CS442
Module 4: Types

University of Waterloo

Winter 2025

1 Getting Stuck

In Module 2, we studied the λ-calculus as a model for programming. We showed how many of the elements of "real-world" programming languages can be built from λ-expressions. But, because these "real-world" constructs were all modeled as λ-expressions, we can combine them in any way we like without regard for "what makes sense".

When we added semantics, in Module 3, we also added direct ("primitive") semantic implementations of particular patterns, such as booleans, rather than expressing them directly as λ-expressions. That change introduces a problem: what happens when you try to use a primitive in a way that makes no sense? For instance, what happens if we try to add two lists as if they were numbers?

Well, let's work out an example that tries to do that, using the semantics of AOE, plus numbers and lists, from Module 3, and the expression (+ (cons 1 (cons 2 empty)) (cons 3 (cons 4 empty))):

- The only semantic rule for + that matches is to reduce the first argument. After several steps, it reduces to [1, 2], so our expression is now (+ [1, 2] (cons 3 (cons 4 empty))).

- The rule for + to reduce the first argument no longer matches, because the first argument is not reducible. But, the rule to reduce the second argument doesn't match, because the first argument is not a number[1]. No rules match, so we can reduce no further.

What happens next? The answer is that nothing happens next! There's no semantic rule that matches the current state of our program, so we can't take another step. This isn't a problem with our semantics; after all, it makes no sense to add lists in this way. But, because of this, applying $\rightarrow^*$ to our code reaches a rather unsatisfying result: (+ [1, 2] (cons 3 (cons 4 empty))).

We call this phenomenon "getting stuck", but to understand what it means to get stuck, first we have to have a goal. Applying $\rightarrow$ repeatedly will reach some conclusion for any program that terminates, so what was different about the conclusion for the above program than a "good" one?

In real software, we usually run software for its side effects: it interacts with the user, or transforms some files, etc. In formal semantics, there is nothing external to the program, and all of the steps are syntactic transformations of the program. So, the question is, what is the syntax for a "finished" program?

Of course, it depends on the language. So, we rely on a topic we set aside in the last module: terminal values. We will consider a program to have run to completion if it reaches a terminal value, usually just called a "value". Values are elements of the syntax that are complete and meaningful on their own, with no further reduction. For instance, in most languages with numbers, a literal number is a value. In the λ-calculus with AOE, an abstraction is a value, as without an application, there's nothing else to do.

---

[1] In the semantics for addition as we wrote them, the left-hand side was a, which referred to numbers, rather than any irreducible expression, but we would run into a similar problem no matter how we implement +. Note also that we used + as a shorthand for list concatenation, but that was a shorthand for the premises, and not part of our language.

This definition is somewhat circular: How do we know when a program has run to completion? It reaches a terminal value. What's a terminal value? A program fragment that can't run any further. It's up to the definer of the semantics to choose values that are both correct for their semantics and intuitively correct.

"Terminal value" is sometimes abbreviated to "term". However, "term" is a very overloaded, well, term, so we will usually call them "values", "terminals", or "terminal values".

> **Aside:** The word "term" is actually related to "terminal" or "terminus". It is the fact that a term has a single, conclusive meaning that makes it "term"! Yes, even in regular English, the term "term" comes from having a complete meaning on its own.

2 Common Terminal Values

We already saw that in the λ-calculus with AOE, an abstraction is a terminal value. In fact, in *most*—but certainly not all—programming language semantics with functions, a function is a value, since there is nothing further to do with a function that has not been called. An implication of this fact is that there is usually an infinite number of possible values, since there is at least an infinite number of possible functions. As a consequence, we can't simply enumerate all of the terminal values; we must describe our values syntactically, as context-free grammars already give us a way of expressing such infinite sets.

Most other values should be fairly obvious. In Module 3, we created extended versions of the λ-calculus with booleans, numbers, lists, and sets. In each case, we introduced syntax for those kinds of values, as well as syntax to do things with them. For instance, with booleans, we introduced true and false, but also if expressions. The new values in that extended λ-calculus are true and false; if expressions are non-terminal.

> **Aside:** Non-terminal, I've heard that term before! Indeed, a non-terminal in a grammar is exactly the same: non-terminals can be reduced, terminals cannot.

In our extended λ-calculus with numbers, numbers are values. With lists, lists of values are values. With sets, sets of values are values. Hopefully, your intuition about what is a completed calculation should align with the definition of a terminal value.

3 Introduction to Types

Now that we've established that our semantics can get stuck, and defined what it means to get stuck, and that reaching a terminal value is *not* getting stuck, what's the solution? Types.

At a basic level, a type is simply a set of semantic objects from which the value of a variable or expression must be taken. That is, each type is a subset of the values of the language. For example, if a variable x is given type int, then there is a set $t_{\texttt{int}}$ (in this case, perhaps, the set of integers) that contains all possible values x can assume.

