \text{ } d_2$</td>
</tr>
</tbody>
</table>

This language should be quite familiar from the previous assignment: it is PCF without type annotations, without the terms associated with sum types, and with three new constructs.

1.2 Dynamics

Because the statics of Dynamic PCF are so permissive, the dynamics have to pick up the slack and detect most errors at run-time. This can be formalized by defining several judgments:

- $d \text{ val}$  
  $d$ is a closed value
- $d \rightarrow d'$ 
  $d$ steps to $d'$
- $d \text{ err}$ 
  $d$ incurs a run-time error
- $d \text{ is num } n$ 
  $d$ is of class $\text{num}$ with value $n$
- $d \text{ is fun } x.d'$ 
  $d$ is of class $\text{fun}$ with body $x.d'$
- $d \text{ is nil}$ 
  $d$ is of class $\text{nil}$
- $d \text{ is cons } d_1, d_2$ 
  $d$ is of class $\text{cons}$ with body $d_1; d_2$

These judgments are defined in Appendix B.

**Task 1.1** (15%). Implement the dynamics of Dynamic PCF in dyn/dynamics.sml.

**Note:** If you are incrementally revising your Homework 2 sec2 solution, you might want to consider starting with Task 2.1 and then modifying that solution as the solution for Task 1.1.
1.3 List programming

Task 1.2 (15%). Implement the following Dynamic PCF list operations in `dyn/listops.dyn`.

- **append xs ys**
  
  Given a list \( xs = [x_1, \ldots, x_n] \) and another list \( ys = [y_1, \ldots, y_m] \), returns \( [x_1, \ldots, x_n, y_1, \ldots, y_m] \).

- **length xs**
  
  Given a list \( xs = [x_1, \ldots, x_n] \), returns \( \bar{n} \).

- **justnums xs**
  
  Given a list, keeps exactly those items that have the tag `num`.
  
  For example, given the list \( xs = [\langle \rangle, \bar{4}, \text{fn}(x:x), \bar{7}] \), the function would return \([\bar{4}, \bar{7}]\).

- **map f xs**
  
  Given a list \( xs = [x_1, \ldots, x_n] \), returns \([f(x_1), \ldots, f(x_n)]\).

- **tabulate n f**
  
  Returns a list of length \( n \), \([f(0), \ldots, f(n-1)]\).

Two style suggestions:

- This language is essentially a Scheme or LISP variant. Accordingly, use lots of parentheses.

- Your code will be easier to understand if you write conditional statements as
  
  \[
  \text{(cond (nil? xs) (nil_case) (cons_case))}
  \]

  instead of the equivalent

  \[
  \text{(cond (xs) (cons_case) (nil_case))}
  \]

2 Hybrid PCF

In Dynamic PCF, all the tagging and tag checking is done implicitly. The programmer has no control over when dynamically classified values are used, because these are the only types of values there are. We now consider an extension to the statically-typed `sec2` language from the previous homework with a new type,
of dynamically classified values.

