Recitation 8:
By-name and by-value PCF, Dynamic and Unityped Languages
15-312: Foundations of Programming Languages
Jeanne Luning Prak, Charles Yuan, Yue Yao
March 7, 2018

1 By-value and By-name Settings

Please also refer to PFPL Supplement: PCF-By-Value for a detailed development.

1.1 A better PCF By-Value

By-name PCF developed in the lecture is just identical to the one presented in PFPL:

• It has \( \lambda \) and \( \text{fix} \) as primitive constructs.
• \( s(\cdot) \) does not evaluate it’s argument. It’s always a value.
• \( (\lambda (x) e_1)(e_2) \) directly substitutes \( e_2 \) into \( e_1 \) without evaluating it.

However, by-value PCF developed in PFPL has a particularly unpleasant rule, namely the dynamics of \( \text{fix} \):

\[
\text{fix } x \text{ is } e \rightarrow [\text{fix } x \text{ is } e/x]e
\]

The rule is particularly unpleasant to the tension between the following two setups:

• A by-value language should always only substitute variables for values.
• The fixed point construct forces us to substitute an expression, which could be an diverging computation instead of just a value, into a variable.

In order to mitigate this tension, our approach is to make an distinction between things are known to be a value, and computation through modal separation. In particular, we would have two separate typing judgments:

• \( \Gamma, x : \tau \vdash v : \tau \). Typing judgment for known values. Notice that values may only be built up with other values.
• \( \Gamma, x : \tau \vdash e \sim \tau \). Typing judgments for computations. Notice that even for computations, variables ranges only over values.
Now we would setup our judgments according to Figure 1 in the PFPL supplement. In particular, there are two rules that we should pay attention to:

\[
\begin{align*}
\Gamma &\vdash e : \tau \\
\Gamma &\vdash e \sim \tau
\end{align*}
\]

This rule essentially says all values can be considered as a (trivial) computation.

\[
\begin{align*}
\Gamma, x : \tau_1 \rightarrow \tau_2, y : \tau_1 &\vdash e \sim \tau_2 \\
\Gamma &\vdash \text{fun}(x,y,e) : \tau_1 \rightarrow \tau_2
\end{align*}
\]

There are a few things to notice about this rule:

- \textit{fun} defines a recursive function where \(x\) stands for the function itself. There is no \(\lambda\) or \textit{fix}, unlike in PFPL PCF-by-value.
- \textit{fun} defines a value, judging by it’s typing rule.
- The recursive call argument \(x\) ranges over value, unlike \textit{fix} operator in PFPL PCF-by-value.

### 1.2 Modal Separation

It’s the same old question: is it possible to (decently) encode by value PCF in terms of by-name PCF. The answer: yes, with the help of computation types \(\text{comp}\{\tau}\). In particular, \(\text{comp}\{\tau}\) is defined through the following judgment:

\[
\begin{align*}
\Gamma &\vdash e \sim \tau \\
\Gamma &\vdash \text{comp}(e) : \tau \\
\Gamma &\vdash e_1 : \text{comp}\{\tau_1\} \\
\Gamma, x : \tau &\vdash e_2 \sim \tau_2 \\
\Gamma &\vdash \text{let}\{e_1;x.e_2\} \sim \tau_2
\end{align*}
\]

where the dynamics are defined using the following rules:

\[
\begin{align*}
\text{comp}(e) &\text{ val} \\
\Gamma &\vdash e \rightarrow e' \\
\text{let}\{\text{comp}(e);x.e\} &\rightarrow \text{let}\{\text{comp}(e');x.e\} \\
v &\text{ val} \\
\text{let}\{\text{comp}(v);x.e\} &\rightarrow [v/x]e
\end{align*}
\]

Essentially \(\text{comp}(\cdot)\) wraps any computation into a value, and \text{let} construct forces a computation and substitute it’s resulting value into \(e_2\).

