Commentary on Practical Foundations for Programming Languages

r/c/s

January - May 2025

1 Introduction

This is my commentary on the book "Practical Foundations for Programming Languages" by Robert Harper, for the CSCI 8980 seminar taught by Professor Eric van Wyk.

The goal of this commentary is to provide someone with a good-but-untrained grasp on discrete mathematics and functional programming exposure an "in" to gaining a deeper appreciation into PL constructs and type safety more generally.

The expressiveness of the Rust typing system (to the point where `embedded-hal` can take the datasheet-y parts of embedded programming and roll it into the type system) is something that I've gained a lot more appreciation for while reading this textbook, so it's a pretty good read for systems programmers getting into the type safety hype. (i.e me!)

You are expected to have a copy of the book with you while reading - this is a companion piece to the book and we will be refering to many parts of the book.

2 Chapters 1 - 3: Prerequisite Knowledge

In this part PFPL attempts to lay out the knowledge needed to get "up-and-running" with the rest of the book. It's honestly almost too dense, but make sure you actually get this part down right - lose sight here and you'll be in a world of pain.

2.1 ASTs, and a primer to structural induction

It's here that we find out first language in PFPL

$$\text{Typ } \tau ::= \texttt{nat}$$

$$\text{Exp } e ::= x | n | e + e | e \times e$$

and in AST-mode, we have

$$\text{Exp } e ::= \texttt{var}[x] | \texttt{num}[n] | \mathit{plus}(e_1, e_2) | \mathit{times}(e_1, e_2)$$

We introduce this language as a means to an end: *structural induction.*
From the textbook:

Suppose that we wish to prove that some property P(a) holds of all ASTs 'a' of a given sort. To show that this is enough to consider all the way 'a' is generated, and show that the property holds in each case under the assumption that it holds for it's constituent ASTs (if any). For this language above, this boils down to:

1. The property holds for any variable x of sort Exp
   - prove that $P(x)$
2. The property holds for any number, $\texttt{num}[n]$:
   - prove that $P(\texttt{num}[n])$
3. Show that so long as (i.e assume) the property holds for a_1 and a_2 (the ASTs for e_1 and e_2), show that the property holds for $\texttt{plus}(a_1; a_2)$ and $\texttt{times}(a_1; a_2)$

I cannot stress this part enough - this is the key to understanding this textbook. It's not the static or dynamic rules. If you don't know where to start, start with this.

2.2 An aside: Substitution

This book typesets substitution in a bit of an obtuse way, and god help you if you end up needing to google this, because other books have their own way of typesetting substitution. We'll use this book's notation just to make it easier. Don't be afraid to come back to this!

- $[e/x]e'$
  - "e replaces x in e'"
  - $[e/x]e = e$
  - $[e/x]i = i$ if there is "no x in i"
  - $[e/x]y' = y, y \neq x$
  - $[e/x]e_1 + e_2 = [e/x]e_1 + [e/x]e_2$

2.3 ABTs

You can think of ABTs as the "memory" or "glue" or "context" of an AST: this is the bit that resolves variables to values in a context (and thus allowing some expression to be evaluated).

For example, "let x be 7 in x + x" would thus be evaluated to $7 + 7$ and then 14. ABTs also let us have "differing scope" - consider "let x be (x * x) in (x + x)". Had we had global scoping this would be impossible, but since the Let binding is only bound to the second AST (the x + x bit), it'll just be (x * x) + (x * x).

2.4 Inductive definitions and derivations

We've all seen how the naturals are defined

$$e ::= 0 | s(e)$$

So let's do something a little cooler - we introduce what's called a "inference rule," which has a structure such that a judgement is at the bottom, and *prerequisite* judgements on top of them.

$$\frac{}{\texttt{zero}\ \text{nat}}$$

$$\frac{e \text{ nat}}{s(e) \text{ nat}}$$

would thus be the *inference rules* for a language that contains *just* the natural numbers. Notice the zero-rule does not have a *prerequisite* judgement - we hold this to be an axiom.

This is the derivation of `succ(succ(zero))` nat:

$$\frac{\dfrac{\overline{\texttt{zero nat}}}{\texttt{succ(zero) nat}}}{\texttt{succ(succ(zero)) nat}}$$

It's important to note right here: *expressions* have derivations, not propositions/properties! (This was the part that got you *bad* in your confused first attempt!)

2.5 Rule induction

Rule induction is merely an instantiation of structural induction - In percise terms, the property P respects the rule

$$\frac{a_1 \text{ J} ... a_k \text{ J}}{a \text{ J}}$$

if $P(a)$ holds whenever $P(a_1), ..., P(a_k)$ do.

It's important to note that $P(a_1), ..., P(a_k)$ doesn't automatically grant you $P(a)$ holding - $P(a_1), ..., P(a_k)$ is merely the *inductive hypothesis* - the fun part is getting that proof!

2.5.1 An example

What fun is a document like this without examples?

Suppose, on the natural numbers, we want to show the property that $S(e)$ nat $\Rightarrow$ e nat (successor of E being a natural means, E is a natural)

What fun is a document like this without examples?

Suppose, on the natural numbers, we want to show the property that $S(e)$ nat $\Rightarrow$ e nat (successor of E being a natural means, E is a natural)

We have the two rules for naturals

$$\frac{}{\texttt{zero nat}}$$

$$\frac{e \text{ nat}}{s(e) \text{ nat}}$$

- We start with the first "form" of natural number, which is `zero`. Since `zero` isn't a $S(e)$...
  - We shrug this off because we don't need the property to be true in this case
  - Recall the implication truth table, if the LHS of an implication is false, then the right hand side is not of our concern
  - We call this "holding vacuously"
- We then move on to the next "form" of natural number, which is $S(e)$. Well,
  - We know that by the second rule, we have an inductive hypothesis that `e` nat.
    - If e is `zero` then the property holds obviously!
    - If $e = S(b)$, then we know b nat. We show that $e = S(a)$ nat, and since $S(a)$ is of form $S(b)$ and b nat, we have $S(b)$ nat by the inductive hypothesis.
      - This case doesn't actually need the inductive hypothesis, can you see how?

3 Chapters 4-6: Statics and Dynamics, Type Safety

(sidenote: for this section, bold paragraphs mean verbatim quotes from the textbook)

Most programming lanugages exhibit a phase distinction between the static and dynamic phases of processing: the static phase consists of parsing and type checking to ensure the program is well-formed; the dynamic phase consists of execution of well-formed programs. A language is said to be "safe" exactly when well-formed programs are well-behaved when executed. Before we start, it might be prudent to define a language **E**, just to see how a language is defined.

$$\text{Typ } \tau ::= \texttt{num} \mid \texttt{str}$$

$$\text{Exp } e ::= x | n | "s" | e_1 + e_2 | e_1 \times e_2 | e_1 \hat{} e_2 | \text{len}(e) | \text{let}(e_1; x.e_2)$$

and in AST form:

$$\text{Typ } \tau ::= \texttt{num} \mid \texttt{str}$$

$$\text{Exp } e ::= x | \texttt{num}[n] | \texttt{str}[s] | \texttt{plus}(e_1; e_2) | \texttt{times}(e_1; e_2) | \texttt{cat}(e_1; e_2) | \texttt{len}(e) | \texttt{let}(e_1; x.e_2)$$

3.1 Statics

Does the expression 'plus(x; num[n])' make sense? Obviously, it depends on if x is of type 'num'. The textbook then claims:

This example is, in fact, illustrative of the general case, in that the only information required about the context of an expression is the type of the variables within whose scope the expression lies. What I interpret from that statement is this: the only information required from the binding tree, is that (in this expression's case) the type of x is indeed `num`, for this expression to be well-typed. I'm not sure why the author decides on this verbiage, but sure.

We're then given the statics of the language **E**, which I won't replicate here - remember that this is commentary rather than a rewrite.

