# Assignment #2, Update 1:
# Recursion, Finite Data Types and Pattern Matching

15-312: Principles of Programming Languages

Out: Tuesday, February 4, 2013
Due: Tuesday, February 18, 2013 1:29PM

## Introduction

In this assignment, we will take look at the fix-point theory of recursion, and implement and analyze PCF with nullary and binary sums and products, considering both explicit eliminatory forms and elimination by pattern matching. As usual, start early, and ask the TAs if you need help.

### Submission

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

```
handin/<yourandrewid>/assn2/assn2.pdf

handin/<yourandrewid>/assn2/sec2/typeops.sml
handin/<yourandrewid>/assn2/sec2/termops.sml
handin/<yourandrewid>/assn2/sec2/typechecker.sml
handin/<yourandrewid>/assn2/sec2/uncheckeddynamics.sml

handin/<yourandrewid>/assn2/syn/abt-util.sml
handin/<yourandrewid>/assn2/syn/pattern.sml
handin/<yourandrewid>/assn2/sec3/typeops.sml
handin/<yourandrewid>/assn2/sec3/termops.sml
handin/<yourandrewid>/assn2/sec3/typechecker.sml
handin/<yourandrewid>/assn2/sec3/inexhaustivedynamics.sml
```

Make sure that your files have the right names (especially `assn2.pdf`!) and are in the correct directories.

## 1 Recursion and Fixed Points

In this section we will look at the theory of recursion underlying Plotkin's PCF. Consider the following definition of the fast exponentiation algorithm to calculate $2^n$ using the standard integer operations of `div` and `mod`.

$$\exp(n) = \begin{cases} 1 & \text{if } n = 0 \\ (\exp(n \texttt{ div } 2))^2 & \text{if } n > 0 \text{ and } n \texttt{ mod } 2 = 0 \\ 2 * \exp(n-1) & \text{if } n > 0 \text{ and } n \texttt{ mod } 2 \neq 0 \end{cases}$$

This definition has a self-reference to $\exp$, the thing that is being defined. We untie this knot by writing a functional $E$ that, if we are given a function $f$ from natural numbers to natural numbers, will allow us to define another function $f'$ from natural numbers $n$ to natural numbers like this:

$$f' \quad = \quad n \mapsto \begin{cases} 1 & \text{if } n = 0 \\ (f(n \texttt{ div } 2))^2 & \text{if } n > 0 \text{ and } n \texttt{ mod } 2 = 0 \\ 2 * f(n-1) & \text{if } n > 0 \text{ and } n \texttt{ mod } 2 \neq 0 \end{cases}$$

The exponentiation function we seek is then the fixed point of the operator $E$, a function $e$ such that $E(e) = e$. We arrive at this fixed point as a limit of a series of approximations of the desired solutions by repeatedly applying the functional $E$. An approximation is a partial function $\phi$ which agrees with $f$ for all inputs for which $\phi$ is defined. In other words, $\phi(n) = m$ implies that $f(n) = m$. The worst approximation function is the function $\perp$ which is undefined for all numbers.

One way to approach this idea is to think about partial functions as their graphs. The *graph* of a partial function $\phi$ is simply the set of tuples $(n, m)$ for which $\phi(n) = m$; for any $n$ there will be at most one $m$ such that $(n, m)$ is in the set. The graph of $\perp$ is the empty set $\{\}$. The set $\{(0, 4), (1, 8), (2, 3), (4, 0)\}$ is the graph of a partial function $g$ where $g(0) = 4$, $g(1) = 8$, $g(2) = 3$, $g(4) = 0$, and where $g$ is undefined on 3 and any number larger or equal to than 5.

We can apply the functional $E$ to *any* function, not just approximations of `exp`. The function $E(\perp)$ has the graph $\{(0, 1)\}$, and the function $E(g)$ has the graph $\{(0, 1), (1, 8), (2, 64), (3, 6), (4, 9), (5, 1), (8, 0)\}$. (The function $g$ cannot be an approximation of $\exp$, because $g$ and $E(g)$ disagree on 0, 2, and 4.)

**Task 1.1** (3%). Write the functional $E$ in PCF (as described in class and in section 2.1) *without* using `fix`. $E$ should have type $(\texttt{nat} \rightharpoonup \texttt{nat}) \rightharpoonup (\texttt{nat} \rightharpoonup \texttt{nat})$ and should have the form

$$E = \lambda\,(f : \texttt{nat} \rightharpoonup \texttt{nat})\ \lambda\,(n : \texttt{nat}) \ldots$$

You can assume that you have expressions `div`, `mod`, and `times`, each with type $\texttt{nat} \rightharpoonup \texttt{nat} \rightharpoonup \texttt{nat}$ and the expected properties. □

**Task 1.2** (3%). Write down the graphs for $E(E(\perp))$ and $E(E(E(\perp)))$. □

According to Kleene's theorem, the *least fixed point* of the function E is given by the limit

$$\lim_{i \geq 0} E^{(i)}(\perp)$$

Now, what we wish to prove is that this least fixed point for E represents a total function.

