OPLSS PL Background Day 3

Robert Harper and Dan Licata

1 Recursive Types

Task 1. Define a recursive type $\mathsf{rec}(t.\tau)$ representing binary trees, as in the following ML datatype:

```
datatype tree =
    Empty
  | Leaf of int
  | Node of tree * tree
```

Define the Empty and Leaf and Node constructors, and define a rec construct for structural recursion over trees (we constructed general recursion $\mathsf{fix}(x.e)$ in class, so you can use this in your definition). I.e. your rec should allow for programming functions that have the form

```
f Empty = e0
f (Leaf x) = e1
f (Node(l,r)) = ... code mentioning l and r and f(l) and f(r) ...
```

Task 2. Church's untyped λ-calculus is a dynamically typed language with only variables, functions, and application

$$d ::= x \mid \lambda x.d \mid d_1(d_2)$$

with the usual operational semantics (here we do call-by-value)

$$\frac{d_2 \;\mathsf{value}}{(\lambda x.d)\, d_2 \mapsto [d_2/x]d} \qquad \frac{d_1 \mapsto d_1'}{d_1\, d_2 \mapsto d_1'\, d_2} \qquad \frac{d_1 \mathsf{value} \quad d_2 \mapsto d_2'}{d_1\, d_2 \mapsto d_1\, d_2'}$$

We represent this with a recursive type $D := \mathsf{rec}(t.t \to t)$ that represents the values of the untyped λ-calculus, as discussed in lecture.

1. Translate every untyped λ-calculus program

$$x_1, \ldots, x_n \vdash d$$

   to a program

$$x_1 : D, \ldots, x_n : D \vdash d^\dagger : D$$

2. Prove that if $d \mapsto d'$ then $d^\dagger \equiv d'^\dagger$, where definitional equality ($\equiv$) is the open extension of evaluation:

   - if $e \mapsto e'$ then $e \equiv e'$.

- it is an equivalence relation (reflexive, symmetric, and transitive).
- it is a congruence

The upshot is that with definitional equality you can "step anywhere in a program" (including places where the dynamic semantics does not let you take a step, like in the body of a function or a case), and take as many steps as you want, either forwards or backwards. Definitional equality is sound (but not complete) for contextual equivalence for this language.

2 Programming and Reasoning in System F

See PFPL Chapters 16, 17, 48 for the definition of System F and logical/contextual equivalence for it.

Weak Definability All of the following type constructors can be defined in System F with only $\rightarrow, \forall$ by "Church encodings":

- $\tau_1 \times \tau_2$ with pairing and projection (see PFPL 10.1 for the rules).
- $\tau_1 + \tau_2$, with `inl` and `inr` and `case` (see PFPL 11.1 for the rules).
- int, with 0, $\texttt{suc}(-)$, and iteration (see the Day One handout).
- $\texttt{list}(\tau)$, with `[]` and `::` and recursion
- $\exists$, with `pack` and `open`

Task 1. Give definitions of the types and their operations. All of the encodings follow a general pattern; try to articulate it.

Task 2. For each type, show that its β steps (elim after intro, e.g. $\texttt{first}\langle e_1, e_2\rangle \mapsto e_1$) are simulated by one or more β steps in System F.

Parametricity However, it is not possible to prove the uniqueness principles for these types using only $\beta\eta$-equality in System F; that requires parametricity. We write $e \sim_\tau e'$ to mean that e and e' are logically related at τ, which is the lifting of the the relations chosen for any type variables that are free in τ. Formally, this is a relation $e \simeq_\tau e'[\eta]$ where η maps each type variable that is free in τ to two types and a relation between them.

- $e \sim_t e'[\eta]$ iff $\eta(t)(e, e')$
- $e \sim_{\tau_1 \rightarrow \tau_2} e'[\eta]$ iff for all $e_1 : \tau_1$ and $e_1' : \tau_1$, $e_1 \sim_{\tau_1} e_1'[\eta]$ implies $(e\, e\_1) \sim_{\tau_2} (e'\, e_1')[\eta]$.
- $e \sim_{\forall t.\tau} e'[\eta]$ iff for all types σ and σ' and (admissible) relations R on σ and σ', $e[\sigma] \sim_\tau e'[\sigma'][\eta, t \hookrightarrow R]$.

