CS442
Module 5: Functional Programming

University of Waterloo

Winter 2025

> "Any sufficiently complicated C or Fortran program contains an ad hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp."
>
> — Philip Greenspun's tenth rule

> "Greenspun's tenth rule is just an arrogant way to say 'programs sometimes use abstraction'."
>
> — Gregor Richards

1 Introduction to Functional Programming

You may have learned Racket in your first year (if you are an undergraduate student at Waterloo and followed a normal curriculum), and you have certainly used OCaml for this course. Congratulations, you've used a functional programming language! In fact, OCaml's second antecedent, ML, was considered for this course as an exemplar of functional programming, but Haskell will be used instead, for reasons which will become clear soon.

First, let's clear one thing up: in common parlance, "functional" is a synonym for "in working order". Thus, there are many obvious jokes around the premise "if it's not functional, then it must be dysfunctional!" Functional in this context, of course, means "of or relating to functions".

Functional languages are based on the construction and evaluation of expressions, and the mathematical equivalence between parameterized expressions and functions. Computations are encapsulated as functions, and functions are combined with each other to express complex computational tasks. Ordering is implicit, and computations are described in terms of the mathematical formula required to achieve their result, rather than an explicit set of steps.

Functional languages are distinguished by the absence (or reduced importance) of *commands*, particularly assignment statements. Through the elimination of commands, which alter program state, functions in the language behave more like functions in the mathematical sense: the output of any function is completely determined by its input, and not on the sequence of previous calls to the function, input/output, nor on any external state. In a *pure functional language*, execution of a function produces no *side effects*—the entire state of the machine, including the values bound to all variable names, remain unchanged after invocation of the function. Languages with this property are said to exhibit *referential transparency*:

Definition 1. A language is referentially transparent if every function has the property that, whenever it is given the same input, it always produces the same output.

More generally, we say that a language is referentially transparent if the meaning (i.e., the value) of an entity (variable, expression, etc.) is defined independently of its external context. However, for our purposes, the above definition will suffice.

An important consequence of referential transparency is that if two entities are declared to be equal, such as a variable bound to a value, then either side of the equals sign should be equivalent in all contexts. For example, if

the variable z is set to the value $f(3)$, then throughout their collective lifetime, the expressions z and $f(3)$ can be used interchangeably[1]. Without referential transparency, we lose this property.

Because of practicality considerations, few languages completely prohibit the use of side effects. Those that do are called "pure functional" languages, and those that don't are called "impure". An obvious example of a pure functional language is the λ-calculus itself. Languages like Scheme, Racket, Lisp, Common Lisp, ML, and OCaml are impure, because they permit side effects through assignment statements and I/O, but have a pure subset and generally discourage the use of impure features. Other languages, such as Haskell, which will serve as the exemplar of functional programming, prohibit assignment and I/O entirely, instead relegating all change in state to a form of explicit specification, called *monads*, which we will come back to in Section 16. Such languages are generally considered pure, and this design decision enforces programming in a functional style much more stringently than impure functional languages do, which is why Haskell is being used as an exemplar. In impure functional languages, z and $f(3)$ from above will *usually* be equivalent, but might not be. For example, consider this `f` and `z` (akin to the above example) written in OCaml:

Program	**Output**

```
1  open Stdio
2  let x = ref 0
3  let f q =
4      x := !x + 1;
5      !x + q
6  let z = f 3
7  let () =
8      printf "%d\n" z;
9      printf "%d\n" (f 3);
10     printf "%d\n" z;
11     printf "%d\n" (f 3)
```

```
4
5
4
6
```

Both printed values for `z` are `4`, but `f 3` ($f(3)$) takes the value `4` when it's bound to `z`, `5` when it's printed on the second line, and `6` when it's printed on the fourth line. OCaml's references break referential transparency.

Video 5.1 (https://student.cs.uwaterloo.ca/~cs442/W25/videos/5.1/): Referential transparency

By definition, in all functional languages, functions are *first-class values*. That is, variables can hold functions, and functions can be passed as values. This is not an uncommon characteristic of non-functional programming languages, but it is certainly a mandatory characteristic of functional languages. Moreover, in functional languages, the difference between a function and a value is often just the presence or absence of parameters. For instance, in OCaml, `let f = 42` is a declaration of a variable, and `let f x = x + 42` is a declaration of a function; and of course, the declaration of a function is really just a declaration of a variable which happens to be bound to a function, which is a value.

Typically, variables in functional languages aren't, uh, variable. That's... needlessly confusing. Actually, they're variable in the mathematical sense, but not the intuitive sense: what "variable" means mathematically is that in different resolutions of an expression or function, a variable may take different values. That context is the binding of the variable. Once bound, a variable's value cannot change; the way that it varies is that it could be bound differently in a different context. This property is *immutability*. Even impure functional languages are usually immutable by default, and usually require some extra construct—such as OCaml's `ref`s and records—to gain mutability. Pure functional languages have no way to express mutable variables per se.

In this module, we'll look at features that are either peculiar to or common among functional languages and how they are defined semantically. We'll mostly do this in the context of Haskell, but we'll need to take a mathematical aside into new extensions of the λ-calculus, namely System F and the parametric λ-calculus, in Sections 10 and 11.

[1] This definition does *not* consider efficiency, and many—but not all!—language implementations will still require extra computational cost to compute $f(3)$ if it's used in place of z.

2 Exemplar: Haskell

> "A monad is a monoid in the category of endofunctors, what's the problem?"
>
> — James Iry, parodying Philip Wadler, *A Brief, Incomplete, and Mostly Wrong History of Programming Languages*[4]

Our exemplar language for functional programming is Haskell. Haskell is a lazy, pure functional language, and, as there are as many as a dozen Haskell programmers who don't have Ph.D.s [citation needed], is vastly more popular than any other language in that category.

Remember, our goal is not to learn exemplars to the point that we can use them for effective programming, merely to use them to demonstrate some concepts of each paradigm. In this module, we will describe semantics and syntax as they relate to Haskell, so that we have something solid to relate them to; when Haskell is an outlier, we will also informally describe how it's done differently in other functional languages.

In the 1980s, a committee was formed to design a state-of-the-art pure functional language. The result was Haskell, named for the logician Haskell Curry. The first report on the Haskell language was published in 1990, and the current Haskell standard is Haskell 2010[6].

Haskell was influenced by ML, and is thus a cousin of OCaml. Some of its syntax will look familiar to what you've seen in OCaml. However, Haskell differs from ML in several important ways.

We will be introducing Haskell syntax as we introduce the relevant concepts. However, most of Haskell will be left unexplored, and this module is not intended to serve as a guide for learning Haskell. If you would like to learn Haskell, you are recommended to use Learn You a Haskell (http://learnyouahaskell.com/).

3 Expressions

All behavior in functional programs is described by *expressions*. Expressions can be made up of numbers, lists, functions, or other data. We combine expressions together to form more complex expressions.

Expressions typed into a Haskell REPL[2], such as `ghci`, are evaluated immediately. If we type `20` into `ghci`, it will respond with `20`.

Expressions can be combined using built-in functions, such as `+` or `*`, to form a compound expression that represents the application of that function. Note that `+` and `*` are *functions*, not operators, but they can be used in a way that resembles operators, by placing the function between its operands. The syntax is similar to OCaml and largely derived from normal mathematical notation. For instance:

```
> 100 + 20
120
> 100 * 20
2000
> 100 + 30 * 5
250
> 5 + 10 + 15 + 20
50
```

Functions can also be called in a way more similar to the λ-calculus, with the function followed by its arguments:

```
> max 5 10
10
> min (-12) 600
-12
> abs (-42)
42
```

[2] i.e., the interactive environment of Haskell. REPL stands for "read-eval-print-loop", so it should be obvious that every expression *read* is *eval*uated.

Again, operators are actually functions, so they may be called in the λ-calculus style as well, merely requiring a bit of syntactic shuffling:

```
> (+) 100 20
120
> (*) 100 20
2000
> (+) 100 ((*) 30 5)
250
```

Just like the λ-calculus, all functions actually take one argument, and multi-argument functions are just repeated application:

```
> add10 = (+) 10
> add10 20
30
> add10 (-10)
0
```

We've also shown a variable binding here; variable bindings in Haskell are less akin to `let` bindings in OCaml, which evaluate their expression ahead of time, and more a simple assertion of equivalence. That is, we've declared that `add10` is a shorthand for writing `(+) 10`. We'll see in Section 14 that in Haskell, these two concepts are actually equivalent.

Aside: The fact that every function is actually single-argument is why both λ-calculus and Haskell eschew parentheses for arguments to functions, in spite of the fact that mathematical notation has them. There's no need for that extra syntax when the number of arguments is unambiguous, and it would be a lot of clutter to add it!

All behavior is encapsulated into functions in this way, so all compound expressions either are function calls or can be reinterpreted as function calls. There is some syntactic sugar for common data types such as lists, but even this can be translated into nothing but function calls.

Since operators are just functions, if you're feeling especially cruel, you can redefine them:

```
> x + y = x - y
> 2 + 2
0
```

Luckily, these redefinitions are contextual. You cannot convince all of Haskell that addition and subtraction are the same thing; you are only allowed to shoot yourself in the foot.

Video 5.2 (https://student.cs.uwaterloo.ca/~cs442/W25/videos/5.2/): Haskell expressions

4 Functions

In functional languages, functions are *first class*. An entity is first class if it can be

- returned from functions,
- passed to functions as arguments, and
- stored as data.

This term is not precise, and different definitions will have different requirements. For instance, it is common to additionally require *function expressiveness*, which means that functions definitions are expressions, which would exclude C's functions from the definition. We won't split hairs about further restrictions on "first class" in this course; if you want to call C's functions first class, then by all means do, and if you don't, then don't. The above restrictions are all that are required to define an entity as first class in these notes.

The formal underpinning of a first-class function is a λ-calculus abstraction. The λ-calculus's abstractions can be expressed in Haskell using a relatively similar syntax, but replacing the hard-to-type λ character with a backslash (\) and the dot with an ASCII arrow (->). For instance, we can represent Church numerals and their mathematical operators:

```
> churchTwo = \f -> \x -> f (f x)
> churchThree = \f -> \x -> f (f (f x))
> churchMul = \m -> \n -> \f -> m (n f)
> churchExp = \m -> \n -> n m
```

Using these functions is a bit unsatisfying, only because the result is also a function, and Haskell doesn't have a built-in way to print a λ function:

```
> churchMul churchTwo churchThree

<interactive>:1:1: error:
    * No instance for (Show ((t0 -> t0) -> t0 -> t0))
        arising from a use of 'print'
        (maybe you haven't applied a function to enough arguments?)
    * In a stmt of an interactive GHCi command: print it
```

But, remember how Church numerals work: they are functions which take an f and an x, and apply f repeatedly to x, n times, where n is the number that the Church numeral represents. So, we can "translate" the resulting Church numeral into numbers that Haskell can print by making f the function to add 1, and $x = 0$:

```
> churchMul churchTwo churchThree ((+) 1) 0
6
> churchExp churchTwo churchThree ((+) 1) 0
8
> churchMul (churchMul churchTwo churchThree) (churchExp churchTwo churchThree) ((+) 1) 0
48
```

Or, we can recover something that looks a bit more like the λ-expression by building it as a string, with the `++` operator, which concatenates strings in Haskell:

```
> churchMul churchTwo churchThree (\x -> "(f " ++ x ++ ")") "x"
"(f (f (f (f (f (f x))))))"
```

Indeed, Haskell is based directly on the λ-calculus, with several of the additional constructs that we added in modules 3 and 4 (and, of course, many more). Haskell is typed, and as we saw in Module 4, typed languages cannot usually directly represent the Y combinator. Haskell is no exception:

```
> y = \f -> (\x -> f (x x)) (\x -> f (x x))

<interactive>:1:23: error:
    * Occurs check: cannot construct the infinite type: t0 ~ t0 -> t
      Expected type: t0 -> t
        Actual type: (t0 -> t) -> t
    * In the first argument of 'x', namely 'x'
      In the first argument of 'f', namely '(x x)'
      In the expression: f (x x)
    * Relevant bindings include
        x :: (t0 -> t) -> t (bound at <interactive>:1:13)
        f :: t -> t (bound at <interactive>:1:6)
        y :: (t -> t) -> t (bound at <interactive>:1:1)
```

Haskell messages are often arcane, but the critical part here is that it can't find a way for the type `t0` to be equivalent to the type `t0 -> t`; these `t` types are Haskell's τ. Recursion is possible because bindings work similarly to the `let` and `let rec` bindings we introduced in Module 4[3].

Haskell also has a shorthand for multi-argument functions. For instance, `\x y -> x + y` is a function to add its two arguments (η-reducible to the `(+)` function). This form is always equivalent to nested functions, such as `\x -> \y -> x + y`, so serves the purpose only of abbreviation. However, in Section 12, we'll see that the translation can be more complicated than this.

[3]There are also other fixed-point combinators than the Y combinator, and there are fixed-point combinators that Haskell *can* represent, but sensible binding is a better solution, so we won't discuss them here.

Video 5.3 (https://student.cs.uwaterloo.ca/~cs442/W25/videos/5.3/): Haskell functions

5 Conditionals and Predicates

Suppose we want to construct a function to compute the absolute value of a number x. The absolute value $|x|$ of x is defined as:

$$|x| = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{if } x < 0 \end{cases}$$

There is, of course, a built-in absolute value function, `abs`, which was demonstrated above; our implementation is unlikely to be as good as the built-in one, but will demonstrate conditions.

We can evaluate this using `if` conditionals. In languages in the C family, `if` is a statement, and the condition affects which statements will be executed; in functional languages, `if` is an expression, and evaluates to either its `then` or `else` branch. Because it's an expression, and because Haskell is typed, we can't just leave out the `else` branch: recall how in our `if` syntax in Module 4, we demanded that each branch have the same type. Well, the `else` branch can't have the same type as the `then` branch if there is no `else` branch, so the `else` branch is mandatory. In languages in the C family, the `?:` operator—sometimes simply called "the ternary operator", which is a dubious name choice since it excludes the possibility of any other operators with three operands—behaves the same, and requires an "else" branch for the same reason. Knowing this, let's write our absolute value function:

```
> absButTerrible = \x -> if x >= 0 then x else -x
> absButTerrible 42
42
> absButTerrible (-42)
42
```

An `if` expression is written (`if` p `then` t `else` e). p is called the *predicate*, or simply *condition*, and determines which of t and e will be evaluated. t and e are called the *then expression* and *else expression*, respectively.

Aside: You may be wondering why we've always put negative numbers inside of parentheses. One unfortunate consequence of Haskell's and OCaml's parenthesis-light function call syntax is that the numeric `-` for negation is almost always ambiguous. `absButTerrible -42` may look right to our eyes, but it's actually trying to subtract `42` from the value `absButTerrible`. You can't subtract a number from a function, so this doesn't work.

Haskell—and most typed functional languages—is picky about the type of the predicate. In C, for instance, we're allowed to use any number or pointer as the predicate, and it's considered to be true if it's not 0 or `NULL`. In Haskell, the predicate must be boolean:

```
> if 42 then 42 else -42

<interactive>:1:4: error:
    * Could not deduce (Num Bool) arising from the literal '42'
[...]
```

What this error is saying is that it couldn't find a way to interpret a `Num` (number) as a `Bool` (boolean).

Of course, since Haskell is based on the λ-calculus, we could also build our own booleans and conditions, in exactly the style that we built $[\![\texttt{true}]\!]$ and $[\![\texttt{false}]\!]$ in the λ-calculus:

```
> true = \x -> \y -> x
> false = \x -> \y -> y
> lambdaIf = \b -> \t -> \f -> b t f
> lambdaIf true "true is true" "true is false"
"true is true"
> lambdaIf false "false is true" "false is false"
"false is false"
```

This is quite pointless, since our λ-flavored `true` and `false` don't behave like Haskell's own true and false, which are named `True` and `False`:

```
> if true then "true is true" else "true is false"

<interactive>:1:4: error:
    * Couldn't match expected type 'Bool'
                  with actual type 'p10 -> p20 -> p10'
```

Precisely what we're doing is passing the $[\![\texttt{true}]\!]$ we defined in Module 2 to the `if` we defined in Module 3, which expects its own primitive types. The flexibility of Haskell in both representing most λ-expressions and all of the primitive extensions we've defined so far make it a useful platform for understanding formal semantics.

