Assignment #3, Update 5:
Dynamic Typing and Refinements

15-312: Principles of Programming Languages

Out: Friday, February 21, 2013
Due: Tuesday, 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, and demonstrate how to prove statements about dynamically typed values using a system of *refinements*.

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/refinements.sml
handin/<yourandrewid>/assn3/dyn/dynamics.sml
handin/<yourandrewid>/assn3/dyn/listops.dyn
handin/<yourandrewid>/assn3/dyn/quicksort.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.

Sort			**Abstract form**	**Concrete form**
Unitype		::=	`ok`	`ok`
Class	c	::=	`fun`	`fun`
			`num`	`num`
			`nil`	`nil`
			`cons`	`cons`
Exp	d	::=	x	x
			`z`	`z`
			$\texttt{s}(d)$	$\texttt{s}(d)$
			$\texttt{num}[n]$	$\bar{n}$
			$\texttt{ifz}(d; d_0; x.d_1)$	$\texttt{ifz}\ d\ \{\texttt{z} \Rightarrow d_0 \mid \texttt{s}(x) \Rightarrow d_1\}$
			$\texttt{let}[\varphi opt](d_0, x.d_1)$	$\texttt{let}\ x = d_0\ \texttt{in}\ d_1$
			$\texttt{lam}(x.d)$	$\texttt{fn}\,(x)\,d$
			$\texttt{ap}(d_1; d_2)$	$d_1\ (d_2)$
			$\texttt{fix}(x.d)$	$\texttt{fix}\ x\ \texttt{is}\ d$
			`triv`	$\langle\rangle$
			$\texttt{pair}(d_1; d_2)$	$\langle d_1, d_2\rangle$
			$\texttt{pr}[\texttt{l}](d)$	$d \cdot \texttt{l}$
			$\texttt{pr}[\texttt{r}](d)$	$d \cdot \texttt{r}$
			$\texttt{instof}[c](d)$	$c?\,d$
			$\texttt{ifnil}(d; d_0; x.y.d_1)$	$\texttt{ifnil}\ d\ \{\langle\rangle \Rightarrow d_0 \mid \langle x, y\rangle \Rightarrow d_1\}$
			$\texttt{cond}(d; d_1; d_2)$	$\texttt{cond}\ d\ d_1\ d_2$

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. The new annotation φopt on the `let` expression can be ignored until we talk about refinements in Section 3.

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 val	d is a closed value
$d \mapsto d'$	d steps to d'
d err	d incurs a run-time error
d isnt c	d is not of class c
d is_num n	d is of class `num` with value n
d is_fun $x.d'$	d is of class `fun` with body $x.d'$
d is_nil	d is of class `nil`
d is_cons d_1, d_2	d is of class `cons` with body $d_1; d_2$

These judgments are defined in Appendix B.

Task 1.1 (10%). 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

A common idiom in Scheme-style dynamic programming is representing the list $[a, b, c, d, e]$ as a series of nested `cons`-tagged values: $\langle a, \langle b, \langle c, \langle d, \langle e, \langle\rangle\rangle\rangle\rangle\rangle\rangle$ in our syntax. The operator `ifnil`, described by Section 18.2 in *PFPL*, is useful for working with such lists, but it is always possible to use `cond` instead. For example, this helper function for list reversal:

```
val revhelper =
fix revhelper is fn (xs) fn (ys)
   ifnil xs
     { <> => ys
     | <x, xs'> => revhelper xs' <x, ys> }
```

can be written with `cond` and `nil?` as follows

```
val revhelper =
fix revhelper is fn (xs) fn (ys)
   (cond (nil? xs)
     (ys)
     (revhelper (xs.r) < xs.l, ys >))
```

Task 1.2 (10%). 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 $xs = [\langle\rangle, \bar{4}, \texttt{fn}\,(x{:}x)\,, \bar{7}]$, 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, $[f(0), \ldots, f(n-1)]$

We suggest you try to use `cond` instead of `ifnil` for practice where possible to get used to programming with this structure. □

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 `(cond (nil? xs) (nil_case) (cons_case))` instead of the equivalent `(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, `dyn`, of dynamically classified values.

Sort			Abstract Form	Concrete Form
Type	τ	::=	$\mathtt{nat}$	$\mathtt{nat}$
			$\mathtt{parr}(\tau_1;\tau_2)$	$\tau_1 \rightharpoonup \tau_2$
			$\mathtt{unit}$	$\mathtt{unit}$
			$\mathtt{prod}(\tau_1;\tau_2)$	$\tau_1 \times \tau_2$
			$\mathtt{void}$	$\mathtt{void}$
			$\mathtt{sum}(\tau_1;\tau_2)$	$\tau_1 + \tau_2$
			$\mathtt{dyn}$	$\mathtt{dyn}$
Class	c	::=	$\mathtt{fun}$	$\mathtt{fun}$
			$\mathtt{num}$	$\mathtt{num}$
			$\mathtt{nil}$	$\mathtt{nil}$
			$\mathtt{cons}$	$\mathtt{cons}$
Exp	h	::=	x	x
			$\mathtt{z}$	$\mathtt{z}$
			$\mathtt{s}(h)$	$\mathtt{s}(h)$
			$\mathtt{ifz}(h;h_0;x.h_1)$	$\mathtt{ifz}\,h\,\{\mathtt{z} \Rightarrow h_0 \mid \mathtt{s}(x) \Rightarrow h_1\}$
			$\mathtt{let}(h_0;x.h_1)$	$\mathtt{let}\,x = h_0\,\mathtt{in}\,h_1$
			$\mathtt{lam}[\tau](x.h)$	$\mathtt{fn}\,(x{:}\tau)\,h$
			$\mathtt{ap}(h_1;h_2)$	$h_1\,(h_2)$
			$\mathtt{fix}[\tau](x.h)$	$\mathtt{fix}\,x{:}\tau\,\mathtt{is}\,h$
			$\mathtt{abort}[\tau](h)$	$\mathtt{abort}[\tau](h)$
			$\mathtt{in}[\tau_1;\tau_2][\mathtt{l}](h)$	$\mathtt{inl}[\tau_1;\tau_2](h)$
			$\mathtt{in}[\tau_1;\tau_2][\mathtt{r}](h)$	$\mathtt{inr}[\tau_1;\tau_2](h)$
			$\mathtt{case}(h;x_1.h_1;x_2.h_2)$	$\mathtt{case}\,h\,\{\mathtt{inl}(x_1) \Rightarrow h_1 \mid \mathtt{inr}(x_2) \Rightarrow h_2\}$
			$\mathtt{triv}$	$\langle\rangle$
			$\mathtt{pair}(h_1;h_2)$	$\langle h_1, h_2\rangle$
			$\mathtt{pr}[\mathtt{l}](h)$	$h \cdot \mathtt{l}$
			$\mathtt{pr}[\mathtt{r}](h)$	$h \cdot \mathtt{r}$
			$\mathtt{new}[c](h)$	$c!\,h$
			$\mathtt{cast}[c](h)$	$h \mathbin{@} c$
			$\mathtt{instof}[c](h)$	$c?\,h$

The introductory form for `dyn` is $\mathtt{new}[c](h)$. 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 `num` and functions of type $\mathtt{dyn} \rightharpoonup \mathtt{dyn}$ can be tagged with `fun`, the unit can be tagged with `nil`, and a pair with type $\mathtt{dyn} \times \mathtt{dyn}$ can be tagged with `cons`.