Task 3. Using relational parametricity, show the following η/uniqueness principles for the System F Church encodings of nullary/binary products:

- For any $\cdot \vdash e : 1$, $e \sim \langle\rangle$.
- For any $\cdot \vdash e : \tau_1 \times \tau_2$, $e \sim \langle \texttt{first}(e), \texttt{second}(e) \rangle$.

  Here is one path to this that I (Dan) like—there many be more direct ways to prove it; if you find one, post it on Piazza!

  - Using parametricity, prove a naturality square for $\forall t.(\tau_1 \to \tau_2 \to t) \to t$: for any
  $$\begin{array}{l} p : \forall t.(\tau_1 \to \tau_2 \to t) \to t \\ g : \sigma_1 \to \sigma_2 \\ f : \tau_1 \to \tau_2 \to \sigma_1 \end{array}$$
  we have
  $$p[\sigma_2](\lambda x.\lambda y.g(fxy)) \sim_{\sigma_2} g(p[\sigma_1]f)$$
  - Using naturality, prove a weak positive η principle for pairs: for any
  $$p : \forall t.(\tau_1 \to \tau_2 \to t) \to t$$
  we have
  $$p[\tau_1 \times \tau_2](\texttt{pair}) \sim p$$
  - Again using naturality, prove the full *positive* η principle for pairs, which characterizes maps *out of* a pair type: for any
  $$\begin{array}{l} p : \forall t.(\tau_1 \to \tau_2 \to t) \to t \\ f : (\forall t.(\tau_1 \to \tau_2 \to t) \to t) \to \sigma \end{array}$$
  we have
  $$f(p) \sim p[\sigma](\lambda a.\lambda b.f(\texttt{pair}\, a\, b))$$
  - Use these to prove the negative η principle for pairs, which characterizes maps *into* / elements of a pair type:
  $$e \sim \langle \texttt{first}(e), \texttt{second}(e) \rangle$$
- For any $\cdot \vdash f : 0 \to \sigma$, $f \sim \lambda x : 0.\texttt{case}(x)[]$.
- For any $\cdot \vdash f : (\tau_1 + \tau_2) \to \sigma$,
  $f \sim \lambda x : \tau_1 + \tau_2.\texttt{case}\, x(\texttt{inl} \cdot y_1 \hookrightarrow f(\texttt{inl} \cdot (y_1)), \texttt{inr} \cdot y_2 \hookrightarrow f(\texttt{inr} \cdot (y_2)))$.

Task 4. State and prove analogous uniqueness properties for `nat`, `list`, $\exists$.

3 Representation Independence

In this problem, you will practice using parametricity to prove two module implementations are observationally equivalent.

The following signature describes queues of integers.

```
structure LQ : QUEUE =
struct
     type queue = int list

     val emp = []
     fun ins (n , l) = l @ [n]
     fun rem l =
        case l
         of [] => NONE
          | x::xs => SOME (x, xs)
end

structure LLQ : QUEUE =
struct
     type queue = (int list) * (int list)

     val emp = ([],[])
     fun ins (n, (front,back)) = (front,n :: back)

     fun rem  (front,back) =
        case (front,back)
         of ([],[]) => NONE
          | (x::xs,\_) => SOME (x, (xs,back))
          | ([],\_) => rem (rev back,[])
end
```

Figure 1: Implementation of Queues

```
signature QUEUE=
sig
   type queue
   val emp : queue
   val ins : int * queue -> queue
   val rem : queue -> (int * queue) option
end
```

In this signature,

- `emp` represents the empty queue.
- `ins` adds an element to the back of a queue.
- `rem` removes the element at the front of the queue and returns it with the remainder of the queue, or `NONE` if the queue is empty.

Taken together, these three values codify the familiar “first-in-first-out” behaviour of a queue.

See Figure 1 for two implementations of this signature.

LQ The first implementation represents a queue with a list where the first element of the list is the front of the queue.

New elements are inserted by being appended to the end of the list. Elements are removed by being pulled off the head of the list. If the list is empty, we know that the queue is empty, so the removal fails.

This implementation is slow in that insertion is always a linear time operation—we have to walk down the whole list each time we add a new element.