Now here’s something very subtle going on: the principal argument of \text{let} construct has to be a value! Further more, the dynamics is only defined for the case where the principal argument is a value.

Unfortunately, there’s more to this madness. Now we are going to change all elimination forms to only allow values at their principal argument, namely:

\[
\begin{align*}
\Gamma &\vdash e_1 : \tau_1 \rightarrow \tau_2 \\
\Gamma &\vdash e_2 : \tau_2 \\
\Gamma &\vdash e_1(e_2) \sim \tau
\end{align*}
\]
\[ \begin{align*} \Gamma \vdash e : \text{nat} & \quad \Gamma \vdash e_0 \sim \tau \quad \Gamma, x : \text{nat} \vdash e_1 \sim \tau \\ \Gamma \vdash \text{ifz} \ e \ \{ z \mapsto e_0 \mid s(x) \mapsto e_1 \} \sim \tau \end{align*} \]

Finally we remove dynamic rules where the principal arguments are non-values. Now let’s take a step back and contemplate on what we have accomplished:

- Did we remove any expressiveness from the language? No, since if we want to have a non-value expression at the principal argument, we can always wrap expression with `comp(·)` and `let`.
- What we did accomplish, is explicitly sequencing computations using `let` constructs. Intuitively our dynamics is divided into two modes: evaluating expressions (taking steps under `let`) and working with values. This is one example of modality.

How does `comp(·)` helps us embed by-name PCF into by-value PCF? Observe:

- We can wrap every by-name elimination form with a `let` construct. E.g., \( e_1(e_2) \) becomes `let\{e_1; v.v(e_2)`.
- A by-name PCF function of type \( \tau_1 \to \tau_2 \) can be defined using by-name PCF through a function of type `comp{\tau_1} \to \tau_2`.
- Correspondingly, calls sites become \( e_1(\text{comp}(e_2)) \)

### 1.3 Recursive type in by-name and by-value Setting

In a by value setting, `fold` evaluates its argument to a value:

\[
\begin{align*}
\frac{e \to e'}{\text{fold}\{t.\tau\}(e) \to \text{fold}\{t.\tau\}(e')} & \quad \frac{v \text{ val}}{\text{fold}\{t.\tau\}(v) \text{ val}}
\end{align*}
\]

In a by name setting, `fold` does not evaluate its argument, leaving behind the possibility of folding a divergent computation:

\[
\overline{\text{fold}\{t.\tau\}(e) \text{ val}}
\]

Now we make the following observation,

`rec\{t.1+t\}` defines a inductive type in by-value setting, and a coinductive type in a by name setting.

`rec\{t.1+t\}` defines `nat` in a by-value setting. It’s witness by an always terminating recursor:

\[
\begin{align*}
z & \triangleq \text{fold}\{t.1+t\}(1 \cdot ()) \\
s(e) & \triangleq \text{fold}\{t.1+t\}(2 \cdot e)
\end{align*}
\]

\[
\text{rec}\{\tau\}(e_0; p.x.e_1) = \text{unroll}(\text{self}(f.\lambda(x : \text{nat}) \text{ifz} \ x \ \{ z \mapsto e_0 \mid s(p) \mapsto (\lambda x : \tau.e_1)\text{unroll}(f)(p)\})(e))
\]

where `ifz` is a shorthand of `case`.

`rec\{t.1+t\}` defines `conat` in by name setting. It’s witness by the existence of \( \omega \):

\[
\omega \triangleq \text{unroll}(\text{self}(x.\text{fold}\{t.1+t\}(2 \cdot \text{unroll}(x))))
\]

It’s possible to regain `conat` in a by-value setting through computation types:

\[
\text{nat}_v \triangleq \text{rec}\{t.\text{comp}\{1+t\}\}
\]