The primary elimination form for `dyn` is $\mathtt{cast}[c](h)$. It performs a tag check on the dynamically classified value h, and if the tags match, it reduces to the underlying value. Additionally, $\mathtt{instof}[c](h)$ takes any value of type `dyn` and returns a value of type $\mathtt{bool} = \mathtt{unit} + \mathtt{unit}$; either $\mathtt{tt} = \mathtt{inl}[\mathtt{unit};\mathtt{unit}](\langle\rangle)$ if

the tag of h is c or $\texttt{ff} = \texttt{inr}[\texttt{unit};\texttt{unit}](\langle\rangle)$ 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 (5%). Implement the statics of Hybrid PCF in `hybrid/typechecker.sml` and the dynamics in `hybrid/dynamics.sml`. □

2.1 Translating Dynamic PCF to Hybrid PCF

The type `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 $n \rightsquigarrow_{\texttt{num}} h$ for translating the untagged portion of numbers, the n in the term $\texttt{num}[n]$, directly to untagged numbers (with type `nat`) in Hybrid PCF:

$$\frac{}{0 \rightsquigarrow_{\texttt{num}} \texttt{z}}(\mathrm{X}_1) \qquad \frac{n \rightsquigarrow_{\texttt{num}} h}{n+1 \rightsquigarrow_{\texttt{num}} \texttt{s}(h)}(\mathrm{X}_2)$$

Some of the rules for the judgment $d \rightsquigarrow h$ are given below:

$$\frac{x \in \Omega}{\Omega \vdash x \rightsquigarrow x}(\mathrm{X}_3)$$

$$\frac{}{\Omega \vdash \texttt{z} \rightsquigarrow \texttt{num!}\,\texttt{z}}(\mathrm{X}_4) \qquad \frac{\Omega \vdash d \rightsquigarrow h}{\Omega \vdash \texttt{s}(d) \rightsquigarrow \texttt{num!}\,(\texttt{s}(h \mathbin{@} \texttt{num}))}(\mathrm{X}_5) \qquad \frac{n \rightsquigarrow_{\texttt{num}} h}{\Omega \vdash \texttt{num}[n] \rightsquigarrow \texttt{num!}\,h}(\mathrm{X}_6)$$

$$\frac{\Omega \vdash d \rightsquigarrow h \qquad \Omega \vdash d_0 \rightsquigarrow h_0 \qquad \Omega, x \vdash d_1 \rightsquigarrow h_1}{\Omega \vdash \texttt{ifz}\,d\,\{\texttt{z} \Rightarrow d_0 \mid \texttt{s}(x) \Rightarrow d_1\} \rightsquigarrow \texttt{ifz}\,(h \mathbin{@} \texttt{num})\,\{\texttt{z} \Rightarrow h_0 \mid \texttt{s}(y) \Rightarrow [(\texttt{num!}\,y)/x]h_1\}}(\mathrm{X}_7)$$

$$\frac{\Omega \vdash d_1 \rightsquigarrow h_1 \qquad \Omega, x \vdash d_2 \rightsquigarrow h_2}{\Omega \vdash \texttt{let}\ x = d_1\ \texttt{in}\ d_2 \rightsquigarrow \texttt{let}\ x = h_1\ \texttt{in}\ h_2}(\mathrm{X}_8) \qquad \frac{\Omega, x \vdash d \rightsquigarrow h}{\Omega \vdash \texttt{fn}\,(x)\,d \rightsquigarrow \texttt{fun!}\,(\texttt{fn}\,(x{:}\texttt{dyn})\,h)}(\mathrm{X}_9)$$

$$\frac{\Omega, x \vdash d \rightsquigarrow h}{\Omega \vdash \texttt{fix}\,x\,\texttt{is}\,d \rightsquigarrow \texttt{fix}\,x{:}\texttt{dyn}\,\texttt{is}\,h}(\mathrm{X}_{10}) \qquad \frac{}{\Omega \vdash \langle\rangle \rightsquigarrow \texttt{nil!}\,\langle\rangle}(\mathrm{X}_{11}) \qquad \frac{\Omega \vdash d_1 \rightsquigarrow h_1 \qquad \Omega \vdash d_2 \rightsquigarrow h_2}{\Omega \vdash \langle d_1, d_2\rangle \rightsquigarrow \texttt{cons!}\,\langle h_1, h_2\rangle}(\mathrm{X}_{12})$$

$$\frac{\Omega \vdash d \rightsquigarrow h}{\Omega \vdash d \cdot \texttt{l} \rightsquigarrow (h \mathbin{@} \texttt{cons}) \cdot \texttt{l}}(\mathrm{X}_{13}) \qquad \frac{\Omega \vdash d \rightsquigarrow h}{\Omega \vdash d \cdot \texttt{r} \rightsquigarrow (h \mathbin{@} \texttt{cons}) \cdot \texttt{r}}(\mathrm{X}_{14})$$

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 `ok` in this case) we can associate a type in the target language with it (just `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 $(\Omega : \texttt{ok}) = x_1 : \texttt{ok}, \ldots, x_n : \texttt{ok}$ and $(\Omega : \texttt{dyn}) = x_1, \ldots, x_n : \texttt{dyn}$.

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

Proof. Rule induction on the derivation of $(\Omega : \texttt{ok}) \vdash d : \texttt{ok}$. 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* $(\Omega : \mathtt{dyn}) \vdash h : \mathtt{dyn}$.

Partial proof. We proceed by rule induction on $\Omega \vdash d \rightsquigarrow h$ with $\mathcal{P}(\Omega \vdash d \rightsquigarrow h) = (\Omega : \mathtt{dyn}) \vdash h : \mathtt{dyn}$

Case (X_3) If $x \in \Omega$ then $\mathcal{P}(\Omega \vdash x \rightsquigarrow x) = (\Omega : \mathtt{dyn}) \vdash x : \mathtt{dyn}$.

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

Case (X_5) If $\mathcal{P}(\Omega \vdash d \rightsquigarrow h)$, then $\mathcal{P}(\Omega \vdash \mathtt{s}(d) \rightsquigarrow \mathtt{num!}\,(\mathtt{s}(h \mathbin{@} \mathtt{num})))$.

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

$$\dfrac{\dfrac{\dfrac{(\Omega : \mathtt{dyn}) \vdash h : \mathtt{dyn}}{(\Omega : \mathtt{dyn}) \vdash h \mathbin{@} \mathtt{num} : \mathtt{nat}}\, H_5}{(\Omega : \mathtt{dyn}) \vdash \mathtt{s}(h \mathbin{@} \mathtt{num}) : \mathtt{nat}}\, \mathtt{nat}\text{-}I_2}{(\Omega : \mathtt{dyn}) \vdash \mathtt{num!}\,(\mathtt{s}(h \mathbin{@} \mathtt{num})) : \mathtt{dyn}}\, H_1$$

Case (X_7) If $\mathcal{P}(\Omega \vdash d \rightsquigarrow h)$, $\mathcal{P}(\Omega \vdash d_0 \rightsquigarrow h_0)$, and $\mathcal{P}(\Omega, x \vdash d_1 \rightsquigarrow h_1)$, then
$\mathcal{P}(\Omega \vdash \mathtt{ifz}\ d\ \{\mathtt{z} \Rightarrow d_0 \mid \mathtt{s}(x) \Rightarrow d_1\} \rightsquigarrow \mathtt{ifz}\ (h \mathbin{@} \mathtt{num})\ \{\mathtt{z} \Rightarrow h_0 \mid \mathtt{s}(y) \Rightarrow [(\mathtt{num!}\, y)/x]h_1\})$.

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

To show: $(\Omega : \mathtt{dyn}) \vdash \mathtt{ifz}\ (h \mathbin{@} \mathtt{num})\ \{\mathtt{z} \Rightarrow h_0 \mid \mathtt{s}(y) \Rightarrow [(\mathtt{num!}\, y)/x]h_1\} : \mathtt{dyn}$

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

The rest of the result follows by derivation:

$$\dfrac{\dfrac{\overset{(1)}{(\Omega : \mathtt{dyn}) \vdash h : \mathtt{dyn}}}{(\Omega : \mathtt{dyn}) \vdash h \mathbin{@} \mathtt{num} : \mathtt{nat}}\, H_5 \quad \overset{(2)}{(\Omega : \mathtt{dyn}) \vdash h_0 : \mathtt{dyn}} \quad \overset{(7)}{(\Omega : \mathtt{dyn}), y : \mathtt{nat} \vdash [\mathtt{num!}\, y/x]h_1 : \mathtt{dyn}}}{(\Omega : \mathtt{dyn}) \vdash \mathtt{ifz}\ (h \mathbin{@} \mathtt{num})\ \{\mathtt{z} \Rightarrow h_0 \mid \mathtt{s}(y) \Rightarrow [(\mathtt{num!}\, y)/x]h_1\} : \mathtt{dyn}}\, \mathtt{nat}\text{-}E$$

The other cases are similar. □

Task 2.2 (10%). Complete the translation by handling $\mathtt{ap}(d_1; d_2)$, $\mathtt{instof}[c](d)$, $\mathtt{ifnil}(d; d_0; x.y.d_1)$, and $\mathtt{cond}(d; d_1; d_2)$.