6 Guards

There is an alternative to `if` for conditionalizing execution: *guards*. Guards are little more than an abbreviation for nested `if` statements, but are common in functional languages, because nested `if`s can become difficult to understand. First, let's start by declaring our functions in a different way:

```
> add10 x = x + 10
```

Recall that previously we said this binding was actually asserting equivalence, rather than binding a variable. This style of binding should make that very clear: we are asserting that writing the expression `add10 x` is equivalent to writing the expression `x + 10`. As a practical matter, this declares a variable named `add10` bound to a function, but mathematically, the declaration is a statement of equivalence.

Recall now our mathematical description of absolute value:

$$|x| = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{if } x < 0 \end{cases}$$

The form to the right of $=$ is a description of *cases*, and Haskell, with a slightly different syntax, allows the same:

```
> :{
| absButTerrible x | x >= 0 = x
|                  | x < 0 = -x
| :}
>
```

This odd `:{ ... :}` syntax is just because the REPL doesn't normally let us do definitions of this form, so let's turn away from the REPL and write an actual Haskell file, which can simply be written like so:

```
absButTerrible x | x >= 0 = x
                 | x < 0 = -x
```

This can be read as "`absButTerrible x` is defined conditionally like so: if `x >= 0`, then `absButTerrible x` is `x`; if `x < 0`, then `absButTerrible x` is `-x`".

These guards can be written as nested `if`s. That brings us to the concept of *syntactic sugar*.

7 Syntactic Sugar

Until this point, we haven't needed to introduce any new semantics, because everything has followed either the semantics of the λ-calculus, or one of the additions we proposed in Module 3. But syntactically, the guards we've seen don't really fit either.

Formal semantics are almost always defined for a reduced *core language*. The core language is a much smaller version of a programming language that includes the semantically interesting parts while excluding many complex language features. Quite often, a formal semantics is trying to demonstrate only one language feature, in which case this core language may be sufficient. But, when a formal semantics is trying to accurately model a full language, rather than just a particular feature, it is associated with a translation—usually informal—from the full language

into the core represented by the formal model. In this case, we can describe it using the double square brackets[4] introduced in Module 2. We can thus describe patterns of the above sort as syntactic sugar like so:

Let the metavariables D range over valid function signatures (i.e., a name followed by formal arguments), C range over case lists, and E range over expressions. Then,

$$\begin{array}{rcll} [\![D\ \ C]\!] & = (& D = [\![C]\!] &) \\ [\![\texttt{|}\ \ E_1\ \texttt{=}\ E_2\ \ C]\!] & = (& \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ [\![C]\!] &) \\ [\![C]\!] = [\![\epsilon]\!] & = (& \texttt{error "Non-exhaustive pattern"} &) \end{array}$$

We haven't yet mentioned `error`, but it does what you probably suspect it does: raises an error if it's evaluated. In this case, it's there because case lists might not cover every possible case, and if they don't, an error is raised at run-time.

The first line in this relation states that a function declaration formed from case statements can be written as a standard function declaration formed with `=`, by rewriting the case statements. The second line says that a given case $E_1 = E_2$ can be rewritten as an if-then-else, where the else expression is the remaining cases. The last line says that the empty case list (i.e., the end of the case list) is an error if it's reached.

Note that this rewriting is semantically correct, but it's likely that the actual Haskell interpreter will do something cleverer. The important thing in semantics is that the *observable* behavior is the same, not that the behavior of the compiler is described precisely.

Using this definition, we would rewrite `absButTerrible` as follows:

```
absButTerrible x = if x >= 0 then x else if x < 0 then -x else error "Non-exhaustive pattern"
```

We'll use this sort of semi-formal rewriting of complex patterns into simple ones frequently to make our formal semantics more compact. In this case, thanks to this rewrite, we're yet to encounter anything that needs any extension to our formal semantics at all.

Exercise 1. Note that we've rewritten the case patterns in terms of the abbreviated syntax we described above (e.g. `absButTerrible x = ...` rather than `absButTerrible = \x -> ...`). Using a similar style, describe the translation of this syntax. If you're already familiar with Haskell and know what's coming in Section 12, then just define the simple case described here, not full pattern matching.

8 Name Binding and Environments

We've cavalierly used bindings above, without being very clear what they mean. Most languages have a *global scope*, in which all declarations are equal. In Scheme, for example, a variable defined with `define` is usable anywhere in the program, even in code written before the `define` itself, so long as the `define` is evaluated before the code using the definition is evaluated. However, there is also a way to bind names *contextually*.

An *environment* is essentially a lookup table, mapping identifier names to values. An environment also associates names with types, in which context it's called a type environment, which we've already seen. The environment forms our abstraction of the computer's memory, so that rather than dealing with pointers and addresses, we have names.

Environments can also nest, and nested environments don't need to have the same definition of any given name. This nesting causes environments to form a tree, with the global scope's environment—if any—as the root. The tree is a run-time phenomenon, as multiple calls to the same function create different environments, but its structure is described by the structure of the code: code in one environment can see variables declared in that environment, and in environments that surround it syntactically. We describe a variable in a given environment as *scoped* to that environment. This is called *lexical scoping*, and is how all modern languages, functional or otherwise, deal with name binding. In Haskell, an environment is declared by a `let...in` expression, which declares a new binding, but only within the context of the expression after `in`; the new environment is a child of the surrounding environment.

[4] Technically speaking, this is the syntax of denotational semantics, but we will be using it far too informally to describe this as a denotational semantics for full Haskell.

For instance, the following two functions both use a variable named `x`, but `x` is bound to different values in each context:

```
fun1 n =
    let x = "A" in n ++ "x is " ++ x

fun2 n =
    let x = "B" in n ++ "x is " ++ x
```

...

```
*Example> fun1 "In fun1, "
"In fun1, x is A"
*Example> fun2 "In fun2, "
"In fun2, x is B"
```

Technically, this example requires a module declaration, but we won't be discussing modules.

> **Aside:** Some people like to snidely deride certain modern languages, in particular JavaScript, as not having lexical scoping, but this is a misunderstanding of lexical scoping. Lexical scoping means that the environment in which variables are scoped can be discovered from only the syntax of the code, not that the environment in which a variable is scoped starts exactly from the text declaring the variable. In the case of JavaScript, variables declared with `var` are scoped to the environment of their surrounding function. Languages with non-lexical scoping include Common Lisp and the Unix shell.

With the exceptions of the `let` bindings above, all of the declarations we've made so far in this module have been in the global scope[5], which is a single, shared environment for all "global" variables. Because this environment is global and shared, recursion and even mutual recursion are easy to express:

```
badlyApproachZero x | x > 0 = badlyApproachZero (-x + 1)
                    | x < 0 = badlyApproachZeroPrime x
                    | x == 0 = x

badlyApproachZeroPrime x | x > 0 = badlyApproachZero x
                         | x < 0 = badlyApproachZeroPrime (-x - 1)
                         | x == 0 = x
```

In this example, `badlyApproachZero` and `badlyApproachZeroPrime` are mutually recursive, but can easily refer to each other since they reside in the same environment, the global scope.

We already looked at formally expressing the semantics of environments, with `let` bindings in Module 4. But, let's recall the semantic and type rules for one part of `let` bindings, to discuss them in the context of environments:

$$\textsc{LetBody}\ \frac{\sigma' = \sigma[x \mapsto V[z/x]] \qquad z \text{ is a fresh variable} \qquad \langle \sigma', M \rangle \rightarrow\ \langle \sigma', M' \rangle}{\langle \sigma, \text{let } x = V \text{ in } M \rangle \rightarrow\ \langle \sigma, \text{let } x = V \text{ in } M' \rangle}$$

$$\text{T\_Let}\ \frac{\Gamma \vdash V : \tau_1 \qquad \{\langle x, \tau_1 \rangle\} + \Gamma \vdash E : \tau_2}{\Gamma \vdash \text{let } x = V \text{ in } E : \tau_2}$$

Our store here, σ, is part of the formal semantics encoding of our *environment*. The other part is Γ, our type environment. Γ only exists in the type judgment, and σ only exists in the semantics, but they are each other's dual. Note how the environments are used: every expression is paired with its environment, and evaluated or judged in that environment. Subexpressions in which new variables have been defined are evaluated or judged in a new context, σ' or $\{\langle x, \tau_1 \rangle\} + \Gamma$, which only exists in that subexpression and its descendants. It is this association between the syntactic nesting of expressions and the nesting of environments that defines lexical scoping.

[5] In Haskell, it's actually a module scope, but again, we won't be discussing modules.

Aside: Not all stores form environments, or at least not all stores form lexically scoped environments. "Store" is just a general term for a variable-value map. Don't be surprised to see things called "store" that don't behave exactly like the store we're describing here. We'll see a very different store in Module 7.

Now, let's consider the case of the global scope. We cannot rewrite our `badlyApproachZero` example in terms of `let`s, or even in terms of `let rec`s[6], because `let rec` as we described it only let us make *one* recursive function, not a pair of mutually recursive functions. So, how do we express a global scope in this way?

To be honest, the usual answer is "we don't", or we only do so informally. Formal semantics describe the reduction of an expression, and it's taken as rote that we have some way of pre-populating σ with all of the variables defined globally, rather than the `empty` we started with while defining the `let rec` semantics in Module 4. But, let's do it formally anyway.

In most statically typed languages, even functional languages, there is actually a different syntax for the global scope than there is for expressions. In Haskell, you cannot simply write an expression in a file and expect it to work. In fact, in the global scope, all you can write is *declarations*. In a pure functional language, those declarations do not, in and of themselves, *do* anything, except for defining the global environment. You then need a starting expression to evaluate, which becomes the left part of the $\rightarrow$ in the first step of evaluation. In Haskell, in an unusual nod to C, the starting place is `main`. For a Haskell file to be usable as a Haskell program, it must define a function `main`, and to run the Haskell program is to evaluate `main`. There's a further caveat because of monads, which we will get to in Section 16, but for the time being, we can just assume we'll be evaluating `main` as a function.

So now, we have two steps: convert our top-level syntax into σ, and then evaluate `main` in σ. This conversion is the step that is rarely described formally, so there is no "usual" name for it; we will give it the simple name *resolve*. Thus, the starting point for a program P is the pair σ and E such that $\sigma = resolve(P)$, $E = \sigma(\texttt{main})\ z$, and z is the argument to the `main` function, such as C's `argv` and `argc`. We thus need a syntax for programs, and a definition for *resolve*; z comes from outside.

The syntax for programs is simply a list of declarations. Haskell is indentation-sensitive, which cannot be expressed in BNF because it does not form a context-free language. In order to make the language of our semantics clearer, and sidestep this problem, we will separate declarations by semicolons, and make a few similar syntactic concessions later.

$$\begin{aligned}
\langle Program \rangle &::= \langle DeclList \rangle \\
\langle DeclList \rangle &::= \langle Decl \rangle \texttt{ ; } \langle DeclList \rangle \\
&\quad\ \mid \epsilon \\
\langle Decl \rangle &::= \langle Var \rangle = \langle Term \rangle
\end{aligned}$$

Now, let's define *resolve* in terms of a program P, where x, V, and L are metavariables ranging over variables, terminal values, and declaration lists, respectively:

$$\textsc{EmptyProgram}\ \frac{}{resolve(\epsilon) = \texttt{empty}} \qquad \textsc{Declaration}\ \frac{\sigma = resolve(L) \qquad \sigma' = \sigma[x \mapsto V]}{resolve(x = V\ \texttt{;}\ L) = \sigma'}$$

An empty program of course defines no variables, so $resolve(\epsilon) = \texttt{empty}$. A program declaring multiple variables can be separated into a single declaration ($x = V$) and "the rest" (L), and *resolve* of that program simply extends the resolution of "the rest" and adds x. Technically, this means that our definition of *resolve* goes right-to-left. While a bit odd, this right-to-left resolution doesn't actually matter so long as names aren't repeated. It is typically the job of type judgment to assure that a name is not multiply defined, rather than formal semantics, and if a name is guaranteed not to be multiply defined, then a σ built left-to-right is indistinguishable from a σ built right-to-left.

A final wrinkle is that in most languages, these declarations may associate names with *expressions*, rather than just terminal values, and those expressions need to be evaluated. In Haskell, names are bound to expressions, but as we will see in Section 14, those expressions *do not* need to be evaluated. For the time being, we will assume they're just associated with values.

[6] OCaml has yet another special syntax for declaring mutually recursive functions like this, but it's not relevant here.

9 Algebraic Data Types

We imagined booleans as being built-in, primitive types in Haskell. This is true, but only because the built-in `if-then-else` syntax can only work if it knows what booleans are. Without that, we could write our own Haskell booleans like so:

```
data Bool = True | False
```

Remember that a type is just a set of values belonging to that type. Previously, we assumed that the actual definition of types and values were external to the language: we defined a language with numbers, and then said that "number" was a type; we defined a language with abstractions, and then said that "abstraction" was a type. Haskell's approach to both types and values is different.

This declaration introduces a new type to the language, *and* introduces two new values to the language. "`True`" and "`False`" needn't be keywords, built into the language; they are defined by this type declaration. This kind of type declaration is called *algebraic data types*, and is common in typed functional languages. This particular declaration enters two new names into our semantic environment, `True` and `False`, but what is the value associated with them?

In fact, it's not important to have an actual, concrete value. The values associated with `True` and `False` are called *symbols*, which are values invented solely to be unique: what is important is that we can, at run-time, know when we are referring to `True`, and know when we are referring to `False`, so all we need is some distinguishable value. You can imagine this value as being a number, if you'd like, with every value declared in this way having a unique number, but that would be a detail of the implementation, not the concept. As far as a user can tell, `True` is `True` and `False` is `False`, and that's all that matters. In our formal semantics, we will represent these symbol values as one-tuples in which the only value is the name itself, in braces; that is, the value of `True` is {`True`}. This approach is *very* loosely lifted from the paper A Static Semantics for Haskell[2], which I would highly advise not reading. In an actual implementation, one could simply generate numbers to represent every symbol, or allocate a small space and use the pointer as the symbol value; in either case, that number or pointer is never actually exposed to the user.

> **Aside:** Actually, there are many different ways of expressing such values in formal semantics as there are formal semantics for languages with algebraic data types, so don't necessarily expect to see symbols of this form everywhere.

In Haskell, `True` and `False` are called *data constructors*. The reason for this will become clearer momentarily, when we parameterize them.

Let's look at how we might introduce symbols to *resolve*:

$$
\begin{aligned}
\langle Decl \rangle &::= \cdots \mid \texttt{data}\ \ \langle Var \rangle = \langle ValueList \rangle \\
\langle ValueList \rangle &::= \langle DataConstr \rangle\ \langle ValueListRest \rangle \\
\langle ValueListRest \rangle &::= \text{“|”}\ \langle DataConstr \rangle\ \langle ValueListRest \rangle \\
&\quad \mid \epsilon \\
\langle DataConstr \rangle &::= \langle Var \rangle \\
\langle Term \rangle &::= \cdots \mid \{\langle Var \rangle\}
\end{aligned}
$$

$$
\textsc{EmptyDataDecl}\ \ \frac{}{resolve(\texttt{data}\ \ n = \epsilon\ \texttt{;}\ \ L) = resolve(L)}
$$

$$
\textsc{DataDecl}\ \ \frac{\sigma = resolve(\texttt{data}\ \ n = M\ \texttt{;}\ \ L) \qquad \sigma' = \sigma[N \mapsto \{N\}]}{resolve(\texttt{data}\ \ n = N\ M\ \texttt{;}\ \ L) = \sigma'}
$$

Note that in DataDecl, n names the *type* we are declaring, while N names the *value* we are declaring. Since the symbol value for N is $\{N\}$, N maps to $\{N\}$. Of course, it is also necessary to remember the name n during type checking, but we're only looking at semantics here. Also, as it happens, Haskell requires that type and data

constructor names are capitalized, and variable names are not; this is helpful for typing Haskell, but unnecessary for our purposes, so we will assume that all of these can have names of the same form.