3.1.1 Inversion for Typing

Suppose that $\Gamma \vdash e : \tau$. If $e = \texttt{plus}(e_1; e_2)$, then $\tau = num$, $\Gamma \vdash e_1 : \texttt{num}$, $\Gamma \vdash e_1 : \texttt{num}$, and similarly for other constructs of the language How would we prove this? We first rewrite it as such

$$\Gamma \vdash e : \tau \Rightarrow e \text{ is } \texttt{plus}(e_1; e_2) \Rightarrow \tau \text{ is } \texttt{num}, \Gamma \vdash e_1 : \texttt{num}, \Gamma \vdash e :_2 \texttt{num}$$

Here's an interesting question: what rule are used in the beginning of the implication?

It's of form $\Gamma \vdash \texttt{plus}(e_1; e_2) : \tau$ for sure - we're using 4.1d: $\frac{\Gamma \vdash e_1 :\texttt{num} \quad \Gamma \vdash e :_2 \texttt{num}}{\Gamma \vdash \texttt{plus}(e_1; e_2) :\texttt{num}}$

So we look at the rule 4.1d, and we immediately see that $\Gamma \vdash \texttt{plus}(e_1; e_2) : \texttt{num}$, so we have $\tau = \texttt{num}$ checked off. Similarly, this rule gives us that by the inductive hypothesis, we have $\Gamma \vdash e_{\{1,2\}} : \texttt{num}$, and hence the proof.

Yeah, it really is that simple. We perform case analysis on the *derivation*, what rules can the expression take? (Note: if we overload '+' for concatenation, this will fail.)

Inversion is a good lemma to cite to tell you something about the language without much justification other than the rules themselves.

3.2 Dynamics

Where statics are concerned with the validity of expressions before they are evaluated, dynamics are concerend with the validity of expressions as they are evaluated - they describe how programs are executed.

We say that a *transition system* is specified with 4 forms of judgement

1. s state, that s is a *state* of the transition system
2. s final, that s is a *final* state
    - This means there is no more step to take past this one
3. s initial, that s is an *initial* state
4. $s \to s'$, where s and s' are states, and that s may "take a step" into s'
    - Obviously, $s \to s'$ can't occur when s final.

There's a trivial proof on if $s \to s'$ and $s' \to s''$ then $s \to^* s''$:

We are to show that $s \to s' \Rightarrow s' \to s'' \Rightarrow s \to^* s''$. We have the rules $\overline{s \to s}$ and $\frac{s \to s' \quad s \to s''}{s \to^* s''}$. That is, to show that 'P(s, s')' holds when $s \to^* s'$, we need to show that 'P(s, s)' holds, and if $s \to s'$ and 'P(s', s")' then 'P(s, s")'.

We know that 'P(s,s)' holds vacuously (i.e the property chain doesn't apply because the first statement is false).

We have $s \to s'$ and $s' \to s''$. We are to show that the property 'P(s', s")' holds, and since it holds by assumption (our proof statement says $s' \to s''$, which implies $s' \to^* s''$ inductively), then we have shown the implication chain, and therefore, our property, to be true.

The book then gives the structural dynamics of **E**, which I will not reproduce here. It also goes on a big thing about *contextual* and *equational* dynamics, but we won't dive too deep into that right now.

3.3 Type safety

This is the big part of PFPL! We want to show *type safety* to be true of our language! But what is type safety?

In terms of our language E here, it means that expressions like adding a string to a int, or concatenating two numbers, will never be valid in E.

Type safety, in general, is the *coherence* between the statics and dynamics - statics will predict that the value of an expression will have a certain "form," and the dynamics will ensure that the expression itself is well-defined. This will ensure that evaluating these expressions will never cause an illegal instruction.

Type safety is given by two properties

1. Preservation: If $e : \tau$ and $e \to e'$ then $e' : \tau$
2. Progress: If $e : \tau$, then either e val or there exists e' such that $e \to e'$

We also introduce the idea of a *canonical form* here. These are useful for proving *progress*.

Canonical forms of E: If e val and $e : \tau$ then

1. If $\tau = \text{num}$ then $e = \texttt{num}[n]$ for some number n
2. If $\tau = \text{str}$ then $e = \texttt{str}[s]$ for some string s

We prove this by induction on rules 4.1 and 5.3:

The property is e val $\Rightarrow (\tau = \text{num} \Rightarrow e = \texttt{num}[n])$ or e val $\Rightarrow (\tau = \text{str} \Rightarrow e = \texttt{str}[s])$

So firstly we have e val, which means by the rules of 5.3, only 5.3a and 5.3b have this property - all the others hold vacuously.

Then we have $\tau = \texttt{num}$, which a bunch of static rules in 4.1 apply to, but only 4.1b really apply here - since we need e val and rules 5.3 give that all these other rules are not e val, only 4.1b apply here. Thus, if e val and $e : \tau$ and $\tau = \text{num}$ then $e = \texttt{num}[n]$ for some n, since 5.3b and 4.1b state so, and same goes for the `str` case.

We have everything we theoretically need to prove type safety of our language above, but we'll save it for the next section where we have a new construct!

4 Chapter 9: System T

System T, so called for **T**otal Functions, or **T**ermination (not for Turing machine as was suggested, because System T is incapable of infinite loops), provides a primitive recursion mechanism.

We define System T with the following grammar:

$$\text{Typ } \tau ::= \texttt{nat} | \tau_1 \to \tau_2$$
$$\text{Exp } e ::= x | z | s(e) | \text{rec } e \ \{z \hookrightarrow e_0 | s(x) \text{ with } y \hookrightarrow e_1\} | \lambda(x : \tau)e | e_1(e_2)$$

$$\text{Typ } \tau ::= \texttt{nat} | arr(\tau_1; \tau_2)$$
$$\text{Exp } e ::= x | z | s(e) | \text{rec}(e)\{e_0; x.y.e_1\} | \text{lam}\{\tau\}(x.e) | ap(e_1; e_2)$$

The statics of **T** are as follows:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}$$

$$\frac{}{\Gamma \vdash z : \texttt{nat}}$$

$$\frac{\Gamma \vdash e : \texttt{nat}}{\Gamma \vdash s(e) : \texttt{nat}}$$

$$\frac{\Gamma \vdash e : nat \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \texttt{nat}, y : \tau \vdash e_1 : \tau}{\Gamma \vdash \texttt{rec}\{e_0; x.y.e_1\}(e) : \tau}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{lam}\{\tau_1\}(x.e) : \texttt{arr}(\tau_1; \tau_2)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{arr}(\tau_2; \tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{ap}(e_1; e_2) : \tau}$$

The dynamics of **T** are as follows:

$$\frac{}{z \ \texttt{val}}$$

$$\frac{[e \ \texttt{val}]}{s(e) \ \texttt{val}}$$

$$\frac{}{\texttt{lam}\{\tau\}(x.e) \ \texttt{val}}$$

(these three are known as the *closure* rules)

$$[\frac{e \to e'}{s(e) \to s(e')}]$$

$$\frac{e_1 \to e_1'}{\texttt{ap}(e_1; e_2) \to \texttt{ap}(e_1'; e_2)}$$

$$[\frac{e_1 \ \texttt{val} \quad e_2 \to e_2'}{\texttt{ap}(e_1; e_2) \to \texttt{ap}(e_1; e_2')}]$$

$$\frac{[e_2 \ \texttt{val}]}{\texttt{ap}(\texttt{lam}\{\tau\}(x.e); e_2) \to [e_2/x]e}$$

$$\frac{e \to e'}{\texttt{rec}\{e_0; x.y.e_1\}(e) \to \texttt{rec}\{e_0; x.y.e_1\}(e')}$$

$$\frac{}{\texttt{rec}\{e_0; x.y.e_1\}(z) : e_0}$$

$$\frac{s(e) \ \texttt{val}}{\texttt{rec}\{e_0; x.y.e_1\}(e) \to [e, \texttt{rec}\{e_0; x.y.e_1\}(e)/x, y]e_1}$$