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

(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. □

Task 2.3 (10%). Implement the translation in `hybrid/translator.sml`. □

2.2 Correctness of translation

The properties we proved in the previous section were necessary *but not sufficient* criteria for the correctness of the translation. Here's an improved attempt at stating a correctness condition:

Proposition 1 (Corretness of translation)**.**
If $\emptyset \vdash d \leadsto h$*, and* $n \leadsto_{\texttt{num}} h_n$*, then* $d \mapsto^* \texttt{num}[n]$ *if and only if* $h \mapsto^* \texttt{num}!\, h_n$*.*

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.

Task 2.4 (5%). This is not a correct translation rule for `ifnil`:

$$\frac{\Omega \vdash d \leadsto h \quad \Omega \vdash d_0 \leadsto h_0 \quad \Omega, x, y \vdash d_1 \leadsto h_1 \quad h_1' = [(h \mathbin{@} \texttt{cons}) \cdot \texttt{l}/x][(h \mathbin{@} \texttt{cons}) \cdot \texttt{r}/y]h_1}{\Omega \vdash \texttt{ifnil}\ d\ \{\langle\rangle \Rightarrow d_0 \mid \langle x, y\rangle \Rightarrow d_1\} \leadsto \texttt{case}\ (\texttt{nil?}\ h)\ \{\texttt{inl}(\_) \Rightarrow h_0 \mid \texttt{inr}(\_) \Rightarrow h_1'\}}$$

Explain why by in terms of Proposition 1 – give (the concrete syntax of) an expression d and explain (informally) why it would serve as a counterexample to Proposition 1 if we used this translation rule. □

2.3 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. $\texttt{ifz}\ \texttt{s}(\texttt{z})\ \{\texttt{z} \Rightarrow \texttt{z} \mid \texttt{s}(x) \Rightarrow x\}$) or ASCII concrete syntax (e.g. `ifz s z { z => z | s x => 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 2.5 (3%). Write the Ackermann function as given above in Dynamic PCF (Section 1).

Task 2.6 (2%). 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 `dyn`). You can use your translation rules above or your implementation if you wish.

Task 2.7 (4%). That must have been a scary chunk of code. Optimize this expression using the fact that the term under the fix (hint for 2.6!) is actually a function of the type $\texttt{dyn} \rightharpoonup \texttt{dyn} \rightharpoonup \texttt{dyn}$. Make sure that your term has this type after your optimization (we'll make it `dyn` again in a bit).

Task 2.8 (4%). 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 $\texttt{nat} \rightharpoonup \texttt{nat} \rightharpoonup \texttt{nat}$, in such a way that there are no checks and the code runs at full speed.

Task 2.9 (2%). However, as external call sites expect a term of type `dyn` tagged with `fun`, where the underlying function also returns a `dyn`, encapsulate the function in 2.8 with input tag checks and output tags such that the resulting expression is a term of type `dyn` again.

3 Refinements

During the optimization task above, you mentally kept track of typing invariants to "guess" the "type" of subexpressions. One way to systematically state these invariants is to use refinements. In a full system of refinement types (like Chapter 24 of the current revision of *PFPL*) we say that any valid refinement *refines a particular type*. But in this assignment we consider refinements for Dynamic PCF where there's only one type, `ok`, to refine, so every refinement φ refines that one type. Refinements characterize sets of well-typed terms that satisfy some property: we write $d \in \varphi$ to say that d satisfies the property φ.

Our particular goal here is to use refinements to characterize expressions that *don't have runtime errors*. The least interesting refinement is $\top$: we want to be able to prove $\texttt{num}[0] \in \top$, for example, and so we'd also like to be able to prove $\texttt{z} \in \top$ because $\texttt{z} \mapsto \texttt{num}[0]$.

If we want to show that any interesting programs are in $\top$, we're going to need to get more specific about our refinements. If we want to prove $\texttt{s}(d) \in \top$ or $\texttt{ifz}(d; d_0; x.d_1) \in \top$, we're going to need to know more than that $d \in \top$. In both cases, we're going to need to know that if the subexpression d evaluates to a value, it evaluates to $\texttt{num}[n]$. We'll express the property that an expression evaluates to a number (if it terminates) with the refinement `num`. To handle `ifz` correctly, we'll also need a hypothetical context with hypotheses of the form $x \in \varphi$.

$$\frac{}{\Sigma, x \in \varphi \vdash x \in \varphi}(\text{ref-var}) \qquad \frac{}{\Sigma \vdash \texttt{num}[n] \in \texttt{num}}(\text{ref-num}) \qquad \frac{}{\Sigma \vdash \texttt{z} \in \texttt{num}}(\text{ref-z}) \qquad \frac{\Sigma \vdash d \in \texttt{num}}{\Sigma \vdash \texttt{s}(d) \in \texttt{num}}(\text{ref-s})$$

$$\frac{\Sigma \vdash d \in \texttt{num} \qquad \Sigma \vdash d_0 \in \varphi \qquad \Sigma, x \in \texttt{num} \vdash d_1 \in \varphi}{\Sigma \vdash \texttt{ifz}(d; d_0; x.d_1) \in \varphi}(\text{ref-ifz})$$

You may have noticed that this is beginning to look a lot like a type system. Refinements have some things in common with types: they allow us to express how our programs behave, and we can use them to prove that certain kinds of errors are impossible. Because they describe the behavior of terms but do not influence the structure of values the way types do, refinements are a little bit more ad-hoc. We've already seen with `s` and `ifz`: the language of refinements influences what programs we can prove error-freedom for. Here's the language we'll be working with in this section:

Sort			**Abstract form**	**Concrete form**
Prop	φ	$::=$	`top`	$\top$
			$\texttt{and}(\varphi_1, \varphi_2)$	$\varphi_1 \,\&\, \varphi_2$
			`num`	`num`
			$\texttt{fun}(\varphi_1, \varphi_2)$	$\varphi_1 \rightharpoonup \varphi_2$
			$\texttt{pair}(\varphi_1, \varphi_2)$	$\varphi_1 \times \varphi_2$
			$\texttt{list}(\varphi)$	$\texttt{list}(\varphi)$

We've already discussed $\top$ and `num`. The refinement $\varphi_1 \,\&\, \varphi_2$ holds of terms that satisfy both the refinement φ_1 and the refinement φ_2. The refinement $\texttt{fun}(\varphi_1, \varphi_2)$ holds of terms that, if they terminate, evaluate to functions that that, when given a value satisfying refinement φ_1, satisfy the refinement φ_2. The refinement $\varphi_1 \times \varphi_2$ similarly holds of terms that, if they terminate, evaluate to pairs of terms satisfying φ_1 and φ_2 respectively. The last refinement, $\texttt{list}(\varphi)$, can be used to classify the the Scheme-style lists we were working with earlier in the assignment.

Here are the remaining rules for refinement checking. The first two rules deal with let binding, which have abstract syntax $\texttt{let}[\varphi opt](d_0, x.d_1)$. The previously-ignored φopt is an optional annotation that allows

us to stipulate what refinement of the argument is supposed to be; it will play an important role soon when we talk about bidirectional refinement checking.