Note that we also could have chosen to have front of the queue be the last element of the list, but then removal would be linear time and we'd have the same problem—we can't escape the fact that one of these operations will be constant time and the other will be linear.

LLQ The second implementation represents a queue with a pair of lists. One list stores the front of the queue, while the other list stores the back of the queue in reverse order. The split between "front" and "back" here can be anywhere in the queue; it depends on the sequence of operations that have been applied to the queue.

New elements are inserted by being put at the head of the reversed back of the queue. Elements are removed in one of two ways:

1. If the front list is not empty, the front of the queue is its head, so we peel it off and return it.
2. If the front list is empty,
3. If the front list is empty, we reverse the reversed back list—now bringing it into order—make that the new front list, take an empty list as the back list, and try remove again on the pair of them.

If both the front and reversed back are empty, we know that the queue is empty, so the removal fails.

If we assume that reverse is linear time, this implementation needs to do a linear time operation on removal sometimes but not every time—in particular, the two-lists implementation has amortized constant time insert and remove, while the one-list implementation will always have at least one operation that's always linear time.

3.1 Observational Equivalence

For this problem, we assume we have products and sums and numbers and lists with the following notions of logical equivalence:

- $e \sim_{\texttt{int}} e'[\eta]$ iff $e \simeq e'$ (both reduce to the same value)
- $e \sim_{\texttt{list(int)}} e'$ iff $e \simeq e'$ (both reduce to the same value). Note that this holds only for lists of the base type `int`.
- $e \sim_{\texttt{unit}} e'[\eta]$ always
- $e \sim_{\tau_1 \times \tau_2} e'[\eta]$ iff $\texttt{first}(e) \sim_{\tau_1} \texttt{first}(e')[\eta]$ and $\texttt{second}(e) \sim_{\tau_2} \texttt{second}(e')[\eta]$
- $e \sim_{\tau_1 + \tau_2} e'[\eta]$ iff ($e \mapsto^* \texttt{inl}(e_1)$ and $e' \mapsto^* \texttt{inl}(e'_1)$ and $e_1 \sim_{\tau_1} e'_1[\eta]$)
  or ($e \mapsto^* \texttt{inr}(e_2)$ and $e' \mapsto^* \texttt{inr}(e'_2)$ and $e_2 \sim_{\tau_2} e'_2[\eta]$)

We regard the SML signature `QUEUE` as the existential type

$$\exists q.q \times (\texttt{int} \times q \to q) \times (q \to (\texttt{int} \times q)\,\texttt{option})$$

where $\tau\,\texttt{option}$ is the sum type $\tau + \langle\rangle$ (with $\texttt{SOME}(e) := \texttt{inl}(e)$ and $\texttt{NONE} := \texttt{inr}\langle\rangle$).

The modules `LQ` and `LLQ` can be regarded as values of this type.

Task 1. Define a relation between the implementations of `LQ.queue` and `LLQ.queue` (i.e. between `int list` and `int list * int list`) that says when an `LQ.queue` represents the same queue as an `LLQ.queue`. Hint: you may want to use the `reverse` and append (`@`) functions on lists.

Task 2. Expand the definition of logical equivalence for the type

$$q \times (\texttt{int} \times q \to q) \times (q \to (\texttt{int} \times q)\,\texttt{option})$$

with q interpreted as your relation — i.e. what does it mean for `emp`, `ins`, `rem` to preserve your relation?

Task 3. Prove that the implementation of `LQ` is related to the the implementation of `LLQ` by the relation described in the previous task. Convince yourself that this shows that the two implementations are logically (and therefore observationally) equivalent, because when $\exists t.\tau$ is closed (has no free type variables), then $e \sim_{\exists t.\tau} e'$ iff $e \mapsto^* \texttt{pack}[\tau_0](e_0)$ and $e' \mapsto^* \texttt{pack}[\tau_0'](e_0')$ and there exists a relation $R(x : \tau_0, y : \tau_0')$ such that $e \sim_\tau e'[t \hookrightarrow R]$.

This example is discussed in PFPL Section 17.4, so look there if you get stuck and want hints.

4 Compiling a Dynamically-Typed Language

The following problem extends the above exercise to a more fully-featured dynamically typed language.