(these are known as the *transition* rules)
Canonical forms: if $e : \tau$ and e val

1. If $\tau = nat$ then either $e = z$ or $e = s(e')$ for some e'
   - $P(e)$: $e : \tau \wedge e$ val $\Rightarrow (\tau = \texttt{nat} \Rightarrow e = z \vee e = s(e')$ for some e'
   - The only expressions that satisfy $e : \tau$ and e val are z, $s(e)$, x, and $\lambda(x : \tau)e$. Of these, only z and $s(e)$ are of $\tau = nat$, by the typing rules, thus the property holds.
   - For all other expressions, the property holds vacuously (since the proposition chain is false)
2. If $\tau = \tau_1 \to \tau_2$ then $e = \lambda(x : \tau_1)e_2$ for some e_2
   - $P(e)$: $e : \tau \wedge e$ val $\Rightarrow (\tau = \texttt{arr}(\tau_1; \tau_2) \Rightarrow e = \lambda(x : \tau_1)e_2$ for some e'
   - The only expressions that satisfy $e : \tau$ and e val are z, $s(e)$, x, and $\lambda(x : \tau)e$. Of these, only $\lambda(x : \tau)e$
   - For all other expression, the property holds vacuously (since the proposition chain is false)

And now for the hard part: proving type safety for System T

1. Preservation: If $e : \tau$ and $e \to e'$ then $e' : \tau$
   - $\texttt{ap}(e_1; e_2) \to \texttt{ap}(e_1'; e_2)$: By the typing rule 9.1f, we have $\tau = \texttt{arr}(\tau_2; \tau)$, by inversion we have that τ_2 exists such that $\Gamma \vdash e_1 : \texttt{arr}(\tau_2; \tau)$ and $\Gamma \vdash e_2 : \tau_2$. Thus this holds by the IH
   - $\texttt{ap}(\texttt{lam}\{\tau\}(x.e); e_2) \to [e_2/x]e$: Since $\texttt{lam}\{\tau\}(x.e) : \texttt{arr}(\tau_2; \tau)$ by Rule 9.1f, and by rule 9.1e $\Gamma, x : \tau_2 \vdash e : \tau$ is our IH, and $[e_2/x]e : \tau$ holds as a result.
   - $\texttt{rec}\{e_0; x.y.e_1\}(e) \to \texttt{rec}\{e_0; x.y.e_1\}(e')$: The typing rule gives us that $e : \texttt{nat}$ and by IH $e' : \texttt{nat}$ so preservation holds in this case.
   - $\texttt{rec}\{e_0; x.y.e_1\}(z) : e_0$: This one is simple. 9.1d gives us that both e_0 and the expression here itself is τ, so it holds for this case.

- $\texttt{rec}\{e_0; x.y.e_1\}(e) \to [e, \texttt{rec}\{e_0; x.y.e_1\}(e)/x, y]e_1$: This is remarkably similar to the application case. We have $\Gamma, x : \texttt{nat}, y : \tau \vdash e_1 : \tau$. x is replaced by $e : \texttt{nat}$, and since y is replaced by $\texttt{rec}\{e_0; x.y.e_1\}(e) : \tau$, so this holds.

2. Progress: If $e : \tau$, then either e val or there exists e' such that $e \to e'$
   - $P(e)$: $e : \tau \Rightarrow (e \text{ val } \vee \exists e' \text{ such that } e \to e')$
   - x: can be any of the below:
   - z: Zero, $\tau = nat$ by 9.1b, z val by 9.2a
   - $s(e)$: Inductive hypothesis e nat, $s(e)$: nat by 9.1b, $s(e)$ val by 9.2b
   - $\texttt{rec}\{e_0; x.y.e_1\}(e)$: $e : nat$ by inversion for typing, if $e = z$ then 9.3f $\texttt{rec}\{e_0; x.y.e_1\}(e) \to e_0$, if $e = s(e')$ then 9.3g $\texttt{rec}\{e_0; x.y.e_1\}(e) \to [e, rec\{e_0; x.y.e_1\}(e)/x, y]e_1$
   - $\texttt{lam}\{\tau\}(x.e)$: $\tau = \texttt{arr}(\tau_1; \tau_2)$ by 9.1e with $\Gamma, x : \tau_1 \vdash e : \tau_2$, $\texttt{lam}\{\tau\}(x.e)$ val by 9.2c
   - $\texttt{ap}(e_1; e_2)$: 9.3d provides $\texttt{ap}(\texttt{lam}\{\tau\}(x.e); e_2) \to [e_2/x]e$

Note that we do not need to be as specific as this example in general, but this is what a "proof that a language is type-safe" looks like.

5 Product and Sum Types

5.1 Product types

Product types, typically $\tau_1 \times \tau_2$, or colloquially a tuple, are an obviously useful construct, so it'd be nice if we can show some notion of type-safety for product types. Structs and objects are a "manifestation" of the product type.

We give tuples as such:

$$\text{Typ } \tau ::= \texttt{unit} | \texttt{prod}(\tau_1; \tau_2)$$

$$\text{Exp } e ::= \texttt{triv} | \texttt{pair}(e_1; e_2) | \texttt{pr}[l](e) | \texttt{pr}[r](e)$$

The `pair` expression is known as the *introduction* form, and the `pr` expressions the *elimination* form. The terms "constructor" and "destructor" apply here as well.

We won't reproduce the statics nor dynamics for product types, but we'll note that 10.2g and h require that despite only the left and right projections being used, we still need "the other side" to be valid. This is weird considering tuples of invalid types "on the other side" seem to work just fine in language practice. This seems to be something about divering computations? (if-then-else)

We show preservation and progress on the pair type:

1. Preservation: If $e : \tau$ and $e \to e'$ then $e' : \tau$
   - Case 1: $e.l : \tau \Rightarrow e.l \to e'$: Really, this is $e.l : \tau \wedge e.l \to e.l' \Rightarrow e.l' : \tau$
     We have the typing rule $\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e.l : \tau_1}$
     Remember the IH: Preservation holds for the top part of the rule.
     We assume $e.l \to e.l'$ and then we have the rule $\frac{e \to e'}{e.l \to e.l'}$ which gives us $e : \tau_1 \times \tau_2$, and the by typing rule $\frac{\Gamma \vdash e' : \tau_1 \times \tau_2}{\Gamma \vdash e.l' : \tau_1}$ the statement holds
   - Case 2: $e.r \to e.r'$ ': `%s/e.l/e.r/g`' on case 1.
   - Case 3: $e.l \to e'.l$: $e \to e' : \tau$, and since τ is $\tau_1 \times tau_2$, $e'.l$ retains τ_1.
   - Case 4: $e.r \to e'.r$: Essentially the same as Case 3

- Case 5: $< e_1, e_2 > \cdot l \to e_1$: By rule 10.1b $< e_1, e_2 >: \tau_1 \times \tau_2$ if $e_1 : \tau_1$ and $e_2 : \tau_2$. $< e_1, e_2 > \cdot l : \tau_1$ by definition.
- Case 6: $< e_1, e_2 > \cdot r \to e_2$: Same as Case 5

2. Progress: If $e : \tau$, then either e val or there exists e' such that $e \to e'$
   - $<>$: unit : 10.2a provides that $<>$ val.
   - $< e_1, e_2 >: \tau_1 \times \tau_2$: either e_1 val and e_2 val which provides $< e_1, e_2 >$ val by rule 10.2b, or $< e_1, e_2 > \to < e_1', e_2 >$ if $e_1 \to e_1'$ and same for e_2
   - $e.l : \tau_1$: Either e val and $e.l : \tau_1$ by rules 10.2e and 10.1c, or $e \to e'$ and $e.l \to e'.l : \tau_1$ by rules 10.2e and 10.1c.
   - $e.r : \tau_2$: Same as $e.l$

This may not seem all that enlightening, but now we've shown that type safety can be proven for a construct like the product type, and is not a property limited to the basic constructs we saw with System T.