The type declaration we just saw declares a *finite* type. The `Bool` type has exactly two values: `True` and `False`. Finite types aren't especially useful, so let's imagine we want a type for lists of booleans. Of course, there are infinitely many possible lists of booleans, so there must be an infinite number of values in this type. How can we express that with algebraic data types?

The answer is that data constructors can have *parameters*, and those parameters are other types. So, leaning on our classic definition of a list—a list is either empty, or a pair of a value and a list—we can build an algebraic data type which represents lists of booleans:

```
data BoolList = Empty | Pair Bool BoolList
```

The `Empty` constructor behaves exactly as `True` or `False` did: it is simply a symbol value, carrying no further information than the fact that `Empty` is `Empty`. On the other hand, `Pair` is not a `BoolList` in and of itself; instead, it is a two-argument function, where the first argument must be a boolean and the second argument must be another `BoolList`. The result of this function is a `Pair` value, associated with the two argument values. That is, `Pair` is a `Bool` → `BoolList` → `BoolList`. For instance, `Pair` refers to a function, but `Pair True Empty` refers to a pair. All values of algebraic data types are effectively n-tuples, where $n-1$ is the number of arguments to the data constructor, and the first element in the tuple is a symbol representing the constructor itself. Therefore, our declaration above gives us a way of making a pair of a boolean and boolean list, and "tagging" that pair so that we can tell later that it was specifically one of our `Pair`s, and not some other pair.

With parameterized data constructors, we gain the ability to describe new, infinite data types. There are an infinite number of boolean lists.

Now, let's look at how we might introduce parameterized data constructors to *resolve*:

$$\langle DataConstr\rangle ::= \cdots \mid \langle Var\rangle \langle Var\rangle^+$$
$$\langle Term\rangle ::= \cdots \mid \{\langle Var\rangle (, \langle Term\rangle)^+\}$$

ParamDataDecl

$$\frac{\sigma = resolve(\texttt{data}\ \ n = M\ \texttt{;}\ \ L) \qquad \sigma' = \sigma[N \mapsto \texttt{\textbackslash}x_1\ \texttt{->}\ \texttt{\textbackslash}x_2\ \texttt{->}\ \cdots\ \texttt{\textbackslash}x_m\ \texttt{->}\ \{N, x_1, x_2, \cdots, x_m\}]}{resolve(\texttt{data}\ \ n = N\ T_1\ T_2\ \cdots T_m\ \ M\ \texttt{;}\ \ L) = \sigma'}$$

The use of $\cdots$ here to represent repeating expressions (in this case, twice in the second premise and once in the conclusion) is a common abbreviation in formal semantics, to avoid needing to break it down into each of its steps. Here, ParamDataDecl shows that an m-argument data constructor is stored in σ' as an m-argument λ function, curried of course, which resolves to the tuple described above. Note that this definition allows *any* value to occupy each of the slots of our algebraic data type; it is the role of type checking to assure that the values are actually of the types declared by T_1 through T_m, which is why they don't appear in σ'.

The boolean list containing `True` then `False` with this declaration is written `Pair True (Pair False Empty)`.

Since data constructors are functions, and functions are curried, we can extract partial applications:

```
PrependTrue = Pair True
TrueFalseList = PrependTrue (Pair False Empty)
```

Video 5.4 (https://student.cs.uwaterloo.ca/~cs442/W25/videos/5.4/): ADTs

Finally, as Haskell's pièce de résistance, algebraic type declarations *themselves* may be parameterized by type, allowing for this general definition for lists of any type:

```
data List a = Empty | Pair a (List a)
```

This only affects types, and not semantics, so we won't show our *resolve* extended to make it work: `Pair` is the same two-argument function.

Aside: Haskell, of course, natively supports lists, so there's no need to explicitly declare a list type like we just have. However, the native list syntax is nothing more than syntactic sugar for a sequence of pairs like this.

10 Parametric Polymorphism and System F

As yet, we've skirted around the issue of polymorphism, but remember that in Module 4, we declared polymorphism as necessary to express even Church numerals. Virtually all typed functional languages, certainly including Haskell, use *parametric polymorphism*. What makes parametric polymorphism so parametric? The fact that types may be parameterized, like functions.

This is formalized in a calculus called System F, also known as the second-order λ-calculus, which was invented independently by Girard[3] and Reynolds[8]. System F looks much like the simply-typed λ-calculus, but includes syntax for abstractions and applications over *types*:

$$
\begin{aligned}
\langle Expr \rangle &::= \langle Var \rangle \mid \langle Abs \rangle \mid \langle App \rangle \mid \langle t\text{-}Abs \rangle \mid \langle t\text{-}App \rangle \\
\langle Var \rangle &::= a \mid b \mid c \mid \cdots \\
\langle Abs \rangle &::= \lambda \langle Var \rangle : \langle Type \rangle . \langle Expr \rangle \\
\langle App \rangle &::= \langle Expr \rangle \langle Expr \rangle \\
\langle t\text{-}Abs \rangle &::= \Lambda \langle TyVar \rangle . \langle Expr \rangle \\
\langle t\text{-}App \rangle &::= \langle Expr \rangle \{ \langle Type \rangle \} \\
\langle Type \rangle &::= \langle PrimType \rangle \mid \langle TyVar \rangle \mid \langle Type \rangle \rightarrow \langle Type \rangle \mid \forall \langle TyVar \rangle . \langle Type \rangle \\
\langle PrimType \rangle &::= t_1 \mid t_2 \mid t_3 \mid \cdots \\
\langle TyVar \rangle &::= \alpha \mid \beta \mid \gamma \mid \cdots
\end{aligned}
$$

Λ is a Greek capital lambda. In System F, Λ behaves like a second level of λ, over types, hence the name second-order λ-calculus. To distinguish applications in the first language from applications in the second, the second-order language uses braces ($\{\}$) for application, but furthermore, only allows applications of explicitly written types. As a consequence, the second-order language is not Turing-complete, since it is impossible to represent recursion. This is actually a good thing, as it means that the halting problem does not haunt our types, and we know that we can always evaluate the second-order component of System F. Otherwise, the second-order language is just λ-calculus, and uses the same reduction rules. For actually "running" System F, that's all there is to it—just reduce the *t-App* if you encounter a *t-App*, and reduce the *App* if you encounter an *App*. Everything interesting about System F comes with types.

Since types can be substituted (by reducing *t-App*), the type system in the simply-typed λ-calculus is insufficient to describe the types and type behavior of System F. In order to have a language to express the types described by Λ, System F expands $\langle Type \rangle$ to include the universal quantifier. For instance, it is now possible for an expression to have the type $\forall \alpha . \alpha \rightarrow \alpha$, which is a function which accepts any type, and returns a value of the same type. Note that α is not a type itself, but a type variable: any particular resolution of this function will have a concrete type, but it may be a different concrete type in different contexts.

In the previous module, we explored the difficulty in expressing Church numerals in the simply-typed λ calculus. We attempted to give Church numerals the type $(t_1 \rightarrow t_1) \rightarrow t_1 \rightarrow t_1$, but found that our definition of $[\![\hat{}]\!]$ didn't work, because it tried to make $t_1 \rightarrow t_1$ equal to $(t_1 \rightarrow t_1) \rightarrow t_1 \rightarrow t_1$. We informally fixed this by making the type $\forall \tau_1 . (\tau_1 \rightarrow \tau_1) \rightarrow \tau_1 \rightarrow \tau_1$, and allowing different Church numerals to have different τ_1s. Now that we can explicitly make type parameters, rather than informally talking about τs, how would we describe a Church numeral?

A Church numeral needed only one core type, t_1, so that is the type we must parameterize. We will simply wrap the definition of a Church numeral, in this case two, in a second-order Λ which makes that a parameter:

$$[\![2]\!] = \Lambda \alpha . \lambda f : \alpha \rightarrow \alpha . \lambda x : \alpha . f(fx)$$

We can recover our original, half-broken Church numerals over t_1 by applying t_1 as our type parameter:

$$\begin{aligned} & (\Lambda\alpha.\lambda f : \alpha \rightarrow \alpha.\lambda x : \alpha.f(fx))\{t_1\} \\ \rightarrow \quad & \lambda f : t_1 \rightarrow t_1.\lambda x : t_1.f(fx) \end{aligned}$$

And now, we can make a version of $[\![\hat{}]\!]$ that is itself parameterized by type:

$$[\![\hat{}]\!] =$$

$$\Lambda\alpha.\lambda m : (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha.\, \lambda n : ((\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha.\, \lambda f : \alpha \rightarrow \alpha.\, \lambda x : \alpha.\, nmfx$$

The types in this expression are quite intimidating, so let's break them down to make things easier. A Church numeral is a function which takes two arguments. The first is a function over the second, which will be applied as many times as the value of the church numeral. A two-argument function is a function returning a function, so it's some $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$. In this case, τ_1 has to be a function over τ_2, so the Church numeral is $(\tau_2 \rightarrow \tau_2) \rightarrow \tau_2 \rightarrow \tau_3$. And finally, since the Church numeral returns the result of the first argument, τ_3 must equal τ_2, so the Church numeral is $(\tau_2 \rightarrow \tau_2) \rightarrow \tau_2 \rightarrow \tau_2$. We thus need only one type argument, τ_2, which we will name α. That explains the type of m. Because we want to be able to choose the real type for α, we parameterize it, which is $\Lambda\alpha$. The result is a function in the second order (Λ) which takes a type as its argument (α), and produces a Church numeral that works with that type, whatever it is.

How about the odd type of n? In $[\![\hat{}]\!]$, n takes m as an argument, but n must also be a Church numeral. So, n must be some $(\tau_4 \rightarrow \tau_4) \rightarrow \tau_4 \rightarrow \tau_4$, but it *also* must accept an argument of m's type, which is $(\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$. That is, $\tau_4 \rightarrow \tau_4 = (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$. We can add some parentheses to the type of m to get $(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$, at which point the relationship between α and τ_4 is clear: $\tau_4 = \alpha \rightarrow \alpha$. So, the type of n is the rather long-winded $((\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$.

To actually *use* the System F variety of Church numerals, we need to fill in all the types in a compatible way, e.g.:

$$[\![\hat{}]\!]\ \{t_1\}(\ [\![\mathsf{2}]\!]\ \{t_1\})(\ [\![\mathsf{2}]\!]\ \{t_1 \rightarrow t_1\})$$

The power of System F's second-order language is that we can instead make the above expression *itself* polymorphic, by replacing t_1 with some parameterized type, such as γ (gamma):

$$\Lambda\gamma.\ [\![\hat{}]\!]\ \{\gamma\}(\ [\![\mathsf{2}]\!]\ \{\gamma\})(\ [\![\mathsf{2}]\!]\ \{\gamma \rightarrow \gamma\})$$

A second, less obvious power of System F is that the second-order language of types is *erasable*. Because type-level substitutions only happen in the type language, and normal λ substitutions only happen in the λ-calculus portion, the actual behavior of any System F expression is dictated by only its λ-calculus component. You can remove all Λ-abstractions and Λ-applications without affecting the meaning of any program. Thus, the only purpose of the type language is for type checking; the concerns of parametric polymorphic types and actual behavior are separated. More precisely:

Definition 2. (Erasure) Given a term E in System F, we define the erasure of E, denoted $erase(E)$, as follows:

$$\begin{aligned} erase(x) = \quad & x \\ erase(\lambda x : \tau.E) = \quad & \lambda x.erase(E) \\ erase(MN) = \quad & erase(M)\ erase(N) \\ erase(\Lambda\alpha.E) = \quad & erase(E) \\ erase(E\{\tau\}) = \quad & erase(E) \end{aligned}$$

Theorem 1. (Erasability) If $E \rightarrow E'$, then either $erase(E) = erase(E')$, or $erase(E) \rightarrow_\beta erase(E')$, up to α-renaming.

The type rules for the System F consist of the rules for the simply-typed λ-calculus, augmented with two new rules to treat type abstractions and type applications. The rule for type abstractions is as follows:

$$\text{T\_TypeAbs}\ \frac{\Gamma \vdash E : \tau \qquad \forall x.\, \langle x, \alpha \rangle \notin \Gamma}{\Gamma \vdash \Lambda\alpha.E : \forall\alpha.\tau}$$

There are several things to note about this rule:

- All Λ-abstractions are of universal types.
- τ may contain α, and indeed, the type of $\Lambda\alpha.E$ could be as simple as $\forall\alpha.\alpha$. In this example, E is directly of the parameterized type.
- α doesn't *have* a type, it *is* a type, so nothing is added to Γ. Γ maps variables to types; it does not map type variables to anything.
- The second premise demands that nothing in the environment has type α already. In particular, this requirement prevents us from constructing ill-formed terms like $\Lambda\alpha.\lambda x : \alpha.\, \Lambda\alpha.x$, in which the inner "$\Lambda\alpha$" attempts to capture the type of x (i.e., the outer α) and thereby divorce x's type from that of α's binding occurrence.

For type language applications, the rule is as follows:

$$\text{T\_TypeApp}\ \frac{\Gamma \vdash E : \forall\alpha.\tau_1}{\Gamma \vdash E\{\tau_2\} : \tau_1[\tau_2/\alpha]}$$

The expression $\tau_1[\tau_2/\alpha]$ is a *type substitution*, and behaves precisely as substitution does over the λ-calculus. In this case, it denotes the replacement of all (free) occurrences of α in τ_1 with τ_2. Its formal definition is analogous to that of substitution for λ-terms. In particular, note that type substitutions are capture-avoiding replacements, in which α-conversions of type variables are performed as needed to prevent capture.

In fact, the type rule for type application is most similar to the *semantic* rule for λ-applications. But, because recursion is syntactically disallowed in the type language of System F, type judgment is guaranteed to terminate.

Exercise 2. Using the type rules for System F, derive the type for $[\![\hat{}]\!]$.

In spite of the fact that System F's type language is erasable, it nonetheless guarantees type soundness, i.e. progress and preservation, which we won't prove here. It is also more powerful than the simply-typed λ-calculus, as demonstrated by the fact that it can successfully represent $[\![\hat{}]\!]$. However, System F still cannot represent the Y combinator, is still strongly normalizing, and is still not Turing-complete. This isn't a problem, since we already had a solution to strong normalization (`let rec`), and this solution works just as well in System F. All we need is an environment that allows self-recursive functions to recover Turing-completeness.

Typed functional languages, including Haskell, base their type language on System F, and this is how parameterized types, like `List a` we defined above, are possible. While `Pair` is a data constructor, and may be used only in the first-level language, `List` is a *type constructor*—i.e., a Λ-abstraction—and may be used only in the type language.

Having to explicitly write type constructors, type applications, and types more generally, is extremely tedious, so most typed functional languages, including Haskell, use some form of *type inference*.

11 Type Inference

Given that the type language of System F is erasable, one might reasonably ask if this erasure can be undone. That is, is it possible to write untyped λ-calculus, *infer* what its types should be, and thus gain type judgment without having to write types? The answer is yes! ... sort of. *Type inference* is the reverse of type erasure:

Definition 3. (Polymorphic Type Inference for System F) For a term E of the untyped λ-calculus, the polymorphic type inference problem (in the context of System F) is to find a well-typed term E' in System F, such that $erase(E') = E$.

As it turns out, in System F, type inference is undecidable. But, we can infer types if we restrict ourselves to inferring a slight restriction of all of the types that System F can represent. Specifically, we will restrict ourselves to inferring type expressions in which the universal quantifier may only be applied to the entire type expression, and not to some subexpression. That is, $\forall\alpha.\alpha \rightarrow \alpha$ is allowed, but $t_1 \rightarrow \forall\alpha.\alpha$ is not, and neither is $(\forall\alpha.\alpha) \rightarrow t_1$. The consequence of this restriction is that we may make functions themselves polymorphic, but a function cannot take an unresolved polymorphic function as its argument, nor can it return an unresolved polymorphic function. This restricts the set of expressions for which we can infer types, but, as it happens, still allows virtually all useful functions.