In particular, `comp{rec\{t.1+t\}}` is NOT a valid encoding.
2 Untyped Languages

In this recitation, we explore two languages: the so-called untyped lambda calculus (Λ) and Dynamic PCF (DPCF). Such languages are often referred to as “untyped” or “dynamically typed” languages. However, we shall see that both of these languages actually have one single type, and every well-formed expression has that type. For this reason, we say that untyped actually means unityped, and that dynamic languages\(^1\) are a special case of static languages\(^2\). Λ is simple enough that having only a single type is unproblematic, but DPCF must incur a major runtime overhead to check that various operations are valid and raise errors if they are not.\(^3\)

3 The “Untyped” Lambda Calculus

Λ only has three possible expressions:

\[
\begin{align*}
& x & \text{variable} \\
& \lambda(x)e & \text{abstraction} \\
& e_1(e_2) & \text{application}
\end{align*}
\]

We will often use the simpler notation \(\lambda x.e\) to represent a lambda term, in accordance with most literature.

Its statics have only one judgment: \(\Gamma \vdash e \text{ ok}\), which determines that an expression contains no free variables.

However, despite its simplicity, Λ is remarkably expressive. It is a Turing-complete language, capable of expressing any computation that a Turing machine, or any other commonly accepted model of computation, can. This is due to the fact that any expression in any other language can be encoded in Λ, through a similar process that we used to encode values in System F. Additionally, it is possible to define general recursion in Λ through the use of fixed-point combinators, the most famous of which is the \(Y\) combinator.

3.1 Definability

We have already seen methods for encoding sums, products, natural numbers, and lists in a language that only has function types: System F. These encodings are almost identical to the encodings for Λ, and so aren’t covered here. However, we will discuss expressing general recursion:

To see the idea behind expressing general recursion in Λ, we’ll use the example of the factorial function. In PCF, we’d write this function as

```plaintext
fix fact : nat -> nat is
    fn (n : nat) ifz(s(z); x.mult(fact(x)(n)))
```

Note that we need to refer to fact in the body of fact. One way we can achieve this in a language without fix is to pass the function to itself as its first argument. So we would have

---

\(^1\)Languages whose “typechecking” is defined in their dynamics rules.

\(^2\)Languages whose typechecking is defined in their statics rules.

\(^3\)These notes are partially derived from 15-312 Spring 2017 course notes by Jake Zimmerman.
This achieves general recursion, but notice that we’ve removed the type annotations from the lambdas. This is because in a language like PCF, this self-referential expression is not well-typed. This is evident from the fact that the \( \text{fact}' \) function immediately takes an argument of the same type as itself. The corresponding type must be infinite, and in fact negative and impossible to express inductively or coinductively. However, in \( \Lambda \), this does not matter. We can define
\[
\lambda \text{fact}. \lambda n. \ldots \text{fact(fact)} \ldots
\]
as we please.

However, writing this by hand is cumbersome, and so we create a function that performs this passing-function-to-itself operation for us. This is known as a fixed-point combinator. For example, the well-known \( Y \) combinator performs this operation:
\[
Y \triangleq \lambda F. (\lambda f. F(f f))(\lambda f. F(f f))
\]
Take a close look at this combinator and make sure you understand why it creates the same kind of self-reference described above. This particular fixed-point combinator was discovered by Haskell Curry, and has the following property:
\[
Y f = f(Y f) = f(f(Y f)) = \ldots
\]
If we give \( Y \) a self-referential function \( f \), it produces an output which is equivalent to its own infinite iteration under \( f \). Mathematically, this is known as a **fixed point** of \( f \), an input which is identical to its corresponding output. This construct allows us to create general recursive expressions. Notice especially how easy it is to introduce divergent computation through this combinator. With \( Y \), we can easily turn self-referential functions into recursive ones. An added advantage is the ease of defining \( f \). Whereas before we had to apply the self-reference explicitly as in \( \text{fact}(\text{fact}) \), this is no longer necessary with the \( Y \) combinator; we may just write \( \text{fact} \).