5.2 Sum Types

Hoare's Billion Dollar Mistake is an example of the sum type. But so is the 'Option' type. We can all kind of agree that the 'Option' type is "the way to do things" when dealing with pointer-y stuff. The reason being is that rather than hiding semantic meaning in 'NULL', the 'Option' type provides us a good way of reasoning about it's value.

Incidentally, we can also show some notion of type safety over sum types, which is why it's better to have explicit sum types rather than a "semantic sum type."

$$\text{Typ } \tau ::= \texttt{void} | \texttt{sum}(\tau_1; \tau_2)$$

$$\text{Exp } e ::= \texttt{abort}\{\tau\}(e) | \texttt{case}(e; x_1.e_1; x_2.e_2) | \texttt{in}\{\tau_1; \tau_2\}[l](e) | \texttt{in}\{\tau_1; \tau_2\}[r](e)$$

Once again, we'll omit reproducing the statics and dynamics for sum types, just go get the textbook PDF.

1. Preservation: If $e : \tau$ and $e \to e'$ then $e' : \tau$
   - Case 1: $l.e \to l.e'$ Really, this is $l.e : \tau_1 \wedge l.e \to l.e' \Rightarrow l.e' : \tau_1$
     We have the typing rule $\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash l.e : \tau_1 + \tau_2}$.
     Assume $l.e \to l.e'$
     Then we have the rule $[\frac{e \to e'}{l.e \to l.e'}]$ (this only really makes sense under eagar eval, anyways..), which gives us $e \to e'$ as a consequence of the IH
     For which we have the rule 11.1b again, this time on e' and $l.e'$ which gives us $\frac{\Gamma \vdash e' : \tau_1}{\Gamma \vdash l.e' : \tau_1 + \tau_2}$ and thus we've proven this case.
   - Case 2: $l.e \to l.e'$ ': `%s/l.e/r.e/g`' on case 1.
   - Case 3: $\texttt{case}\ l.e\{l.x_1 \hookrightarrow e_1 | r.x_2 \hookrightarrow e_2\} \to [e/x_1]e_1$: Rule 11.1d provides that $\Gamma, x_1 : \tau_1 \vdash e_1 : \tau$, which is provided by the substitution $[e/x_1]$
   - Case 4: $\texttt{case}\ r.e\{l.x_1 \hookrightarrow e_1 | r.x_2 \hookrightarrow e_2\} \to [e/x_2]e_2$: Same as Case 3
   - Case 5: $\texttt{case}\ e\{l.x_1 \hookrightarrow e_1 | r.x_2 \hookrightarrow e_2\} \to \texttt{case}\ e'\{l.x_1 \hookrightarrow e_1 | r.x_2 \hookrightarrow e_2\}$: 11.1d provides that $e : \tau_1 + \tau_2$ and by the IH, $e \to e' : \tau_1 + \tau_2$
   - Case 6: $\texttt{abort}(e) \to \texttt{abort}(e')$: Same as Case 5, really, but that $e :$ `void`

2. Progress: If $e : \tau$, then either e val or there exists e' such that $e \to e'$:

   - Case $\texttt{abort}(e) : \tau$: Since e is always of type `void`, it's never a value, thus $e \to e'$, and there will always be a $\texttt{abort}(e) \to \texttt{abort}(e')$.
   - Case $\texttt{l.e} : \tau_1 + \tau_2$: Either e val and `l.e` val by 11.2b, or $e \to e'$ and $\texttt{l.e} \to l.e'$ by 11.2e
   - Case $\texttt{r.e} : \tau_1 + \tau_2$: Same as the case above
   - Case $\texttt{case}\ e\{l.x_1 \hookrightarrow e_1 | r.x_2 \hookrightarrow e_2\}$: Either $e \to e'$ in which case $\texttt{case}\ e\{l.x_1 \hookrightarrow e_1 | r.x_2 \hookrightarrow e_2\} \to \texttt{case}\ e'\{l.x_1 \hookrightarrow e_1 | r.x_2 \hookrightarrow e_2\}$ or e val, which implies e is of form $l.e$ or $r.e$ (as a consequence of cases 11.1(b,c) and 11.2(b,c)) and this expression will transition to $[e/x_1]e_1$ or $[e/x_2]e_2$.

and thus, we have shown type safety of sum types, another signficant construct of progrmming languages.

6 Inductive and Coinductive types

The primary motivation for inductive and coinductive types is to provide us with a mechanism for generating types like the natural numbers in later namely System F and the untyped lambda calculus. This is a core foundation of the other topics, so make sure you know it well (even if you can't pull the proofs off right away)

6.1 Inductive types

The natural numbers serve as a good gateway as an inductive type, since it's inductively defined and all.

The book redefines the introduction and elimination forms for the natural numbers as such:

$$\frac{\Gamma \vdash e : \text{unit} + \text{nat}}{\Gamma \vdash \text{fold}_{\text{nat}}(e) : nat}$$

$$\frac{\Gamma, x : \text{unit} + \tau \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \text{nat}}{\text{rec}_{\text{nat}}(x.e_1; e_2) : \tau}$$

And the dyanmics as such:

$$\frac{}{\text{fold}_{\text{nat}}(e)\ \text{val}}$$

$$\frac{e_2 \to e_2'}{\text{rec}_{\text{nat}}(x.e_1; e_2) \to \text{rec}_{\text{nat}}(x.e_1; e_2)}$$

$$\frac{}{\text{rec}_{\text{nat}}(x.e_1; \text{fold}_{\text{nat}}(e_2)) \to [\text{map}\{t.\text{unit+t}\}(y.\text{rec}_{\text{nat}}(x.e_1; y))(e_2)/x]e_1}$$

To define a doubling function over this type, we have

$$\texttt{rec}(x.\texttt{case}\ x\{l.<> \hookrightarrow fold(l.<>) | r.x' \hookrightarrow r.fold(r.fold(r.x'))\})(\texttt{fold}(e))$$

Harper doesn't give a clear indication about the motivations behind this formulation of inductive and coinductive types. I think this is an (somewhat abandoned/unclear) effort at showing that the constructor/recursor form as the basis for all inductive types.

6.2 Coinductive types

Instead of giving a robust definition of what a coinductive type is, we'll instead venture to show a coinductive type: this is the definition of a *stream* of natural numbers

$$\frac{\Gamma \vdash e : \texttt{stream}}{\Gamma \vdash \texttt{hd}(e) : nat}$$

$$\frac{\Gamma \vdash e : \texttt{stream}}{\Gamma \vdash \texttt{tl}(e) : stream}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash e_1 : \texttt{nat} \quad \Gamma, x : \tau \vdash e_2 : \tau}{\Gamma \vdash \texttt{strgen}\ x\ \texttt{is}\ e\ \texttt{in}\ \ < hd \hookrightarrow e_1, tl \hookrightarrow e_2 >: stream}$$

We also get these dynamic rules:

$$\frac{}{\texttt{strgen}\ x\ \texttt{is}\ e\ \texttt{in}\ \ < hd \hookrightarrow e_1, tl \hookrightarrow e_2 >\ \text{val}}$$

$$\frac{e \to e'}{\texttt{hd}(e) \to \texttt{hd}(e')}$$

$$\frac{}{\texttt{hd}(\texttt{strgen}\ x\ \texttt{is}\ e\ \texttt{in}\ \ < hd \hookrightarrow e_1, tl \hookrightarrow e_2 >) \to [e/x]e_1}$$

$$\frac{e \to e'}{\texttt{tl}(e) \to \texttt{tl}(e')}$$

$$\frac{}{\texttt{tl}(\texttt{strgen}\ x\ \texttt{is}\ e\ \texttt{in}\ \ < hd \hookrightarrow e_1, tl \hookrightarrow e_2 >) \to \texttt{strgen}\ x\ \texttt{is}\ [e/x]e_2\ \texttt{in}\ \ < hd \hookrightarrow e_1, tl \hookrightarrow e_2 >}$$

So a "stream of natural numbers" is really just the expression $\texttt{strgen}\ x\ \texttt{is z in}\ \ < hd \hookrightarrow x, tl \hookrightarrow s(x) >$

and "stream of even naturals" would be $\texttt{strgen}\ x\ \texttt{is z in}\ \ < hd \hookrightarrow x, tl \hookrightarrow s(s(x)) >$.