Let's consider what it means to infer types. We are operating over the untyped λ-calculus, which of course has no type judgment. We would like to find System F–like types for the λ-expressions. We have a typing judgment for System F, *and* we have a function, *erase*, which converts System F expressions to untyped λ-calculus expressions. A logical starting point, therefore, is to apply *erase* to the System F expressions in the typing rules, thereby getting rules that are defined in terms of the untyped λ-calculus. It is then up to us to determine if those rules make sense or help us[7]. This process of erasing chunks of type judgments isn't a formal process so much as an experiment, so we'll represent it with a squiggly arrow ($\rightsquigarrow$). We'll start with T_TypeAbs and T_TypeApp:

$$erase(\text{T\_TypeAbs})\ \frac{\Gamma \vdash erase(E) : \tau \qquad \forall x.\, \langle x, \alpha\rangle \notin \Gamma}{\Gamma \vdash erase(\Lambda\alpha.E) : \forall\alpha.\tau} \quad \rightsquigarrow \quad \text{T\_Gen}\ \frac{\Gamma \vdash E : \tau \qquad \forall x.\, \langle x, \alpha\rangle \notin \Gamma}{\Gamma \vdash E : \forall\alpha.\tau}$$

$$erase(\text{T\_TypeApp})\ \frac{\Gamma \vdash erase(E) : \forall\alpha.\tau_1}{\Gamma \vdash erase(E\{\tau_2\}) : \tau_1[\tau_2/\alpha]} \quad \rightsquigarrow \quad \text{T\_Spec}\ \frac{\Gamma \vdash E : \forall\alpha.\tau_1}{\Gamma \vdash E : \tau_1[\tau_2/\alpha]}$$

Note that once we erase the type language portions of the expressions, the expressions in the premises and conclusions of these two rules become the same! The T_Spec rule, called *specialization*, says that if E is of a polymorphic type $\forall\alpha.\tau_1$, then we can substitute α for some τ_2. With the goal of undoing erasure, this is certainly true: we could rewrite E as $E\{\tau_2\}$, which erases to E. Note that T_Spec doesn't actually specify τ_2—it doesn't appear in the premises—and so this judgment is true for *any* τ_2. Of course, E would never be of a polymorphic type, except that we've also created T_Gen.

The T_Gen rule, called *generalization*, says that if E is of type τ, then we can instead view it as a polymorphic type $\forall\alpha.\tau$. Again, with the goal of undoing erasure, this is certainly true: we could rewrite E as $\Lambda\alpha.E$, which erases to E.

These rules allow us to generalize or specialize types, but we still need a way of getting types in the first place. If we apply erasure to the T_Abstraction rule from Module 4, the mystery is solved:

$$erase(\text{T\_Abstraction})\ \frac{\{\langle x, \tau_1\rangle\} + \Gamma \vdash erase(E) : \tau_2}{\Gamma \vdash erase(\lambda x : \tau_1.\, E) : \tau_1 \rightarrow \tau_2} \quad \rightsquigarrow \quad \text{T\_Ascr}\ \frac{\{\langle x, \tau_1\rangle\} + \Gamma \vdash E : \tau_2}{\Gamma \vdash (\lambda x.\, E) : \tau_1 \rightarrow \tau_2}$$

The T_Ascr rule, type ascription, allows us to "invent" a type for the variable of an abstraction; note how τ_1 is not actually restricted, except insofar as E's types work with x assigned the type τ_1. Again, we could rewrite $\lambda x.\, E$ as $\lambda x : \tau_1.\, E$, which erases to $\lambda x.\, E$.

Erased versions of the rules for T_Variable and T_Application are uninteresting, as they had no explicit types to erase anyway. We will use them verbatim.

What we have just derived are the type rules for the *polymorphic λ-calculus*. Syntactically, the polymorphic λ-calculus *is* the untyped λ-calculus. It is distinguished only in having types, even if they're not written. Thus, the polymorphic λ-calculus is an example of a statically typed language in which types are not written.

[7] Of course, these are course notes, so you can probably guess that this process will be fruitful.

Our new set of type rules is odd. Our first observation is that the type rules for the simply-typed λ-calculus possess a property that our new set of type rules lacks. This property is known as *syntax-directedness*. A formal semantic system is *syntax-directed* if it has the following two properties:

- There is no more than one rule in the system for each expression in the language.
- The semantics of an expression is defined completely in terms of the semantics of its immediate subexpressions.

Because the type rules for the simply-typed λ-calculus are syntax-directed, they possess a useful second property: they're deterministic. That is, for a given expression, the type rules can only be applied in one way. Based on the syntactic structure of an expression E, we can determine the unique rule that matches E's structure. We apply this rule and then recurse on the subexpressions. Thus, type derivation in the simply-typed λ-calculus is a completely mechanical process.

The type rules for the polymorphic λ-calculus, on the other hand, are not syntax-directed. For a given expression E, there can be as many as three rules that we can apply: the rule based on the syntactic structure of E, as well as the rules for specialization and generalization. Worse yet, both of those rules, as well as the rule for type ascription, can be applied in infinitely many different ways, for every possible type. Because these type rules are not syntax-directed, using them to derive types is complicated. But, because the types have not been written into the underlying syntax, to derive types with these rules *is* to infer types. Luckily, there is a well-known algorithm, described and proved by Milner and Damas[1], that can perform type inference in the polymorphic λ-calculus, known as Hindley-Milner type inference[8]. We now focus our attention on their algorithm.

11.1 Unification

Consider how we informally resolved the types for Church numerals, and more specifically, for $[\![\hat{}]\!]$. We derived that m and n must be of some types $(\tau_2 \to \tau_2) \to \tau_2 \to \tau_2$ and $(\tau_3 \to \tau_3) \to \tau_3 \to \tau_3$, respectively. And, since $[\![\hat{}]\!]$ includes the expression nm, m's type must be n's argument type; i.e., $\tau_3 \to \tau_3 = (\tau_2 \to \tau_2) \to \tau_2 \to \tau_2$. This relationship is true if $\tau_3 = \tau_2 \to \tau_2$. This process of discovering the relationship between abstract types is called *unification*, and is the heart of type inference.

Because the type rules for the polymorphic λ-calculus are not syntax-directed, rather than build the type of an expression directly from the types of its immediate subexpressions, Milner's algorithm uses type variables to accumulate incomplete type information about the immediate subexpressions. The algorithm treats the incomplete type information gathered from the subexpressions as a set of equations to be solved. It then solves the equations, thus obtaining a type for the original expression.

The first step to understanding Milner's algorithm is to consider the problem of solving type equations. To solve type equations is to relate types, as we did with Church numerals above, and the algorithm to do so is known as *Robinson's Unification Algorithm*[9].

Our adaptation of Robinson's unification algorithm works on type expressions with unquantified type variables. Solutions of type equations are expressed in terms of substitutions in type expressions. We first mentioned type substitutions in our description of System F, in the type rule for specialization. Because we consider only unquantified type expressions here, the expression $\tau_1[\tau_2/\alpha]$ simply denotes the type τ_1 with all instances of α replaced by τ_2; that is, the problem of capturing in substitution that we had in Module 2 does not occur. A substitution may have any number of steps $[\tau_n/\alpha_m]$, including zero, so we represent an empty substitution as $[]$.

The solution of the type equation $\tau_1 = \tau_2$ is called a *unifier*:

Definition 4. (Unifier) Given types τ_1 and τ_2, a unifier of τ_1 and τ_2 is a substitution S such that $\tau_1 S = \tau_2 S$.

Consider, for example, the types $\tau_1 = \alpha \to (\beta \to \alpha)$ and $\tau_2 = \alpha \to (\text{nat} \to \alpha)$, assuming we have the "nat" primitive type. By setting both α and β to "nat", we specialize both τ_1 and τ_2 to $\text{nat} \to (\text{nat} \to \text{nat})$. Thus, the substitution $[\text{nat}/\alpha][\text{nat}/\beta]$ is a unifier of τ_1 and τ_2. On the other hand, if we only set β to nat, then τ_1 and τ_2 both specialize to $\alpha \to (\text{nat} \to \alpha)$. Thus, $[\text{nat}/\beta]$ is also a unifier of τ_1 and τ_2.

[8] An algorithm described by Milner and Damas is called Hindley-Milner type inference because the world is cruel. It is sometimes, but rarely, instead called Damas–Hindley–Milner type inference.

It is likely that the latter unifier is more useful, as it leaves the type more open to further unification, by leaving α. In general, we prefer unifiers that constrain the specialization of type variables as little as possible, and in particular, we seek a *Most General Unifier*:

Definition 5. (Most General Unifier) Let τ_1 and τ_2 be type expressions. A substitution S is a Most General Unifier (MGU) of τ_1 and τ_2 if S is a unifier of τ_1 and τ_2, and for every unifier S' of τ_1 and τ_2, there exists a substitution S'' such that $S' = S'' \circ S$.

It is a fact that if τ_1 and τ_2 can be unified (that is, if they possess a unifier), then they possess an MGU. The MGUs of a given pair of expressions are equivalent to one another, up to renaming of variables. Robinson's unification algorithm takes two type expressions—like τ_3 and $\tau_2 \rightarrow \tau_2$—as parameters, and returns an MGU for them, if one exists.

Before we look at the algorithm, let's take a moment to recall the space we're operating in. We have several kinds of types: primitive types, arrow types, universal types, polymorphic types, and abstract types.

- Primitive types are types such as t_1 and "nat". Unification cannot change primitive types, because the code will not function with a different type.
- Arrow types are constructed types with a $\rightarrow$, i.e., function types. We haven't previously called them arrow types, but as we now have two kinds of constructed types, it is useful to distinguish them. Unification cannot substitute an arrow type for a non-arrow type—functions will still be functions!—but may be able to substitute one or both sides of the arrow.
- Universal types are constructed types with a $\forall$, i.e., universally quantified types. If an expression has type $\forall\alpha.\alpha \rightarrow \alpha$, that means that the expression is a function which takes a value of any type and returns a value of the same type. Unification won't actually interact with universal types; this is the restriction we made at the beginning of this section. Instead, *after* unification, we will generalize using T_Gen, which produces universal types.
- Polymorphic types—i.e., type variables—are the parameter of type abstractions, and the operand of universal types, such as α. All polymorphic types are defined by some surrounding universal quantifier such as $\forall\alpha.\alpha \rightarrow \alpha$. Our goal in unification is to discover specializations necessary to unify polymorphic types with any of the above types.
- Abstract types are types such as τ_1, and they do not actually exist in the language. Any given abstract type is some real type in the unification algorithm, and we write the unification algorithm with an abstract type to indicate that a step works (or fails) with any type.

Unification is defined as follows:

Definition 6. (Unification) The unification algorithm, U, takes as input type expressions τ_1 and τ_2, and returns a most general unifier for them, if one exists, as follows:

U_PrimSelf	$U(t,t) = []$ (i.e., the empty substitution)
U_PrimErr	$U(t_1,t_2) = \textbf{error}(t_1 \neq t_2)$ (cannot unify different primitive types)
U_PrimArrow	$U(t, \tau_1 \rightarrow \tau_2) = \textbf{error}$ (cannot unify non-function with function)
U_ArrowArrow	$U(\tau_1 \rightarrow \tau_2, \tau_3 \rightarrow \tau_4) = S_1S_2$ where $S_1 = U(\tau_1,\tau_3), S_2 = U(\tau_2S_1, \tau_4S_1)$
U_Spec	$U(\alpha,\tau) = \begin{cases} \textbf{error} & \text{if } \tau \text{ is a constructed type and } \alpha \text{ appears in } \tau \\ [\tau/\alpha] & \text{otherwise} \end{cases}$
U_Comm	$U(\tau_1,\tau_2) = U(\tau_2,\tau_1)$ (U is commutative)

Unifying a primitive type with itself produces a trivial substitution (U_PrimSelf). Naturally, as $t = t$ in the first place, no substitution is needed to relate the two.

A primitive type cannot be unified with a different primitive type (U_PrimErr, for example, integers and strings cannot be unified), or with a function type (U_PrimArrow).

To unify two function types (U_ArrowArrow), we first unify their parameter types, obtaining a substitution S_1. We then apply S_1 to the result types of the two expressions, and then unify the result types, obtaining a substitution S_2. The result is the composition of S_1 and S_2. Note that the application of S_1 to τ_2 and τ_4 is necessary to ensure that any type information obtained during unification of τ_1 and τ_3 is enforced during unification of τ_2 and τ_4. This process could equivalently be done in the opposite order; it is unnecessary to unify the parameter types before the return types, only to assure that the substitution from one is enforced in the other.

To unify a type variable α with any type expression τ (U_Spec), we simply return the substitution that replaces α with τ, with one exception: if τ contains α, but is not equal to α, then the unification is circular and must fail (for example, the type equation $\alpha = \alpha \rightarrow \alpha$ is circular and has no solution). This safeguard against circularity is known as the occurs-check. Note that U_Spec also implies that $U(\alpha, \alpha)$, i.e., the unification of a type variable with itself, is $[\alpha/\alpha]$. This substitution is of course pointless, so practical implementations will instead produce [].

Finally, U is commutative (U_Comm), so, for instance, $U(\tau_1 \rightarrow \tau_2, t)$ can be reversed to fit U_PrimArrow. When taking unification as an algorithm, rather than just a mathematical description, it's important to check each case before swapping the arguments, to avoid an infinite recursion, of course.

11.2 Algorithm W

With unification, we may finally describe the polymorphic type inference algorithm of Milner and Damas itself: Algorithm W.

Algorithm W, or simply W, forms the basis for the type system of Haskell and many other typed functional languages, including OCaml. W takes as input a type environment and an expression, and returns a substitution and the type of the expression in the context of that substitution. Note that this substitution applies to the type environment; substitution over a type environment is simply substitution over each of the types in the type environment.

Algorithm W for the polymorphic λ-calculus is outlined here:

Definition 7. (Algorithm W) Algorithm W takes as input a type environment Γ and an expression E, and returns a pair of a substitution and a type expression.

W_Var
$$W(\Gamma, x) = \langle [], \tau \rangle \text{ where } \tau = \Gamma(x)$$

W_Abs
$$W(\Gamma, (\lambda x.\, E)) = \langle S, (\alpha S) \rightarrow \tau \rangle$$
where α is a new type variable,
$\langle S, \tau \rangle = W(\{\langle x, \alpha \rangle\} + \Gamma, E)$

W_App
$$W(\Gamma, (E_1 E_2)) = \langle S_1 S_2 S_3, \alpha S_3 \rangle$$
where $\langle S_1, \tau_1 \rangle = W(\Gamma, E_1)$,
$\langle S_2, \tau_2 \rangle = W(\Gamma S_1, E_2)$,
α is a new type variable
$S_3 = U(\tau_1 S_2, \tau_2 \rightarrow \alpha)$,

Once W has terminated, any type variables introduced by W that remain in the type returned by the algorithm are then universally quantified by T_Gen.

We see that W works by decomposing the given expression according to its structure, typing the constituent parts, and then using unification and composition to construct a type for the original expression. W is syntax-directed: there is exactly one case for each production of abstract syntax in the polymorphic λ-calculus, and the

type of each expression is built directly from the types of its immediate subexpressions. We can therefore examine the operation of W case-by-case.

Finding the type of a variable is trivial (W_Var); we simply look it up in the environment Γ. As we did not specialize any type variables, we return the empty substitution.

To find the type of an abstraction (W_Abs), we invent a new type variable α through T_Ascr, and bind the parameter variable x to α in Γ while typing the body E of the abstraction. During the typing of E, the variable α may be specialized to a more specific type. If it is, the specialization will form part of the substitution S that is returned, replacing the invented α. Thus, we complete the typing of $\lambda x.\, E$ by applying S to α and using the resulting type as the parameter component of the type we return. The result component of the type is τ, the type returned by the typing of E. We also pass up the substitution S generated by the recursive invocation.