$$\frac{\Sigma \vdash d_0 \in \varphi_1 \qquad \Sigma, x \in \varphi_1 \vdash d_1 \in \varphi_2}{\Sigma \vdash \mathtt{let}[\mathtt{none}](d_0, x.d_1) \in \varphi_2}(\text{ref-}\mathtt{let}) \qquad \frac{\Sigma \vdash d_0 \in \varphi_1 \qquad \Sigma, x \in \varphi_1 \vdash d_1 \in \varphi_2}{\Sigma \vdash \mathtt{let}[\mathtt{some}(\varphi_1)](d_0, x.d_1) \in \varphi_2}(\text{ref-}\mathtt{let}\text{-anno})$$

$$\frac{\Sigma, x \in \varphi_1 \vdash d \in \varphi_2}{\Sigma \vdash \mathtt{lam}(x.d) \in \varphi_1 \rightharpoonup \varphi_2}(\text{ref-}\mathtt{fun}) \qquad \frac{\Sigma \vdash d_1 \in \varphi_1 \rightharpoonup \varphi_2 \qquad \Sigma \vdash d_2 \in \varphi_1}{\Sigma \vdash \mathtt{ap}(d_1; d_2) \in \varphi_2}(\text{ref-}\mathtt{ap})$$

$$\frac{\Sigma, x \in \varphi \vdash d \in \varphi}{\Sigma \vdash \mathtt{fix}(x.d) \in \varphi}(\text{ref-}\mathtt{fix}) \qquad \frac{\Sigma \vdash d_1 \in \varphi_1 \Sigma \vdash d_2 \in \varphi_2}{\Sigma \vdash \langle d_1, d_2 \rangle \in \varphi_1 \times \varphi_2} \qquad \frac{\Sigma \vdash d \in \varphi_1 \times \varphi_2}{\Sigma \vdash d \cdot \mathtt{l} \in \varphi_1}(\text{ref-}\mathtt{prl})$$

$$\frac{\Sigma \vdash d \in \varphi_1 \times \varphi_2}{\Sigma \vdash d \cdot \mathtt{r} \in \varphi_2}(\text{ref-}\mathtt{prr}) \qquad \frac{}{\Sigma \vdash \langle\rangle \in \mathtt{list}(\varphi)}(\text{ref-}\langle\rangle) \qquad \frac{\Sigma \vdash d_1 \in \varphi \Sigma \vdash d_2 \in \mathtt{list}(\varphi)}{\Sigma \vdash \langle d_1, d_2 \rangle \in \mathtt{list}(\varphi)}(\text{ref-}\mathtt{cons})$$

$$\frac{\Sigma \vdash d \in \mathtt{list}(\varphi') \qquad \Sigma \vdash d_0 \in \varphi \qquad \Sigma, x \in \varphi', y \in \mathtt{list}(\varphi') \vdash d_1 \in \varphi}{\Sigma \vdash \mathtt{ifnil}(d; d_0; x.y.d_1) \in \varphi}(\text{ref-}\mathtt{ifnil}\text{-}\mathtt{list})$$

$$\frac{\Sigma \vdash d \in \varphi_1 \times \varphi_2 \qquad \Sigma, x \in \varphi_1, y \in \varphi_2 \vdash d_1 \in \varphi}{\Sigma \vdash \mathtt{ifnil}(d; d_0; x.y.d_1) \in \varphi}(\text{ref-}\mathtt{ifnil}\text{-}\mathtt{pair}) \qquad \frac{\Sigma \vdash d \in \top}{\Sigma \vdash \mathtt{instof}[c](d) \in \top}(\text{ref-}\mathtt{instof})$$

$$\frac{\Sigma \vdash d \in \top \qquad \Sigma \vdash d_1 \in \varphi \qquad \Sigma \vdash d_2 \in \varphi}{\Sigma \vdash \mathtt{cond}(d; d_1; d_2) \in \varphi}(\text{ref-}\mathtt{cond}) \qquad \frac{\Sigma \vdash d \in \varphi_1 \qquad \Sigma \vdash d \in \varphi_2}{\Sigma \vdash d \in \varphi_1 \,\&\, \varphi_2}(\text{ref-}\wedge)$$

$$\frac{\Sigma \vdash d \in \varphi_1 \qquad \varphi_1 \leq \varphi_2}{\Sigma \vdash d \in \varphi_2}(\text{ref-entails})$$

The last rule is an important one. If a refinement φ *entails* another refinement φ', which we write as $\varphi \leq \varphi'$, then any term satisfying φ also satisfies φ'. Refinement entailment is how we can use (ref-$\mathtt{z}$) and still prove that $\mathtt{z} \in \top$. These are the rules for refinement entailment:

$$\frac{}{\varphi \leq \top}(\leq\top) \qquad \frac{\varphi \leq \varphi_1 \qquad \varphi \leq \varphi_2}{\varphi \leq \varphi_1 \,\&\, \varphi_2}(\leq\&\text{-I}) \qquad \frac{\varphi_1 \leq \varphi}{\varphi_1 \,\&\, \varphi_2 \leq \varphi}(\leq\&\text{-E1}) \qquad \frac{\varphi_2 \leq \varphi}{\varphi_1 \,\&\, \varphi_2 \leq \varphi}(\leq\&\text{-E2})$$

$$\frac{}{\mathtt{num} \leq \mathtt{num}}(\leq\mathtt{num}) \qquad \frac{\varphi_1' \leq \varphi_1 \qquad \varphi_2 \leq \varphi_2'}{\varphi_1 \rightharpoonup \varphi_2 \leq \varphi_1' \rightharpoonup \varphi_2'}(\leq\rightharpoonup) \qquad \frac{\varphi \leq \varphi'}{\mathtt{list}(\varphi) \leq \mathtt{list}(\varphi')}(\leq\mathtt{list})$$

$$\frac{\varphi_1 \leq \varphi' \qquad \varphi_2 \leq \mathtt{list}(\varphi')}{\varphi_1 \times \varphi_2 \leq \mathtt{list}(\varphi')}(\leq\times\mathtt{list}) \qquad \frac{\varphi_1 \leq \varphi_1' \qquad \varphi_2 \leq \varphi_2'}{\varphi_1 \times \varphi_2 \leq \varphi_1' \times \varphi_2'}(\leq\times)$$

We can prove by induction that entailment is reflexive ($\varphi \leq \varphi$ for all φ) and transitive (if $\varphi_1 \leq \varphi_2$ and $\varphi_2 \leq \varphi_3$ then $\varphi_1 \leq \varphi_3$).

Task 3.1 (5%). Show that $\mathtt{fix}\ nums\ \mathtt{is}\ \mathtt{fn}\,(n)\,\mathtt{ifz}\ n\ \{\mathtt{z} \Rightarrow \langle\rangle \mid \mathtt{s}(m) \Rightarrow \langle m, nums\ (m)\rangle\}$ satisfies the refinement $\mathtt{num} \rightharpoonup \mathtt{list}(\mathtt{num})$. You can give a derivation or write out the judgments and rules linearly. □

Task 3.2 (5%). The entailment rule for functions is odd: the first premise is $\varphi_1' \leq \varphi_1$, whereas we might expect it to be the other way around. Show why this alternate rule:

$$\frac{\varphi_1 \leq \varphi_1' \qquad \varphi_2 \leq \varphi_2'}{\varphi_1 \rightharpoonup \varphi_2 \leq \varphi_1' \rightharpoonup \varphi_2'}(\leq\text{broken})$$

cannot be right by showing that you can use it to give a refinement to an expression which will always fail at runtime. (The relationship between lists and pairs may be helpful here, but it's not necessary.) □

3.1 Bidirectional Refinement Checking

The rules for refinement checking are different from type checking rules in some important ways. For one, each term d may have many possible refinements, even infinitely many (e.g. $\texttt{lam}(x.x)$ has refinements $\top \rightharpoonup \top$ and $(\top \rightharpoonup \texttt{num}) \rightharpoonup (\top \rightharpoonup \texttt{num})$ and $((\top \rightharpoonup \top) \rightharpoonup \top) \rightharpoonup \top$ and so on. Thus, we simply cannot write a function that *synthesizes* a unique refinement given just a term and context, as we have always done so far with types.

However, sometimes we can still synthesize refinements. We know that $\texttt{z}$ and $\texttt{s}(d)$ have a best refinement of $\texttt{num}$, and if we know that d synthesizes the refinement $\varphi_1 \times \varphi_2$, then we can synthesize the refinement of $d \cdot \texttt{l}$ to be φ_1. Even better, if we know that d synthesizes $\varphi_1 \rightharpoonup \varphi_2$, then we can try to see if $d\ (d')$ synthesizes φ_2, and now we have an easier problem remaining: we have to *check* the term d' against the refinement φ_1. With the exception of $\texttt{let}$, which takes many forms, we can pretty cleanly separate out all our Dynamic PCF language constructs into terms s that synthesize refinements and terms e that need to have their refinements checked.