Why do we get these guarantees with the \( Y \) combinator? We argued that \( Y \) satisfies the fixed-point relation above. As we saw when we studied PCF, one way of analyzing the fixed-point combinator is that a **functional** can be generated for a recursive specification of a function which takes a self-reference and “verifies” it. The equality between the functional applied to a candidate solution, and the candidate itself, is sufficient to show the correctness of the candidate, which is deemed a solution. If we start with the candidate \( Y f \), then \( Y f = f(Y f) \) holds, proving the correctness of the candidate.

The \( Y \) combinator is not meant as a particularly practical method of writing recursive functions, nor is \( \Lambda \) particularly practical as a programming language. However, it is a theoretically powerful construct that encodes recursion directly into the lambda calculus.

### 3.2 Untyped = Unityped

\( \Lambda \) is called **untyped**, but in fact, it can be easily embedded in a typed language with recursive types, such as FPC. The type of every expression in \( \Lambda \) is
\[
\text{rec}\{t.t \rightarrow t\}
\]
and expressions can be translated into **FPC** as

\[
x^\dagger \triangleq x
\]

\[
\lambda x.e^\dagger \triangleq \text{fold}(\lambda(x : \text{rec}\{t.t \rightarrow t\})e^\dagger)
\]

\[
e_1(e_2)^\dagger \triangleq \text{unfold}(e_1^\dagger)(e_2^\dagger)
\]

Thus, we say that **Λ** is actually **untyped**, with every well-formed expression having type **rec\{t.t \rightarrow t\}**. Indeed, every expression in the lambda calculus is implicitly a function that takes its own type and returns its own type. In this sense, dynamic typing is simply a particular instantiation of static typing! It’s possible to reason about a supposedly untyped language within the framework of recursive types and gain some of the advantages of type safety.

### 4 Dynamic PCF

**Λ** gets away with only having a single recursive type because of its simplicity. However, if we want to add other primitives to a language, this doesn’t work so well. This is clear even in a language which only contains functions and numbers: the statics no longer guarantee that we can’t apply a number to an argument or case on whether a function is zero or successor. To handle this, such a language must check, at runtime, that an operation that is meant to be performed on numbers is actually being performed on a number, and similarly for functions.

This is the principle behind Dynamic **PCF**. **DPCF** is an modification of **PCF** which has only a single **type** of expressions, but multiple **classes** of value that are checked as a program executes. Its syntax looks almost identical to **PCF**, but without type annotations:

\[
\text{Exp} \quad d ::= \quad x \quad \text{variable} \\
\quad \text{num}[n] \quad \text{numeral}^4 \\
\quad z \quad \text{zero} \\
\quad \text{s}(d) \quad \text{successor} \\
\quad \text{ifz } d \{ z \mapsto d_0 \mid \text{s}(x) \mapsto d_1 \} \quad \text{zero test} \\
\quad \lambda(x)d \quad \text{abstraction} \\
\quad d_1(d_2) \quad \text{application} \\
\quad \text{fix } x \text{ is } d \quad \text{recursion}
\]

The statics of DPCF are the same as that of **Λ**: they simply check that an expression contains no free variables. However, the dynamics are much more involved. Central to them is the notion of **class checking**, which is defined by the judgments **is_fun**, **is_num**, **isnt_fun**, and **isnt_num**. Class judgments are only defined on values, and expose the underlying structure of the value, as follows:

<table>
<thead>
<tr>
<th>num[n] is_num n</th>
<th>(\lambda(x)d) is_fun x.d</th>
<th>num[n] isnt_fun</th>
<th>(\lambda(x)d) isnt_num</th>
</tr>
</thead>
</table>

The reason class judgments are only defined on values is that they are used when a transition rule needs to rely on the structure of a value: it is impossible to define dynamics rules for checking if \(\lambda(x)d\) is zero or successor, or for substituting an argument into the body of num[n]. If the class check fails, we use the judgment **d err**, and then propagate errors through the dynamics.