<table>
<thead>
<tr>
<th>Sort</th>
<th>Abstract Form</th>
<th>Concrete Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>( \tau ::= )</td>
<td>( \tau )</td>
</tr>
<tr>
<td></td>
<td>( \text{nat} )</td>
<td>( \text{nat} )</td>
</tr>
<tr>
<td></td>
<td>( \text{parr}(\tau_1; \tau_2) )</td>
<td>( \tau_1 \to \tau_2 )</td>
</tr>
<tr>
<td></td>
<td>( \text{unit} )</td>
<td>( \text{unit} )</td>
</tr>
<tr>
<td></td>
<td>( \text{prod}(\tau_1; \tau_2) )</td>
<td>( \tau_1 \times \tau_2 )</td>
</tr>
<tr>
<td></td>
<td>( \text{void} )</td>
<td>( \text{void} )</td>
</tr>
<tr>
<td></td>
<td>( \text{sum}(\tau_1; \tau_2) )</td>
<td>( \tau_1 + \tau_2 )</td>
</tr>
<tr>
<td></td>
<td>( \text{dyn} )</td>
<td>( \text{dyn} )</td>
</tr>
<tr>
<td>Class</td>
<td>( c ::= )</td>
<td>( c )</td>
</tr>
<tr>
<td></td>
<td>( \text{fun} )</td>
<td>( \text{fun} )</td>
</tr>
<tr>
<td></td>
<td>( \text{num} )</td>
<td>( \text{num} )</td>
</tr>
<tr>
<td></td>
<td>( \text{nil} )</td>
<td>( \text{nil} )</td>
</tr>
<tr>
<td></td>
<td>( \text{cons} )</td>
<td>( \text{cons} )</td>
</tr>
<tr>
<td>Exp</td>
<td>( h ::= )</td>
<td>( h )</td>
</tr>
<tr>
<td></td>
<td>( x )</td>
<td>( x )</td>
</tr>
<tr>
<td></td>
<td>( z )</td>
<td>( z )</td>
</tr>
<tr>
<td></td>
<td>( s(h) )</td>
<td>( s(h) )</td>
</tr>
<tr>
<td></td>
<td>( \text{ifz}(h; h_0; x.h_1) )</td>
<td>( \text{ifz} h { z \Rightarrow h_0 \mid s(x) \Rightarrow h_1 } )</td>
</tr>
<tr>
<td></td>
<td>( \text{let}(h_0; x.h_1) )</td>
<td>( \text{let} x = h_0 \text{ in } h_1 )</td>
</tr>
<tr>
<td></td>
<td>( \text{lam}<a href="x.h">\tau</a> )</td>
<td>( \text{fn}(x:\tau) h )</td>
</tr>
<tr>
<td></td>
<td>( \text{ap}(h_1; h_2) )</td>
<td>( h_1(h_2) )</td>
</tr>
<tr>
<td></td>
<td>( \text{fix}<a href="x.h">\tau</a> )</td>
<td>( \text{fix } x:\tau \text{ is } h )</td>
</tr>
<tr>
<td></td>
<td>( \text{abort}<a href="h">\tau</a> )</td>
<td>( \text{abort}<a href="h">\tau</a> )</td>
</tr>
<tr>
<td></td>
<td>( \text{in}[\tau_1; \tau_2]<a href="h">l</a> )</td>
<td>( \text{inl}<a href="h">\tau_1; \tau_2</a> )</td>
</tr>
<tr>
<td></td>
<td>( \text{in}[\tau_1; \tau_2]<a href="h">r</a> )</td>
<td>( \text{inr}<a href="h">\tau_1; \tau_2</a> )</td>
</tr>
<tr>
<td></td>
<td>( \text{case}(h; x_1.h_1; x_2.h_2) )</td>
<td>( \text{case } h { \text{inl}(x_1) \Rightarrow h_1 \mid \text{inr}(x_2) \Rightarrow h_2 } )</td>
</tr>
<tr>
<td></td>
<td>( \text{triv} { } )</td>
<td>( \langle \rangle )</td>
</tr>
<tr>
<td></td>
<td>( \text{pair}(h_1; h_2) )</td>
<td>( \langle h_1, h_2 \rangle )</td>
</tr>
<tr>
<td></td>
<td>( \text{pr}<a href="h">l</a> )</td>
<td>( h \cdot l )</td>
</tr>
<tr>
<td></td>
<td>( \text{pr}<a href="h">r</a> )</td>
<td>( h \cdot r )</td>
</tr>
<tr>
<td></td>
<td>( \text{new}<a href="h">c</a> )</td>
<td>( c! h )</td>
</tr>
<tr>
<td></td>
<td>( \text{cast}<a href="h">c</a> )</td>
<td>( h # c )</td>
</tr>
<tr>
<td></td>
<td>( \text{instof}<a href="h">c</a> )</td>
<td>( c? h )</td>
</tr>
</tbody>
</table>

The introductory form for \( \text{dyn} \) is \( \text{new}[c](h) \). It tags the underlying value \( h \) with the tag \( c \) if \( h \) is of an appropriate type for that class. Natural numbers can be tagged with \( \text{num} \) and functions of type \( \text{dyn} \to \text{dyn} \) can be tagged with \( \text{fun} \), the unit can be tagged with \( \text{nil} \), and a pair with type \( \text{dyn} \times \text{dyn} \) can be tagged with \( \text{cons} \).

The primary elimination form for \( \text{dyn} \) is \( \text{cast}[c](h) \). It performs a tag check on the dynamically classified value \( h \), and if the tags match, it reduces to the underlying value. Additionally, \( \text{instof}[c](h) \) takes any value of type \( \text{dyn} \) and returns a value of type \( \text{bool} = \text{unit} + \text{unit} \); either \( \text{tt} = \text{inl}[\text{unit}; \text{unit}](\langle \rangle) \) if the tag of \( h \) is \( c \) or \( \text{ff} = \text{inr}[\text{unit}; \text{unit}](\langle \rangle) \) if the tags don’t match. The new rules needed on top of your sec2 PCF implementation from the last homework are given in Appendix C.

**Task 2.1** (10%). Implement the statics of Hybrid PCF in \text{hybrid/typechecker.sml} and the dynamics in \text{hybrid/dynamics.sml}.  

Copyright © 2015, Carnegie Mellon University
2.1 Translating Dynamic PCF to Hybrid PCF

The type \( \text{dyn} \) allows the programmer to explicitly perform the tagging (new) and tag checks (cast) that Dynamic PCF was implicitly (and thus pervasively) performing within its dynamics. This can be seen more clearly by performing a translation from Dynamic PCF to Hybrid PCF.