Sort			**Concrete form**
Synth	s	::=	x
			$\texttt{z}$
			$\texttt{s}(e)$
			$\texttt{let}\ x = s_0\ \texttt{in}\ s_1$
			$\texttt{let}\ x : \varphi = e_0\ \texttt{in}\ s_1$
			$s_1\ (e_2)$
			$s \cdot \texttt{l}$
			$s \cdot \texttt{r}$
			$c?\ e$
Check	e	::=	s
			$\texttt{ifz}\ s\ \{\texttt{z} \Rightarrow e_0 \mid \texttt{s}(x) \Rightarrow e_1\}$
			$\texttt{let}\ x = s_0\ \texttt{in}\ e_1$
			$\texttt{let}\ x : \varphi = e_0\ \texttt{in}\ e_1$
			$\texttt{fn}\ (x)\ e$
			$\texttt{fix}\ x\ \texttt{is}\ e$
			$\langle\rangle$
			$\langle e_1, e_2\rangle$
			$\texttt{ifnil}\ s\ \{\langle\rangle \Rightarrow e_1 \mid \langle x, y\rangle \Rightarrow e_2\}$
			$\texttt{cond}\ s\ e_1\ e_2$

In some respects, this separation severely limits the language. We can no longer, for instance, write programs like $(\texttt{fn}\ (x)\ \texttt{s}(x))(\texttt{s}(\texttt{z}))$: The expression $\texttt{fn}\ (x)\ \texttt{s}(x)$ can only have its refinement checked, and the left half

of an application must synthesize its refinement. On the other hand, in regular programming practice we don't really care about defining recursive functions and then immediately applying them: instead, we define them at a let-binding, where it's not so burdensome to give a refinement.

Synthesis

$$\frac{}{\Sigma, x \in \varphi \vdash x \Rightarrow \varphi} \qquad \frac{}{\Sigma \vdash \mathtt{z} \Rightarrow \mathtt{num}} \qquad \frac{\Sigma \vdash e \Leftarrow \mathtt{num}}{\Sigma \vdash \mathtt{s}(e) \Rightarrow \mathtt{num}} \qquad \frac{\Sigma \vdash e_0 \Rightarrow \varphi_0 \qquad \Sigma, x \in \varphi_0 \vdash s_1 \Rightarrow \varphi}{\Sigma \vdash \mathtt{let}\ x = e_0\ \mathtt{in}\ s_1 \Rightarrow \varphi}$$

$$\frac{\Sigma \vdash e_0 \Leftarrow \varphi_0 \qquad \Sigma, x \in \varphi_0 \vdash s_1 \Rightarrow \varphi}{\Sigma \vdash \mathtt{let}\ x : \varphi_0 = e_0\ \mathtt{in}\ s_1 \Rightarrow \varphi} \qquad \frac{\Sigma \vdash s_1 \Rightarrow \varphi_1 \rightharpoonup \varphi_2 \qquad \Sigma \vdash e_2 \Leftarrow \varphi_1}{\Sigma \vdash s_1\ (e_2) \Rightarrow \varphi_2}$$

$$\frac{\Sigma \vdash s \Rightarrow \varphi_1 \times \varphi_2}{\Sigma \vdash s \cdot \mathtt{l} \Rightarrow \varphi_1} \qquad \frac{\Sigma \vdash s \Rightarrow \varphi_1 \times \varphi_2}{\Sigma \vdash s \cdot \mathtt{r} \Rightarrow \varphi_2} \qquad \frac{\Sigma \vdash e \Leftarrow \top}{\Sigma \vdash c?\, e \Rightarrow \top}$$

Checking

$$\frac{\Sigma \vdash e_0 \Rightarrow \varphi_0 \qquad \Sigma, x \in \varphi_0 \vdash s_1 \Leftarrow \varphi}{\Sigma \vdash \mathtt{let}\ x = e_0\ \mathtt{in}\ s_1 \Leftarrow \varphi} \qquad \frac{\Sigma \vdash e_0 \Leftarrow \varphi_0 \qquad \Sigma, x \in \varphi_0 \vdash s_1 \Leftarrow \varphi}{\Sigma \vdash \mathtt{let}\ x : \varphi_0 = e_0\ \mathtt{in}\ s_1 \Leftarrow \varphi}$$

$$\frac{\Sigma \vdash e \Rightarrow \mathtt{num} \qquad \Sigma \vdash s_0 \Leftarrow \varphi \qquad \Sigma, x \in \mathtt{num} \vdash s_1 \Leftarrow \varphi}{\Sigma \vdash \mathtt{ifz}\ e\ \{\mathtt{z} \Rightarrow s_0 \mid \mathtt{s}(x) \Rightarrow s_1\} \Leftarrow \varphi} \qquad \frac{\Sigma, x \in \varphi_1 \vdash e \Leftarrow \varphi_2}{\Sigma \vdash \mathtt{fn}\,(x)\, e \Leftarrow \varphi_1 \rightharpoonup \varphi_2}$$

$$\frac{\Sigma, x \in \varphi \vdash e \Leftarrow \varphi}{\Sigma \vdash \mathtt{fix}\ x\ \mathtt{is}\ e \Leftarrow \varphi} \qquad \frac{}{\Sigma \vdash \langle\rangle \Leftarrow \top} \qquad \frac{}{\Sigma \vdash \langle\rangle \Leftarrow \mathtt{list}(\varphi)} \qquad \frac{\Sigma \vdash e_1 \Leftarrow \top \qquad \Sigma \vdash e_2 \Leftarrow \top}{\Sigma \vdash \langle e_1, e_2\rangle \Leftarrow \top}$$

$$\frac{\Sigma \vdash e_1 \Leftarrow \varphi_1 \qquad \Sigma \vdash e_2 \Leftarrow \varphi_2}{\Sigma \vdash \langle e_1, e_2\rangle \Leftarrow \varphi_1 \times \varphi_2} \qquad \frac{\Sigma \vdash e_1 \Leftarrow \varphi \qquad \Sigma \vdash e_2 \Leftarrow \mathtt{list}(\varphi)}{\Sigma \vdash \langle e_1, e_2\rangle \Leftarrow \mathtt{list}(\varphi)}$$

$$\frac{\Sigma \vdash s \Rightarrow \mathtt{list}(\varphi') \qquad \Sigma \vdash e_0 \Leftarrow \varphi \qquad \Sigma, x \in \varphi', y \in \mathtt{list}(\varphi') \vdash e_1 \Leftarrow \varphi}{\Sigma \vdash \mathtt{ifnil}\ s\ \{\langle\rangle \Rightarrow e_0 \mid \langle x, y\rangle \Rightarrow e_1\} \Leftarrow \varphi}$$

$$\frac{\Sigma \vdash s \Rightarrow \varphi_1 \times \varphi_2 \qquad \Sigma, x \in \varphi_1, y \in \varphi_2 \vdash e_1 \Leftarrow \varphi}{\Sigma \vdash \mathtt{ifnil}\ s\ \{\langle\rangle \Rightarrow e_0 \mid \langle x, y\rangle \Rightarrow e_1\} \Leftarrow \varphi} \qquad \frac{\Sigma \vdash e \Leftarrow \top \qquad \Sigma \vdash e_1 \Leftarrow \varphi \qquad \Sigma \vdash e_2 \Leftarrow \varphi}{\Sigma \vdash \mathtt{cond}\ e\ e_1\ e_2 \Leftarrow \varphi}$$

$$\frac{\Sigma \vdash s \Rightarrow \varphi_1 \qquad \varphi_1 \leq \varphi_2}{\Sigma \vdash s \Leftarrow \varphi_2}$$

It's straightforward to prove by simultaneous rule induction that $\Sigma \vdash e \Leftarrow \varphi$ implies $\Sigma \vdash e \in \varphi$ and $\Sigma \vdash e \Rightarrow \varphi$ implies $\Sigma \vdash e \in \varphi$. Such a theorem is necessary, because while we can prove is a version of progress and preservation for the original set of refinement rules, the bidirectional refinement rules fail preservation: a term that synthesizes a refinement, like "$\mathtt{let}\ f : \mathtt{num} \rightharpoonup \mathtt{num} = \mathtt{fn}\,(x)\,\mathtt{s}(x)\ \mathtt{in}\ f(\mathtt{s}(\mathtt{z}))$," can step to a term "$(\mathtt{fn}\,(x)\,\mathtt{s}(x))(\mathtt{s}(\mathtt{z}))$" that does not.