For example, the rules for **app** are defined as:

\[
\frac{d_1 \mapsto d'_1}{d_1(d_2) \mapsto d'_1(d_2)} \
\frac{d_1 \text{ err}}{d_1(d_2) \text{ err}}
\]

\[^4\text{The numeric literal construct is added for convenience. It should be treated as having identical semantics to inductively built natural numbers, and we will elide the obvious rules.}\]
Despite this, **DPCF** can be shown type safe. We simply modify our progress theorem to account for the error judgment:

**Theorem** (*Progress*). If $d_{\text{ok}}$, then either $d_{\text{val}}$, or $d_{\text{err}}$, or there exists $d'$ such that $d \rightarrow d'$.

This is a much weaker theorem than before, as our code may now error at runtime despite passing all static checks, but it still ensures that execution of a program in **DPCF** will never get “stuck” if it is well formed.

As an example of a program in **DPCF**, consider the following implementation of addition:

```plaintext
fix plus is
  fn (n) fn (m)
  ifz n {
    z => m
    | s n' => s (plus n' m)
  }
```

Note the lack of type annotations. If this function is evaluated with $n$ and $m$ as numbers, it will return a number. But let’s think about what happens if $n$ is not a number—the `ifz` construct expects a number, so we will receive a runtime error. If $m$ is not a number and $n$ is nonzero, then the `s` construct in the recursive case will fail. But if $m$ is not a number and $n$ is zero, then this implementation simply returns $m$. Puzzling! The behavior of the program has become very difficult to predict, a consequence of the lack of types.

Furthermore, the evaluation of this program is likely to be highly inefficient. It will be filled with runtime checks for whether a term is a number or is a function, which slows things down quite a bit.

## 5 Hybrid PCF

Dynamic **PCF** is more interesting and easier to work with than the lambda calculus, but it doesn’t come close to **PCF** in terms of safety. Every expression in **DPCF** now has the possibility of erroring at runtime, without any way for us to ensure runtime safety. What if instead of replacing all types in **PCF** with a dynamic interpretation, we simply added dynamic types to **PCF**?
The result is a language called Hybrid PCF, or HPCF:

\[
\begin{align*}
\text{Cls} & \quad l \quad ::= \quad \text{num} \quad \text{number} \\
& \quad \quad \quad \quad \quad \text{fun} \quad \text{function} \\
\text{Typ} & \quad \tau \quad ::= \quad \text{nat} \quad \text{natural} \\
& \quad \quad \quad \quad \quad \quad \quad \tau_1 \to \tau_2 \quad \text{function} \\
& \quad \quad \quad \quad \quad \quad \quad \text{dyn} \quad \text{dynamic} \\
\text{Exp} & \quad d \quad ::= \quad x \quad \text{variable} \\
& \quad \quad \quad \quad \quad \text{num}[n] \quad \text{numeral} \\
& \quad \quad \quad \quad \quad \text{z} \quad \text{zero} \\
& \quad \quad \quad \quad \quad \text{s}(d) \quad \text{successor} \\
& \quad \quad \quad \quad \quad \text{ifz } d \{ \text{z } \mapsto d_0 \ | \ \text{s}(x) \mapsto d_1 \} \quad \text{zero test} \\
& \quad \quad \quad \quad \quad \text{λ}(x)d \quad \text{abstraction} \\
& \quad \quad \quad \quad \quad \text{d}_1(d_2) \quad \text{application} \\
& \quad \quad \quad \quad \quad \text{fix } x \text{ is } d \quad \text{recursion} \\
& \quad \quad \quad \quad \quad \text{l!e} \quad \text{tag} \\
& \quad \quad \quad \quad \quad \text{e @ l} \quad \text{cast} \\
& \quad \quad \quad \quad \quad \text{l?e} \quad \text{test}
\end{align*}
\]

This extension of PCF includes all the operations and type structure of PCF, except with the new notion of **classes**. There are two classes, one for numbers and one for functions, which roughly correspond to the number and function types, but have some differences:

1. Classes are checked at runtime, not compile-time. This has notable consequences for performance, as we shall see.
2. Classes encode limited information about the underlying data. Notably, a function is just a “function” in terms of class, with no information about argument or return value. (As we shall see, the implicit argument and result are both of dynamic type.)
3. Classes are not necessarily accurate. An assumption that a value is of some class which is incorrect will be met with a runtime error.