**Task 1.3** (9%). Let $\phi_i$ be the approximation $E^{(i)}(\perp)$ which is $E$ applied $i$ times to $\perp$. Show that, for each natural number $m$, there exists $i$ such that $\phi_i(m)$ is defined. You may use strong induction and may cite the following fact without proof: If $\phi_i(m)$ is defined, then $\phi_j(m)$ is defined for all $j \geq i$. □

# 2 PCF with Nullary and Binary Products and Sums

For this section of the assignment, we will be working on PCF extended with natural numbers, nullary and binary product and sum types and a primitive let operator for convenience.

## 2.1 PCF

The language $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$, also known as Plotkin's PCF, integrates functions and natural numbers using *general recursion*, a means of defining self-referential expressions. In contrast to those in $\mathcal{L}\{\texttt{nat} \to\}$, evaluation of expressions in $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ may not terminate. The statics and dynamics of PCF (with the addition of natural numbers and a primitive `let` operator) are given in Appendix B.1 and C.1, respectively.

The crucial characteristic embodied in PCF is the fixed point characterization of recursive definitions, which we examined above. That is, we can obtain the least fixed point of any functional definable in the language. Using this, we may solve any set of recursion equations we like, with the proviso that the solution may not be *total*. You can read more about PCF in Chapter 10 of *PFPL*.

| | | | | |
|---|---|---|---|---|
| Type | $\tau$ | $::=$ | $\texttt{nat}$ | $\texttt{nat}$ |
| | | | $\texttt{parr}(\tau_1;\tau_2)$ | $\tau_1 \rightharpoonup \tau_2$ |
| Exp | $e$ | $::=$ | $\texttt{z}$ | $\texttt{z}$ |
| | | | $\texttt{s}(e)$ | $\texttt{s}(e)$ |
| | | | $\texttt{ifz}(e;e_0;x.e_1)$ | $\texttt{ifz}\, e\, \{z \Rightarrow e_0 \mid \texttt{s}(x) \Rightarrow e_1\}$ |
| | | | $\texttt{lam}[\tau](x.e)$ | $\texttt{fn}\,(x:\tau)\,e$ |
| | | | $\texttt{let}(e_1;x.e_2)$ | $\texttt{let}\, x = e_1\, \texttt{in}\, e_2$ |
| | | | $\texttt{fix}[\tau](x.e)$ | $\texttt{fix}\, x:\tau\, \texttt{is}\, e$ |

## 2.2 Product Types

The *binary product* of two types $\tau_1 \times \tau_2$ consists of an ordered pair of values, one from each type. The *explicit eliminatory forms* for this type are called *projections*. They select the first or the second component of the pair, respectively. One can also consider the type of the nullary product `unit` which has no eliminatory form and contains only one value, $\langle\rangle$.

| | | | | |
|---|---|---|---|---|
| Type | $\tau$ | $::=$ | $\texttt{unit}$ | $\texttt{unit}$ |
| | | | $\texttt{prod}(\tau_1;\tau_2)$ | $\tau_1 \times \tau_2$ |
| Exp | $e$ | $::=$ | $\texttt{triv}$ | $\langle\rangle$ |
| | | | $\texttt{pair}(e_1;e_2)$ | $\langle e_1, e_2\rangle$ |
| | | | $\texttt{pr}[\texttt{l}](e)$ | $e \cdot \texttt{l}$ |
| | | | $\texttt{pr}[\texttt{r}](e)$ | $e \cdot \texttt{r}$ |

The statics and dynamics of nullary and binary products are given in Appendix B.2 and C.2, respectively.

## 2.3 Sum Types

Most data structures involve alternatives such the distinction between a leaf and an interior node in a tree. `datatype` declarations in ML can be thought of as a labeled sum of different types. Here we will consider the more primitive notion of nullary and binary sums, from which all finite sums can be built.

The *binary sum* type, $\tau_1 + \tau_2$ represents a choice between two types, where the selection is made by which of the two introductory forms is used. The explicit eliminatory form of the binary sum type is by a case analysis over these possibilities.

The *nullary sum* type, `void`, can be considered to be a sum with no elements. That is, there is no introductory form. The eliminatory form, `abort`, aborts the computation in the event that a value of type `void` is produced (which cannot happen). Sometimes, `void` is written as `0` for this reason. Note that `void` should not be confused with `unit` (often written `1`), although many languages (deriving from C) do so.