To find the type of an application (W_App), we first type the rator E_1, obtaining a type τ_1 and a substitution S_1. We then type the rand E_2 using the updated environment ΓS_1 (to ensure that any type information gathered in the typing of E_1 is enforced during typing of E_2), obtaining the type τ_2 and the substitution S_2. Because we are typing an application, E_1 must be a function type, the parameter component of which must match the type of E_2. Thus, we now unify $\tau_1 S_2$ with $\tau_2 \rightarrow\ \alpha$, where α is a new type variable (we apply S_2 to τ_1 to ensure that both expressions being unified reflect the information obtained during both recursive applications of W). The type we return is α, updated by application of S_3 to reflect the type constraints enforced by the unification. The substitution we return is the composition of S_1, S_2, and S_3. Note that application is the only step which requires the unification algorithm, U.

11.3 Type Inference in Practice

Real languages which use type inference, such as Haskell and OCaml, always allow types to be specified explicitly as well. This is both for documentation and because polymorphic types sometimes do not express the user's actual intent; they can easily be *too* polymorphic.

Type inference allows programs to be fully statically typed, even if few or no types are written. However, it is distinct from dynamic typing, because types are reasoned about from the code alone, and distinct from gradual and optional typing (if you happen to be familiar with these concepts), because every expression is given a precise type. Type inference does not weaken type checking, but allows sophisticated types to be used without having to write them, or often, even work them out.

The downside of type inference is that error messages can be incomprehensible, even to those familiar with type inference. However, knowing the concepts of type inference make reading such type errors easier. With a knowledge of type inference, you're recommended to go back to Section 4 and read the error message that Haskell produced when we attempted to type the Y combinator.

Exercise 3. Extend W for let bindings as defined in Module 4.

Exercise 4. Extend W to support optional explicit type specification, i.e., simply-typed λ-calculus abstractions in the polymorphic λ-calculus.

12 Pattern Matching

Now that we have both algebraic data types and parametric polymorphism, we can discuss one of the fundamental features of typed functional languages: pattern matching.

Consider our `List` type from Section 9:

```
data List a = Empty | Pair a (List a)
```

The types are erasable[9], so setting aside types, this is a definition of two data constructors, `Empty` and `Pair`, of

[9] Actually, Haskell lets you have, for instance, two different definitions of `Pair` that apply to different types, and needs types to disambiguate them, so types are only fully erasable if all names are unique. We will discuss this problem in a different context in Module 8, so we'll ignore it for now and assume that all names are unique.

which the latter takes two arguments. But, data constructors in our semantics generate symbols, and symbols aren't actually part of the written language, just part of our semantics. Indeed, the tuples that we generate in our semantics don't even have any operations to unpack them; they only exist as values. So, how would we actually use a `List` in practice?

The answer is that there is a construct similar to the guards we saw in Section 6, but much more powerful: pattern matching. All typed languages with ADTs[10] support pattern matching, to avoid exposing symbols as a construct in the language. Untyped functional languages such as Scheme/Racket usually expose symbols, but often have pattern matching as well.

In Haskell, we can match based on a list like so:

```
sumList l =
    case l of
        Empty -> 0
        Pair x y -> x + sumList y
```

There's a lot to unpack in the "`case`" construction. Each line associates a data constructor with an expression, but it does even more: the data constructor can have unbound variables, and implicit `let` bindings are built to bind them. So, rewritten as a nested if statement, this would look something like

```
sumList = \l ->
    if (l[0] = #Empty) then 0
    else if (l[0] = #Pair) then
        let x = l[1] in
        let y = l[2] in
            x + sumList y
```

But, this syntax is purely fictional. Both `l[n]`, to extract a value from the tuple we're using in our semantics to represent ADTs, and `#`, to name symbols, don't exist in the actual language. That being said, this is how the above pattern would actually be implemented.

Patterns are actually far more powerful than this; in general, *any* expression composed of data constructors, variables, and values is allowed. For instance, we can write a function that sums the first two elements of a list by nesting `Pair`:

```
sumFirstTwo l =
    case l of
        Pair x (Pair y _) -> x + y
```

This example also demonstrates _, the "don't care" pattern, which behaves like a variable (in that it matches anything), but doesn't actually bind a variable. Like with guards, the `else` branch of any pattern matching is a call to `error`.

Values in patterns are an unusual case, but also give us an easy way to write base cases for recursive functions over non-recursive data types:

```
sumNums x =
    case x of
        0 -> 0
        _ -> x + sumNums (x-1)
```

Patterns are strictly ordered, so a more general pattern (such as _) must come after a more specific pattern (such as `0`).

Implementing patterns involves reducing the left-hand side of `->` using a reduction rule specific to patterns. It is reduced to a value which may have "holes", labeled by variable names. The value produced by that reduction is compared to the value passed to `case`, with a comparator that supports nested tuples, and matches a hole to any value. The right-hand side of `->` is then evaluated in an environment that associates the variables defined by the holes with the values defined by the case value.

Haskell also supports a shorthand for pattern matching, in a syntax that resembles declaring a function multiple times:

```
sumNums 0 = 0
sumNums x = x + sumNums (x-1)
```

[10] ... that this author is aware of

Like with `case` expressions, the order is critical, as this is just syntactic sugar for a `case` expression.

12.1 Semantics of Patterns

Since patterns are strictly ordered, the first step to defining their formal semantics is to rewrite them in a way that's more similar to the familiar `if-then-else` expression. Haskell—and most languages with pattern matching—don't have such a "match/else" expression, so we'll just have to invent one. In the language of our semantics, and not the source language, we will suppose that the expression `match E1 : P then E2 else E3` evaluates `E2` if the expression `E1` evaluates to a value which matches the pattern `P`, and `E3` otherwise. `E2` will be evaluated in a context in which the variables in `P` have been bound.

We will also simplify the process of matching by asserting that patterns are only one level deep; that is, we may `match E1 : Pair a b then ...`, but we may not `match E1 : Pair a (Pair b) then ...`. Again, the latter may be rewritten in terms of the former, albeit with a somewhat more complicated structure to the resulting `match-then-else` expressions.

Now, let's write the rules for `match`. Remember that if the expression evaluates to an ADT of any kind, the result will be some tuple $\{Sym, ...\}$, where Sym is the symbol for the data constructor, and the rest is the arguments. To match, we must evaluate both sides. Since we will produce `let`-bindings, we will evaluate with a store σ.

First, the usual rules to evaluate both sides. P here is a metavariable over patterns, which are syntactically just expressions; the difference comes in type judgment, in that they may have unbound variables, as described above.

$$\textsc{MatchLeft}\ \frac{\langle \sigma, E_1 \rangle \to \langle \sigma, E_1' \rangle}{\langle \sigma, \text{match } E_1 : P \text{ then } E_2 \text{ else } E_3 \rangle \to \langle \sigma, \text{match } E_1' : P \text{ then } E_2 \text{ else } E_3 \rangle}$$

$$\textsc{MatchRight}\ \frac{\langle \{\}, P \rangle \to \langle \{\}, P' \rangle}{\langle \sigma, \text{match } E_1 : P \text{ then } E_2 \text{ else } E_3 \rangle \to \langle \sigma, \text{match } E_1 : P' \text{ then } E_2 \text{ else } E_3 \rangle}$$

Note that P is reduced *without* anything in the store! This is because all variables in patterns are supposed to be unbound; they are bound *by* the matching.

Now, once both sides have been reduced, we can perform our match. First, we need to convert the unbound variables in P into `let`-bindings:

$$\textsc{MatchBind}\ \frac{\begin{array}{c} P = \{y, \__1, \__2, \cdots \__n, x, z_1, z_2, \cdots, z_m\} \\ P' = \{y, \__1, \__2, \cdots, \__n, \_, z_1, z_2, \cdots, z_m\} \\ V = \{y, M_1, M_2, \cdots, M_n, N, M_1', M_2', \cdots, M_m'\} \end{array}}{\langle \sigma, \text{match } V : P \text{ then } E_1 \text{ else } E_2 \rangle \to \langle \sigma, \text{match } V : P' \text{ then let } x = N \text{ in } E_1 \text{ else } E_2 \rangle}$$

This is a bit wordy, so let's go part-by-part:

- The $P =$ premise demands that the pattern have some number n (which may be 0) of throw-away `_` matches, then a variable name x, then m more variables.
- The $P' =$ line specifies that after this step of matching, our pattern will have one more `_`, replacing x.
- The $V =$ line associates the appropriate element—the one after n other elements—of the value we're matching against with the name N.
- The conclusion expands E_1 into "let $x = N$ in E_1", so that E_1 will be evaluated in a context with x defined as matched in V.

When all the binding is done, we need to actually evaluate the expression:

$$\textsc{MatchThen}\ \frac{P = \{y, \__1, \__2, ..., \__n\} \qquad V = \{y, M_1, M_2, ..., M_n\}}{\langle \sigma, \text{match } V : P \text{ then } E_1 \text{ else } E_2 \rangle \rightarrow \langle \sigma, E_1 \rangle}$$

This simply demands that P and V be of the same form; in particular, their symbol, y, must be the same.

Finally, if the symbol is wrong, the match fails:

$$\textsc{MatchElse}\ \frac{P = \{y, \cdots\} \qquad V = \{z, \cdots\} \qquad y \neq z}{\langle \sigma, \text{match } V : P \text{ then } E_1 \text{ else } E_2 \rangle \rightarrow \langle \sigma, E_2 \rangle}$$

Note that nothing in our rules checked for the case that the pattern and value matched have different sizes, e.g., that P is an n-tuple and V is an m-tuple, so they would get stuck if this occurred. Our data constructors only ever produce tuples of a fixed length, so it is the job of type judgment, and the rest of our semantic rules, to assure that the semantics do not get stuck in this way.

13 Errors and Exceptions

Haskell uses a special *error* type to indicate erroneous situations. Errors carry with them an error message, as a string. In most other languages, including OCaml, *exceptions* behave in a similar way to errors. For example, calling `sumFirstTwo` with a list of length less than two.

Semantically, since errors are, well, errors, virtually all formal semantics simply ignore them. But, we don't shy away from formally defining things in this course, so let's do it anyway, and add errors to the λ-calculus:

$$\langle Term \rangle ::= \cdots \mid \textbf{error}$$
$$\langle Expr \rangle ::= \cdots \mid \textbf{error}$$

$$\textsc{ErrorLeft}\ \frac{}{\textbf{error}\ E \rightarrow \ \textbf{error}} \qquad \textsc{ErrorRight}\ \frac{}{V\ \textbf{error} \rightarrow \ \textbf{error}}$$

In short, if an error occurs in any subexpression during application, then the expression is itself an error. This propagates upwards through all applications, so that if a program encounters an error anywhere, the entire program will be in error. With the λ-calculus, we only needed to explicitly show this propagation with two rules, but imagine if we had many more rules! Even adding the primitives we introduced in Module 3 would involve a dozen new, and extremely boring, rules, which is why they're usually ignored in formal semantics.

More interesting than errors are exceptions, but Haskell has no exceptions per se[11].

14 Lazy Evaluation

Recall that when we discussed the λ-calculus, we considered three reduction strategies: Applicative Order Reduction (AOR), Applicative Order Evaluation (AOE), and Normal Order Reduction (NOR). AOR and AOE proceed by always reducing the leftmost, innermost redex (in the case of AOE, ignoring redices inside of abstractions), while NOR always reduces the leftmost, outermost redex. As a result, under AOR and AOE, arguments to a function are always reduced completely before they are passed to the function. Conversely, under NOR, arguments are passed to the function first, and then evaluated as needed. This "as needed" form of evaluation is called *lazy evaluation*, and with lazy evaluation, arguments that aren't explicitly used (or even variable bindings that aren't explicitly used!) will never be evaluated at all. AOR and AOE are forms of *eager evaluation*, which are similar to how almost all programming languages evaluate.

[11] Monads, monads, monads! If you're familiar with Haskell, you're screaming at me that of course it has exceptions, but its exceptions are a library feature relating to monads, not a language feature.

As we learned from the Standardization Theorem, a reduction under NOR will always reach a normal form, if one exists. On the other hand, AOR and AOE—in this section, we will use "AOR" to refer to either—can get caught in infinite reduction sequences, even if a normal form exists. From this point of view, it would make sense to base the reduction strategies of functional programming languages on NOR rather than AOR. However, we've just stated that most languages are based on eager evaluation. Why should this be? The answer is that in general, eager strategies tend to be more efficient and predictable than lazy strategies. Consider the following expression:

$$(\lambda x.\, xxxxx)((\lambda y.\, y)z)$$

This expression reduces to the normal form $zzzzz$. However, notice that it takes six steps to reach the normal form under NOR, but only two to reach it under AOR[12]. This phenomenon manifests itself any time a function makes use of its argument more than once. If the argument is not reduced before being passed to the function, then it is replicated by the function and must then be reduced multiple times.

There are, however, two situations in which NOR can reach a normal form faster than AOR. First, as we have seen, AOR can fall into avoidable infinite reductions, whereas NOR is guaranteed to reach a normal form if one exists. In this case, of course, NOR reaches the normal form faster than AOR because AOR never reaches it. Any finite amount of time is less than infinite time. The second case is when NOR can avoid evaluation steps entirely. Consider the following expression:

$$(\lambda x.\, z)((\lambda a.\, \lambda b.\, \lambda c.\, \lambda d.\, \lambda e.\, e)yyyyy)$$

This expression reduces to the normal form z. However, in this case, AOR requires six steps of reduction to reach the normal form, while NOR requires only one. This expression illustrates the other important efficiency advantage that NOR has over AOR: if a function does not use one of its arguments, then NOR never reduces the argument, while AOR does reduce it. In other words, NOR does not bother to perform reductions whose results will just be thrown away, while AOR does.

Haskell is a lazily evaluated language, which means it mostly follows NOR. Actually, as Haskell has no interest in normal forms per se, but is a language with actual primitive values, it follows NOE: normal order evaluation. NOE is to NOR as AOE is to AOR: it follows the same order, but never reduces inside an abstraction. This is "lazy" because we only reduce expressions at the latest possible time, when we absolutely must in order to get a value; since an abstraction *is* a value, we don't need to reduce inside of it at all!

All of the primitive semantic rules we defined in Module 3 work for any of these evaluation orders, but our informal definition of pattern matching doesn't quite. For pattern matching to work lazily, the left-hand side is evaluated fully, and the comparator evaluates fully only the parts of the case expression necessary to make the comparison.

In addition, lazy evaluation allows us to change our definitions of `let` bindings and global declarations: both can allow expressions to be bound, rather than values, because those expressions will only be evaluated when needed anyway.

This lazy evaluation strategy makes certain patterns natural in Haskell, but nonsense in other languages. For instance, consider this function:

```
allTheNumbersFrom = \x -> Pair x (allTheNumbersFrom (x+1))
```

This function generates a list of every integer greater than or equal to the integer `x`. That list is, of course, infinitely long. And yet, this function works fine: if we call `allTheNumbersFrom 1`, for example, we get a list. If we get the first element of that list, it's `1`. The second element is `2`. Of course, if we try to go over the entire list, we'll never get to the end, because the list is infinitely long, but that didn't stop Haskell from representing it.

Let's see why this works by getting the third element of one of these infinite lists. We'll start with these additional (rather silly) definitions of `third`, `second`, and `first`:

```
third = \x -> case x of Pair _ y -> second y
second = \x -> case x of Pair _ y -> first y
first = \x -> case x of Pair y _ -> y
```

[12] Tongue-in-cheek exercise: Find the exercise from Module 2 to which this example is a solution.