Not every possible subset of the values is a type, since most subsets are useless; we're trying to define types to achieve a particular goal. Our problem was that a particular value could not behave in the way that we tried to make it behave—specifically, in our example, we tried to add a list like an integer—so if we could categorize values by the way they behave, we could have at least identified the problem.

More broadly, one defines types based on semantic meaning. For instance, numbers and lists are different types, because you can do different things with them. You can't add two lists, and so we can now give meaning to the error: you tried to use addition on a type to which that doesn't apply. But, we can go further than merely giving meaning to the error: we know, by definition, that cons always produces a list, and that + needs an argument of the number type, so we can reject the program just by inspecting it, without having to run it until it gets stuck. That is the goal of type systems.

Categorizing values in this way can be useful even if we're not facing getting stuck. For example, the expression $\lambda x. \lambda y. x$ might represent the value `true` when viewed as boolean data, but it might represent the function that takes two arguments and returns the first when viewed as a function. Both of these views of $\lambda x. \lambda y. x$ are valid, but presumably, a programmer only intended one of them in any context. The set of booleans—in the λ-calculus *not* extended with primitive booleans—is much smaller than the set of functions, but it is also a *subset* of the set of functions. If the programmer had written down that they intended it to be a boolean, then that documents how they intend it be used, even though in the λ-calculus, functions are largely interchangeable.

The first use of types occurred around the year 1900, when mathematicians were formulating theories of the foundations of mathematics. Types were used to augment formal systems to avoid falling into variants of Russell's paradox[2]. Type theory remained a highly specialized and rather obscure field until the 1970s when its connection with programming languages became clear. Throughout its history, type theory has remained closely related to logic and deduction systems; the connection between type theory and logic is known as the Curry-Howard Isomorphism, and its nature should become clear in the discussion that follows, but we will first focus on the practical use of type systems in programming languages.

4   Static vs. Dynamic Typing

In a statically typed programming language, types can be reasoned from the code itself, without needing to execute it. In many statically typed languages, the types are written in the code, though this is not technically a requirement. Statically typed languages include OCaml, C and C++, and Java.

For instance, if I write the program in Figure 1 in OCaml and attempt to compile it, the compiler will refuse with the error in Figure 2. Reasoning only from the code, without executing the program—indeed, as OCaml is a compiled language, I could not possibly execute the program, since the compiler never produced an executable—the compiler could tell us that our types were violated.

```
let x = [1;2] + [3;4]
```

Figure 1: OCaml program with a type violation

```
File "ex.ml", line 1, characters
  8-13:
Error: This expression has type
  'a list but an expression was
  expected of type int
```

Figure 2: Error from the program in Figure 1

As we set out previously, the goal of types in general is to prevent programs that might get stuck. The goal of static typing is to catch this kind of error before the program actually runs. In a statically typed language, every code fragment's types are checked to be consistent—by that language's definition of "consistent"—before execution.

This also means that, in most cases, code will be checked for correctness even if it's never used. For instance, if the code in Figure 1 were in `if false`, it would *still* be rejected, even though the code is unreachable. In fact, static types *cannot* refuse *only* programs that would actually get stuck at run-time. Rice's theorem[3] guarantees this. As a consequence, the programs accepted by a static type system will always be smaller than the programs that would terminate according to our semantic rules (at least, if the type system is correct).

In a dynamically typed programming language, types are reasoned about only during execution (or not at all). Type errors arise only at run-time, and only careful code authorship can guarantee an absence of type errors. Dynamically typed languages include Smalltalk, JavaScript, Python, and nearly all languages that fall under the umbrella term "scripting language", which is a term so ill defined as to be almost meaningless.

---

[2]R is the set of all sets that are not members of themselves. Then, $R \in R \Leftrightarrow R \notin R$

[3]All non-trivial semantic properties of programs are undecidable.

For instance, if I write the program in Figure 3 in GNU Smalltalk and run it, the program will begin execution, but immediately crash with the error in Figure 4.

```
{1. 2} + {3. 4}
```

Figure 3: GNU Smalltalk program with a type violation

```
Object: Array new: 2 error: did
  not understand #+
MessageNotUnderstood(Exception)>>
  signal (ExcHandling.st:254)
Array(Object)>>doesNotUnderstand: #+
  (SysExcept.st:1448)
UndefinedObject>>executeStatements
  (ex.st:1)
```