Note that this definition of classes exactly characterizes “types” in languages like Python, which are dynamically checked. When Python refers to “types”, it really refers to “classes”!

Now when we write a program, if we would like then we are able to use dynamic typed expressions. Expressions of type dyn are introduced via the tag construct and eliminated via the cast construct. These constructs are remarkably similar to the extensible type exn in Standard ML! Similarly to extensibles, expressions of arbitrary type become dynamic when tagged with the appropriate label (class), and recovered when the label is stripped. Unlike extensibles, the cast operation is inherently unsafe. If a cast is performed on an expression of incorrect class, the dynamic behavior requires us to raise a runtime error. We’ll come back to the similarity here when we discuss dynamic classification later in the course.

We give the programmer the ability to test for the class of a dynamic expression using a test construct. As we shall see, this construct is flawed in some major ways.

### 5.1 Statics

The statics for HPCF are the same as for PCF except for the dynamic components:

\[
\begin{align*}
\Gamma \vdash e : \text{nat} & \quad \implies \quad \Gamma \vdash e : \text{dyn} \\
\Gamma \vdash \text{num} \mathbf{!} e : \text{dyn} & \quad \implies \quad \Gamma \vdash \text{fun} \mathbf{!} e : \text{dyn}
\end{align*}
\]
5.2 Dynamics

$$\Gamma \vdash e : \text{dyn} \quad \Gamma \vdash e @ \text{num} : \text{nat} \quad \Gamma \vdash e @ \text{fun} : \text{dyn} \rightarrow \text{dyn}$$

$$\Gamma \vdash e : \text{dyn} \
\Gamma \vdash l \ ? \ e : \text{bool}$$

Note that we use a boolean type here. We don’t specify which interpretation of booleans should be preferred—perhaps a sum, or a number, or baked into the language. Now that we’ve seen how to do each of these approaches, we can get creative!

Note how weak the statics are for this system. From an expression of dynamic type, it is statically legal to cast it to either a number or a function. Only at runtime will the error be caught.

5.2 Dynamics

The dynamics for the new cases are eager in the arguments of tagging, casting, and testing. These are the interesting rules:

$$\begin{align*}
\text{e val} & (l! e) \rightarrow e \\
\text{l ! e val} & (l! e) \rightarrow l'' \err \\
\text{l ! e val} & l \neq l'' \rightarrow true \\
\text{l ! e val} & l'' \neq l \rightarrow false
\end{align*}$$

These rules indicate that tagged values are values, and we may either successfully cast or receive a runtime error in event of an unsuccessful cast. We can test for whether a value is of a particular label, receiving a boolean in return. Note that the error judgment should be propagated throughout the rules for it to be complete.

5.3 Boolean Blindness

The class instance test construct leaves much to be desired. For this section, we can make our point a bit clearer using some Java-like syntax. Consider the following snippet:

```java
Object x = ...
if (x instanceof SomeClass) {
    SomeClass y = (SomeClass) x;
}
```

This is an arguably reasonable piece of Java code, but it falls prey to **boolean blindness**: the phenomenon that a reliance on boolean tests about our data tells us nothing statically true about the data. Here, we would like to safely cast x to SomeClass, and do so via an `instanceof` test. But from the typechecker’s view, `instanceof` may be an arbitrary boolean predicate, which simply returns a boolean value and continues execution. We are still left with the expression x, about which we have learned nothing from a static point of view! Arguably the programmer is now somewhat more secure in the cast, but there is no static proof of safety here. Contrast this with the ML `case` construct, which statically ensures the safety of each of its branches at compile-time. Boolean blindness is a crutch for a language without statically checked sum types!