4.1 Syntax

The abstract syntax of Dyn is as follows:

$$\begin{array}{rcl} d & ::= & x \mid (\mathsf{num}\ n) \mid (\mathsf{plus}\ d_1\ d_2) \mid (\mathsf{num?}\ d) \\ & & \mid (\mathsf{fun}\ (x)\ d) \mid (\mathsf{app}\ d_1\ d_2) \mid (\mathsf{fun?}\ d) \\ & & \mid \mathsf{nil} \mid (\mathsf{if}\ d\ d_1\ d_2) \\ & & \mid (\mathsf{cons}\ d_1\ d_2) \mid (\mathsf{car}\ d) \mid (\mathsf{cdr}\ d) \mid (\mathsf{cons?}\ d) \\ & & \mid \mathsf{error} \end{array}$$

This syntax is inspired by some dynamically typed languages such as Scheme.

As we will see in the dynamic semantics below, each value in this language is tagged with a *class*, which identifies it as a number or a function or nil or a cons. $(\mathsf{num}\ n)$ is a number constant, $(\mathsf{plus}\ d_1\ d_2)$ adds two numbers, and $(\mathsf{num?}\ d)$ checks whether the argument has class number. x is a variable; $(\mathsf{fun}\ (x)\ d)$ is a function, where x stands for the argument and is bound in d; $(\mathsf{app}\ d_1\ d_2)$ is function application; and $(\mathsf{fun?}\ d)$ checks whether the argument has class function. The term nil is simply a distinguished constant that we can check for with $(\mathsf{if}\ d\ d_1\ d_2)$, which evaluates to d_1 on any **non**-nil value and to d_2 on nil—think of nilas "false". $(\mathsf{cons}\ d_1\ d_2)$ creates a pair of d_1 and d_2, with projections $(\mathsf{car}\ d)$ and $(\mathsf{cdr}\ d)$.[1] $(\mathsf{cons?}\ d)$ checks whether its argument has class cons. Finally error is a distinguished error term.

To simplify things, we do not include booleans true and false. Therefore, $(\mathsf{fun?}\ d)$ and $(\mathsf{num?}\ d)$ and $(\mathsf{cons?}\ d)$ return d itself if d has the specified class, and nil otherwise. E.g. $(\mathsf{num?}\ (\mathsf{num}\ n))$ evalutes to $(\mathsf{num}\ n)$, while E.g. $(\mathsf{fun?}\ (\mathsf{num}\ n))$ evalutes to nil. If you think about it for a minute, you'll see why we do not include a $(\mathsf{nil?}\ d)$ that follows this pattern; if you like, you can define $(\mathsf{nil?}\ d)$ so that it returns some non-nil value (e.g. the number 1) when given nil and nil otherwise using $(\mathsf{if}\ d\ d_1\ d_2)$.

The term cons can be used to simulate pairs, lists, trees, and so on:

$(\mathsf{cons}\ (\mathsf{num}\ 1)\ (\mathsf{num}\ 2))$	the pair (1,2)
$(\mathsf{cons}\ (\mathsf{num}\ 1)\ \mathsf{nil})$	the list [1]

4.2 Static Semantics

Recall that the static semantics of a language answer the question "Which are the well-formed programs?" DYN is "dynamically typed" in the sense that its static semantics admit essentially any program that you can write down as well-formed. For example, the following are all well-formed DYN programs:

$$(\mathsf{plus}\ (\mathsf{num}\ 1)\ (\mathsf{num}\ 2))$$
$$(\mathsf{cons}\ (\mathsf{num}\ 1)\ \mathsf{nil})$$
$$(\mathsf{app}\ (\mathsf{fun}\ (x)\ x)\ (\mathsf{num}\ 2))$$
$$(\mathsf{if}\ (\mathsf{fun}\ (x)\ x)\ (\mathsf{num}\ 4)\ \mathsf{nil})$$

$$(\mathsf{plus}\ (\mathsf{num}\ 1)\ \mathsf{nil})$$
$$(\mathsf{app}\ (\mathsf{num}\ 4)\ (\mathsf{num}\ 5))$$

$$(\mathsf{app}\ (\mathsf{fun}\ (x)\ (\mathsf{app}\ x\ x))\ (\mathsf{fun}\ (x)\ (\mathsf{app}\ x\ x)))$$

However, we do give DYN a simple static semantics which simply checks that variables are bound before they are used. This is presented in the form a judgement $\Gamma \vdash d\ \mathsf{ok}$, where Γ is a collection of hypotheses of the form $x\ \mathsf{ok}$. The rules for this judgement are presented in Figure 2.