Now, let's do our reduction, using NOE[13]:

```
   third (allTheNumbersFrom 1)
→ (x-> case x of Pair _ y -> second y) (allTheNumbersFrom 1)
→ case (allTheNumbersFrom 1) of Pair _ y -> second y
→ case ((x-> Pair x (allTheNumbersFrom (x+1))) 1) of Pair _ y -> second y
→ case (Pair 1 (allTheNumbersFrom (1+1))) of Pair _ y -> second y
→ let y = (allTheNumbersFrom (1+1)) in second y
→ second (allTheNumbersFrom (1+1))
→ (x-> case x of Pair _ y -> first y) (allTheNumbersFrom (1+1))
→ case (allTheNumbersFrom (1+1)) of Pair _ y -> first y
→ case ((x-> Pair x (allTheNumbersFrom (x+1))) (1+1)) of Pair _ y -> first y
→ case (Pair (1+1) (allTheNumbersFrom (1+1+1))) of Pair _ y -> first y
→ let y = (allTheNumbersFrom (1+1+1)) in first y
→ first (allTheNumbersFrom (1+1+1))
→ (x-> case x of Pair y _ -> y) (allTheNumbersFrom (1+1+1))
→ case (allTheNumbersFrom (1+1+1)) of Pair y _ -> y
→ case ((x-> Pair x (allTheNumbersFrom (x+1))) (1+1+1)) of Pair y _ -> y
→ case (Pair (1+1+1) (allTheNumbersFrom (1+1+1+1))) of Pair y _ -> y
→ let y = (1+1+1) in y
→ 1+1+1
→ 2+1
→ 3
```

Consider in particular the last few lines: when we evaluated `first`, its argument still had an infinite recursion, but since we didn't *use* that part of the argument, the fact that it was infinite is irrelevant. And, in the very end, since we hadn't yet needed to evaluate our addition—remember, `+` is a function too—all of the addition happens at the end.

This style of evaluation is wildly different from that used by other programming languages. In terms of the *meaning* of any expression, it is strictly better, since we can define infinite data structures without filling all of memory. It may seem unintuitive, but it fits nicely with another feature of Haskell: *Haskell is pure*. In a pure functional language, it doesn't actually *matter* if a particular subexpression is evaluated, if it doesn't contribute to the final value; whether it was evaluated or not is unobservable, except in how long your program runs.

There is a more subtle benefit in how it handles recursion. Most functional programming languages need to handle one case of recursion specially: so-called *tail recursion*. Tail recursion is when the very last thing a function does is call itself, with a different argument. The obvious implementation of recursion would simply add a new stack frame, and then all stack frames return in sequence when the base case is reached. The problem with this is that for very large data structures, the stack is easily exhausted. Other functional languages handle this case by clearing the current stack frame *before* making the recursive call, so that the same stack space is used repeatedly. Haskell needs no special handling: it just returns the unevaluated expression.

The compiler magic required to make Haskell perform well in spite of lazy evaluation is nothing short of herculean. The simple fact is that as lazy as Haskell is, computers are neither lazy nor functional. It is the job of Haskell compiler to determine which expressions are *definitely* evaluated, put them in an evaluation order, and make light functions of all the expressions which are *maybe* evaluated. Expressions which can't be proved to be evaluated exactly once are *memoized*, which means that memory is allocated to store their value, and then that value is used when the expression is evaluated again, so that if a variable is used multiple times, it doesn't involve fully evaluating an expression every time. In spite of all this, the major implementation of Haskell, `ghc`, produces code

[13] We're also simplifying `let` expressions to use substitution, just to avoid letting the reduction get out of hand.

with performance roughly on par with C compilers, at least in well-behaved code.

Laziness can be added ad hoc to any programming language with first-class functions. For instance, an OCaml function which behaves similarly to the Haskell `allTheNumbersFrom` above can be constructed by wrapping the problematic infinite recursion in a function, and infinite lists in an algebraic data type:

```
type 'a lazyList =
    | Empty
    | Pair of 'a * (unit -> 'a lazyList)

let rec allTheNumbersFrom x =
    Pair (x, fun _ -> (allTheNumbersFrom (x+1)))

let first (Pair (x, _)) = x
let second (Pair (_, x)) = first (x())
let third (Pair (_, x)) = second (x())
```

This version is more difficult to use, since we must explicitly call the function to get more of the list, with `x()` (the application of `x` with `unit`). However, this pattern, in particular with infinite lists, is sufficiently useful that it's made its way into the syntactic sugar of several languages, such as Python, in the form of *generators*.

Laziness requires pureness for the results to be unsurprising, but pureness comes at a great cost. In a language like OCaml, if you want to print something, you just call the `printf` function, which returns a `unit`. OCaml is evaluated in AOE order, so it's quite obvious when the actually printing will be done; the point in the code where you've textually written `printf "Hi"` is also when `"Hi"` will be printed. In Haskell, there are no functions that return `unit`, since that's wholly meaningless in a pure language: expressions have no behavior other than to evaluate a value, so an expression that evaluates to `unit` couldn't actually *do* anything. When an expression does something other than evaluate to a value, such as print, that other behavior is called a *side effect*. All practical programming languages, including Haskell, allow side effects, but laziness and purity significantly change how they can be expressed.

Before we discuss the implications of purity on side effects, though, let's discuss how they're represented in the semantics of an impure language like OCaml.

15 Impure State and Assignment

Consider OCaml, in which a reference can be created with `ref v`, where `v` is its initial value[14]. Because references change their value, the order of evaluation of any function that uses references is extremely important; OCaml follows AOE, as this is a fairly intuitive "left-to-right" order. But, that's only half the story. How would we *formally* describe references and state?

Recall how we described a store, σ. The store itself isn't useful for storing references, because it's transient; taking a step does not actually change the store, we simply add to the store when we encounter `let` bindings in subexpressions. What's more important is that it gave us a new way to think about our reduction in general: rather than *just* reducing an expression, we paired that expression with a store. In order to represent state, we add one more element to that tuple: a *heap*.

A heap is a map, like the store, which associates references with their values. In formal semantics, heaps are often named Σ (capital sigma), but this is much less universal than σ for stores; heaps are sometimes named H, or even $Heap$, as some kind of perverse anglophilia. We add Σ to the tuple we reduce over, so instead of just reducing over a pair of a store and an expression, we now reduce over a triple of a heap, store, and expression: $\langle \Sigma, \sigma, E \rangle$. Unlike the store as we defined it, however, taking a reduction step can actually *change* the heap. That is, there are rules that look like this:

$$\frac{\cdots}{\langle \Sigma, \sigma, E \rangle \rightarrow \langle \Sigma', \sigma, E' \rangle}$$

Thus, as reduction proceeds, later expressions are evaluated in a changed heap. We start off our reduction with

[14] As you've learned, this is actually an abbreviation for records in OCaml, but we'll stick to just references to keep this description simpler.

$\Sigma = \{\}$, and we cannot forget the state of the heap until the expression part of our triple has reduced to a value; i.e., reduction is complete.

Heaps are called such not by their analogy to the heap data structure (which they are not analogous to), but by their analogy to the heap of a program, i.e., physical memory. As the program runs, the heap is changed, and so that change should be reflected in the reduction steps.

Our store mapped variable names to values, but the heap cannot be indexed so nicely, because references don't actually have names, and even if they did, multiple references may have the same name or multiple names may have the same reference. Instead, like with symbols in abstract data types, we're going to need another sort of value created just to be unique. These values are called *labels*. You can think of them as memory addresses, as they are the indices into our heap, but our formal heap is just a simple map, with no particular structure, so it's better just to think of labels as arbitrary values. A reference will take the value of a label, and accessing references will involve using that label to index the heap. Labels are terminal values, insofar as they are not, in and of themselves, reducible.

Now, let's extend a language—specifically, the λ-calculus with `let` bindings—to include references using OCaml's syntax, and labels:

$$\begin{aligned}
\langle Term \rangle &::= \cdots \mid \langle Label \rangle \\
\langle Expr \rangle &::= \cdots \\
&\quad \mid \text{ref } \langle Expr \rangle \\
&\quad \mid \,! \langle Expr \rangle \\
&\quad \mid \langle Expr \rangle \;:=\; \langle Expr \rangle
\end{aligned}$$

Note that we haven't actually defined $\langle Label \rangle$. $\langle Label \rangle$ does not form part of the actual user language—a user cannot write a label, they can only write a "ref", which may become a label—so all that's important in the definition of $\langle Label \rangle$ is that there are infinitely many unique labels.

The typical metavariable for labels is ℓ, a script lowercase L, which is distinct from l, because it's not confusing enough to have both upper and lowercase, and both Greek and Latin letters, so some jackass added different typefaces.

Now, let's define the semantics for each of our new expressions. Let the metavariable E range over expressions, V range over terminal values, ℓ range over labels, Σ range over heaps, σ range over stores. Then,

$$\textsc{RefEval}\ \frac{\langle \Sigma, \sigma, E \rangle \to \langle \Sigma', \sigma, E' \rangle}{\langle \Sigma, \sigma, \text{ref } E \rangle \to \langle \Sigma', \sigma, \text{ref } E' \rangle} \qquad \textsc{Ref}\ \frac{\ell \text{ is a fresh label in } \Sigma \qquad \Sigma' = \Sigma[\ell \mapsto V]}{\langle \Sigma, \sigma, \text{ref } V \rangle \to \langle \Sigma', \sigma, \ell \rangle}$$

$$\textsc{DerefEval}\ \frac{\langle \Sigma, \sigma, E \rangle \to \langle \Sigma', \sigma, E' \rangle}{\langle \Sigma, \sigma, !E \rangle \to \langle \Sigma', \sigma, !E' \rangle} \qquad \textsc{Deref}\ \frac{\Sigma(\ell) = V}{\langle \Sigma, \sigma, !\ell \rangle \to \langle \Sigma, \sigma, V \rangle}$$

$$\textsc{AssgLeft}\ \frac{\langle \Sigma, \sigma, E_1 \rangle \to \langle \Sigma', \sigma, E_1' \rangle}{\langle \Sigma, \sigma, E_1 := E_2 \rangle \to \langle \Sigma', \sigma, E_1' := E_2 \rangle} \qquad \textsc{AssgRight}\ \frac{\langle \Sigma, \sigma, E \rangle \to \langle \Sigma', \sigma, E' \rangle}{\langle \Sigma, \sigma, V := E \rangle \to \langle \Sigma', \sigma, V := E' \rangle}$$

$$\textsc{Assg}\ \frac{\Sigma' = \Sigma[\ell \mapsto V]}{\langle \Sigma, \sigma, \ell := V \rangle \to \langle \Sigma', \sigma, V \rangle}$$

Exercise 5. Perform the reduction in AOE for this program: $(\lambda x.\ [\![\texttt{false}]\!]\ (x := 2)(!x))(\text{ref } 0)$

To those more familiar with C and C++, "ref" itself is essentially `malloc`, "!" is `*`, and "$x := y$" is `*x = y`. Because Σ is actually changed by "ref" and ":=", the order of evaluation can completely change the meaning of a program, and expressions with results which are ultimately discarded, such as the first argument to $[\![\texttt{false}]\!]$, can nonetheless contribute to the final value of the program. Although nothing makes state of this sort *technically* incompatible with lazy evaluation, the behavior of a program with state and lazy evaluation would be so difficult to predict, it's simply never done.

Aside: You may have noticed that virtually all of our formal semantics for added features have some rather boring steps saying "if you have an expression instead of a value as a subexpression, evaluate that first". Some formal semantics attempt to abbreviate this, but that's often very unclear; it's still more common to simply include all of these dull ordering steps.

The same basic idea—adding an extra element to the tuple that represents our program state—can also be extended to any other kind of impure behavior, such as standard I/O (with a list of input characters and output characters) and even files (with a tree of files).

Note that you will see many different ways of expressing the operands to $\rightarrow$ when state is included. It's not uncommon to write the triple as $\Sigma; \sigma; E$ instead of $\langle \Sigma, \sigma, E \rangle$. Some works prefer to group both state components, something like $\langle \Sigma, \sigma \rangle ; E$. These differences are purely superficial; the critical thing is that our reduction is now over not just the expression, but *state*.

We see that in an impure language, the order of evaluation will significantly impact the meaning of a program. It affects when the heap is changed, and the heap affects the evaluation of the rest of the program. Let's now return our discussion to Haskell, and bridge the gap between pureness and side effects.

16 Monads

At long last, that thing that makes Haskell such an intimidating language to use: it's *monads*!

Forget the hype: "monad" is just a pretentious word for "procedure". A monad is an ordered sequence of actions[15]; nothing more, nothing less. What makes monads unique is how these procedures are embedded into the language. The building blocks of procedures—commands—are *values* in Haskell. Haskell preserves its purity by manipulating impure actions as values, rather than actually performing them. In a sense, Haskell *describes* impure behavior, rather than *doing* impure behavior.

Monads bundle the side-effecting behavior into a sort of black box, and allow us to link those black boxes into explicitly specified chains. Those chains form the specific ordering that makes stateful behavior predictable.

The concept of a monad comes from category theory. Monads were first applied to computer science by Eugenio Moggi[7], and their use was popularized by the Haskell community, in particular by Philip Wadler[10] and Simon Peyton Jones[5]. Monads are a rather abstract concept—this is why they are considered so difficult to understand—so we shall introduce them through the motivating example of input and output, and then generalize.

16.1 The `IO` Monad and Haskell's Id and Ego

Haskell is really two languages. I call the language we've been discussing until now the *Haskell Id*[16]. The Haskell Id cannot communicate with the world. It cannot perform input and output at all. All it can do is compute, with no side effects.

So, if we want to perform input and output, how do we? There is another language, which I call the *Haskell Ego*. The Haskell Ego is the personality; it communiacates with the user, and is the only component capable of I/O.

Aside: The terms "Haskell Id" and "Haskell Ego" are unique to these notes. Don't expect to find them elsewhere. The Haskell Superego is, of course, the Haskell programmer.

The Haskell Ego is not pure. It's not even functional! It's an imperative language. But, a Haskell programmer doesn't directly write commands in the Haskell Ego. Instead, it works like this: a Haskell Id program is an expression that should evaluate to a *command* in the Haskell Ego language. The Haskell Ego interpreter then runs

[15] The ordering might be abstract, in the same way that a C compiler might reorder your code for optimization purposes.
[16] This is "id" as in "id and ego", not "ID" as in "identification".

that command, and the result may be a Haskell Id expression, as well as side effects. If it's a Haskell Id expression, then that's evaluated to get a Haskell Ego command, and so forth. Our $\rightarrow$ (now $\rightarrow_{Id}$) works with NOE semantics, but the value it creates must be a command in the language of the Haskell Ego. Then $\rightarrow_{Ego}$ evaluates that, and the rules for $\rightarrow_{Ego}$ (which we won't formalize) can result in an expression in the Haskell Id. So, the program executes with $\rightarrow^*_{Id}$, then $\rightarrow_{Ego}$, then $\rightarrow^*_{Id}$, etc., until the Ego is commanded to stop.

All side effects are under $\rightarrow_{Ego}$, but $\rightarrow_{Ego}$ will always have a fully evaluated value from the Haskell Id (a command). So, the Haskell Id arrow is always $\langle \sigma, E \rangle \rightarrow_{Id} \langle \sigma, E' \rangle$, with no Σ, so no mutation, while the Haskell Ego arrow is always $\langle \Sigma, \sigma, V \rangle \rightarrow_{Ego} \langle \Sigma', \sigma, E \rangle$, with a value that could not have progressed in the Id language.

With this new view, a Haskell program is supposed to evaluate to a command. Note: evaluate *to* a command, not *evaluate a* command. That command is a *value* to the Haskell Id—the language we've been using until this point. That command is an instruction for the other language, the Haskell Ego, which may then produce a Haskell Id expression to evaluate. This is how Haskell manages to be pure: the language you actually use can only create commands; it cannot run them. The interpreter that runs them is separate; it is the ego.

Why all this bother? Because making commands be values means that they have a *type*. It brings the type checker into side effects.

The real name for the Haskell Ego is the I/O monad.

The monad for performing I/O is called, appropriately, `IO`. `IO` is a polymorphic type, as an I/O function may return something as well as performing a side effect; for instance, `getLine` is a value of type `IO String`. Note that the type `IO String` has no arrows in it. `getLine` is not a function! This is the id-ego distinction: the Haskell Id cannot get a line, it can only create a value that describes getting a line. Remember, functions are *pure*, so `getLine` couldn't have been a function.