| | | | | |
|---|---|---|---|---|
| Type | $\tau$ | ::= | $\texttt{void}$ | $\texttt{void}$ |
| | | | $\texttt{sum}(\tau_1; \tau_2)$ | $\tau_1 + \tau_2$ |
| Exp | $e$ | ::= | $\texttt{abort}[\tau](e)$ | $\texttt{abort}[\tau](e)$ |
| | | | $\texttt{in}[\tau_1; \tau_2][\texttt{l}](e)$ | $\texttt{inl}[\tau_1; \tau_2]\ e$ |
| | | | $\texttt{in}[\tau_1; \tau_2][\texttt{r}](e)$ | $\texttt{inr}[\tau_1; \tau_2]\ e$ |
| | | | $\texttt{case}(e; x_1.e_1; x_2.e_2)$ | $\texttt{case}\ e\ \{\texttt{inl}\ x_1 \Rightarrow e_1 \mid \texttt{inr}\ x_2 \Rightarrow e_2\}$ |

The statics and dynamics of binary sums are given in Appendix B.3 and C.3, respectively. Note that our concrete syntax for expressions here has deviated from the syntax in the book. The changes make the concrete syntax more verbose, but if we left out type annotations it would be much trickier to do typechecking on expressions.

### 2.4 Theory

**Task 2.1** (5%). State the canonical forms lemma for the language described in Sections 2.1-2.3 above. Please state it completely and precisely in terms of the judgments $e\ \mathsf{val}$ and $\Gamma \vdash e : \tau$. □

**Task 2.2** (15%). Prove the progress theorem for this language: if $\emptyset \vdash e : \tau$ then either $e\ \mathsf{val}$ or there exists an $e'$ such that $e \mapsto e'$. You may cite the canonical forms lemma you stated above and any appropriate inversion lemmas without proof.

As a simplification, you may consider the sublanguage without functions (that is, without $\rightharpoonup$-I, $\rightharpoonup$-E, $\rightharpoonup_{\mathsf{v}}$, $\mathrm{ap}^1_{\mathsf{s}}$, $\mathrm{ap}^2_{\mathsf{s}}$, and $\mathrm{ap}_{\mathsf{e}}$). □

### 2.5 Implementation

We will now implement this language, using the ABT infrastructure that you developed in the previous assignment. We have included our reference ABT implementation in the code for this assignment, and you should use that.

**Task 2.3** (15%). Implement the statics and dynamics of this language by filling in the unimplemented functions in the `sec2` directory. You will need to implement parts of the following structures:

- `TermOps`
- `TypeOps`
- `TypeChecker`
- `UncheckedDynamics` – the `Abort` exception should be raised iff a value of type `void` is encountered in the dynamics. This will never happen in a correct implementation. □

## 3 Pattern Matching

Nested pattern matching is a natural and convenient generalization of the elimination forms for most finite datatypes. For example, you may want to write a function that takes a pair containing a Boolean and a nat and doubles the nat if the Boolean is true or squares it if the Boolean is false. We could write it like this:

$$
\begin{array}{l}
\texttt{fn}\,(n : (\texttt{unit} + \texttt{unit}) \times \texttt{nat}) \\
\quad \texttt{match}\, n\, \{ \\
\qquad \langle \texttt{inl}\langle\rangle, x\rangle \Rightarrow \text{plus}(x)(x) \\
\qquad \mid \langle \texttt{inr}\langle\rangle, x\rangle \Rightarrow \text{times}(x)(x) \\
\quad \}
\end{array}
$$

Notice that the `match` operator introduced above[1] is matching against the full nested structure of the argument, $n$, not just its outermost form. Each pattern also binds variables within the scope of a rule (i.e. $x$, above), to stand for any expression that appears in that position.

In this section, we will implement the language of patterns, based on the language described in Chapter 13 of PFPL. Our language is based on PCF with sums and products as described above, but removes the explicit elimination forms (bracketed in Appendix B and C) in favor of the more general `match` operator:

| | | | | |
|---|---|---|---|---|
| Exp | $e$ | ::= | $\texttt{match}[p_1 \ldots p_n](e; \vec{x_1}.e_1; \ldots; \vec{x_n}.e_n)$ | $\texttt{match}\, e\, \{p_1 \Rightarrow e_1 \mid \ldots \mid p_n \Rightarrow e_n\}$ |
| Pattern | $p$ | ::= | $\texttt{wild}$ | $\_$ |
| | | | $x$ | $x$ |
| | | | $\texttt{z}$ | $\texttt{z}$ |
| | | | $\texttt{s}(p)$ | $\texttt{s}(p)$ |
| | | | $\texttt{triv}$ | $\langle\rangle$ |
| | | | $\texttt{pair}(p_1; p_2)$ | $\langle p_1, p_2\rangle$ |
| | | | $\texttt{in}[\texttt{l}](p)$ | $\texttt{inl}\ p$ |
| | | | $\texttt{in}[\texttt{r}](p)$ | $\texttt{inr}\ p$ |

Here, each pairing $(p_i, \vec{x_i}.e_i)$ corresponds to the rule $p_i \Rightarrow e_i$ in the concrete syntax. This ABT syntax differs from the presentation of in *PFPL*, because our ABT infrastructure does not support different sorts of expressions in arguments to operators. Therefore, patterns have been separated out as parameters of the `match` operator.