The judgment \( \Omega \vdash d \rightsquigarrow h \) says that the Dynamic PCF term \( d \) translates to the Hybrid PCF term \( h \); they are allowed to share the free variables in \( \Omega = x_1, \ldots, x_n \). It requires an auxiliary judgment \( n \sim_{\text{num}} h \) for translating the untagged portion of numbers, the \( n \) in the term \( \text{num}[n] \), directly to untagged numbers (with type \( \text{nat} \)) in Hybrid PCF:

\[
\frac{0 \sim_{\text{num}} z}{\Omega \vdash x \sim x} \quad \frac{n \sim_{\text{num}} h}{n + 1 \sim_{\text{num}} s(h)}
\]

Some of the rules for the judgment \( d \sim h \) are given below:

\[
\Omega \vdash x \sim x \quad \frac{\Omega \vdash d \rightsquigarrow h}{\Omega \vdash s(d) \sim_{\text{num}} (s(h) \text{ @ num})} \quad \frac{\Omega \vdash n \rightsquigarrow h}{\Omega \vdash \text{num}[n] \sim_{\text{num}} h}
\]

\[
\Omega \vdash d \sim h \quad \Omega \vdash d_0 \sim h_0 \quad \Omega, x \vdash d_1 \sim h_1
\]

\[
\Omega \vdash \text{ifz } d \{ z \Rightarrow d_0 \mid s(x) \Rightarrow d_1 \} \sim (s(h @ \text{num}) \{ z \Rightarrow h_0 \mid s(y) \Rightarrow ((\text{num!})/x)h_1 \})\]

\[
\Omega \vdash x \vdash d_1 \sim h_1 \quad \Omega, x \vdash d_2 \sim h_2
\]

\[
\Omega \vdash \text{let } x = d_1 \text{ in } d_2 \sim \text{let } x = h_1 \text{ in } h_2 \quad \Omega \vdash \text{let } x = d \sim h
\]

\[
\Omega \vdash \text{fix } x \vdash d \sim_{\text{fix }} x : \text{dyn is } h \quad \Omega \vdash () \sim_{\text{nil!}} ()
\]

\[
\Omega \vdash d \sim h \quad \Omega \vdash d_1 \sim h_1 \quad \Omega \vdash d_2 \sim h_2
\]

\[
\Omega \vdash d \sim h \quad \Omega \vdash (d_1, d_2) \sim \text{cons!} (h_1, h_2)
\]

\[
\Omega \vdash d \sim h \quad \Omega \vdash d \sim (h \text{ @ cons}) \cdot l
\]

\[
\Omega \vdash d \sim h \quad \Omega \vdash d \sim (h \text{ @ cons}) \cdot r
\]

For such a translation to be correct, a necessary (but not sufficient!) condition is that the translation be type-directed. That is, for each type in the source language (just \( \text{ok} \) in this case) we can associate a type in the target language with it (just \( \text{dyn} \) in this case) and prove that the translation respects that relation. To specify these theorems we’ll use a new piece of notation: if \( \Omega = x_1, \ldots, x_n \), then \( (\Omega : \text{ok}) = x_1 : \text{ok}, \ldots, x_n : \text{ok} \) and \( (\Omega : \text{dyn}) = x_1, \ldots, x_n : \text{dyn} \).

**Theorem 1** (Translations exist). If \( (\Omega : \text{ok}) \vdash d : \text{ok} \), then there exists an \( h \) such that \( \Omega \vdash d \rightsquigarrow h \).

**Proof.** Rule induction on the derivation of \( (\Omega : \text{ok}) \vdash d : \text{ok} \). Because we’ve left the definition of this boring judgment implicit, we’ll ignore this theorem for now. (It’s not true unless we give some more translation rules, though!)

**Theorem 2** (Translations are well-typed). If \( \Omega \vdash d \rightsquigarrow h \) then \( (\Omega : \text{dyn}) \vdash h : \text{dyn} \).

**Partial proof.** We proceed by rule induction on \( \Omega \vdash d \rightsquigarrow h \) with \( \mathcal{P}(\Omega \vdash d \rightsquigarrow h) = (\Omega : \text{dyn}) \vdash h : \text{dyn} \).
# Proposition 1

Case \((X_3)\) If \(x \in \Omega\) then \(\mathcal{P}(\Omega \vdash x \leadsto x) = (\Omega : \text{dyn}) \vdash x : \text{dyn}\).

Because \(x \in \Gamma\), we have that \(x : \text{dyn} \in (\Omega : \text{dyn})\). The result follows by rule \text{var} of the statics of HPCF.

Case \((X_5)\) If \(\mathcal{P}(\Omega \vdash d \leadsto h)\), then \(\mathcal{P}(\Omega \vdash s(d) \leadsto \text{num}!(s(h @ \text{num}))\).