Inside of `getLine`'s value is some code that allows the Haskell Ego to read a line from standard input. But, to the Haskell Id, it's just a black box. Think of it like having a pointer to some machine code that will read a line, and that pointer is not a callable Haskell function, just a pointer.

Similarly, there is `putStrLn`, which is of type `String -> IO ()`. Yes, `putStrLn` *is* a function, but calling it does not write a string to standard output. It's a function so that it can make an I/O black box specific to the string you use as an argument; that I/O black box itself contains the code that allows Haskell to write the string out. That I/O is paired with `()`, which, like OCaml, is a unit value containing no information: `putStrLn` has no return except for its boxed behavior.

Conceptually, we may think of the type `IO a` as follows:

```
IO a = World -> (a, World)
```

That is, given a distinguished type `World` that represents the state of the entire world (or at least, Σ), we see that an `IO` action may be thought of as a function that takes the entire world as input, and produces a value of type `a` and a new world as output. Indeed, that's exactly how our formal semantics modeled side effects above: Σ is the world—all the state outside of our code—and `a` is the type that E actually evaluates to. Since Haskell is a real programming language, the "entire world" includes the computer screen and keyboard; input actions affect the state of the keyboard (one or more characters is removed from an input buffer), and output actions affect the state of the screen (characters appear on the display). This is purely conceptual, since we cannot write functions over the entire world, so instead, an `IO` is just a black box.

Let's imagine that we want to write a very simple program using `getLine` and `putStrLn` that simply reads in one line and writes it back out again. We can imagine `getLine` and `putStrLn` pictorially as shown in Figure 1. The "λ" half of each box shows its behavior in E, the Id, the expression within our language. The "World" half of each box shows its behavior on the world. Since both interact with the world, both have an input and output there. `getLine` has no input on the λ side, as it needs no information from the program to do its job. Its output on the λ side actually flows across from the World side. `putStrLn` requires a character from the program in order to change the world (in the small way of writing out a line), and returns only unit on the λ side.

What we want to do is compose `getLine` and `putStrLn` into a single program that performs both actions, as in Figure 2. Note that the λ side of the composed box has no input; our composed machine is just a value, not a function. The state of the world in this composed version "flows" through `getLine`, into `putStrLn`, and out of our

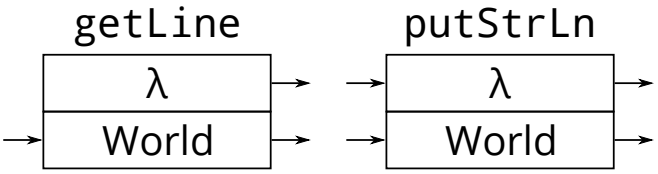

Figure 1: The `getLine` and `putStrLn` primitives.

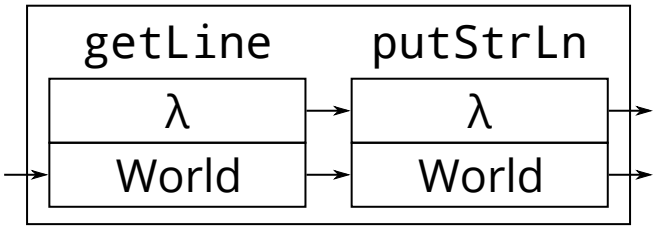

Figure 2: `getLine` and `putStrLn` composed.

combined program.

The combinator used to combine `getLine` and `putStrLn` in this way is written `>>=`, and called "bind". In the case of `IO`, `>>=` has the following type (note that `::` is used to explicitly specify the type of a declaration):

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

In this particular case, the type variable `a` will be substituted for `String`, so that `IO String` is the type of `getLine`, and the type variable `b` will be substituted for `()`, so that `a -> IO b` is the type of `putStrLn`. Our combined action, which we call `echo`, is then written as follows:

```
echo = getLine >>= putStrLn
```

`echo` is a non-function *value*, of type `IO ()`. It specifies, but does not actually perform, the I/O. Thus, nothing we've done so far is impure. All we've done is *describe* the side-effecting behavior; we haven't actually done anything with side effects.

The command, in Haskell Ego terms, is to get a line, and produce the expression `putStrLn x`, where `x` is the read line, for the Haskell Id. That Haskell Id expression then produces a command to print out the line for the Haskell Ego.

In general, the notation `f >>= g` specifies (but does not perform!) the following sequence of actions:

- the action `f` is performed first;
- the result produced by `f` is passed as an argument to the function `g`, yielding another action;
- the resulting action is performed, and its result returned as the result of evaluating the entire expression.

We've done all this work to produce a value that describes the side effects to be performed, so how do we use that to say what our *program* does? Recall that in Section 8, we said that in a language in which the global syntax only allows declarations, we need to know some starting point, and that in Haskell, that starting point is `main`. But, we said there that `main` is a function; that is not strictly true. `main` is an `IO` monad! The behavior of the entire Haskell program is to run the I/O described by the value of the expression declared as `main`. So, we can make a Haskell program which performs `echo` like so:

```
main = echo
```

A Haskell Id program is ultimately a specification of the I/O behavior which must be performed throughout the program, and it is the Haskell Ego which actually causes that I/O to occur. It is in this convoluted way that Haskell remains a pure functional language, but is capable of I/O. I/O can never be pure, but Haskell—as long as we only consider the Haskell Id to be Haskell—cannot perform I/O; it can only describe it.

Aside: OK, there is `unsafePerformIO`, which converts an `IO a` into an `a` by performing the I/O whenever it's evaluated. `unsafePerformIO` is named so for a reason, and isn't used frequently in Haskell code. The reason it's called "unsafe" rather than just "impure" is mainly laziness: in a lazy language, it's really hard to predict when—or if—the expression will be evaluated, so it's unsafe to assume that this `unsafePerformIO` will actually perform its I/O, and very unsafe to assume it will perform it at a given time.

Additionally, in the Haskell REPL `ghci`, if an expression given by the user evaluates to an `IO`, then the action specified by that `IO` is performed immediately, so for instance, we can make `ghci` echo like so:

```
> getLine >>= putStrLn
Hello, Haskell!
Hello, Haskell!
>
```

Note that the second line is the user input, and the third line is `ghci`'s output.

Suppose now that we would like to combine our `echo` program with itself, to produce a program that requests a line, prints it, and then requests and prints a second line. We cannot simply write `echo >>= echo`, because `echo` is not a function, so its type does not match the type expected by the second parameter of `>>=`, and therefore the expression contains a type error.

On the other hand, the output of the first echo, namely `()`, conveys no information, and is not useful in further computation. So, we can solve our problem by chaining the first `echo` to a function that ignores its input and then performs the second `echo`. We can capture this behavior in a new combinator, `>>`, called "then":

```
(>>) = \f -> \g -> f >>= (\_ -> g)
```

We can then implement our desired program as follows:

```
echoecho = echo >> echo
```

As another example of `>>`, the following is an `IO` action to read a line and print it twice:

```
echoTwice = getLine >>= (\l -> (putStrLn l >> putStrLn l))
```

Consider now the case where we would like to return a value to the user, rather than just print to the screen. Suppose we would like to read three lines from standard input and return the first and third, as a pair. Such a program must have type `IO (String, String)`; we proceed as follows:

```
myIOAction = getLine >>= \l1 ->
             getLine >>
             getLine >>= \l3 ->
             return (l1, l3)
```

Here, after the third `getLine`, there were no I/O actions left to perform; however, we still need a way to return the pair (c1, c3) to the user. Using `(l1, l3)` here will cause a type error, because monadic binding (`>>=`) always produces some `IO b`. `(l1, l3)` has type `(String, String)`, rather than the required `IO (String, String)`. This is because `l1` and `l3` are still part of the I/O—they were defined by lines we read—so we can only specify how we return them in a black box, and not actually return them. For this reason, we need a new primitive, `return`. The `return` primitive has no effect on World; its purpose is to take a value and wrap it inside the `IO` monad, so that it can be used in `IO` actions. It allows expressions in only the λ-calculus side to be within our `IO` specification.

We now have the machinery necessary to consider a more complex monadic computation. The following action retrieves lines of text until an empty line is found, and returns the entire block (note that `++` is used to concatenate strings in Haskell):

```
1 getBlock = getLine >>= \l ->
2            if l == "" then
3                return ""
4            else
5                getBlock >>= \ls ->
6                return (l ++ "\n" ++ ls)
```

16.2 Generalizing Monads

But what was the point of all this?

In an impure language, there is (typically) no way to express the side effects of a function in its types. There is no way to label side-effecting functions as special, non-referentially-transparent entities. Values of the `IO` monad are of the `IO` monad type, and are not functions, so you cannot accidentally use them as functions. The type system therefore represents the referential transparency—or absence thereof—of any behavior, by making side-effecting commands have a different type from functions. Put briefly: monads gave the control of type systems to side effects.

But, the side effects don't *have* to be I/O. There are many kinds of secondary behaviors that *could* be expressed without monads, but to which monads can give types and structure.

Now that we have some experience programming with monadic I/O, we can consider, in the abstract, what it means to be a monad. The formal definition of a monad is as follows:

Definition 8. (Monad) A monad is a triple $\langle M, \texttt{>>=}_M, \text{return}_M \rangle$, where M is a type constructor and

- $\text{return}_M : \alpha \to M\{\alpha\}$
- $\texttt{>>=}_M : M\{\alpha\} \to (\alpha \to M\{\beta\}) \to M\{\beta\}$

such that the following laws are satisfied:

- $(\text{return}_M a)\texttt{>>=}k = ka$
- $m\texttt{>>=}(\text{return}_M) = m$
- $m\texttt{>>=}((\lambda a.\, ka)\texttt{>>=}(\lambda b.\, hb)) = (m\texttt{>>=}(\lambda a.\, ka))\texttt{>>=}(\lambda b.\, hb)$

To summarize, monads are a way of boxing up behavior with ordering (through binding), such that the result of one behavior must be determined before continuing to the next behavior. Taken abstractly, the ordering doesn't have to be literal, as it is with the `IO` monad, but is simply that a behavior depends on the result of the last behavior.

By now, we have seen several examples of monadic programming, and it has become apparent that certain programming patterns emerge. In particular, the second argument of `>>=` is often an explicit λ-abstraction. To hide some of the complexity and focus attention on the underlying structure of the computation, Haskell supports a syntactic sugar for these common monad patterns, `do`. We use the `do` notation to remove explicit invocations of `>>=`. For example, our previous example, `myIOAction` can be rewritten using `do` notation as follows:

```
1 myIOAction =
2     do  l1 <- getLine
3         getLine
4         l3 <- getLine
5         return (l1, l3)
```

Recall that in Section 13, we discussed how errors propagate in Haskell. That was a rather brutish way to handle errors, so let's look instead to handling errors with monads.

We will use monads to handle errors in much the same way that we might use exceptions in another language. Rather than raise an exception in the pure functional setting, a computation that encounters an error will return a special token to indicate the error, rather than a normal value. We can model this "normal value or error token" behavior with a data declaration:

```
data Maybe a = Nothing | Just a
```

This `Maybe` type is functionally identical to OCaml's `option` type. For example, a division function which wishes to carefully handle error cases would return `Nothing` if the divisor is 0 (to avoid a division by zero error), and `Just` (the quotient) if the divisor is non-zero. A typical caller would need to pattern match to actually use this division function, but we can use monadic composition to make it simpler.

We can make a monad of `Maybe` by defining the monadic `>>=` and `return` functions for `Maybe`'s patterns:

```
(>>=) Nothing _ = Nothing
(>>=) (Just x) g = g x

return = Just
```

Remember that data constructors are still functions, so we can simply rebind `return` as `Just`.

The combinator `>>=`, which combines monadic computations, first examines the value of its first argument. If the first argument is `Nothing`, then there was a program error, and the `Nothing` is propagated without executing the second action. This is similar to how **error** is propagated in an application without concern for the rest of the expression. If the first argument is `Just x`, then there was no program error, and computation may proceed: the function g may be applied to the result of the previous computation. The combinator `return`, which just wraps its argument into the `Maybe` monad, is equivalent to `Just`.

We may now implement a safe integer division operation, using the Maybe monad:

```
safeDiv x y =
    if y == 0 then
        Nothing
    else
        Just (div x y)
```

Note that Haskell does support `/`, but it's floating-point division, and we wanted integer division, which in Haskell is named `div`.

Suppose now we wish to define a function f that takes inputs a, b, and c, and computes $a/b + b/c$. We may create a safe version of f using the `Maybe` monad as follows:

```
safef a b c =
    do
        r1 <- safeDiv a b
        r2 <- safeDiv b c
        return (r1 + r2)
```

Notice the imperative feel of `safef`; in this case, everything we've written is functional and pure, but monadic binding gives ordering to pure functions. Also notice that there is no explicit error handling inside of `safef`. All of the mechanics of propagating `Nothing` have been confined to the definition of the `Maybe` monad itself. Nevertheless, if `b` is 0, so that `safeDiv a b` fails, the error (`Nothing`) will propagate through the remainder of `safef`, so that the remaining computations, `safeDiv b c` and `return (r1 + r2)`, will be skipped.

The `IO` monad supports variables and state, but in fact, we don't need to involve the Haskell Ego just to manage state; we may also use monads to model mutable state, and assignment, in a purely functional setting. We consider a "state transformer" on a given state type `s` and result type `a` to be a function that takes an old state (of type `s`) as a parameter, and returns a result (of type `a`) and a new state (of type `s`) as results:

```
data State s a = ST (s -> (a, s))
```

Remember that monads box an *action*, not a value, and that action in this case is a transformation over the state, along with the function over `a`.

The data type `State` is the type constructor for our state transformers, and `ST` is the data constructor for a given transformer. We observe that, for any state type `s`, the type constructor `State s` is a monad, given definitions of `>>=` and `return`:

```
(>>=) (ST f) g = ST (\s0 ->
    let (a, s1) = f s0 in
    let (ST h) = g a in
    let (b, s2) = h s1 in
        (b, s2)
)

return x = ST (\s -> (x, s))
```

Our `return` combinator maps a value to a function that returns that value, while leaving the state unchanged. Our `>>=` produces a function that threads the initial state, `s0`, through the computations `f` and `g`. First `f`, a function on states, is applied to the initial state `s0`, producing a value `a` and a new state `s1`. The function `g`, of type `a -> State s b`, is then applied to `a`, producing a state computation, `h`, which is then applied to the "current" state, `s1`,

to produce a new value `b` and a final state `s2`. Thus, the initial state `s0` is mapped to `s1` after `f` is applied to it, and then to `s2` after `g a` is applied to it. The result is the pair `(b, s2)`, consisting of the final value `b` and the final state `s2`.

The state monad moves the changes to Σ into the explicit domain of Haskell code: a function in the `State` monad operates over the space of E, and returns a value describing the operation over the state of Σ. If we make Σ a single variable, instead of a whole map, then we can get or put the value of that variable with pure state functions:

```
get = \a -> \s -> (s, s)
put v = \a -> \s -> ((), v)
```

The `get` function ignores its input and sets the result and state to the same value. The `put` function puts a given value into the state.

Although `IO`'s behavior is essentially magic—you cannot "open" an `IO` value and retrieve the underlying code to perform I/O—the concepts of monadic composition apply exactly the same when the monad is decomposable and even purely functional, as in `State`.

16.3 Monads Elsewhere

Although monads are only strictly necessary in a pure functional language, their ability to explicitly specify behavior that is usually only implicitly specified, such as state change and I/O, makes them useful even in an impure setting. Although monads are not *necessary* in OCaml, they are now supported, and can be used to make I/O and stateful behavior part of the checked type of functions. Similarly, Scala, an object-oriented language with functional features that runs on the Java platform, supports monads, even though its surrounding environment is anything but pure.

However, any language which both allows impure behavior and allows monads must necessarily sacrifice some assurance: you *can* use monads to explicitly specify impure behavior, but you don't *have* to. Haskell is the largest-scale language to *require* monads for all impure behavior.