### 3.1 Statics

The typing rule for the match expression is as follows:

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash rs : \tau \rightsquigarrow \tau'}{\Gamma \vdash \texttt{match}\, e\, \{rs\} : \tau'}(\text{match})$$

Here, $rs$ is a sequence of one or more rules of the form $p \Rightarrow e$. As one would expect, we first synthesize a type for $e$, which can be of any type, $\tau$. We must then check that the provided rules can match values of type $\tau$, and that each branch produces an expression of the same type, $\tau'$, so that the `match` expression can be assigned a single type. That is the purpose of the new judgment form $\Gamma \vdash rs : \tau \rightsquigarrow \tau'$. It can be defined as follows:

$$\frac{\Gamma \vdash p_1 \Rightarrow e_1 : \tau \rightsquigarrow \tau' \qquad \cdots \qquad \Gamma \vdash p_n \Rightarrow e_n : \tau \rightsquigarrow \tau'}{\Gamma \vdash p_1 \Rightarrow e_1 \mid \ldots \mid p_n \Rightarrow e_n : \tau \rightsquigarrow \tau'}(\text{rules})$$

[1] This is similar to the `case` operator in SML, but should not to be confused with the simpler `case` operator that serves as the elimination form for sums in Section 2 and can be written as a special case of `match`.

As you might expect, to check a sequence of rules, you must check each rule. The judgment to check a rule is read "the rule $p \Rightarrow e$ transforms type $\tau$ to $\tau'$" and is inductively defined as follows:

$$\frac{\Lambda \Vdash p : \tau \qquad \Gamma\ \Lambda \vdash e : \tau'}{\Gamma \vdash p \Rightarrow e : \tau \rightsquigarrow \tau'}(\text{rule})$$

In other words, to check a rule, you must first check that the pattern can match values of type $\tau$, and extract the variables bound by the pattern, as well as their types. That is the purpose of the judgment $\Lambda \Vdash p : \tau$. Then, these variables must be added to the context to synthesize a type for the branch as a whole, which is what the second premise expresses.

In the judgment $\Lambda \Vdash p : \tau$, there is a context $\Lambda$ for variables present in the pattern. This judgment is similar to the $\Gamma \vdash e : \tau$ judgment except that that we cannot weaken $\Lambda$ by adding arbitrary variables not present in the pattern to it. In other words, each variable in the context must be present *exactly once* in the pattern (and *vice versa*). So, each well-formed pattern matching values of a certain type has a unique $\Lambda$ associated with it. This means that this judgment can be thought of as a function that takes $p$ and $\tau$ as input and outputs $\Lambda$ if it exists. Note that a variable may appear only once in a pattern. This judgment is inductively defined as follows:

$$\frac{}{x{:}\tau \Vdash x : \tau}(\text{pat-var}) \qquad \frac{}{\emptyset \Vdash \_ : \tau}(\text{pat-}\_) \qquad \frac{}{\emptyset \Vdash \langle\rangle : \texttt{unit}}(\text{pat-}\langle\rangle) \qquad \frac{}{\emptyset \Vdash \texttt{z} : \texttt{nat}}(\text{pat-z})$$

$$\frac{\Lambda \Vdash p : \texttt{nat}}{\Lambda \Vdash \texttt{s}(p) : \texttt{nat}}(\text{pat-s}) \qquad \frac{\Lambda_1 \Vdash p : \tau_1}{\Lambda_1 \Vdash \texttt{inl}(p) : \tau_1 + \tau_2}(\text{pat-inl}) \qquad \frac{\Lambda_2 \Vdash p : \tau_2}{\Lambda_2 \Vdash \texttt{inr}(p) : \tau_1 + \tau_2}(\text{pat-inr})$$

$$\frac{\Lambda_1 \Vdash p_1 : \tau_1 \qquad \Lambda_2 \Vdash p_2 : \tau_2 \qquad dom(\Lambda_1) \cap dom(\Lambda_2) = \emptyset}{\Lambda_1\ \Lambda_2 \Vdash \langle p_1, p_2\rangle : \tau_1 \times \tau_2}(\text{pat-}\times)$$

### 3.2 Dynamics

The dynamics for `match` are defined in terms of a substitution $\theta$ from variables to values. We define $\theta$ as a *substitution* of the form $\{x \hookrightarrow e_1, \ldots, x_k \hookrightarrow e_k\}$, and we define the simultaneous substitution of $\theta$ in e as $\hat{\theta}(e) = [e_1, \ldots, e_k/x_1, \ldots x_k]e$. The judgment $\theta : \Lambda$ states that "the substitution $\theta$ is well-typed and agrees with type context $\Lambda$". It is inductively defined as follows.

$$\frac{}{\emptyset : \emptyset}(\theta\Lambda_1) \qquad \frac{\theta : \Lambda \qquad e : \tau}{(\theta \otimes x \hookrightarrow e) : (\Lambda, x : \tau)}(\theta\Lambda_2)$$