Knowing this, you might ask why we do not introduce some equivalent to the `case` construct for eliminating dynamic types and instead rely on the boolean-blind class instance test operator. Since dynamic types can error out anyway, we have no way of propagating static information about a dynamic value. We could define syntactical sugar for “pattern matching” on a dynamic value, but it would never give us the safety of static checking back anyway.
5.4 Optimization of Hybrid PCF

Let’s port our DPCF implementation of addition to HPCF:

```plaintext
fix plus : dyn is
  fun ! fn (n : dyn)
    fun ! fn (m : dyn)
      ifz (n @ num) {
        z => m
        | s n' => num ! (s (((plus @ fun) (num ! n')) @ fun m) @ num))
    }
end
```

Our type annotations are back, but they’re not very helpful. If you examine the code, it’s clear that it has the exact same runtime behavior as the DPCF program, since all computation is dynamic. In fact, we have reified the dynamic check behavior: every instance of tag and cast is a dynamic operation that will slow the program down, and now we write them explicitly.

But now that we have the whole PCF type system, we can optimize this implementation. If we are fully convinced a dynamic value has some class, we may as well strip away some pairs of tagging and casting.\(^5\) We can repeat this process until we make a realization: this addition function only has well-defined behavior when both of its arguments are numbers, and it returns a number in that case. So we are really being quite superfluous when we say that it has type `dyn`. We might as well write it like this:

```plaintext
let val plus =
  fix plus : nat -> nat -> nat is
    fn (n : nat) fn (m : nat)
      ifz n {
        z => m
        | s n' => s (plus n' m)
      }
    in
      fun ! fn (n : dyn) fun ! fn (m : dyn)
        num ! plus (n @ num) (m @ num)
  end
```

So, after doing all this optimization to reduce the dynamic overhead, we have come full circle. This is just a PCF function with static types, and a wrapper around it that casts the incoming arguments.\(^6\) Why did we need dynamic types in the first place?

### 6 Dynamic vs. Static

The debate between dynamic and static types is perennial. Proponents of dynamic types often claim:

1. It’s easy to write dynamic programs. Many programs you write tend to “just work”, without compiler complaints.

\(^5\)For the curious, this is one operating principle of just-in-time (JIT) compilers for languages like JavaScript!
\(^6\)The astute observer will notice that the behavior described earlier, about what happens when \(m\) is not a number, has changed. Think about whether we are justified in doing this optimization in light of that fact.
2. Dynamic code is more concise because of its lack of type annotations.

3. Dynamic code is more flexible in behavior, for example being able to handle a wide variety of inputs.

However, each of these points is not particularly valid. A static type system counters each:

1. Dynamic programs may have a easier learning curve, but reasoning about your code and designing large-scale systems is next to impossible without real type checking. Runtime errors abound!

2. An ML-style type inference system often results in code that is arguably more concise than their dynamically typed counterparts.

3. Flexibility is a two-edged sword—now a user must guess which capabilities are or are not supported by a dynamic function. Even worse, flexibility in function return values is downright negative, as the user will have to make sense of the output!

In addition, there are features that only static type systems support:

1. A system of modules which enforces data abstraction and safety

2. Generic programming driven by the type of data (like map)

3. The remarkable experience that programs satisfying a type specification must be correct. We didn’t get into this much, but a concept called parametricity in polymorphic languages (like System F, or ML) means that the type of an expression says a lot about it! For example, there is only one pure function of type \( 'a \rightarrow 'a \), and only one function of type \( 'a \rightarrow 'b \rightarrow 'a \). When you utilize the type system to its full capabilities, it tends to be the case that if your program compiles, it must be correct!

In the end, dynamic typing is a particular mode of use of static types, and dynamic languages can be analyzed in a type-based framework and compared to more powerful type systems. They have an easier learning curve and considerable allure, but ultimately certain things are only possible in a static framework.