4.3 Dynamic Semantics

One result of admitting so many well-formed programs in the static semantics is that we then have to deal with them in the dynamic semantics. Like many dynamically typed languages, DYN is intended to be safe in the sense that the operational behavior of any well-formed program is completely specified: you can prove progress and preservation for DYN. To accomplish this, we make the programs that would have been ruled out by a type system result in run-time errors. For example, here are the intended results of the programs

[1] The names come from a programming language called Lisp, and were chosen because of features of the IBM 704 computer (`http://en.wikipedia.org/wiki/CAR_and_CDR`)—think of $(\mathsf{car}\ d)$ as "first projection" and $(\mathsf{cdr}\ d)$ as "second projection".

$\boxed{\Gamma \vdash d\,\mathsf{ok}}$

$$\frac{}{\Gamma, x\,\mathsf{ok} \vdash x\,\mathsf{ok}}$$

$$\frac{}{\Gamma \vdash (\mathsf{num}\ n)\,\mathsf{ok}} \qquad \frac{\Gamma \vdash d_1\,\mathsf{ok} \quad \Gamma \vdash d_2\,\mathsf{ok}}{\Gamma \vdash (\mathsf{plus}\ d_1\ d_2)\,\mathsf{ok}} \qquad \frac{\Gamma \vdash d_1\,\mathsf{ok}}{\Gamma \vdash (\mathsf{num?}\ d_1)\,\mathsf{ok}}$$

$$\frac{\Gamma, x\,\mathsf{ok} \vdash d\,\mathsf{ok}}{\Gamma \vdash (\mathsf{fun}\ (x)\ d)\,\mathsf{ok}} \qquad \frac{\Gamma \vdash d_1\,\mathsf{ok} \quad \Gamma \vdash d_2\,\mathsf{ok}}{\Gamma \vdash (\mathsf{app}\ d_1\ d_2)\,\mathsf{ok}} \qquad \frac{\Gamma \vdash d\,\mathsf{ok}}{\Gamma \vdash (\mathsf{fun?}\ d)\,\mathsf{ok}}$$

$$\frac{}{\Gamma \vdash \mathsf{nil}\,\mathsf{ok}} \qquad \frac{\Gamma \vdash d\,\mathsf{ok} \quad \Gamma \vdash d_1\,\mathsf{ok} \quad \Gamma \vdash d_2\,\mathsf{ok}}{\Gamma \vdash (\mathsf{if}\ d\ d_1\ d_2)\,\mathsf{ok}}$$

$$\frac{\Gamma \vdash d_1\,\mathsf{ok} \quad \Gamma \vdash d_2\,\mathsf{ok}}{\Gamma \vdash (\mathsf{cons}\ d_1\ d_2)\,\mathsf{ok}} \qquad \frac{\Gamma \vdash d\,\mathsf{ok}}{\Gamma \vdash (\mathsf{car}\ d)\,\mathsf{ok}} \qquad \frac{\Gamma \vdash d\,\mathsf{ok}}{\Gamma \vdash (\mathsf{cdr}\ d)\,\mathsf{ok}} \qquad \frac{\Gamma \vdash d\,\mathsf{ok}}{\Gamma \vdash (\mathsf{cons?}\ d)\,\mathsf{ok}}$$

$$\frac{}{\Gamma \vdash \mathsf{error}\,\mathsf{ok}}$$

Figure 2: Static Semantics for DYN

mentioned above:

Program	Result
(plus (num 1) (num 2))	(num 3)
(cons (num 1) nil)	(cons (num 1) nil)
(app (fun (x) x) (num 2))	(num 2)
(if (fun (x) x) (num 4) nil)	nil
(plus (num 1) nil)	error
(app (num 4) (num 5))	error
(app (fun (x) (app x x)) (fun (x) (app x x)))	infinite loop

As you can see, the operational semantics must detect error states by testing the *classes* of values at run-time, and step to error. This run-time information indicating what “type” a value has is not necessary in a statically typed language, because these errors are ruled out by the type checker.