Given a pattern $p$ and a value $e$, we need to find a $\theta$ which is the correct substitution to apply when $p$ matches $e$. This is given by the judgment $\theta \Vdash p \lhd e$ and is inductively defined as follows.

$$\frac{}{x \hookrightarrow e \Vdash x \lhd e}(\text{sub-var}) \qquad \frac{}{\emptyset \Vdash \_ \lhd e}(\text{sub-wild}) \qquad \frac{}{\emptyset \Vdash \langle\rangle \lhd \langle\rangle}(\text{sub}\langle\rangle) \qquad \frac{}{\emptyset \Vdash \texttt{z} \lhd \texttt{z}}(\text{sub-z})$$

$$\frac{\theta \Vdash p \lhd e}{\theta \Vdash \texttt{s}(p) \lhd \texttt{s}(e)}(\text{sub-s}) \qquad \frac{\theta_1 \Vdash p_1 \lhd e_1 \qquad \theta_2 \Vdash p_2 \lhd e_2 \qquad dom(\theta_1) \cap dom(\theta_2) = \emptyset}{\theta_1 \otimes \theta_2 \Vdash \langle p_1, p_2\rangle \lhd \langle e_1, e_2\rangle}(\text{sub}\times)$$

$$\frac{\theta \Vdash p \lhd e}{\theta \Vdash \texttt{inl}\ p \lhd \texttt{inl}\ e}(\text{sub}+_1) \qquad \frac{\theta \Vdash p \lhd e}{\theta \Vdash \texttt{inr}\ p \lhd \texttt{inr}\ e}(\text{sub}+_2)$$

We also need to define inductively the conditions for a match failure. This is defined by the judgment, $e \perp p$.

$$\frac{e_1 \perp p_1}{\langle e_1, e_2\rangle \perp \langle p_1, p_2\rangle}(\text{mfail}\times_1) \qquad \frac{e_2 \perp p_2}{\langle e_1, e_2\rangle \perp \langle p_1, p_2\rangle}(\text{mfail}\times_2) \qquad \frac{}{\mathtt{s}(e) \perp \mathtt{z}}(\text{mfail-}\mathtt{nat}_1)$$

$$\frac{}{\mathtt{z} \perp \mathtt{s}(p)}(\text{mfail-}\mathtt{nat}_2) \qquad \frac{e \perp p}{\mathtt{s}(e) \perp \mathtt{s}(p)}(\text{mfail-}\mathtt{nat}_3) \qquad \frac{}{\mathtt{inl}\ e \perp \mathtt{inr}\ p}(\text{mfail}+_1)$$

$$\frac{e \perp p}{\mathtt{inl}\ e \perp \mathtt{inl}\ p}(\text{mfail}+_2) \qquad \frac{}{\mathtt{inr}\ e \perp \mathtt{inl}\ p}(\text{mfail}+_3) \qquad \frac{e \perp p}{\mathtt{inr}\ e \perp \mathtt{inr}\ p}(\text{mfail}+_4)$$

Finally, we can define the stepping rules for when an expression is a pattern. We first evaluate the argument to the match first. We then apply the first rule which matches our value. The match results in a checked runtime error when there are no rules left. This is the only form of runtime error possible in this assignment; the dynamic semantics needs rules like the ones in Homework 1 that pass on errors that occur in subterms; we have omitted these for brevity.

$$\frac{e \mapsto e'}{\mathtt{match}\ e\ \{rs\} \mapsto \mathtt{match}\ e'\ \{rs\}}(\mathtt{match}_\mathsf{s}) \qquad \frac{e\ \mathsf{val}}{\mathtt{match}\ e\ \{\}\ \mathsf{err}}(\mathtt{match}_\mathsf{e})$$

$$\frac{e\ \mathsf{val} \qquad \theta \Vdash p_0 \lhd e}{\mathtt{match}\ e\ \{p_0 \Rightarrow e_0 \mid rs\} \mapsto \hat{\theta}(e_0)}(\mathtt{match}^1_\mathsf{e}) \qquad \frac{e\ \mathsf{val} \qquad e \perp p_0 \qquad \mathtt{match}\ e\ \{rs\} \mapsto e'}{\mathtt{match}\ e\ \{p_0 \Rightarrow e_0 \mid rs\} \mapsto e'}(\mathtt{match}^2_\mathsf{e})$$

### 3.3 Theory

Now that we have defined our language, we are ready to prove some theorems about it. The first important result is the completeness of pattern matching rules. This simply states that either a value $e$ matches a pattern $p$ of the same type or the match fails by our failure rules.