Figure 4: Error from the program in Figure 3

Because types are only checked during execution, if an expression or statement in the language would definitely cause a type error, but it is never actually reached, then the program may run flawlessly. Furthermore, as variables can generally store values of any type, the same code may or may not evoke type errors depending on what values are passed to it, and this can make it exceedingly difficult to discover the original source of an error. Consider for example the code in Figure 5, which has a definite type error but only in a block that's never reached, and has a conditional type error in a block that is reached. The error produced, in Figure 6, is only produced on the very last invocation of the run:right: method.

```
1  Object subclass: Example [
2     run: x right: y [
3         true ifTrue: [ ^x+y ]
4             ifFalse: [ ^{1. 2} + {3. 4} ].
5     ]
6  ]
7
8  Example new run: 1 right: 2.
9  Example new run: 1.5 right: (3/4).
10 Example new run: {49. 50} right: {51. 52}.
```

Figure 5: Smalltalk program with conditional type violation

```
Object: Array new: 2 error: did not
  understand #+
MessageNotUnderstood(Exception)>>
  signal (ExcHandling.st:254)
Array(Object)>>doesNotUnderstand: #+
  (SysExcept.st:1448)
Example>>run:right: (ex.st:3)
UndefinedObject>>executeStatements
  (ex.st:9)
```

Figure 6: Error from the program in Figure 5

In semantics, a language is statically typed if types can be determined without using $\rightarrow^*$—that is, without "running" a program. We will see in Section 8 how this determination is formally described. Dynamically typed semantics simply get stuck, and thus, among authors of semantics, "dynamically typed" and "untyped" usually mean the same thing; because these notes focus on semantics, you should assume that if we say "typed" without specifying further, we mean statically typed.

Statically typed languages usually offer guarantees, while dynamically typed languages offer flexibility. However, the distinction between them is only *when* types are reasoned about; not all statically typed languages guarantee that type errors will not occur, and not all dynamically typed languages allow type errors. For that distinction, we need strong typing.

5   Strong vs. Weak Typing

As a preface, note that "strong" and "weak" typing are used in many different ways by different languages. We will use the definition usually used in language semantics, but bear in mind that other definitions are used "in the wild". Some language propaganda uses the term "strong typing" so loosely that it basically just means "whatever we do that our major competitor doesn't do"; I'm looking at you, Python.

In our definition, a strongly typed language is a language in which type errors are guaranteed not to happen at run-time. In terms of semantics, a language is strongly typed if we can reject all programs which would get stuck without having to actually take the steps and get stuck. A weakly typed language allows type errors at run-time, or semantically, may get stuck.

As it turns out, when we attempt to apply this definition outside of semantics, it is, at best, wishy-washy. Generally speaking, OCaml is considered to be a strongly typed language. After all, there is no way to compile a program that attempts to add two lists with `+`, so that kind of type error simply cannot arise at run-time. However, OCaml will let you compile a match expression that misses cases, and that can be argued to be a type error that occurs at run-time, so perhaps it would be better to say that the language "OCaml programs which compile without warnings" is strongly typed? Alas, not even that is true: the type of numbers which are valid divisors in division is "all numbers except zero" (that is, we can't divide by zero), but OCaml doesn't have such a type, and doesn't force you to check before dividing.

This problem descends quickly into philosophy. Is division by zero a type error, or some other kind of error? That just depends on whether you choose to accept "non-zero number" as a type, and almost no programming languages do. This makes the definition of strong typing circular: type errors are guaranteed not to happen, and type errors are those errors that we can prevent from happening.

Similarly, Java is usually considered strongly typed, but it allows any reference type to have the value `null`, even though trying to use `null` as that reference type will usually raise a type error.

Conversely, C and C++ are usually considered to be weakly typed, but they do prevent many type errors.

In spite of being quite loosely defined, strong and weak typing are nonetheless useful categories; after all, catching 99% of a certain category of error is better than catching 0%. As mentioned, OCaml and Java are generally considered strongly typed. Not all statically typed languages are strongly typed: C and C++ allow you to cast pointers in any way you please, and more importantly, to attempt to use them after they've been deleted or before they've been allocated, so they are certainly weakly typed. Most dynamically typed languages are considered to be weakly typed, including all of the dynamically typed languages listed above.

6   Memory Safety

Typing and memory safety are commonly conflated, but are not the same thing. Memory safety is not meaningful in formal semantics, because in formal semantics, we don't have memory in this sense (i.e., RAM). Informally, a memory-safe language is a language in which data written to a particular location in memory with one type can never be read or overwritten with an incompatible type until that memory is deallocated and reallocated, and references cannot be held to memory after it's been deallocated or before it's been allocated. Most languages with automatic memory management (such as garbage collection) are memory safe, such as Java. Most languages with manual memory management (`malloc` and `free`) are not memory safe, such as C and C++. Some languages combine the two, having memory-safe and non-memory-safe parts, such as Rust. As memory safety is barely meaningful for semantics, we won't discuss it again until Module 10.