The result follows by derivation from the induction hypothesis that \((\Omega : \text{dyn}) \vdash h : \text{dyn}:\)

\[
\begin{align*}
(\Omega : \text{dyn}) & \vdash h \text{ : dyn} & H_5 \\
(\Omega : \text{dyn}) & \vdash h \circ \text{num : nat} \quad \text{nat-I}_2 \\
(\Omega : \text{dyn}) & \vdash \text{num}!(s(h @ \text{num})) \text{ : dyn} & H_1
\end{align*}
\]

Case \((X_7)\) If \(\mathcal{P}(\Omega \vdash d \leadsto h)\), \(\mathcal{P}(\Omega \vdash d_0 \leadsto h_0)\), and \(\mathcal{P}(\Omega, x \vdash d_1 \leadsto h_1)\), then

\(\mathcal{P}(\Omega \vdash \text{ifz } d \{ z \Rightarrow d_0 | s(x) \Rightarrow d_1 \} \leadsto \text{ifz } (h @ \text{num}) \{ z \Rightarrow h_0 | s(y) \Rightarrow [(\text{num}!y)/x]h_1 \})\).

1) \((\Omega : \text{dyn}) \vdash h \text{ : dyn} \quad \text{by I.H.}
2) \((\Omega : \text{dyn}) \vdash h_0 \text{ : dyn} \quad \text{by I.H.}
3) \((\Omega : \text{dyn}), x : \text{dyn} \vdash h_1 \text{ : dyn} \quad \text{by I.H.}

To show: \((\Omega : \text{dyn}) \vdash \text{ifz } (h @ \text{num}) \{ z \Rightarrow h_0 | s(y) \Rightarrow [(\text{num}!y)/x]h_1 \} \text{ : dyn}

4) \((\Omega : \text{dyn}), y : \text{nat}, x : \text{dyn} \vdash h_1 \text{ : dyn} \quad \text{by Weakening lemma for Hybrid PCF on (3)}
5) \((\Omega : \text{dyn}), y : \text{nat} \vdash y : \text{nat} \quad \text{by rule \text{var}}
6) \((\Omega : \text{dyn}), y : \text{nat} \vdash \text{num}!y : \text{dyn} \quad \text{by rule } H_1 \text{ on (5)}
7) \((\Omega : \text{dyn}), y : \text{nat} \vdash [\text{num}!y/x]h_1 : \text{dyn} \quad \text{by Substitution lemma for Hybrid PCF on (6) and (4)}

The rest of the result follows by derivation:

\[
\begin{align*}
1) & \quad (\Omega : \text{dyn}) \vdash h \text{ : dyn} & H_5 \\
2) & \quad (\Omega : \text{dyn}) \vdash h_0 : \text{dyn} & H_5 \\
3) & \quad (\Omega : \text{dyn}), y : \text{nat} \vdash y : \text{nat} \quad \text{by rule var} \\
4) & \quad (\Omega : \text{dyn}), y : \text{nat} \vdash \text{num}!y : \text{dyn} & H_5 \\
5) & \quad (\Omega : \text{dyn}), y : \text{nat} \vdash [\text{num}!y/x]h_1 : \text{dyn} & \text{nat-E}
\end{align*}
\]

The other cases are similar. \qed

To fully guarantee correctness of our translation we need the following proposition to hold:

**Proposition 1** (Correctness of translation).

If \(\emptyset \vdash d \leadsto h\), and \(n \leadsto_{\text{num}} h_n\), then \(d \mapsto^* \text{num}[n]\) if and only if \(h \mapsto^* \text{num}!h_n\).

While we won’t actually be able to prove this theorem with the techniques we’ve learned so far, a theorem statement is a really useful thing to have around, because it helps us know when we’ve definitely messed up. Keep this theorem in mind when writing the rest of the translation.

**Task 2.2** (18%). Complete the translation by handling \(\text{ap}(d_1; d_2)\), \(\text{instof}[c](d)\), and \(\text{cond}(d; d_1; d_2)\).

(a) Write down the remaining rules for the judgment \(\Omega \vdash d \leadsto h\).

(b) Prove the cases of Theorem 2 corresponding to your new rules. You may cite any basic lemmas about Hybrid PCF (e.g. canonical forms, inversion, substitution, etc.) without proof. \qed
Task 2.3 (20%). Implement the translation in `hybrid/translator.sml`. In the code the variables for DPCF terms and for HPCF terms have different SML types, so \( \Omega \) will be a mapping from DPCF variables to HPCF variables, rather than a list of shared variables. When you encounter a variable binding, you should create a new HPCF variable and add the mapping from the DPCF variable to the HPCF variable to the variable translation. The variable case can then simply be a lookup.

2.2 Optimizing the Hybrid Language

Now we’ll write some code in the hybrid language – or at least we’ll have our translation write some code for us, and then we’ll try to clean up the result. Your answers in this section must be in well-formatted concrete syntax - either TeX concrete syntax (e.g. \( \text{ifz } s(z) \{ z \Rightarrow z \mid s(x) \Rightarrow x \} \)) or ASCII concrete syntax (e.g. \( \text{ifz } s \ z \{ z \Rightarrow z \mid s \ x \Rightarrow x \} \)).