**Task 3.1** (15%). Prove the following theorem:

**Theorem 1.** *Suppose that $e : \tau$, $e$ $\mathsf{val}$ and $\Lambda \Vdash p : \tau$. Then either there exists $\theta$ such that $\theta : \Lambda$ and $\theta \Vdash p \lhd e$ or $e \perp p$.*

You do not need to show the cases corresponding to sums types and patterns. You may assume the following structural properties about contexts and substitutions without proof:

1. If $\theta : \Lambda$, then $dom(\theta) = dom(\Lambda)$
2. If $\theta_1 : \Lambda_1$ and $\theta_2 : \Lambda_2$ then $\theta_1 \otimes \theta_2 : \Lambda_1\ \Lambda_2$.

You may also cite appropriate inversion lemmas for the rules above as needed without proof. □

Now that we have added a lot of machinery to the language, let's make sure preservation still holds.

**Task 3.2** (15%). Prove the preservation theorem for this language: if $\emptyset \vdash e : \tau$ and $e \mapsto e'$ then $\emptyset \vdash e' : \tau$. For this question, you only need to deal with the three rules in the dynamic semantics above that involve patterns. You can use the lemmas below without proof:

**Lemma 1.** *Suppose that* $e : \tau$, $e$ val, $\Lambda \Vdash p : \tau$ *and* $\theta \Vdash p \lhd e$, *then* $\theta : \Lambda$.

**Lemma 2.** *(Substitution) If* $\theta : \Lambda$ *and* $\Gamma\ \Lambda \vdash e : \tau$, *then* $\Gamma \vdash \hat{\theta}(e) : \tau$

You may also cite appropriate inversion lemmas for the rules above as needed without proof. □

### 3.4 Implementation

**Task 3.3** (20%). Implement the statics and dynamics of this language by filling in the unimplemented functions in the `sec3` directory. You will need to implement parts of the following structures:

- `TermOps` – Be careful: the arity of a match is determined in part by its patterns!
- `TypeOps`
- `Pattern` – note that patterns do *not* use the ABT infrastructure, because variables in patterns behave differently from variables in terms – they must be used only once, and they are not given meaning (within the pattern) by substitution! As such, the pattern that we write as $\langle\langle x, \_\rangle, \texttt{inl}\ y\rangle$ will be represented as the pattern $\texttt{pair}(\texttt{pair}(\texttt{var}, \texttt{wild}), \texttt{in}[\texttt{l}](\texttt{var}))$. The abstractor $\vec{x_i}.e_i$ associated with this pattern must then have a valence of 2. That is, it should bind the two variables that appeared in the pattern: $x, y.e_i$.

  If we typecheck the pattern above against a type like $((\texttt{unit} + \texttt{unit}) \times \texttt{unit}) \times (\texttt{nat} + \texttt{void})$ it will produce the list of types $[(\texttt{unit} + \texttt{unit}), \texttt{nat}]$. Your implementation of pattern matching can similarly produce a list of values $\vec{v}$ if successful, which can then be substituted for the list of variables $\vec{x_i}$ in $e_i$.
- `ABT_Util` – you'll want to implement and use the new function `separatebinders`.
- `TypeChecker`
- `InexhaustiveDynamics` – Remember that the error propagation rules for our language have been omitted for brevity, but you must propagate match failure errors correctly in your code. □

# A Putting The Code Together

You can compile your files using `CM.make "sources.cm"`. We have provided three (!) ways to test the final implementation: an interpreter, a test harness and a reference implementation. All of these are based on a parser we provide for you (see examples below).

## A.1 Interpreter

As usual, to run the interpreter, execute `TopLevel.repl();`. This will provide a command-line interpreter that will provide two basic commands, `step` and `eval`. These commands do not take a mode argument any more as we have only one kind of dynamics.

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 a sample session of the interpreter.

```
command ::=
  step <exp>;
| step;
| eval <exp>;
| eval;

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

exp ::=
  z
| s <exp>
| ifz <exp> { z => <exp> | s <ident> => <exp> }
| fn (<ident> : <ty>) <exp>
| let <decls> in <exp> end (* whitespace is required *)
| fix <ident> : <ty> is <exp>
| <>
| <<exp>, <exp>>
| <exp>.l
| <exp>.r
| abort[<ty>] <exp>
| inl[<ty>,<ty>] <exp>
| inr[<ty>,<ty>] <exp>
| case e { inl <ident> => exp | inr <ident> => exp }
| match e { <rules> }
| (<exp>)

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

decl ::= val <ident> = <exp>

rules ::=
  <pat> => <exp>
| <pat> => <exp> | <rules>
```

```
pat ::=
  _
| <ident>
| z
| s <pat>
| <>
| <<pat>, <pat>>
| inl <pat>
| inr <pat>
| (<pat>)

ident ::= (* a letter followed by any number of letters and numbers *)

Example interpreter session:

- TopLevel.repl();
->eval (fix sum : nat->nat->nat is fn(x:nat) fn(y:nat)
        ifz x {z => y | s u => s(sum u y)})
       (s z) (s (s z));
Statics : exp has type : nat
 s(s(s(z))) VAL

->eval case inl[nat,void] z { inl x => <s x, x> | inr p => abort[nat*nat] p };
Statics : exp has type : prod(nat, nat)
 pair(s(z), z) VAL

->eval <z, s z>.l;
Statics : exp has type : nat
 z VAL

->eval inr[nat,unit] ((fn (x:unit) x) <>);
Statics : exp has type : sum(nat, unit)
 in[nat; unit][r](triv) VAL
```

## A.2 Test Harness

Another way to test your code is by `TestHarness.runalltests(v);` where `v` is a `bool` indicating whether you want verbose output or not. This is mostly just a framework set up for you, in `tests.sml`, with a few simple test cases. You are responsible for handing in a working solution. Although not sufficient, this means handing in a well-tested implementation. You need to come up with test cases to exercise your code. In order to generate a comprehensive suite of tests, you are encouraged to share test cases with your classmates.