The mechanism in which coinduction really works is just the "tail" portion where successive substitutions of the "inner expression" based on the "tl" section is done.

Of course, these "stream" types are exactly what `iostream` and all the other streams in C++ are.

7 System F

We now come to System F and dispense ourselves of the simple types we saw in System T:

$$\text{Typ} ::= t | \tau_1 \to \tau_2 | \forall(t.\tau)$$

$$\text{Exp} ::= x | \lambda(x : t)e | e_1(e_2) | \Lambda(t)e | e[t]$$

The typing rules change a little - we add a new type of static judgement called *type formation* judgement.

$$\frac{}{\Delta, t\ \text{type} \vdash t\ \text{type}}$$

$$\frac{\Delta \vdash \tau_1\ \text{type} \quad \Delta \vdash \tau_2\ \text{type}}{\Delta \vdash \text{arr}(\tau_1; \tau_2)\ \text{type}}$$

$$\frac{\Gamma, t \text{ type} \vdash \tau \text{ type}}{\Gamma \vdash \text{all}(t.\tau) \text{ type}}$$

We won't reproduce the rest of the typing judgement, they're the same as in System T, but we'll have

$$\frac{\Delta\Gamma \vdash e_1 : \text{arr}(\tau_2; \tau) \quad \Delta\Gamma \vdash e_2 : \tau_2}{\Delta\Gamma \vdash \text{ap}(e_1; e_2) : \tau}$$

$$\frac{\Gamma, t \text{ type} \Gamma \vdash e : \tau}{\Delta\Gamma \vdash \text{Lam}(t.e) : \text{all}(t.\tau)}$$

$$\frac{\Delta\Gamma \vdash e : \text{all}(t.\tau') \quad \Gamma \vdash \tau \text{ type}}{\Delta\Gamma \vdash \text{App}\{\tau\}(e) : [\tau/t]\tau'}$$

for the new constructs. The dynamics are what you expect, as well.

Writing the proof that System F is type-safe is not actually all that enlightening. Why? The book says that it is type safe, so we'll go with that (plus, the proofs look awfully similar to the ones from System T). Rather, let's talk about what the book really doesn't:

So why do this in the first place?

The real motivation for System F is the ability to create any inductive/coinductive type!

We can create any desired inductive/coinductive type with System F alone.

For example, the nat type can be defined as such:

$$\text{nat} \triangleq \forall(t.t \to (t \to t) \to t)$$

$$\text{z} \triangleq \Lambda(t)\lambda(z : t)\lambda(s : t \to t)z$$

$$\text{s}(e) \triangleq \Lambda(t)\lambda(z : t)\Lambda(s : t \to t)s(e[t](z)(s))$$

$$\texttt{iter}\{e_1; x.e_2\}(e_0) \triangleq e_0[\tau](e_1)(\lambda(x : \tau)e_2)$$

In a similar fashion, sum types can be encoded as such:

$$\tau_1 + \tau_2 \triangleq \forall(r.(\tau_1 \to r) \to (\tau_2 \to r) \to r)$$

$$< e_1, e_2 > \triangleq \Lambda(r)\lambda(x : \tau_1 \to \tau_2 \to r)x(e_1)(e_2)$$

$$l.e \triangleq \Lambda(r)\lambda(x : \tau_1 \to r)\lambda(y : \tau_2 \to r)x(e)$$

$$r.e \triangleq \Lambda(r)\lambda(x : \tau_1 \to r)\lambda(y : \tau_2 \to r)y(e)$$

$$\texttt{case}e\{l.x_1 \hookrightarrow e_1 | r.x_2 \hookrightarrow e_2\} \triangleq e[p](\lambda(x_1 : \tau_1)e_1)(\lambda(x_2 : \tau_2)e_2)$$

and of course - the ability to create types like τ `list` and $\tau_1 \times \tau_2$ without knowing what τ is extremely powerful - the fact that we can prove type safety for this mechanism is something to marvel at.

7.1 Parametricity

Parametricity properties are properties of a program in System F that can be proved knowing only it's type. For example, if $i : \forall(t.t \to t)$ then we know for certain that i is the identity function, without knowing what the expression i is, and similarly, $b : \forall(t.t \to t \to t)$ then we know that b is either $\Lambda(t)(\lambda x : t)(\lambda y : t)x$ or $\Lambda(t)(\lambda x : t)(\lambda y : t)y$.

Parametricity properties allow us to examine how a given expression might behave in evaluation, without evaluating the function itself - extending the utility of the "static" rules.

7.2 Definability

Harper claims that you can write an evaluator for System T in System F. (Remember that it is impossible to do so in System T itself). I'm not sure if this is actually true without resorting to the whole "type inference is undecidable" mechanism, which I think is cheating. This is a fun exercise in the future.

The same diagonalization argument applies for System F, and so System F is not Turing complete - we'll have to go to System PCF/FPC for Turing completeness.

7.3 An aside: how concepts from System F can solve some common frustrations of strongly-typed languages

Brian Kernighan wrote in an article in 1981 about some of his pet peeves of the strongly-typed language Pascal. Most notably, in Pascal, the length of an array is actually part of the type itself, but Pascal does not afford the programmer the luxury of parametric polymorphism a-la System F.

This piece of Pascal decls below

```
var     arr10 : array [1..10] of integer;
        arr20 : array [1..20] of integer;
```

will cause `arr10` and `arr20` to be of different types. Brian notes that writing a `sort` for "interger arrays" is not possible under Pascal.

In a hypothetical Pascal-with-parametric-polymorpism, however, we could probably have something akin to

```
procedure sort(a: all(t.array t of integer));
...
```

which would allow for writing said `sort()` function while preserving the length embedding in the type.

`sort` isn't the only function that would be difficult to type in Pascal's typing system. Consider character string concatenation:

```
procedure concat(a: ????, b: ????)
```

Well now we have a big problem - what is the type of the function `concat()`? If `concat()` returns a type `array [1..MAXSTR] of char`, which is also the type of a and b, you'd get a type error if the length of a and b together go over `MAXSTR`. Clearly, string expressions should be as large as it needs to be. Parametricity allows us to assign valid types to expressions with the "array" datatype, thus making more programs typable.

Going back a bit, `sort` could also be extended to beyond the integers to all types where comparison between them is type-safe. OCaml provides us with this sorting function for it's `List` type:

```
List.sort : ('a -> 'a -> int) -> 'a list -> 'a list
```

and JavaScript gives us this sorting function for it's `Array` type:

```
Array.prototype.sort(compareFn: (a, b) => Number)
```

This allows us to abstract away the implementation details of specific algorithms (like sorting) in a manner that we can trust is type safe (since the mechanism behind parametric polymorphism, System F, is type safe). Indeed, this is where most programmers see parametric polymorphism in action.

Unfortunately, parametric polymorphism isn't in Pascal proper, so Brian's Pascal implementation of Software Tools resorts to a `[1..MAXCHAR] of char` type and then manually checks for null-terminators, which defeats the whole point of Pascal's type safety.

8 Existential Types

Harper writes somewhere down in the middle of this chapter that the "abstract" types in this chapter do not exist in runtime. This is probably the best way to introduce existential types IMO - as a compile-time construct that helps the programmer and the type-checker.