The dynamic semantics for DYN is presented in Figures 3 and 4 as a judgement $d_1 \mapsto_{\mathsf{d}} d_2$. Premises of the form $d \neq d'$ mean syntactic inequality. Notice the extra run-time checks in the instruction rules. For example, plus cannot just go ahead and add the numbers in question; it must check that the arguments have the right class and signal an error if they do not. We will gain a better understanding of these class tests by looking at their compilation.

$\boxed{d\;\mathsf{value_d}}$

$$\frac{}{(\mathsf{num}\ n)\ \mathsf{value_d}} \qquad \frac{}{(\mathsf{fun}\ (x)\ d)\ \mathsf{value_d}} \qquad \frac{}{\mathsf{nil}\ \mathsf{value_d}} \qquad \frac{d_1\ \mathsf{value_d} \quad d_2\ \mathsf{value_d}}{(\mathsf{cons}\ d_1\ d_2)\ \mathsf{value_d}}$$

$\boxed{d_1 \mapsto_{\mathsf{d}} d_2}$

Numbers:

$$\frac{d_1 \mapsto_{\mathsf{d}} d_1'}{(\mathsf{plus}\ d_1\ d_2) \mapsto_{\mathsf{d}} (\mathsf{plus}\ d_1'\ d_2)} \qquad \frac{d_1\ \mathsf{value_d} \quad d_2 \mapsto_{\mathsf{d}} d_2'}{(\mathsf{plus}\ d_1\ d_2) \mapsto_{\mathsf{d}} (\mathsf{plus}\ d_1\ d_2')} \qquad \frac{n_1 + n_2 = n}{(\mathsf{plus}\ (\mathsf{num}\ n_1)\ (\mathsf{num}\ n_2)) \mapsto_{\mathsf{d}} (\mathsf{num}\ n)}$$

$$\frac{d_1\ \mathsf{value_d} \quad d_2\ \mathsf{value_d} \quad d_1 \neq (\mathsf{num}\ \_)}{(\mathsf{plus}\ d_1\ d_2) \mapsto_{\mathsf{d}} \mathsf{error}} \qquad \frac{d_1\ \mathsf{value_d} \quad d_2\ \mathsf{value_d} \quad d_2 \neq (\mathsf{num}\ \_)}{(\mathsf{plus}\ d_1\ d_2) \mapsto_{\mathsf{d}} \mathsf{error}}$$

$$\frac{d \mapsto_{\mathsf{d}} d'}{(\mathsf{num?}\ d) \mapsto_{\mathsf{d}} (\mathsf{num?}\ d')} \qquad \frac{}{(\mathsf{num?}\ (\mathsf{num}\ n)) \mapsto_{\mathsf{d}} (\mathsf{num}\ n)} \qquad \frac{d\ \mathsf{value_d} \quad d \neq (\mathsf{num}\ \_)}{(\mathsf{num?}\ d) \mapsto_{\mathsf{d}} \mathsf{nil}}$$

Functions:

$$\frac{d_1 \mapsto_{\mathsf{d}} d_1'}{(\mathsf{app}\ d_1\ d_2) \mapsto_{\mathsf{d}} (\mathsf{app}\ d_1'\ d_2)} \qquad \frac{d_1\ \mathsf{value_d} \quad d_2 \mapsto_{\mathsf{d}} d_2'}{(\mathsf{app}\ d_1\ d_2) \mapsto_{\mathsf{d}} (\mathsf{app}\ d_1\ d_2')}$$

$$\frac{d_2\ \mathsf{value_d}}{(\mathsf{app}\ (\mathsf{fun}\ (x)\ d)\ d_2) \mapsto_{\mathsf{d}} [d_2/x]\ d} \qquad \frac{d_1\ \mathsf{value_d} \quad d_2\ \mathsf{value_d} \quad d_1 \neq (\mathsf{fun}\ (\_)\ \_)}{(\mathsf{app}\ d_1\ d_2) \mapsto_{\mathsf{d}} \mathsf{error}}$$

$$\frac{d \mapsto_{\mathsf{d}} d'}{(\mathsf{fun?}\ d) \mapsto_{\mathsf{d}} (\mathsf{fun?}\ d')} \qquad \frac{}{(\mathsf{fun?}\ (\mathsf{fun}\ (x)\ d)) \mapsto_{\mathsf{d}} (\mathsf{fun}\ (x)\ d)} \qquad \frac{d\ \mathsf{value_d} \quad d \neq (\mathsf{fun}\ (\_)\ \_)}{(\mathsf{fun?}\ d) \mapsto_{\mathsf{d}} \mathsf{nil}}$$