## A.3 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 (if you find a bug in our implementation, extra credit to you!)

# B Statics

Rules for explicit eliminatory forms are bracketed as explained in Section 3.

## B.1 PCF

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}(\text{var}) \qquad \frac{}{\Gamma \vdash \mathtt{z} : \mathtt{nat}}(\mathtt{nat}\text{-}\mathrm{I}_1) \qquad \frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash \mathtt{s}(e) : \mathtt{nat}}(\mathtt{nat}\text{-}\mathrm{I}_2)$$

$$\left[\frac{\Gamma \vdash e : \mathtt{nat} \qquad \Gamma \vdash e_0 : \tau \qquad \Gamma, x : \mathtt{nat} \vdash e_1 : \tau}{\Gamma \vdash \mathtt{ifz}(e; e_0; x.e_1) : \tau}(\mathtt{nat}\text{-E})\right] \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau}{\Gamma \vdash \mathtt{let}(e_1; x.e_2) : \tau}(\text{let})$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathtt{lam}[\tau_1](x.e) : \tau_1 \rightharpoonup \tau_2}(\rightharpoonup\text{-I}) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \rightharpoonup \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash \mathtt{ap}(e_1, e_2) : \tau_2}(\rightharpoonup\text{-E}) \qquad \frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \mathtt{fix}[\tau](x.e) : \tau}(\text{fix})$$

## B.2 Products

$$\frac{}{\Gamma \vdash \langle\rangle : \mathtt{unit}}(\mathtt{unit}\text{-I}) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2\rangle : \tau_1 \times \tau_2}(\times\text{-I}) \qquad \left[\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathtt{pr}[\mathtt{l}](e) : \tau_1}(\times\text{-E}_1)\right] \qquad \left[\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathtt{pr}[\mathtt{r}](e) : \tau_2}(\times\text{-E}_2)\right]$$

## B.3 Sums

$$\frac{\Gamma \vdash e : \mathtt{void}}{\Gamma \vdash \mathtt{abort}[\tau](e) : \tau}(\mathtt{void}\text{-E}) \qquad \frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathtt{in}[\tau_1; \tau_2][\mathtt{l}](e) : \tau_1 + \tau_2}(+\text{-I}_1) \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathtt{in}[\tau_1; \tau_2][\mathtt{r}](e) : \tau_1 + \tau_2}(+\text{-I}_2)$$

$$\left[\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad \Gamma, x_1{:}\tau_1 \vdash e_1 : \tau \qquad \Gamma, x_2{:}\tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathtt{case}(e; x_1.e_1; x_2.e_2) : \tau}(+\text{-E})\right]$$

# C Dynamics (Eager, Left-to-Right)

Rules for explicit eliminatory forms are bracketed as explained in Section 3.

## C.1 PCF with Natural Numbers and Let

$$\frac{}{\mathtt{z}\ \mathsf{val}}(\mathrm{nat}^1_{\mathsf{v}}) \qquad \frac{e\ \mathsf{val}}{\mathtt{s}(e)\ \mathsf{val}}(\mathrm{nat}^2_{\mathsf{v}}) \qquad \frac{}{\mathtt{lam}[\tau](x.e)\ \mathsf{val}}(\rightharpoonup_{\mathsf{v}}) \qquad \frac{e \mapsto e'}{\mathtt{s}(e) \mapsto \mathtt{s}(e')}(\mathrm{s}_{\mathsf{s}})$$