We extend System F with these constructs:

$$\text{Typ } \tau := \texttt{some}(t.\tau)$$

$$\text{Exp } := \texttt{pack}\{t.\tau\}\{\rho\}(e) | \texttt{open}\{t.\tau\}\{\rho\}(e_1; t, x.e_2)$$

The type is called the "interface," the "pack" expression is known as the "interface," and the "open" expression the "client."

The point of existential types is to provide some guarantees to the interfacing code about the behavior of some underlying system without exposing the underlying system itself - abstraction.

There are some important info given to you by the statics, so we'll reproduce the important ones here. It's safe to say that **FE** is type-safe with an appeal to PFPL.

$$\frac{\Delta \vdash \rho \text{ type} \quad \Delta, t \text{ type} \vdash \tau \text{ type} \quad \Delta\Gamma \vdash e : [\rho/t]\tau}{\Delta\Gamma \vdash \texttt{pack}\{t.\tau\}\{\rho\}(e) : \texttt{some}(t.\tau)}$$

$$\frac{\Delta\Gamma \vdash e_1 : \texttt{some}(t.\tau) \quad \Delta, t \text{ type } \Gamma \vdash \tau \text{ type} \quad \Delta \vdash \tau_2 \text{ type}}{\Delta\Gamma \vdash \texttt{open}\{\tau_2\}(e_1; t, x.e_2) : \tau_2}$$

We see that the type of the client τ_2 cannot not involve the abstract type t in the last static rule here. This is here to prevent the client from attempting to export a value of the abstract type. So say, if a "tree" client to the "bucket" interface tries to return a node, it can't.

The body of the client, e_2 is type checked without knowing the representation type t. The client is in effect polymorphic in the type variable t. That is, the interface can specify what the t is and the client must obey.

Once again, the abstract types (that is, the $\texttt{some}(t.\tau)$ type) does not exist in the compiled code. This is thus a **zero-cost** abstraction.

8.1 Rust and how existential types abstract protocol implementors

The Rust crate `embedded-hal` provides "traits" for various bus devices like I2C and SPI. While the operation of these busses is outside the scope of this paper, we'll see how the Rust `embedded-hal` allows embedded developers to write platform-independent code for embedded devices. We use an example of reading from an SD card, which is an example of an SPI device.

In `embedded-sdmmc` we see the implementation for an "SD card" device (abbreviated):

```
impl<SPI, DELAYER> SdCardInner<SPI, DELAYER>
where
    SPI: embedded_hal::spi::SpiDevice<u8>,
    DELAYER: embedded_hal::delay::DelayNs,
{
    /*
        Removed stuff that we don't really need and removed for brevity
    */

    fn read_byte(&mut self) -> Result<u8, Error> {
        self.transfer_byte(0xFF)
    }
    fn write_byte(&mut self, out: u8) -> Result<(), Error> {
        let _ = self.transfer_byte(out)?;
        Ok(())
    }
    fn transfer_byte(&mut self, out: u8) -> Result<u8, Error> {
        let mut read_buf = [0u8; 1];
        self.spi
            .transfer(&mut read_buf, &[out])
            .map_err(|_| Error::Transport)?;
        Ok(read_buf[0])
    }
    fn write_bytes(&mut self, out: &[u8]) -> Result<(), Error> {
        self.spi.write(out).map_err(|_e| Error::Transport)?;
        Ok(())
    }
    /// Send multiple bytes and replace them with what comes back over the SPI bus.
    fn transfer_bytes(&mut self, in_out: &mut [u8]) -> Result<(), Error> {
        self.spi
            .transfer_in_place(in_out)
            .map_err(|_e| Error::Transport)?;
        Ok(())
    }
    /// Spin until the card returns 0xFF, or we spin too many times and
    /// timeout.
    fn wait_not_busy(&mut self, mut delay: Delay) -> Result<(), Error> {
        loop {
            let s = self.read_byte()?;
            if s == 0xFF {
                break;
            }
            delay.delay(&mut self.delayer, Error::TimeoutWaitNotBusy)?;
        }
        Ok(())
    }
}
```

We can see that the SdCardInner type is dependent on the two types `embedded_hal::spi::SpiDevice<u8>` and `embedded_hal::delay::DelayNs`. This is akin to the SdCard type using the "open" client expression in **FE**.

In fact, when creating an SDCard type, we do use an expression that is exactly like the "open" type:

```
let (spi, cs) = arduino_hal::Spi::new(/* omitted */); // Replace with device-specific SPI init
let spi_bus = embedded_hal_bus::RefCellDevice::new(
    &spi, cs, /* omitted */
); // spi_bus is an example of the "open" expression!
let sdcard = embedded_sdmmc::SdCard::new(spi_bus);
```

In `spi.rs` we see this fragment (abbreviated):

```
pub trait SpiBus<Word: 'static + Copy = u8>: ErrorType {
    async fn read(&mut self, words: &mut [Word]) -> Result<(), Self::Error>;
    async fn write(&mut self, words: &[Word]) -> Result<(), Self::Error>;
    async fn transfer(&mut self, read: &mut [Word], write: &[Word]) -> Result<(), Self::Error>;
    async fn transfer_in_place(&mut self, words: &mut [Word]) -> Result<(), Self::Error>;
    async fn flush(&mut self) -> Result<(), Self::Error>;
}
```

We see the functions that the `SdCardInner` type uses are defined right here, but not implemneted. Needless to say, actual SPI bus function implementations are full of platform specific code.

We note that the first and third hypotheses in Rule 17.1b guarantees that in $\Delta\Gamma \vdash \texttt{pack}\{t.\tau\}\{\rho\} : \texttt{some}(t.\tau)$ we have *some* implementation of all functions specified in t. (The second hypothesis merely asserts that t *has* said functions). This is mirrored in the Rust docs, which state:

```
Traits are implemented for specific types through separate implementations.

Trait functions may omit the function body by replacing it with a semicolon.
This indicates that the implementation must define the function. If the trait
function defines a body, this definition acts as a default for any
implementation which does not override it. Similarly, associated constants
may omit the equals sign and expression to indicate implementations must
define the constant value. Associated types must never define the type, the
type may only be specified in an implementation.
```

(Side comment: It's really gratifying to find a rule in a concrete language like Rust reflect the rules in the textbook!)

We'll omit the Arudino and Pi Pico HAL crates here, but their function headers will include the `impl` keyword, signalling to the compiler that they do indeed implement `SpiBus`.

Once again, we'll note that `SpiBus` isn't a concrete type, but rather a "trait," Rust's keyword for indicating an existential type. Here we can see the SPI functions used by the SdCardInner type, which exists thanks to the `arduino_hal::Spi` crate "implementing" the `SpiBus` trait. Adapting code for other boards will be a matter of just using the `SpiBus` trait.

You'll notice that SdCardInner itself is "impl"-ing something - that's the larger SdCard type, which itself might does not need specify reliance on an SPI abstract type as there's also the SDIO bus that an SD card can operate on.