Task 3.3 (15%). Using the reference implementation of Dynamic PCF, rewrite your list functions from Task 1.2 in `dyn/quicksort.dyn` so that they pass the refinement checker. You should also add two new functions:

- `partition : (nums -> top) -> list num -> (list num * list num)`

  Takes a function f and a list $xs = [x_1, \ldots, x_n]$ and returns two lists $\langle as, bs \rangle$, both of which are subsequences of xs. The first list as contains all the x_i such that $f(x_i)$ was non-`nil`, and the second list contains all the x_i such that $f(x_i)$ returned $\langle\rangle$.

- `quicksort : list num -> list num`

  Implements a sorting algorithm: takes a list of numbers and returns a list with the same elements, sorted least to greatest. It's relatively straightforward to implement Quicksort using the `partition` function, but any sort that passes the refinement checker will do.

Two hints. First, our sample solution does not use any additional refinement annotations beyond those given in the `quicksort.dyn` stub file. Second, it will be necessary to turn some of the instance of `cond` into instances of `ifnil` in order to pass the refinement checker.

Task 3.4 (10%). The function `justnums` can be given the refinement `list top -> list top`, but it really seams like that function ought to have the refinement `list top -> list num`. Briefly explain why this is not possible in our system and how this relates to what *PFPL* calls "Boolean blindness."

Propose an extension to the dynamics of Dynamic PCF and/or the refinement type system and show how it could be used to rewrite `justnums` such that it has the desired refinement.

Bonus task: Implement bidirectional refinement checking in `dyn/refinements.sml`. As a (substantial!) additional challenge, implement the rules for refinement intersection:

$$\frac{\Sigma \vdash e \Leftarrow \varphi_1 \qquad \Sigma \vdash e \Leftarrow \varphi_2}{\Sigma \vdash e \Leftarrow \varphi_1 \,\&\, \varphi_2} \qquad \frac{\Sigma \vdash e \Rightarrow \varphi_1 \,\&\, \varphi_2}{\Sigma \vdash e \Rightarrow \varphi_1} \qquad \frac{\Sigma \vdash e \Rightarrow \varphi_1 \,\&\, \varphi_2}{\Sigma \vdash e \Rightarrow \varphi_2}$$

The checking rule is quite easy to implement, but the two synthesis rules make type synthesis very nondeterministic. It will require some non-trivial programming to get this to work correctly.

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. This is the second column in the table in which introduce the syntax for a language. We provide below the grammar that the interpreter accepts, as well as two sample sessions of the interpreter.

```
cls ::= num | fun | nil | cons
ident ::= (* a letter followed alphanumeric characters, _, or ' *)
numeral ::= (* a series of digits *)

(* DYNAMIC PCF *)
command ::=
  step <dexp>;
| step;
| eval <dexp>;
| eval;
| loadfile <filename>;  (* Filename of file containing a <dexp> *)

pred ::=
  top
| <pred> & <pred>
| num
| <pred> -> <pred>
| <pred> * <pred>
| list <pred>

dexp ::=
  z
| s <dexp>
| <numeral>
| ifz <dexp> { z => <dexp> | s <ident> => <dexp> }
| let <decls> in <dexp> end
| fn (<ident>) <dexp>
| <dexp>(<dexp>)
| fix <ident> is <dexp>
| <>
| <<dexp>, <dexp>>
| <dexp>.l
| <dexp>.r
```

```
| <cls>? <dexp>
| ifnil <dexp> { <> => <dexp> | <<ident>, <ident>> => <dexp> }
| cond <dexp> <dexp> <dexp>
| (<dexp>)

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

decl ::=
  val <ident> = <dexp>          (* <dexp> should synthesize a refinement *)
| val <ident> : <pred> = <dexp> (* <dexp> should check against <pred> *)

(* HYBRID PCF INTERPRETER *)
command ::=
  step <hexp>;
| step;
| eval <hexp>;
| eval;
| trans <dexp>;
| loadfile <filename>;  (* Filename of file containing a <hexp> *)
| transfile <filename>; (* Filename of file containing a <dexp> *)

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

hexp ::=
  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>.l
| <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:

```
$ rlwrap sml -m dyn/sources.cm
- TopLevel.repl ();
->loadfile dyn/addone.dyn;
 --> let(lam(x@12. cond(instance[num](x@12), s(x@12), cond(instance[fun...
->eval;
 pair(num[5], pair(lam(y@134. s(ap(lam(x@135. x@135), y@134))), pair(pa...
->step (cond (nil? (fn (x) x)) (1) (2));
 --> cond(triv, s(z), s(s(z)))
->step;
 --> s(s(z))
->step;
 --> s(s(num[0]))
->eval;
 num[2] VAL

$ rlwrap sml -m hybrid/sources.cm
- TopLevel.repl ();
->transfile dyn/addone.dyn;
Statics : exp has type dyn
 --> let(new[fun](lam[dyn](x@45. case(instance[nil](case(instance[num](...
->eval;
Statics : exp has type dyn
 new[cons](pair(new[num](s(s(s(s(s(z)))))), new[cons](pair(new[fun](lam...
->trans <s z, <>>;
Statics : exp has type dyn
 --> new[cons](pair(new[num](s(cast[num](new[num](z)))), new[nil](triv)))
->step;
Statics : exp 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

Unless you do the bonus portion of the assignment, you will need to use the reference interpreter to test your quicksort implementation against the refinement type checker.

The reference interpreter for Dynamic PCF, which does refinement type checking, can be loaded like this:

```
$ rlwrap sml @SMLload=/afs/andrew/course/15/312/bin/ref_impl_dyn
- TopLevel.repl ();
```

The reference interpreter for Hybrid PCF (which does not include an implementation of the translation from Dynamic PCF to Hybrid PCF) can be run like this:

```
$ rlwrap sml @SMLload=/afs/andrew/course/15/312/bin/ref_impl_hybrid
- TopLevel.repl ();
```

B Dynamic PCF

The definition of the judgment $\Gamma \vdash d : \mathtt{ok}$ is omitted; it's really boring.

B.1 Values

$$\frac{}{\mathtt{num}[n]\ \mathsf{val}}(\mathrm{D}_1) \qquad \frac{}{\mathtt{lam}(x.d)\ \mathsf{val}}(\mathrm{D}_2) \qquad \frac{}{\langle\rangle\ \mathsf{val}}(\mathrm{D}_3) \qquad \frac{d_1\ \mathsf{val} \qquad d_2\ \mathsf{val}}{\langle d_1, d_2\rangle\ \mathsf{val}}(\mathrm{D}_4)$$

B.2 Tag checking

$$\frac{}{\mathtt{num}[n]\ \mathsf{is\_num}\ n}(\mathrm{D}_5) \qquad \frac{}{\mathtt{num}[n]\ \mathsf{isnt}\ \mathtt{fun}}(\mathrm{D}_6) \qquad \frac{}{\mathtt{num}[n]\ \mathsf{isnt}\ \mathtt{nil}}(\mathrm{D}_7) \qquad \frac{}{\mathtt{num}[n]\ \mathsf{isnt}\ \mathtt{cons}}(\mathrm{D}_8)$$

$$\frac{}{\mathtt{lam}(x.d)\ \mathsf{is\_fun}\ x.d}(\mathrm{D}_9) \qquad \frac{}{\mathtt{lam}(x.d)\ \mathsf{isnt}\ \mathtt{num}}(\mathrm{D}_{10}) \qquad \frac{}{\mathtt{lam}(x.d)\ \mathsf{isnt}\ \mathtt{nil}}(\mathrm{D}_{11}) \qquad \frac{}{\mathtt{lam}(x.d)\ \mathsf{isnt}\ \mathtt{cons}}(\mathrm{D}_{12})$$

$$\frac{}{\langle\rangle\ \mathsf{is\_nil}}(\mathrm{D}_{13}) \qquad \frac{}{\langle\rangle\ \mathsf{isnt}\ \mathtt{num}}(\mathrm{D}_{14}) \qquad \frac{}{\langle\rangle\ \mathsf{isnt}\ \mathtt{fun}}(\mathrm{D}_{15}) \qquad \frac{}{\langle\rangle\ \mathsf{isnt}\ \mathtt{cons}}(\mathrm{D}_{16})$$

$$\frac{d_1\ \mathsf{val} \qquad d_2\ \mathsf{val}}{\langle d_1, d_2\rangle\ \mathsf{is\_cons}\ d_1, d_2}(\mathrm{D}_{17}) \qquad \frac{d_1\ \mathsf{val} \qquad d_2\ \mathsf{val}}{\langle d_1, d_2\rangle\ \mathsf{isnt}\ \mathtt{num}}(\mathrm{D}_{18}) \qquad \frac{d_1\ \mathsf{val} \qquad d_2\ \mathsf{val}}{\langle d_1, d_2\rangle\ \mathsf{isnt}\ \mathtt{fun}}(\mathrm{D}_{19}) \qquad \frac{d_1\ \mathsf{val} \qquad d_2\ \mathsf{val}}{\langle d_1, d_2\rangle\ \mathsf{isnt}\ \mathtt{nil}}(\mathrm{D}_{20})$$