$$\left[\frac{e \mapsto e'}{\mathtt{ifz}(e; e_0; x.e_1) \mapsto \mathtt{ifz}(e'; e_0; x.e_1)}(\mathrm{ifz}_{\mathsf{s}})\right] \quad \left[\frac{}{\mathtt{ifz}(\mathtt{z}; e_0; x.e_1) \mapsto e_0}(\mathrm{ifz}^1_{\mathsf{e}})\right] \quad \left[\frac{\mathtt{s}(e)\ \mathsf{val}}{\mathtt{ifz}(\mathtt{s}(e); e_0; x.e_1) \mapsto [e/x]e_1}(\mathrm{ifz}^2_{\mathsf{e}})\right]$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{ap}(e_1; e_2) \mapsto \mathtt{ap}(e_1'; e_2)}(\mathrm{ap}^1_{\mathsf{s}}) \qquad \frac{e_1\ \mathsf{val} \qquad e_2 \mapsto e_2'}{\mathtt{ap}(e_1; e_2) \mapsto \mathtt{ap}(e_1; e_2')}(\mathrm{ap}^2_{\mathsf{s}}) \qquad \frac{e_2\ \mathsf{val}}{\mathtt{ap}(\mathtt{lam}[\tau](x.e); e_2) \mapsto [e_2/x]e}(\mathrm{ap}_{\mathsf{e}})$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{let}(e_1; x.e_2) \mapsto \mathtt{let}(e_1'; x.e_2)}(\mathrm{let}_{\mathsf{s}}) \quad \frac{e_1\ \mathsf{val}}{\mathtt{let}(e_1; x.e_2) \mapsto [e_1/x]e_2}(\mathrm{let}_{\mathsf{e}}) \quad \frac{}{\mathtt{fix}[\tau](x.e) \mapsto [\mathtt{fix}[\tau](x.e)/x]e}(\mathrm{fix}_{\mathsf{s}})$$

## C.2 Products

$$\frac{}{\langle\rangle\ \mathsf{val}}(\mathtt{unit}_{\mathsf{v}}) \qquad \frac{e_1\ \mathsf{val} \qquad e_2\ \mathsf{val}}{\langle e_1, e_2\rangle\ \mathsf{val}}(\times_{\mathsf{v}}) \qquad \frac{e_1 \mapsto e_1'}{\langle e_1, e_2\rangle \mapsto \langle e_1', e_2\rangle}(\times^1_{\mathsf{s}}) \qquad \frac{e_1\ \mathsf{val} \qquad e_2 \mapsto e_2'}{\langle e_1, e_2\rangle \mapsto \langle e_1, e_2'\rangle}(\times^2_{\mathsf{s}})$$

$$\left[\frac{e \mapsto e'}{\mathtt{pr}[\mathtt{l}](e) \mapsto \mathtt{pr}[\mathtt{l}](e')}(\mathrm{prl}_{\mathsf{s}})\right] \quad \left[\frac{e \mapsto e'}{\mathtt{pr}[\mathtt{r}](e) \mapsto \mathtt{pr}[\mathtt{r}](e')}(\mathrm{prr}_{\mathsf{s}})\right] \quad \left[\frac{e_1\ \mathsf{val} \qquad e_2\ \mathsf{val}}{\mathtt{pr}[\mathtt{l}](\langle e_1, e_2\rangle) \mapsto e_1}(\mathrm{prl}_{\mathsf{e}})\right] \quad \left[\frac{e_1\ \mathsf{val} \qquad e_2\ \mathsf{val}}{\mathtt{pr}[\mathtt{r}](\langle e_1, e_2\rangle) \mapsto e_2}(\mathrm{prr}_{\mathsf{e}})\right]$$

## C.3 Sums

$$\frac{e \mapsto e'}{\mathtt{abort}[\tau](e) \mapsto \mathtt{abort}[\tau](e')}(\mathrm{abort}_{\mathsf{s}}) \qquad \frac{e\ \mathsf{val}}{\mathtt{in}[\tau_1; \tau_2][\mathtt{l}](e)\ \mathsf{val}}(+^1_{\mathsf{v}}) \qquad \frac{e\ \mathsf{val}}{\mathtt{in}[\tau_1; \tau_2][\mathtt{r}](e)\ \mathsf{val}}(+^2_{\mathsf{v}})$$

$$\frac{e \mapsto e'}{\mathtt{in}[\tau_1; \tau_2][\mathtt{l}](e) \mapsto \mathtt{in}[\tau_1; \tau_2][\mathtt{l}](e')}(+^1_{\mathsf{s}}) \qquad \frac{e \mapsto e'}{\mathtt{in}[\tau_1; \tau_2][\mathtt{r}](e) \mapsto \mathtt{in}[\tau_1; \tau_2][\mathtt{r}](e')}(+^2_{\mathsf{s}})$$

$$\left[\frac{e \mapsto e'}{\mathtt{case}(e; x_1.e_1; x_2.e_2) \mapsto \mathtt{case}(e'; x_1.e_1; x_2.e_2)}(\mathrm{case}_{\mathsf{s}})\right]$$

$$\left[\frac{e\ \mathsf{val}}{\mathtt{case}(\mathtt{in}[\tau_1; \tau_2][\mathtt{l}](e); x_1.e_1; x_2.e_2) \mapsto [e/x_1]e_1}(\mathrm{case}^1_{\mathsf{e}})\right]$$

$$\left[\frac{e\ \mathsf{val}}{\mathtt{case}(\mathtt{in}[\tau_1; \tau_2][\mathtt{r}](e); x_1.e_1; x_2.e_2) \mapsto [e/x_2]e_2}(\mathrm{case}^2_{\mathsf{e}})\right]$$