8.2 Modules, OCaml (they're more powerful than existentials!)

So back in Spring 2022 I had this assignment about OCaml modules. (Funnily enough, with Eric too!) that somehow very few people solved. It was quite intuitive to me, so it came as a surprise. Modules allow for behavior that look like existential types (but is not strictly existential types, as the module system is more powerful, as we'll see)

8.2.1 Modules acting like existential types

Say we want to implement a set, abstracting away the underlying data structure.

```
module type SetS = sig
    type 'a set
    val empty: 'a set
    val insert: 'a -> 'a set -> 'a set
    val is_elem: 'a -> 'a set -> bool
end
```

A minimal implementation using a List would thus look something like this:

```
open Set
module ListM :Set.SetS = struct
  type 'a set = 'a list
let empty = []
let insert (v: 'a) (ls: 'a list) : 'a list =  v::ls
let rec is_elem (v: 'a) (ls: 'a list) : bool =
    List.fold_left  (fun accum x -> accum || (if x = v then true else false)) false ls
end
```

and in execution of code that asks for a "`Set.SetS` type" (which doesn't *really* exist, we instead "replace" it with the `ListM` ones.

```
module HiddenWordsSet(S: Set.SetS) = struct
    (* ... *)
    let check_for_words (f: string) (t: string) (words: string list) (es: string S.set) :
    (string * string *  string) list =
        List.fold_left (fun lst y -> if (S.is_elem y es) then (t,f,y)::lst else lst) [] words
    (* ... *)
end
```

Notice that if we just "blinded out" the module parameter's typing, we can actually do alpha-renaming over the code inside `HiddenWordsSet`, and that would still be correct. This is the beauty of existential types - they're a compile-time construct that aid the programmer, and is completely invisible.

You can, with some effort, emulate the convenience of existential types with C macros, and this is somewhat common practice in embedded C programming, but that tends to lead to inscrutible C errors compiler errors, and potential run-time errors due to manual macro substitutions that a competent type-checker can find

8.2.2 Modules are more powerful than existential types

Below is a more complicated set implementation, this time a red-black tree:

```
module TreeM : Set.SetS = struct
  type color = Red | Black
  type 'a rbtree = Leaf | Node of color * 'a * 'a rbtree * 'a rbtree
  type 'a set = 'a rbtree
  let empty = Leaf
  let balance = function
  | Black, z, Node (Red, y, Node (Red, x, a, b), c), d
```

```
  | Black, z, Node (Red, x, a, Node (Red, y, b, c)), d
  | Black, x, a, Node (Red, z, Node (Red, y, b, c), d)
  | Black, x, a, Node (Red, y, b, Node (Red, z, c, d)) ->
    Node (Red, y, Node (Black, x, a, b), Node (Black, z, c, d))
  | a, b, c, d -> Node (a, b, c, d)
  let insert (x: 'a) (ts: 'a rbtree) : 'a rbtree =
    let rec ins = function
      | Leaf -> Node (Red, x, Leaf, Leaf)
      | Node (color, y, a, b) as s ->
        if x < y then balance (color, y, ins a, b)
        else if x > y then balance (color, y, a, ins b)
        else s
    in
    match ins ts with
    | Node (_, y, a, b) -> Node (Black, y, a, b)
    | Leaf -> (* guaranteed to be nonempty *)
      failwith "RBT insert failed with ins returning leaf"
  let rec is_elem (x: 'a) (ts: 'a rbtree) : bool =
    match ts with
    | Leaf -> false
    | Node (_, y,l,r) -> if x < y then is_elem x l
                         else if x > y then is_elem x r
                         else true
end
```

Notice the addition of the `color` type, and how the rest of the Red-Black tree implementation relies on $\Delta \vdash$ `color` type. We can no longer perform the alpha-renaming trick in order to eliminate the "module parameter," which means modules can come with their own "internal" context (it's own $\Delta\Gamma$) (these are typically called "namespaces"). Notice that the rules state a single context $\Delta\Gamma$, which means modules are strictly more expressive than existential types.

9 System PCF

Robert Harper's blog has an article about how Computer Science educators tend to teach about recursion in a very wrong and very bad way. Harper highlights that recursion is "one of the most important and beautiful concepts in programming" and I'm very much inclined to agree.

It's also in this article that Harper gives a very concise definition of the `fix` construct in System PCF. We'll first show the example provided in the blog, then show the static and dynamic rules of the `fix` operator as shown in the book. This order of operations, I think, serve to illustrate the power of `fix` as a general recursive construct better than the book itself does.

(Note: I've been told that Harper and others did eventually succeed in reforming the CMU CS curriculum, which was the original intent of the article.)

Say we want to define the function `fact` recursively. We write the expression as such:

```
fix f is fun n : nat in case n { zero => 1 | succ(n') => n * f n' }
```

As the blogpost claims: by α-conversion, if we evaluate the function f, we find that f is merely

$$\lambda(n : \texttt{nat})\ \text{case}\ (n)\{z \Rightarrow 1 | s(n') => n \times f(n')\} : nat \rightharpoonup nat$$

and there is nothing actually all that interesting about it, other than that f is a "name" that we've given to the expression above that refers back to itself. Recursion by itself does not require any notion of a call stack, and simple α-conversion can implement recursion.

We call f a "fixed point" (and Harper tells us in no uncertain terms that it is a "fixed point" and not a "fixpoint" in his blog) as it allows *partial* functions - ones that diverge (i.e loop on seemingly forever) on some or even all inputs.

PCF is very similar to System T:

$$\text{Typ } \tau ::= \texttt{nat} | \tau_1 \rightharpoonup \tau_2$$

$$\text{Exp } e ::= x | z | s(e) | \lambda(x : \tau) e | e_1(e_2) | \texttt{ifz } e\{z \hookrightarrow e_0 | s(x) \hookrightarrow e_1\} | \text{fix } x : \tau \text{ is } e$$

$$\text{Typ } \tau ::= \texttt{nat} | arr(\tau_1; \tau_2)$$

$$\text{Exp } e ::= x | z | s(e) | \text{lam}\{\tau\}(x.e) | ap(e_1; e_2) | \texttt{ifz}\{e_0; x.e_1\}(e) | \texttt{fix}\{\tau\}(x.e)$$

We replace the `rec` construct with two constructs - a "if-zero" construct and a "fix" construct (which is the same construct discussed earlier)

The statics and dynamics from System T remain, but for the new constructs we have new static rules:

$$\frac{\Gamma \vdash e : \text{ nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \text{ nat} \vdash e_1 : \tau}{\Gamma \vdash \texttt{ifz}\{e_0; x.e_1\}(e) : \tau}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \texttt{fix}\{\tau\}(x.e) : \tau}$$

and of course, dynamic rules rules

$$\frac{e \to e'}{\texttt{ifz}\{e_0; x.e_1\}(e) \to \texttt{ifz}\{e_0; x.e_1\}(e')}$$

$$\frac{}{\texttt{ifz}\{e_0; x.e_1\}(z) \to e_0}$$

$$\frac{s(e) \text{ val}}{\texttt{ifz}\{e_0; x.e_1\}(s(e)) \to [e/x]e_1}$$

$$\frac{}{\texttt{fix}\{\tau\}(x.e) \to [\texttt{fix}\{\tau\}(x.e)/x]e}$$

This last rule creates the self-referential nature for the fixed point operator - please refer back to the example above, and the earlier part of the commentary on substitution.

9.1 Type safety for PCF

A confusing part of the book is the big gap between System T and System F (total languages) and System PCF. I suppose it's for good reason (covering inductive and coinductive data types, polymorphism, and existential types are all important constructs in programming languages), but this creates a false sense that type safety means all expressions converge to a value - it does not.

In progress, when the property "if e : *tau* then e val or there exists e' such that $e \to e'$, it REALLY means "or" in the logical sense. This property is the reason why type safety for PCF (and thus, type safety for recursion) can happen.