The Ackermann function is a cute (and notoriously slow) function.

\[
A(m, n) = \begin{cases} 
  n + 1 & \text{if } m = 0 \\
  A(m - 1, 1) & \text{if } n = 0 \\
  A(m - 1, A(m, n - 1)) & \text{otherwise}
\end{cases}
\]

Task 2.4 (4%). Write the Ackermann function as given above in Dynamic PCF (Section 1).

Task 2.5 (4%). Translate the function from the previous task into Hybrid PCF, making explicit all necessary tags and checks and not introducing any optimization (i.e. all subexpressions should be of type \( \text{dyn} \)). You can use your translation rules above or your implementation if you wish.

Task 2.6 (6%). That must have been a scary chunk of code. Optimize this expression using the fact that the term under the fix (hint for 2.5!) is actually a function of the type \( \text{dyn} \rightarrow \text{dyn} \rightarrow \text{dyn} \). Make sure that your term has this type after your optimization (we’ll make it \( \text{dyn} \) again in a bit).

Task 2.7 (6%). Now, assume that your inputs are numbers. If you look closely at your code then you should be able to verify that whenever your function returns a value, it is always a number. So you can just transform your function to type \( \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} \), in such a way that there are no checks and the code runs at full speed.

Task 2.8 (2%). However, as external call sites expect a term of type \( \text{dyn} \) tagged with \( \text{fun} \), where the underlying function also returns a \( \text{dyn} \), encapsulate the function in 2.7 with input tag checks and output tags such that the resulting expression is a term of type \( \text{dyn} \) again.
A Putting The Code Together

As usual, you can compile your files using `CM.make "sources.cm"` in the appropriate directory. We have provided two ways to test the final implementation: an interpreter and a reference implementation. All of these are based on a parsers we provide for you.

In order to generate a comprehensive suite of tests, you are encouraged to share test cases (that do not involve list manipulation) with your classmates.

A.1 Interpreter

As usual, to run the interpreter, execute `TopLevel.repl();`. The REPLs have new features in this assignment: you can ask to load files, and the Hybrid PCF interpreter can be given a Dynamic PCF term to translate and then evaluate.

The syntax for each term construct is as close as possible to the concrete syntax mentioned for it. (The concrete syntax is the second column in the table which introduces the syntax for the language.) We provide below the grammar that the interpreter accepts, as well as two sample sessions of the interpreter.

\[
\begin{align*}
\text{cls} & ::= \text{num} | \text{fun} | \text{nil} | \text{cons} \\
\text{ident} & ::= (*) \text{a letter followed by alphanumeric characters, \_ or ’ *)} \\
\text{numeral} & ::= (*) \text{a series of digits *)}
\end{align*}
\]

\[
\begin{align*}
\text{directives} & ::= \\
& \text{step <dexp>; } \\
& | \text{step;} \\
& | \text{eval <dexp>; } \\
& | \text{eval;} \\
& | \text{use <filename>; } (* \text{Filename of file containing a <dexp> *})
\end{align*}
\]

\[
\begin{align*}
\text{dexp} & ::= \\
& \text{<ident>} \\
& | \text{z} \\
& | \text{s <dexp>} \\
& | \text{<numeral>} \\
& | \text{ifz <dexp> \{ z \Rightarrow <dexp> | s \text{ <ident> \Rightarrow <dexp> } \}} \\
& | \text{let <decls> in <dexp> end} \\
& | \text{fn (<ident>) <dexp>} \\
& | \text{<dexp> <dexp>} \\
& | \text{fix <ident> is <dexp>} \\
& | <> \\
& | \text{<<dexp>, <dexp>>} \\
& | \text{<dexp>.l} \\
& | \text{<dexp>.r} \\
& | \text{<cls>? <dexp>} \\
& | \text{cond <dexp> <dexp> <dexp>} \\
& | (\text{<dexp>})
\end{align*}
\]

\[
\begin{align*}
\text{decls} & ::= \text{<decl> | <decl> <decls>}
\end{align*}
\]

\[
\begin{align*}
\text{decl} & ::= \\
& \text{<ident>}
\end{align*}
\]
val <ident> = <dexp>