B.3 Core PCF

$$\frac{}{\mathtt{z} \mapsto \mathtt{num}[0]}(\mathrm{D}_{21}) \qquad \frac{d \mapsto d'}{\mathtt{s}(d) \mapsto \mathtt{s}(d')}(\mathrm{D}_{22}) \qquad \frac{d\ \mathsf{err}}{\mathtt{s}(d)\ \mathsf{err}}(\mathrm{D}_{23}) \qquad \frac{d\ \mathsf{is\_num}\ n}{\mathtt{s}(d) \mapsto \mathtt{num}[n+1]}(\mathrm{D}_{24})$$

$$\frac{d\ \mathsf{isnt}\ \mathtt{num}}{\mathtt{s}(d)\ \mathsf{err}}(\mathrm{D}_{25}) \qquad \frac{d \mapsto d'}{\mathtt{ifz}(d; d_0; x.d_1) \mapsto \mathtt{ifz}(d'; d_0; x.d_1)}(\mathrm{D}_{26}) \qquad \frac{d\ \mathsf{err}}{\mathtt{ifz}(d; d_0; x.d_1)\ \mathsf{err}}(\mathrm{D}_{27})$$

$$\frac{d\ \mathsf{is\_num}\ 0}{\mathtt{ifz}(d; d_0; x.d_1) \mapsto d_0}(\mathrm{D}_{28}) \qquad \frac{d\ \mathsf{is\_num}\ n+1}{\mathtt{ifz}(d; d_0; x.d_1) \mapsto [\mathtt{num}[n]/x]d_1}(\mathrm{D}_{29}) \qquad \frac{d\ \mathsf{isnt}\ \mathtt{num}}{\mathtt{ifz}(d; d_0; x.d_1)\ \mathsf{err}}(\mathrm{D}_{30})$$

$$\frac{d_0 \mapsto d_0'}{\mathtt{let}[\varphi opt](d_0, x.d_1) \mapsto \mathtt{let}[\varphi opt](d_0', x.d_1)}(\mathrm{D}_{31}) \qquad \frac{d_0\ \mathsf{err}}{\mathtt{let}[\varphi opt](d_0, x.d_1)\ \mathsf{err}}(\mathrm{D}_{32})$$

$$\frac{d_0\ \mathsf{val}}{\mathtt{let}[\varphi opt](d_0, x.d_1) \mapsto [d_0/x]d_1}(\mathrm{D}_{33}) \qquad \frac{d_1 \mapsto d_1'}{\mathtt{ap}(d_1; d_2) \mapsto \mathtt{ap}(d_1'; d_2)}(\mathrm{D}_{34}) \qquad \frac{d_1\ \mathsf{err}}{\mathtt{ap}(d_1; d_2)\ \mathsf{err}}(\mathrm{D}_{35})$$

$$\frac{d_1\ \mathsf{val} \qquad d_2 \mapsto d_2'}{\mathtt{ap}(d_1; d_2) \mapsto \mathtt{ap}(d_1; d_2')}(\mathrm{D}_{36}) \qquad \frac{d_1\ \mathsf{val} \qquad d_2\ \mathsf{err}}{\mathtt{ap}(d_1; d_2)\ \mathsf{err}}(\mathrm{D}_{37}) \qquad \frac{d_1\ \mathsf{is\_fun}\ x.d \qquad d_2\ \mathsf{val}}{\mathtt{ap}(d_1; d_2) \mapsto [d_2/x]d}(\mathrm{D}_{38})$$

$$\frac{d_1\ \mathsf{isnt}\ \mathtt{fun} \qquad d_2\ \mathsf{val}}{\mathtt{ap}(d_1; d_2)\ \mathsf{err}}(\mathrm{D}_{39}) \qquad \frac{}{\mathtt{fix}(x.d) \mapsto [\mathtt{fix}(x.d)/x]d}(\mathrm{D}_{40})$$

B.4 Pairs

$$\frac{d_1 \mapsto d_1'}{\langle d_1, d_2\rangle \mapsto \langle d_1', d_2\rangle}(\mathrm{D}_{41}) \qquad \frac{d_1 \ \mathsf{err}}{\langle d_1, d_2\rangle \ \mathsf{err}}(\mathrm{D}_{42}) \qquad \frac{d_1 \ \mathsf{val} \qquad d_2 \mapsto d_2'}{\langle d_1, d_2\rangle \mapsto \langle d_1, d_2'\rangle}(\mathrm{D}_{43}) \qquad \frac{d_1 \ \mathsf{val} \qquad d_2 \ \mathsf{err}}{\langle d_1, d_2\rangle \ \mathsf{err}}(\mathrm{D}_{44})$$

$$\frac{d \mapsto d'}{\mathtt{pr}[\mathtt{l}](d) \mapsto \mathtt{pr}[\mathtt{l}](d')}(\mathrm{D}_{45}) \qquad \frac{d \ \mathsf{err}}{\mathtt{pr}[\mathtt{l}](d) \ \mathsf{err}}(\mathrm{D}_{46}) \qquad \frac{d \ \mathsf{is\_cons} \ d_1, d_2}{\mathtt{pr}[\mathtt{l}](d) \mapsto d_1}(\mathrm{D}_{47}) \qquad \frac{d \ \mathsf{isnt\ cons}}{\mathtt{pr}[\mathtt{l}](d) \ \mathsf{err}}(\mathrm{D}_{48})$$

$$\frac{d \mapsto d'}{\mathtt{pr}[\mathtt{r}](d) \mapsto \mathtt{pr}[\mathtt{r}](d')}(\mathrm{D}_{49}) \qquad \frac{d \ \mathsf{err}}{\mathtt{pr}[\mathtt{r}](d) \ \mathsf{err}}(\mathrm{D}_{50}) \qquad \frac{d \ \mathsf{is\_cons} \ d_1, d_2}{\mathtt{pr}[\mathtt{r}](d) \mapsto d_2}(\mathrm{D}_{51}) \qquad \frac{d \ \mathsf{isnt\ cons}}{\mathtt{pr}[\mathtt{r}](d) \ \mathsf{err}}(\mathrm{D}_{52})$$

B.5 Tag checking

$$\frac{d \mapsto d'}{\mathtt{instof}[c](d) \mapsto \mathtt{instof}[c](d')}(\mathrm{D}_{53}) \qquad \frac{d \ \mathsf{err}}{\mathtt{instof}[c](d) \ \mathsf{err}}(\mathrm{D}_{54}) \qquad \frac{d \ \mathsf{isnt}\ c}{\mathtt{instof}[c](d) \mapsto \langle\rangle}(\mathrm{D}_{55})$$

$$\frac{}{\mathtt{instof}[\mathtt{num}](\mathtt{num}[n]) \mapsto \mathtt{num}[1]}(\mathrm{D}_{56}) \qquad \frac{}{\mathtt{instof}[\mathtt{fun}](\mathtt{lam}(x.d)) \mapsto \mathtt{num}[1]}(\mathrm{D}_{57})$$

$$\frac{}{\mathtt{instof}[\mathtt{nil}](\langle\rangle) \mapsto \mathtt{num}[1]}(\mathrm{D}_{58}) \qquad \frac{d_1 \ \mathsf{val} \qquad d_2 \ \mathsf{val}}{\mathtt{instof}[\mathtt{cons}](\langle d_1, d_2\rangle) \mapsto \mathtt{num}[1]}(\mathrm{D}_{59})$$