1. Preservation: If $e : \tau$ and $e \to e'$ then $e' : \tau$
    - $\texttt{ifz}\{e_0; x.e_1\}(e) \to \texttt{ifz}\{e_0; x.e_1\}(e')$: $e \to e'$ as the inductive hypothesis, then by the static rule for `ifz`, this holds.
    - $\texttt{ifz}\{e_0; x.e_1\}(z) \to e_0$: The static rule for `ifz` specifies that the construct itself and e_0 have the same type τ

- $\texttt{ifz}\{e_0; x.e_1\}(s(e)) \to [e/x]e_1$: $e :$ nat trivially, then by the static rule for `ifz` we know that $x :$ nat needs to hold for e_1 to have the same type τ as the `ifz` construct. We swap x with e, so this property holds.
- $\texttt{fix}\{\tau\}(x.e) \to [\texttt{fix}\{\tau\}(x.e)/x]e$: By the static rule, $x : \tau$ for $\texttt{fix}\{\tau\}(x.e) : \tau$. We replace x in e with $\texttt{fix}\{\tau\}(x.e) : \tau$, so the typing judgement holds (since τ is the same for x and the `fix` expression)

2. Progress: If $e : \tau$ then either e val or there exists e' such that $e \to e'$
   - $\texttt{ifz}\{e_0; x.e_1\}(e)$ transitions to e_0 or $[e/x]e_1$ depending on e. Since $e :$ nat, there is always a transition state.
   - $\texttt{fix}\{\tau\}(x.e)$ always transitions to $[\texttt{fix}\{\tau\}(x.e)/x]e$. No matter what. Progress holds trivially for the fixed point.

9.2 An aside: a comment on type safety that taught me quite a bit

Up till this point, we've talked about type safety as this infallable thing ("type-safe programs cannot go wrong"). So here's a statement that has a probability of shattering your mind:

"C is perfectly type safe."

This is not my quote, this is Harper's.

"Type safety is an internal coherence property stating that the statics and the dynamics fit together. It is always true by definition, and therefore never interesting. True by definition meaning that if it weren't true, you'd change the definition."

(and that's why I've not bothered with furnishing type safety for many constructs discussed - it's simply not interesting. Harper seems to have this opinion too, which I think past me would not.)

The more important point that Harper makes in this comment is that "a language L is minimally sensible iff it enjoys type safety in the sense of L. It makes no sense to ask whether L is "type safe" in some independent sense that L' may enjoy but L might not."

Harper points out that Unix never properly implemented C dynamics, and "absurdities" like "segmentation fault" exist, and programmers have to be aware about "voodoo magic" (my term) like the call stack, `errno`s and magic pointers you write to in order to create side effects. None of this is in the C language directly, and is an issue with how we implement them.

Harper encounters the same sort of issues in how others teach recursion - when talking about recursion in beginner programming courses, we have to set asides about the call stack, "stack overflows," and all that jazz because the way we implement recursion in real languages do not reflect the dynamics we see in System PCF. In our PCF dynamics it is very much advantageous to use recursion but recursion is seens as a faux-pas in beginner programming courses.

Of course, PFPL has a section about control flow and mutable state, but we're not going to cover all that. I just found this comment and blog post to be very interesting and pertinent to PCF.

10 Logical Relations, or "weird symbols get thrown at you a lot"

So this is the sort-of "famous" part of PFPL, and frankly I still don't understand it all that well. The gist that I took away is that we need a coherent understanding of what "equality" is, and logical relations is the means to that end.

We get thrown a bunch of symbols, but here are the ones that I think are important.

Kleene equivalence: Written as $e \simeq e'$, it's defined as $e \simeq e'$ iff $\exists n \geq 0$ s.t $e \to^\star \overline{n}$ and $e \to^\star \overline{n}$. That is - Kleene equivalence is when two expressions evaluate to the same value.

Observational equivalence: Written as $e \cong e'$, it's defined as $\Gamma \vdash e \cong e' : \tau$ iff $C\{e\} \simeq C\{e'\}$ for every program context C. That is, for all possible experiments you can do on e and e', the outcome is the same for e and e'.

So.. . is Kleene equivalence just observational equivalence? No! This is where we acknowledge that side effects are a thing. (PFPL actually does go quite a bit into reasoning about concurrency, but our skimming skips past that)

Harper tries to establish observational equivalence as "the coarsest congruent correlation" between expressions, but honestly I don't have the mathematical background to parse that. The important part comes next.

Logical equivalence is defined to be a family of relations $e \sim_\tau e'$ between closed expressions of type τ. That is, $e \sin_{\mathtt{nat}} e'$ iff $e \simeq e'$, and $e \sin_{\tau_1 \to \tau_2} e'$ iff $e_1 \sim_{\tau_1} e_1' \Rightarrow e(e_1) \sim_{\tau_2} e'(e_1')$

Harper then claims "logical equivalence and observational equivalence coincide," and provides a proof of that, which flew over my head.

But why do we care?

10.1 Sketch: System T is normalizing.

(Note: none of this is original, this is from Ahmed (2023))

Amal Ahmed wrote a pretty concise document that is much easier to read for OPLSS 2023. We want to show that System T (in Ahmed it's called the Simply-Typed Lambda Calculus) is normalizing (i.e all computations in System T terminate), as a foil to understanding how logical relations are used.

Ahmed starts out trying to prove System T to be normalizing thru induction on the derivations alone, but when we get to the application case, we find that $(\lambda(x : \tau)e)(e_2 : \tau) \to [e_2/x]e$, and we don't know anything else about $[e_2/x]e$ - if e terminates or not. The inductive hypothesis is not enough to solve this.

Logical relations are a predicate $P_\tau(e)$ over an expression of a type. They have three principles

- The relation should have well-typed terms, i.e $\cdot \vdash e : \tau$
- The property of interest P should be baked in the logical relation
  - In this instance, we want to say that normalization is baked in the logical relation
  - I don't understand this principle
- The property should be preserved in the elimination forms
  - This means that we need to have a way to continue after consuming our value of type τ

We construct the normalization predicates for System T as such:

$$N_{\mathtt{nat}}(e) \triangleq \cdot \vdash e : \mathtt{nat} \wedge e \Downarrow$$

$$N_{\tau_1 \to \tau_2} \triangleq \cdot e : \tau_1 \to \tau_2 \wedge \forall e'. N_{\tau_1}(e') \Rightarrow N_{\tau_2}(ee')$$

Notice that these predicates are constructed exactly like the definition for logical equivalence layed out in PFPL.

The proof for normalization then becomes as such:

- For all terms e, $\cdot \vdash e : \tau$ then $N_\tau(e)$
  - It's difficult to do this for lambda abstraction. We need to show that $\cdot \vdash \lambda(x : \tau_1)e : \tau_1 \to \tau_2$ and $\lambda(x : \tau_1)e \Downarrow$, then show that for every term e, if $N_{\tau_1}(e')$ then $N_{\tau_2}(\lambda x : \tau_1)ee'...$ where we get stuck, since e is not a closed term.

- We thus generalize this case into this: For all contexts Γ, substitutions γ, and terms e, if $\Gamma \vdash e : \tau$ and $\gamma \vDash \Gamma$ then $N_\tau(\gamma(e))$. There are some limits to γ (that's where the "dom" stuff comes from), but for now this will do.

- For all terms e, if $N_\tau(e)$ then $e \Rightarrow$

  - This is actually so easy you'll question it. This is just saying if e is normalizing then e will eventually evaluate into a value. The property of interest is in the definition of the logical relation

Then, with a lemma for substitution $\Gamma \vdash e : \tau$ and $\gamma \vDash \Gamma$ then $\cdot \vdash \gamma(e) : \tau$ and a lemma for reduction $\cdot \vdash e : \tau \wedge e \to e'$ then $N_\tau(e) \Leftrightarrow N_\tau(e')$, we can prove normalization for System T.

The case for application now becomes showing $N_\tau(\gamma(e_1 e_2)$, which is equivalent to showing $N_\tau(\gamma(e_1)\gamma(e_2))$. By the IH, we have $N_{\tau_2 \to \tau}(e_1)$, and unwrapping the definition gives us $\forall e', N_{\tau_1}(e') \Rightarrow N_{\tau_2}(\gamma(e_1 e))$. Let e' be $\gamma(e_2)$ and τ_1 as τ. The other IH gives us $N_{\tau_1}(\gamma(e_2))$, which gives us $N_\tau(\gamma(e_1)\gamma(e_2))$

Note that the abstraction case gets ugly if we endeavor to only use logical relations here, but I think using a combination of typical syntactic analysis and LR when-you-need-it is enough to convince myself.