(* HYBRID PCF INTERPRETER *)

directive ::=  
  step <hexp>;  
| step;  
| eval <hexp>;  
| eval;  
| trans <dexp>;  
| use <filename>;  
(* If filename ends in ".dyn" use will load and translate a DPCF term  
* from the file, otherwise it will load an HPCF term from the file.  
*)

ty ::= nat | <ty> -> <ty> | unit | <ty> * <ty> | void | <ty> + <ty> | dyn

hexp ::=  
  <ident>  
| z  
| s <hexp>  
| <numeral>  
| ifz <hexp> { z => <hexp> | s <ident> => <hexp> }  
| let <decls> in <hexp> end  
| fn (<ident> : <ty>) <hexp>  
| <hexp> <hexp>  
| fix <ident> : <ty> is <hexp>  
| abort[<ty>] <exp>  
| inl[<ty>,<ty>] <exp>  
| inr[<ty>,<ty>] <exp>  
| case e { inl <ident> => exp | inr <ident> => exp }  
| <>  
| <<hexp>, <hexp>>  
| <hexp>.1  
| <hexp>.r  
| <cls>! <hexp>  
| <hexp> @ <cls>  
| <cls>? <hexp>  
| ( <hexp> )

decls ::= <decl> | <decl> <decls>

decl ::= val <ident> = <hexp>
Here are examples of running the both interpreters:

```bash
$ rlwrap sml -m dyn/sources.cm
- TopLevel.repl ();
->use "dyn/addone.dyn";
|--> (Let ((addone@10, (Lam (x@12 . (Cond ((InstOf (Num, x@12)), (S x@...)
->eval;
(Pair ((Num 5), (Pair ((Lam (y@23 . (S (Ap ((Lam (x@236 . x@236)), y@...
->step (cond (nil? (fn (x) x)) (s z) (s (s z)));
|--> (Cond (Triv, (S Z), (S (S Z)))))
->step;
|--> (S (S Z))
->step;
|--> (S (S (Num 0)))
->eval;
(Num 2) VAL
```

```bash
$ rlwrap sml -m hybrid/sources.cm
- TopLevel.repl ();
->use "dyn/addone.dyn";
Statics: term has type Dyn
|--> (Let ((addone@69, (New (Fun, (Lam ((x@71, Dyn) . (Case ((InstOf (... Statics: term has type Dyn
(New (Cons, (Pair ((New (Num, (S (S (S (S (S Z))))))), (New (Cons, (Pa...
->trans <s z, <>>;
Statics: term has type Dyn
|--> (New (Cons, (Pair ((New (Num, (S (Cast ((New (Num, Z))), Num)))))),...
->step;
Statics: term has type Dyn
|--> (New (Cons, (Pair ((New (Num, (S Z))), (New (Nil, Triv)))))))
->step;
(New (Cons, (Pair ((New (Num, (S Z))), (New (Nil, Triv))))))) VAL
```

### A.2 Reference Implementation

Finally, we have included the solution to both sections of this assignment as a binary heap image, `ref_impl`. You can load it into SML by passing in the `@SMLload=ref_impl` flag. Your solution should behave just like ours.

### B Dynamic PCF

The definition of the judgment \( \Gamma \vdash d : \text{ok} \) is omitted; it’s really boring.

#### B.1 Values

\[
\begin{align*}
\text{num}[n] & \text{val} (D_1) \\
\text{lam}(x.d) & \text{val} (D_2) \\
\{\} & \text{val} (D_3) \\
(d_1 \text{val}, d_2 \text{val}) & \text{val} (D_4)
\end{align*}
\]
B.2 Tag checking

\[
\begin{align*}
\text{num}[n] \text{ is num } n \quad & (D_5) \\
\text{num}[n] \text{ isnt fun } & (D_6) \\
\text{num}[n] \text{ isnt nil } & (D_7) \\
\text{num}[n] \text{ isnt cons } & (D_8) \\
\lambda x.d \text{ is fun } x.d \quad & (D_9) \\
\lambda x.d \text{ isnt num } & (D_{10}) \\
\lambda x.d \text{ isnt nil } & (D_{11}) \\
\lambda x.d \text{ isnt cons } & (D_{12}) \\
\langle \rangle \text{ is nil } & (D_{13}) \\
\langle \rangle \text{ isnt num } & (D_{14}) \\
\langle \rangle \text{ isnt fun } & (D_{15}) \\
\langle \rangle \text{ isnt cons } & (D_{16}) \\
\langle d_1, d_2 \rangle \text{ is cons } d_1, d_2 & (D_{17}) \\
\langle d_1, d_2 \rangle \text{ is cons } d_1, d_2 & (D_{18}) \\
\langle d_1, d_2 \rangle \text{ is cons } d_1, d_2 & (D_{19}) \\
\langle d_1, d_2 \rangle \text{ is cons } d_1, d_2 & (D_{20})
\end{align*}
\]