$$\frac{d \mapsto d'}{\mathtt{ifnil}(d; d_0; x.y.d_1) \mapsto \mathtt{ifnil}(d'; d_0; x.y.d_1)}(\mathrm{D}_{60}) \qquad \frac{d \ \mathsf{err}}{\mathtt{ifnil}(d; d_0; x.y.d_1) \ \mathsf{err}}(\mathrm{D}_{61})$$

$$\frac{d \ \mathsf{is\_nil}}{\mathtt{ifnil}(d; d_0; x.y.d_1) \mapsto d_0}(\mathrm{D}_{62}) \qquad \frac{d \ \mathsf{is\_cons} \ d_l, d_r}{\mathtt{ifnil}(d; d_0; x.y.d_1) \mapsto [d_l/x][d_r/y]d_1}(\mathrm{D}_{63})$$

$$\frac{d \ \mathsf{isnt}\ \mathtt{nil} \qquad d \ \mathsf{isnt\ cons}}{\mathtt{ifnil}(d; d_0; x.y.d_1) \ \mathsf{err}}(\mathrm{D}_{64})$$

$$\frac{d \mapsto d'}{\mathtt{cond}\ d\ d_1\ d_2 \mapsto \mathtt{cond}\ d'\ d_1\ d_2}(\mathrm{D}_{65}) \qquad \frac{d \ \mathsf{err}}{\mathtt{cond}\ d\ d_1\ d_2 \ \mathsf{err}}(\mathrm{D}_{66})$$

$$\frac{d \ \mathsf{isnt}\ \mathtt{nil}}{\mathtt{cond}\ d\ d_1\ d_2 \mapsto d_1}(\mathrm{D}_{67}) \qquad \frac{d \ \mathsf{is\_nil}}{\mathtt{cond}\ d\ d_1\ d_2 \mapsto d_2}(\mathrm{D}_{68})$$

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 val, and $h \mapsto h'$ judgments.

C.1 Statics of `dyn` fragment

$$\frac{\Gamma \vdash h : \mathtt{nat}}{\Gamma \vdash \mathtt{new}[\mathtt{num}](h) : \mathtt{dyn}}(\mathrm{H}_1) \qquad \frac{\Gamma \vdash h : \mathtt{dyn} \rightharpoonup \mathtt{dyn}}{\Gamma \vdash \mathtt{new}[\mathtt{fun}](h) : \mathtt{dyn}}(\mathrm{H}_2) \qquad \frac{\Gamma \vdash h : \mathtt{unit}}{\Gamma \vdash \mathtt{new}[\mathtt{nil}](h) : \mathtt{dyn}}(\mathrm{H}_3)$$

$$\frac{\Gamma \vdash h : \mathtt{dyn} \times \mathtt{dyn}}{\Gamma \vdash \mathtt{new}[\mathtt{cons}](h) : \mathtt{dyn}}(\mathrm{H}_4) \qquad \frac{\Gamma \vdash h : \mathtt{dyn}}{\Gamma \vdash \mathtt{cast}[\mathtt{num}](h) : \mathtt{nat}}(\mathrm{H}_5) \qquad \frac{\Gamma \vdash h : \mathtt{dyn}}{\Gamma \vdash \mathtt{cast}[\mathtt{fun}](h) : \mathtt{dyn} \rightharpoonup \mathtt{dyn}}(\mathrm{H}_6)$$

$$\frac{\Gamma \vdash h : \mathtt{dyn}}{\Gamma \vdash \mathtt{cast}[\mathtt{nil}](h) : \mathtt{unit}}(\mathrm{H}_7) \qquad \frac{\Gamma \vdash h : \mathtt{dyn}}{\Gamma \vdash \mathtt{cast}[\mathtt{cons}](h) : \mathtt{dyn} \times \mathtt{dyn}}(\mathrm{H}_8)$$

$$\frac{\Gamma \vdash h : \mathtt{dyn}}{\Gamma \vdash \mathtt{instof}[c](h) : \mathtt{unit} + \mathtt{unit}}(\mathrm{H}_9)$$

C.2 Dynamics of `dyn` fragment

$$\frac{h\ \mathsf{val}}{\mathtt{new}[c](h)\ \mathsf{val}}(\mathrm{H}_{10}) \qquad \frac{h \mapsto h'}{\mathtt{new}[c](h) \mapsto \mathtt{new}[c](h')}(\mathrm{H}_{11}) \qquad \frac{h\ \mathsf{err}}{\mathtt{new}[c](h)\ \mathsf{err}}(\mathrm{H}_{12})$$

$$\frac{h \mapsto h'}{\mathtt{cast}[c](h) \mapsto \mathtt{cast}[c](h')}(\mathrm{H}_{13}) \qquad \frac{h\ \mathsf{err}}{\mathtt{cast}[c](h)\ \mathsf{err}}(\mathrm{H}_{14}) \qquad \frac{h\ \mathsf{val}}{\mathtt{cast}[c](\mathtt{new}[c](h)) \mapsto h}(\mathrm{H}_{15})$$

$$\frac{h\ \mathsf{val} \qquad c \neq c'}{\mathtt{cast}[c](\mathtt{new}[c'](h))\ \mathsf{err}}(\mathrm{H}_{16}) \qquad \frac{h \mapsto h'}{\mathtt{instof}[c](h) \mapsto \mathtt{instof}[c](h')}(\mathrm{H}_{17}) \qquad \frac{h\ \mathsf{err}}{\mathtt{instof}[c](h)\ \mathsf{err}}(\mathrm{H}_{18})$$

$$\frac{h\ \mathsf{val}}{\mathtt{instof}[c](\mathtt{new}[c](h)) \mapsto \mathtt{in}[\mathtt{unit};\mathtt{unit}][\mathtt{l}](\langle\rangle)}(\mathrm{H}_{19}) \qquad \frac{h\ \mathsf{val} \qquad c \neq c'}{\mathtt{instof}[c](\mathtt{new}[c'](h)) \mapsto \mathtt{in}[\mathtt{unit};\mathtt{unit}][\mathtt{r}](\langle\rangle)}(\mathrm{H}_{20})$$

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

$$\frac{h\ \mathsf{err}}{\mathtt{s}(h)\ \mathsf{err}}(\mathrm{H}_{21}) \qquad \frac{h\ \mathsf{err}}{\mathtt{ifz}(h; h_0; x.h_1)\ \mathsf{err}}(\mathrm{H}_{22}) \qquad \frac{h\ \mathsf{err}}{\mathtt{let}(h; x.h_1)\ \mathsf{err}}(\mathrm{H}_{23}) \qquad \frac{h_1\ \mathsf{err}}{\mathtt{ap}(h_1; h_2)\ \mathsf{err}}(\mathrm{H}_{24})$$

$$\frac{h_1\ \mathsf{val} \qquad h_2\ \mathsf{err}}{\mathtt{ap}(h_1; h_2)\ \mathsf{err}}(\mathrm{H}_{25}) \qquad \frac{h\ \mathsf{err}}{\mathtt{abort}[\tau](h)\ \mathsf{err}}(\mathrm{H}_{26}) \qquad \frac{h\ \mathsf{err}}{\mathtt{in}[\tau_1;\tau_2][\mathtt{l}](h)\ \mathsf{err}}(\mathrm{H}_{27}) \qquad \frac{h\ \mathsf{err}}{\mathtt{in}[\tau_1;\tau_2][\mathtt{r}](h)\ \mathsf{err}}(\mathrm{H}_{28})$$

$$\frac{h_1\ \mathsf{err}}{\langle h_1, h_2\rangle\ \mathsf{err}}(\mathrm{H}_{29}) \qquad \frac{h_1\ \mathsf{val} \qquad h_2\ \mathsf{err}}{\langle h_1, h_2\rangle\ \mathsf{err}}(\mathrm{H}_{30}) \qquad \frac{h\ \mathsf{err}}{\mathtt{pr}[\mathtt{l}](h)\ \mathsf{err}}(\mathrm{H}_{31}) \qquad \frac{h\ \mathsf{err}}{\mathtt{pr}[\mathtt{r}](h)\ \mathsf{err}}(\mathrm{H}_{32})$$

$$\frac{h\ \mathsf{err}}{\mathtt{case}(h; x.h_1; y.h_2)\ \mathsf{err}}(\mathrm{H}_{33})$$