Nil:

$$\frac{d \mapsto_{\mathsf{d}} d'}{(\mathsf{if}\ d\ d_1\ d_2) \mapsto_{\mathsf{d}} (\mathsf{if}\ d'\ d_1\ d_2)} \qquad \frac{d\ \mathsf{value_d} \quad d \neq \mathsf{nil}}{(\mathsf{if}\ d\ d_1\ d_2) \mapsto_{\mathsf{d}} d_1} \qquad \frac{}{(\mathsf{if}\ \mathsf{nil}\ d_1\ d_2) \mapsto_{\mathsf{d}} d_2}$$

Figure 3: Dynamic Semantics for DYN

Cons:

$$\frac{d_1 \mapsto_{\mathsf{d}} d_1'}{(\mathsf{cons}\ d_1\ d_2) \mapsto_{\mathsf{d}} (\mathsf{cons}\ d_1'\ d_2)} \qquad \frac{d_1\ \mathsf{value_d} \quad d_2 \mapsto_{\mathsf{d}} d_2'}{(\mathsf{cons}\ d_1\ d_2) \mapsto_{\mathsf{d}} (\mathsf{cons}\ d_1\ d_2')}$$

$$\frac{d \mapsto d'}{(\mathsf{car}\ d) \mapsto_{\mathsf{d}} (\mathsf{car}\ d')} \qquad \frac{(\mathsf{cons}\ d_1\ d_2)\ \mathsf{value_d}}{(\mathsf{car}\ (\mathsf{cons}\ d_1\ d_2)) \mapsto_{\mathsf{d}} d_1} \qquad \frac{d\ \mathsf{value_d} \quad d \neq (\mathsf{cons}\ \_\ \_)}{(\mathsf{car}\ d) \mapsto_{\mathsf{d}} \mathsf{error}}$$

$$\frac{d \mapsto d'}{(\mathsf{cdr}\ d) \mapsto_{\mathsf{d}} (\mathsf{cdr}\ d')} \qquad \frac{(\mathsf{cons}\ d_1\ d_2)\ \mathsf{value_d}}{(\mathsf{cdr}\ (\mathsf{cons}\ d_1\ d_2)) \mapsto_{\mathsf{d}} d_2} \qquad \frac{d\ \mathsf{value_d} \quad d \neq (\mathsf{cons}\ \_\ \_)}{(\mathsf{cdr}\ d) \mapsto_{\mathsf{d}} \mathsf{error}}$$

$$\frac{d \mapsto_{\mathsf{d}} d'}{(\mathsf{cons?}\ d) \mapsto_{\mathsf{d}} (\mathsf{cons?}\ d')} \qquad \frac{(\mathsf{cons}\ d_1\ d_2)\ \mathsf{value_d}}{(\mathsf{cons?}\ (\mathsf{cons}\ d_1\ d_2)) \mapsto_{\mathsf{d}} (\mathsf{cons}\ d_1\ d_2)} \qquad \frac{d\ \mathsf{value_d} \quad d \neq (\mathsf{cons}\ \_\ \_)}{(\mathsf{cons?}\ d) \mapsto_{\mathsf{d}} \mathsf{nil}}$$