B.3 Core PCF

\[
\begin{align*}
\mathbf{z} & \mapsto \text{num[0]} & (D_{21}) \\
\mathbf{s}(d) & \mapsto \mathbf{s}(d') & (D_{22}) \\
\mathbf{d} & \mapsto \mathbf{d}' & (D_{23}) \\
\mathbf{d} & \mapsto \mathbf{d}' & (D_{24}) \\
\mathbf{d} & \mapsto \mathbf{d}' & (D_{25}) \\
\text{ifz}(d; d_0; x.d_1) & \mapsto \text{ifz}(d'; d_0; x.d_1) & (D_{26}) \\
\mathbf{d} & \mapsto \mathbf{d}' & (D_{27}) \\
\text{let}(d_0, x.d_1) & \mapsto \text{let}(d_0', x.d_1) & (D_{28}) \\
\mathbf{d} & \mapsto \mathbf{d}' & (D_{29}) \\
\mathbf{d} & \mapsto \mathbf{d}' & (D_{30}) \\
\mathbf{d} & \mapsto \mathbf{d}' & (D_{31}) \\
\mathbf{d} & \mapsto \mathbf{d}' & (D_{32}) \\
\mathbf{d} & \mapsto \mathbf{d}' & (D_{33}) \\
\mathbf{d} & \mapsto \mathbf{d}' & (D_{34}) \\
\mathbf{d} & \mapsto \mathbf{d}' & (D_{35}) \\
\mathbf{d} & \mapsto \mathbf{d}' & (D_{36}) \\
\mathbf{d} & \mapsto \mathbf{d}' & (D_{37}) \\
\mathbf{d} & \mapsto \mathbf{d}' & (D_{38}) \\
\mathbf{d} & \mapsto \mathbf{d}' & (D_{39}) \\
\mathbf{d} & \mapsto \mathbf{d}' & (D_{40}) \\
\mathbf{d} & \mapsto \mathbf{d}' & (D_{41}) \\
\mathbf{d} & \mapsto \mathbf{d}' & (D_{42}) \\
\mathbf{d} & \mapsto \mathbf{d}' & (D_{43}) \\
\mathbf{d} & \mapsto \mathbf{d}' & (D_{44}) \\
\mathbf{d} & \mapsto \mathbf{d}' & (D_{45}) \\
\mathbf{d} & \mapsto \mathbf{d}' & (D_{46}) \\
\mathbf{d} & \mapsto \mathbf{d}' & (D_{47}) \\
\mathbf{d} & \mapsto \mathbf{d}' & (D_{48}) \\
\mathbf{d} & \mapsto \mathbf{d}' & (D_{49}) \\
\mathbf{d} & \mapsto \mathbf{d}' & (D_{50}) \\
\mathbf{d} & \mapsto \mathbf{d}' & (D_{51}) \\
\mathbf{d} & \mapsto \mathbf{d}' & (D_{52})
\end{align*}
\]

B.4 Pairs

\[
\begin{align*}
\langle d_1, d_2 \rangle & \mapsto \langle d_1', d_2 \rangle & (D_{41}) \\
\langle d_1, d_2 \rangle & \mapsto \langle d_1, d_2' \rangle & (D_{42}) \\
\langle d_1, d_2 \rangle & \mapsto \langle d_1', d_2' \rangle & (D_{43}) \\
\langle d_1, d_2 \rangle & \mapsto \langle d_1, d_2' \rangle & (D_{44}) \\
\mathbf{pr}[1](d) & \mapsto \mathbf{pr}[1](d') & (D_{45}) \\
\mathbf{pr}[1](d) & \mapsto \mathbf{pr}[1](d') & (D_{46}) \\
\mathbf{pr}[1](d) & \mapsto \mathbf{pr}[1](d') & (D_{47}) \\
\mathbf{pr}[1](d) & \mapsto \mathbf{pr}[1](d') & (D_{48}) \\
\mathbf{pr}[r](d) & \mapsto \mathbf{pr}[r](d') & (D_{49}) \\
\mathbf{pr}[r](d) & \mapsto \mathbf{pr}[r](d') & (D_{50}) \\
\mathbf{pr}[r](d) & \mapsto \mathbf{pr}[r](d') & (D_{51}) \\
\mathbf{pr}[r](d) & \mapsto \mathbf{pr}[r](d') & (D_{52})
\end{align*}
\]
B.5 Tag checking

\[
\begin{align*}
    d & \mapsto d' \quad \text{(D}_{53}\text{)} \\
    \text{instof}[c](d) & \mapsto \text{instof}[c](d') \quad \text{(D}_{54}\text{)} \\
    d & \text{err} \quad \text{instof}[c](d) & \mapsto \text{err} \quad \text{(D}_{55}\text{)} \\
    d & \text{isnt} c \quad \text{instof}[c](d) & \mapsto \langle \rangle \\
    \text{instof[fun]}(\text{lam}(x. d)) & \mapsto \text{num}[1] \quad \text{(D}_{56}\text{)} \\
    \text{instof[num]}(\text{num}[n]) & \mapsto \text{num}[1] \quad \text{(D}_{57}\text{)} \\
    \text{instof[nil]}(\langle \rangle) & \mapsto \text{num}[1] \quad \text{(D}_{58}\text{)} \\
    d & \mapsto d' \quad \text{cond} d_1 d_2 \mapsto \text{cond} d'_1 d'_2 \quad \text{(D}_{60}\text{)} \\
    d & \text{err} \quad \text{cond} d_1 d_2 \text{ err} \quad \text{(D}_{61}\text{)} \\
    d & \text{isnt} \text{ nil} \quad \text{cond} d_1 d_2 \mapsto d_1 \quad \text{(D}_{62}\text{)} \\
    d & \text{is} \text{ nil} \quad \text{cond} d_1 d_2 \mapsto d_2 \quad \text{(D}_{63}\text{)} \\
\end{align*}
\]
C Hybrid PCF