17 Implementation and Continuation-Passing Style

A note: this section suffers from "nowhere-else-to-stick-it-ism". It has nothing to do with monads, or even Haskell, but is sufficiently important to functional languages that it would be unwise to leave it out.

Implementing functional language is complicated by the fact that no matter how much functional programmers may want it, computers are not functional; they operate in an explicit sequence of steps. However, our formal semantics always described steps, through the $\rightarrow$ morphism, so clearly it is not impossible to describe functional languages in this fashion. In this section, we discuss *continuations*, an idea borrowed from formal semantics, and *Continuation-Passing Style* (CPS), an intermediate representation based on them.

Evaluation of expressions (and execution of programs) proceeds in several steps, each corresponding roughly to one step of β-reduction in the λ-calculus. With each step of the evaluation, we associate a *continuation*. A continuation is an entity (we may think of continuations as functions) that represents the remainder of the computation to be performed. If $E \rightarrow E'$, then the continuation is a version of E in which the part reduced by $E \rightarrow E'$ is factored out, and made an argument. A continuation takes as input the value currently being computed and applies the remainder of the computation to this value. The result is the final result computed by the entire program. Consider, for example, the following expression:

$$(3 + 5) * 4$$

The continuation of the subexpression 3 is $\lambda x.\,(x + 5) * 4$. The continuation of the subexpression $3 + 5$ is $\lambda x.\,x * 4$. The continuation of the entire expression, by convention, is simply the identity function, $\lambda x.\,x$.

Typically, during the compilation process, a compiler will transform source code into a convenient *intermediate representation*, which is typically based on a much simpler language, and upon which optimizing transformations are more easily performed. There are several well-known intermediate representations. One of these, known as *Continuation-Passing Style* (CPS), is particularly popular in compilers for functional languages.

CPS is based on making the notion of a continuation of an expression explicit. Each function is transformed so that it accepts an additional parameter, representing the function's continuation. Then, rather than return a value, the function passes that value to its continuation. Because CPS is more easily described with AOE than NOR, we will discuss it in the concept of OCaml. Consider the following simple OCaml function:

```
let f x = 3
```

This function could be rewritten to accept continuations as follows:

```
let f x k = k 3
```

In this way, functions in CPS never return, they simply pass the values they compute to their continuation. When the computation is finished, the continuation is the identity function, and this continuation is passed the final value of the computation.

Consider now the following, more complex, example:

```
let rec filter l p =
    match l with
    | [] -> []
    | x :: xs ->
        if p x then
            x :: (filter xs p)
        else
            (filter xs p)
```

This function takes a list and a predicate, and filters out all values from the list which match the predicate. This function could be rewritten in CPS as follows:

```
let rec filter l p k1 =
    match l with
    | [] -> k1 []
    | x :: xs ->
        let k2 b = (
            if b then
                let k3 l2 = (
                    let r = x :: l2
                      in k1 r
                ) in filter xs p k3
            else
                let k4 l2 = (
                    k1 l2
                ) in filter xs p k4
        ) in p x k2
```

From this example, we can observe the major characteristics of CPS. First, we note that CPS names all intermediate expressions (in this example, the only instance of this behavior is the binding of the name `r` to the expression `x :: l2`). In this way, CPS makes data-flow explicit. Second, CPS names all points of control flow: each recursive call to filter is made with a separate continuation, and these continuations (`k3` and `k4`) are distinct from the continuation `k2` passed to the predicate `p`, and the top-level continuation `k1`. In this way, CPS makes control flow explicit. (This is also why we used OCaml instead of Haskell to discuss continuation passing; while Haskell's control flow is well-defined, it's less obvious).

Tracing through the CPS code, we see that, upon invocation of filter, the first step of the computation is the invocation of `p`. The result (either `true` or `false`) is passed to the continuation `k2`, which tests the value computed by p and invokes filter recursively with either the continuation `k3` or the continuation `k4`, both of which eventually pass control back to the caller via invocation of the continuation `k1`.

Explicit control flow and data-flow make optimizations based on control flow analysis and data-flow analysis particularly easy. Consider, for example, the second recursive call to filter in the original function:

```
let k4 l2 = (
    k1 l2
) in filter xs p k4
```

In general, CPS formulations of a tail call—a recursive call which is the last behavior in a function—will have this form. The definition of `k4` is trivial—it is η-reducible to `k1`—and so we could easily replace the above code with simply the following:

```
filter xs p k1
```

That is, we simply reuse the top-level continuation `k1`. As simple as this transformation is, what we have actually done is optimize away tail recursion: the continuation requires no context from this function call, so the context from this function call can be discarded. Tail-call elimination is one of the fundamental optimizations of functional languages, and CPS makes it trivial.

In Haskell, this style is less needed, because using NOR, we evaluate the *outermost* expression, instead of the *innermost*. Tail calls naturally melt away, because what we return is not the result of a recursive call, but the *expression* of the recursive call. Thus, while there are areas of Haskell to which CPS is applicable, different techniques are used in lazy languages.

18 Fin

In the next module, we will look at another style of declarative programming, logic programming. Assignment 3 will focus on implementing functional languages like you've seen in this module.

References

[1] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, 1982.

[2] Karl-Filip Faxén. A static semantics for Haskell. *Journal of functional programming*, 12(4-5):295, 2002.

[3] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Éditeur inconnu, 1972.

[4] James Iry. A brief, incomplete, and mostly wrong history of programming languages. *One Div Zero* blog, Thursday, May 7, 2009. http://james-iry.blogspot.com/2009/05/brief-incomplete-and-mostly-wrong.html.

[5] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. *NATO SCIENCE SERIES SUB SERIES III COMPUTER AND SYSTEMS SCIENCES*, 180:47–96, 2001.

[6] Simon Marlow et al. Haskell 2010 language report. *Available on: https://www. haskell. org/onlinereport/haskell2010*, 2010.

[7] Eugenio Moggi. *Computational lambda-calculus and monads*. University of Edinburgh, Department of Computer Science, Laboratory for . . . , 1988.

[8] John C Reynolds. Towards a theory of type structure. In *Programming Symposium*, pages 408–425. Springer, 1974.

[9] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965.

[10] Philip Wadler. How to declare an imperative. *ACM Computing Surveys (CSUR)*, 29(3):240–263, 1997.

Appendix A Full Semantics

We have introduced many features of functional languages, each independently. This appendix serves as a unified reference. This is the mostly-complete syntax, semantics, and *resolve* for what we might call "untyped λ-Haskell", i.e., an untyped language with Haskell-like features (excluding some semantics that were left as exercises):

A.1 Syntax

$$
\begin{aligned}
\langle Program \rangle &::= \langle DeclList \rangle \\
\langle DeclList \rangle &::= \langle Decl \rangle \ \mathbf{;} \ \langle DeclList \rangle \\
&\quad \mid \epsilon \\
\langle Decl \rangle &::= \langle Var \rangle = \langle Expr \rangle \mid \text{data } \langle Var \rangle = \langle ValueList \rangle \\
\langle ValueList \rangle &::= \langle DataConstr \rangle \, \langle ValueListRest \rangle \\
\langle ValueListRest \rangle &::= \text{“|”} \, \langle DataConstr \rangle \, \langle ValueListRest \rangle \\
&\quad \mid \epsilon \\
\langle DataConstr \rangle &::= \langle Var \rangle \, \langle Var \rangle^* \\
\langle Expr \rangle &::= \langle Var \rangle \\
&\quad \mid \langle Abs \rangle \\
&\quad \mid \langle App \rangle \\
&\quad \mid \text{true} \\
&\quad \mid \text{false} \\
&\quad \mid \text{if } \langle Expr \rangle \text{ then } \langle Expr \rangle \text{ else } \langle Expr \rangle \\
&\quad \mid \text{match } \langle Expr \rangle : \langle Expr \rangle \text{ then } \langle Expr \rangle \text{ else } \langle Expr \rangle \\
&\quad \mid \langle Num \rangle \\
&\quad \mid \langle NumExp \rangle \\
&\quad \mid \text{let } \langle Var \rangle = \langle Expr \rangle \text{ in } \langle Expr \rangle \\
&\quad \mid \mathbf{error} \\
&\quad \mid (\langle Expr \rangle) \\
\langle Term \rangle &::= \langle Var \rangle \mid \langle Abs \rangle \mid \text{true} \mid \text{false} \mid \langle Num \rangle \mid \mathbf{error} \\
\langle Var \rangle &::= (\text{any valid ID}) \\
\langle Abs \rangle &::= \lambda \, \langle Var \rangle \, . \, \langle Expr \rangle \\
\langle App \rangle &::= \langle Expr \rangle \, \langle Expr \rangle \\
\langle Num \rangle &::= 0 \mid 1 \mid \cdots \\
\langle NumExp \rangle &::= \langle NumBinOps \rangle \, \langle Expr \rangle \, \langle Expr \rangle \\
\langle NumBinOps \rangle &::= + \mid - \mid * \mid /
\end{aligned}
$$

A.2 Semantics

$$\textsc{Application}\ \frac{}{\langle \sigma, (\lambda x.\, M_1) M_2 \rangle \to \langle \sigma, M_1[M_2/x] \rangle} \qquad \textsc{ReduceLeft}\ \frac{\langle \sigma, M_1 \rangle \to \langle \sigma, M_1' \rangle}{\langle \sigma, M_1 M_2 \rangle \to \langle \sigma, M_1' M_2 \rangle}$$

$$\textsc{Add}\ \frac{a + b = c}{\langle \sigma, (+\ a\ b) \rangle \to \langle \sigma, c \rangle}$$

$$\textsc{AddLeft}\ \frac{\langle \sigma, M \rangle \to \langle \sigma, M' \rangle}{\langle \sigma, (+\ M\ N) \rangle \to \langle \sigma, (+\ M'\ N) \rangle} \qquad \textsc{AddRight}\ \frac{\langle \sigma, M \rangle \to \langle \sigma, M' \rangle}{\langle \sigma, (+\ a\ M) \rangle \to \langle \sigma, (+\ a\ M') \rangle}$$

$$\textsc{Sub}\ \frac{a - b = c \qquad c \in \mathbb{N}}{\langle \sigma, (-\ a\ b) \rangle \to \langle \sigma, c \rangle}$$

$$\textsc{SubLeft}\ \frac{\langle \sigma, M \rangle \to \langle \sigma, M' \rangle}{\langle \sigma, (-\ M\ N) \rangle \to \langle \sigma, (-\ M'\ N) \rangle} \qquad \textsc{SubRight}\ \frac{\langle \sigma, M \rangle \to \langle \sigma, M' \rangle}{\langle \sigma, (-\ a\ M) \rangle \to \langle \sigma, (-\ a\ M') \rangle}$$

$$\textsc{IfTrue}\ \frac{}{\langle \sigma, \text{if true then } E_1 \text{ else } E_2 \rangle \to \langle \sigma, E_1 \rangle} \qquad \textsc{IfFalse}\ \frac{}{\langle \sigma, \text{if false then } E_1 \text{ else } E_2 \rangle \to \langle \sigma, E_2 \rangle}$$

$$\textsc{IfExpr}\ \frac{\langle \sigma, E_1 \rangle \to \langle \sigma, E_1' \rangle}{\langle \sigma, \text{if } E_1 \text{ then } E_2 \text{ else } E_3 \rangle \to \langle \sigma, \text{if } E_1' \text{ then } E_2 \text{ else } E_3 \rangle}$$

$$\textsc{LetDistinct}\ \frac{x \in FV[V] \qquad z \text{ is a fresh variable} \qquad M' = M[z/x]}{\langle \sigma, \text{let } x = V \text{ in } M \rangle \to \langle \sigma, \text{let } z = V \text{ in } M' \rangle}$$

$$\textsc{LetBody}\ \frac{x \notin FV[V] \qquad \sigma' = \sigma[x \mapsto V] \qquad \langle \sigma', M \rangle \to \langle \sigma', M' \rangle}{\langle \sigma, \text{let } x = V \text{ in } M \rangle \to \langle \sigma, \text{let } x = V \text{ in } M' \rangle}$$

$$\textsc{Variable}\ \frac{\sigma[x] = E}{\langle \sigma, x \rangle \to \langle \sigma, E \rangle} \qquad \textsc{LetResolution}\ \frac{}{\langle \sigma, \text{let } x = E \text{ in } V \rangle \to \langle \sigma, V[E/x] \rangle}$$

$$\textsc{MatchLeft}\ \frac{\langle \sigma, E_1 \rangle \to \langle \sigma, E_1' \rangle}{\langle \sigma, \text{match } E_1 : P \text{ then } E_2 \text{ else } E_3 \rangle \to \langle \sigma, \text{match } E_1' : P \text{ then } E_2 \text{ else } E_3 \rangle}$$

$$\textsc{MatchRight}\ \frac{\langle \{\}, P \rangle \to \langle \{\}, P' \rangle}{\langle \sigma, \text{match } E_1 : P \text{ then } E_2 \text{ else } E_3 \rangle \to \langle \sigma, \text{match } E_1 : P' \text{ then } E_2 \text{ else } E_3 \rangle}$$

$$\textsc{MatchBind}\ \frac{\begin{array}{c} P = \{y, \__1, \__2, \cdots \__n, x, z_1, z_2, \cdots, z_m\} \\ P' = \{y, \__1, \__2, \cdots, \__n, \_, z_1, z_2, \cdots, z_m\} \\ V = \{y, M_1, M_2, \cdots, M_n, N, M_1', M_2', \cdots, M_m'\} \end{array}}{\langle \sigma, \text{match } V : P \text{ then } E_1 \text{ else } E_2 \rangle \to \langle \sigma, \text{match } V : P' \text{ then let } x = N \text{ in } E_1 \text{ else } E_2 \rangle}$$

$$\textsc{MatchThen}\ \frac{P = \{y, \__1, \__2, ..., \__n\} \qquad V = \{y, M_1, M_2, ..., M_n\}}{\langle \sigma, \text{match } V : P \text{ then } E_1 \text{ else } E_2 \rangle \to \langle \sigma, E_1 \rangle}$$

$$\textsc{MatchElse}\ \frac{P = \{y, \cdots\} \qquad V = \{z, \cdots\} \qquad y \neq z}{\langle \sigma, \text{match } V : P \text{ then } E_1 \text{ else } E_2 \rangle \to \langle \sigma, E_2 \rangle}$$

$$\textsc{ErrorLeft}\ \frac{}{\langle \sigma, \mathbf{error} E \rangle \to \langle \sigma, \mathbf{error} \rangle} \qquad \textsc{ErrorRight}\ \frac{}{\langle \sigma, V\ \mathbf{error} \rangle \to \langle \sigma, \mathbf{error} \rangle}$$

A.3 Global Name Resolution

$$\textsc{EmptyProgram}\ \frac{}{resolve(\epsilon) = \texttt{empty}}$$

$$\textsc{Declaration}\ \frac{\sigma = resolve(L) \qquad \sigma' = \sigma[x \mapsto V]}{resolve(x = V\ \texttt{;}\ L) = \sigma'}$$

$$\textsc{EmptyDataDecl}\ \frac{}{resolve(\texttt{data}\ \ n = \epsilon\ \texttt{;}\ \ L) = resolve(L)}$$

$$\textsc{DataDecl}\ \frac{\sigma = resolve(\texttt{data}\ \ n = M\ \texttt{;}\ \ L) \qquad \sigma' = \sigma[N \mapsto \{N\}]}{resolve(\texttt{data}\ \ n = N\ M\ \texttt{;}\ \ L) = \sigma'}$$

ParamDataDecl

$$\frac{\sigma = resolve(\texttt{data}\ \ n = M\ \texttt{;}\ \ L) \qquad \sigma' = \sigma[N \mapsto \texttt{\textbackslash}x_1\ \texttt{->}\ \texttt{\textbackslash}x_2\ \texttt{->}\ \cdots\ \texttt{\textbackslash}x_m\ \texttt{->}\ \{N, x_1, x_2, \cdots, x_m\}]}{resolve(\texttt{data}\ \ n = N\ T_1\ T_2\ \cdots T_m\ \ M\ \texttt{;}\ \ L) = \sigma'}$$

Rights