Error Propagation:

$$\frac{}{(\mathsf{plus}\ \mathsf{error}\ d) \mapsto_{\mathsf{d}} \mathsf{error}} \qquad \frac{d\ \mathsf{value_d}}{(\mathsf{plus}\ d\ \mathsf{error}) \mapsto_{\mathsf{d}} \mathsf{error}} \qquad \frac{}{(\mathsf{num?}\ \mathsf{error}) \mapsto_{\mathsf{d}} \mathsf{error}}$$

$$\frac{}{(\mathsf{app}\ \mathsf{error}\ d) \mapsto_{\mathsf{d}} \mathsf{error}} \qquad \frac{d\ \mathsf{value_d}}{(\mathsf{app}\ d\ \mathsf{error}) \mapsto_{\mathsf{d}} \mathsf{error}} \qquad \frac{}{(\mathsf{fun?}\ \mathsf{error}) \mapsto_{\mathsf{d}} \mathsf{error}}$$

$$\frac{}{(\mathsf{if}\ \mathsf{error}\ d_1\ d_2) \mapsto_{\mathsf{d}} \mathsf{error}} \qquad \frac{}{(\mathsf{cons}\ \mathsf{error}\ d) \mapsto_{\mathsf{d}} \mathsf{error}} \qquad \frac{d\ \mathsf{value_d}}{(\mathsf{cons}\ d\ \mathsf{error}) \mapsto_{\mathsf{d}} \mathsf{error}}$$

$$\frac{}{(\mathsf{car}\ \mathsf{error}) \mapsto_{\mathsf{d}} \mathsf{error}} \qquad \frac{}{(\mathsf{cdr}\ \mathsf{error}) \mapsto_{\mathsf{d}} \mathsf{error}}$$

$$\frac{}{(\mathsf{cons?}\ \mathsf{error}) \mapsto_{\mathsf{d}} \mathsf{error}}$$

Figure 4: Dynamic Semantics for DYN, Continued

4.4 Compiler

As a target of compilation, we use a language with integers, recursive functions, labeled products and sums (see the Day 1 implementation problem, or PFPL Chapters 10-11), and recursive types:

$$\tau \quad ::= \quad t \mid \mathsf{int} \mid \tau_1 \to \tau_2 \mid \langle l_1 \hookrightarrow \tau_1, \ldots, l_n \hookrightarrow \tau_n \rangle \mid [l_1 \hookrightarrow \tau_1, \ldots l_n \hookrightarrow \tau_n] \mid \mathsf{rec}(t.\tau)$$

Task 1. Define a type dyn that can represent all of the classes of data in the DYN language. Informally, these include:

- numbers
- functions from values to values, which *can error when applied*
- nil
- a cons of two values

The idea now is to compile a DYN program to a program of type dyn. One wrinkle is how we will deal with the run-time errors that arise in DYN programs. We would like to show how error emerges from sum types, so we use options to model errors. This means *the code for checking and propogating errors must be generated by the compilation*, instead of being built into the target language. (This is a common trade-off in compilers: the less you put into the target language, the more complex compilation is, but the easier it is to implement the target language.) The dynamic semantics of DYN step to error when a primitive is applied to a wrong class of argument; for example:

$$(\mathsf{plus}\ (\mathsf{num}\ 1)\ \mathsf{nil}) \mapsto_{\mathsf{d}} \mathsf{error}$$

To cover this possibility we compile a DYN term to term of type dyn option(which is definable in the above language as $[\mathsf{some} \hookrightarrow \tau, \mathsf{none} \hookrightarrow \mathsf{unit}]$), where the none case means that the DYN term resulted in an error and the $\mathsf{some}(v)$ case represents an actual value.

There is one subtlety, however: what should the compilation of this program be?

$$(\mathsf{fun}\ (\_)\ (\mathsf{plus}\ (\mathsf{num}\ 1)\ \mathsf{nil}))$$

Because functions can error when applied, when we compile the body of a function, we will get a program of type dyn option rather than dyn. You may need to change your definition of dyn if you didn't catch this above.

Task 2. Define a compilation function, and prove that it translates a DYN program

$$x_1\ \mathsf{ok}, \ldots, x_n\ \mathsf{ok} \vdash d\ \mathsf{ok}$$

to a program

$$x_1{:}\mathsf{dyn}, \ldots, x_n{:}\mathsf{dyn} \vdash d : \mathsf{dyn\ option}$$

The variables in the context are *not* given an option type, because DYNis call-by-value, so variables are always substituted by values, which correspond to elements of dyn, not dyn option.

Task 3. Prove that your compilation preserves the operational semantics. In particular, if $d_1 \mapsto_{\mathsf{d}} d_2$ then $d_1^\dagger \equiv d_2^\dagger$, where $\equiv$ means definitional equality (see the definition in Section 1).