Hybrid PCF’s dynamic semantics are a direct extension of the dynamics and statics of PCF from the previous homework. In that homework, we did not make the error propagation rules explicit; we will give all those rules here, but we will only give the new cases of \( \Gamma \vdash h : \tau, h \text{ val}, \text{ and } h \mapsto h' \) judgments.

C.1 Statics of dyn fragment

\[
\Gamma \vdash h : \text{nat} \\
\Gamma \vdash \text{new}[\text{num}] (h) : \text{dyn} \\
\Gamma \vdash h : \text{dyn} \rightarrow \text{dyn} \\
\Gamma \vdash \text{new}[\text{fun}] (h) : \text{dyn} \\
\Gamma \vdash h : \text{unit} \\
\Gamma \vdash \text{new}[\text{nil}] (h) : \text{dyn} \\
\Gamma \vdash h : \text{dyn} \times \text{dyn} \\
\Gamma \vdash \text{cast}[\text{num}] (h) : \text{nat} \\
\Gamma \vdash h : \text{dyn} \\
\Gamma \vdash \text{cast}[\text{nil}] (h) : \text{unit} \\
\Gamma \vdash h : \text{dyn} \\
\Gamma \vdash \text{cast}[\text{cons}] (h) : \text{dyn} \times \text{dyn} \\
\Gamma \vdash h : \text{dyn}
\]

C.2 Dynamics of dyn fragment

\[
\frac{h \text{ val}}{\text{new}[c] (h) \text{ val}} (H_{10}) \\
\frac{h \mapsto h'}{\text{new}[c] (h) \rightarrow \text{new}[c] (h')} (H_{11}) \\
\frac{h \text{ err}}{\text{new}[c] (h) \text{ err}} (H_{12}) \\
\frac{h \mapsto h'}{\text{cast}[c] (h) \rightarrow \text{cast}[c] (h')} (H_{13}) \\
\frac{h \text{ err}}{\text{cast}[c] (h) \text{ err}} (H_{14}) \\
\frac{h \text{ val}}{\text{cast}[c] (\text{new}[c] (h)) \rightarrow h} (H_{15}) \\
\frac{h \text{ val}}{\text{cast}[c] (\text{new} [c'] (h)) \text{ err}} (H_{16}) \\
\frac{h \mapsto h'}{\text{instof}[c] (h) \rightarrow \text{instof}[c] (h')} (H_{17}) \\
\frac{h \text{ err}}{\text{instof}[c] (h) \text{ err}} (H_{18}) \\
\frac{h \text{ val}}{\text{instof}[c] (\text{new}[c] (h)) \rightarrow \text{inunit}; \text{unit}; \text{unit}[1]()} (H_{19}) \\
\frac{h \text{ val}}{\text{instof}[c] (\text{new} [c'] (h)) \rightarrow \text{inunit}; \text{unit}; \text{unit}[r]()} (H_{20})
\]

C.3 Dynamics of PCF with sums and products (error propagation rules only)

\[
\frac{h \text{ err}}{s(h) \text{ err}} (H_{21}) \\
\frac{h \text{ err}}{\text{ifz}(h; h_0; x_1 h_1) \text{ err}} (H_{22}) \\
\frac{h \text{ err}}{\text{let}(h; x; h_1) \text{ err}} (H_{23}) \\
\frac{h \text{ err}}{\text{ap}(h_1; h_2) \text{ err}} (H_{24}) \\
\frac{h_1 \text{ err}}{h_2 \text{ err}} (H_{25}) \\
\frac{h \text{ err}}{\text{abort}[	au](h) \text{ err}} (H_{26}) \\
\frac{h \text{ err}}{\text{in}[	au_1; \tau_2][1](h) \text{ err}} (H_{27}) \\
\frac{h \text{ err}}{\text{in}[	au_1; \tau_2][r](h) \text{ err}} (H_{28}) \\
\frac{h \text{ err}}{\text{case}(h; x; h_1; y; h_2) \text{ err}} (H_{29}) \\
\frac{h \text{ err}}{h_1 \text{ err}(h_2) \text{ err}} (H_{30}) \\
\frac{h \text{ err}}{h_1 \text{ val}(h_2) \text{ err}} (H_{31}) \\
\frac{h \text{ err}}{h_1 \text{ err}(h_2) \text{ err}} (H_{32}) \\
\frac{h \text{ err}}{pr[x](h) \text{ err}} (H_{33})
\]