7   Pathological Cases Collapse Everything (Or: This is All Meaningless)

Is assembly language statically typed or dynamically typed[4]? We don't ascribe a type to every register, but in fact, we *do* write types... implicitly, in the operation. In MIPS assembly, for example, using `mult` is an indication that our two registers both contain 32-bit signed integers, and using `multu` is an indication that they contain 32-bit unsigned integers. Using `lw` or `sw` is an indication that a register contains a pointer. This author likes to use the term "operational types" to distinguish this kind of language where types are on the operations instead of on the values, since he finds the static-dynamic categorization poorly applicable.

Is assembly language strongly typed or weakly typed? If I try to load a word from address 0, then that will crash. Except, that's not a property of assembly language, it's a behavior of the memory management unit (MMU); and,

---

[4]Setting aside so-called "typed assembly language" dialects.

if you're using a standard such as POSIX, there may be a well-defined set of steps to take (raising a segmentation fault), from which your program can recover and continue to run. So, if we formally model assembly language *and* POSIX, then our semantics wouldn't even get stuck here. And, on a sufficiently limited system without an MMU, assembly code cannot crash at all[5], which would seem to trivially classify it as strongly typed. You can't get type errors at run-time if you can't get any errors at run-time 🙎.

Are shell scripts statically typed or dynamically typed? This may seem like an absurd question, but consider this: all variables in the shell contain strings. Sometimes those strings may, to the programmer, represent numbers or lists or anything else, but as far as the shell is concerned, they're strings. So, it's easy to reason about types in a shell script statically: $\forall x. x \in$ strings. More broadly, it's hard to classify "singly typed" programming languages as statically or dynamically typed, because it's meaningless to reason about such a paucity of types. But, this pathology can easily be extended to describe all dynamically typed languages as statically typed as well: all values are of the type "all values".

Are shell scripts strongly typed or weakly typed? There is no such thing as a type error in the shell, since everything is a string. A command may not be found, but there's a well-defined behavior for what to do when a command isn't found; nothing "gets stuck", and a shell script will always run to completion, even if every step along the way fails catastrophically[6].

The issue we're circling around is that getting stuck isn't actually meaningful. More precisely, getting stuck is a property of our formal semantics, and not a property of the language that our semantics models. A programming language cannot simply blow up the Universe, so it has to do *something* in all of the circumstances that we would define as "getting stuck" in formal semantics. Perhaps it throws an exception (like Java with `null`), perhaps it raises a signal (like assembly and C on POSIX-compliant systems), or perhaps it simply sets a flag (like a command not found in shell), but it doesn't "get stuck".

This is a mismatch that arises naturally from attempting to mathematically model a real system, and is not avoidable or resolvable. It is simply important to sometimes come up for air, so to speak, and determine whether you've defined your semantics in a way that getting stuck models something you really care about and want to avoid, or it's simply a mathematical anomaly.

> **Aside:** You will sometimes see languages divided into "compiled" vs. "interpreted" languages, and often those terms will be associated with static or dynamic types. However, compilation or interpretation is not a property of the language at all, but its implementation. C is usually classified as a compiled language, but C interpreters, such as PicoC, exist as well. Smalltalk is usually classified as an interpreted language, but most Smalltalk "interpreters" are actually Just-in-Time compilers, which are, well, compilers. I would recommend avoiding the terms "compiled language" and "interpreted language" entirely, since they conflate languages and implementations.

We will set aside this discussion by focusing on defined semantics, rather than languages per se. We can argue over whether a particular implementation of assembly with POSIX is strongly typed, but a particular set of formal semantics either can get stuck or it cannot. In that context, in the rest of this module, we will only discuss statically typed, strongly typed languages.

8 The Simply-Typed λ-Calculus

To begin our study of type systems, we need a typed language. We create one by augmenting the λ-calculus with a simple sublanguage of types. We call the resulting language the simply-typed λ-calculus[1].

---

[5]It can jump to an invalid instruction, but the CPU always has some well-defined behavior for invalid instructions.

[6]Unless the -e option is used, and of course, completion may be `exit` rather than the end of the script.

The simply-typed λ-calculus has the following syntax:

$$\langle Expr \rangle ::= \langle Var \rangle \mid \langle Abs \rangle \mid \langle App \rangle \mid (\langle Expr \rangle)$$
$$\langle Var \rangle ::= a \mid b \mid c \mid \cdots$$
$$\langle Abs \rangle ::= \lambda \langle Var \rangle : \langle Type \rangle . \langle Expr \rangle$$
$$\langle App \rangle ::= \langle Expr \rangle \langle Expr \rangle$$
$$\langle Type \rangle ::= \langle PrimType \rangle \mid \langle Type \rangle \rightarrow \langle Type \rangle$$
$$\langle PrimType \rangle ::= t_1 \mid t_2 \mid \cdots$$

The core syntax of λ-calculus is largely unchanged. The only difference is that we now associate a type with the variable of each abstraction. The intuition is that abstractions will only accept arguments whose type matches the type of the variable—i.e., the formal parameter.

In addition to the core calculus, we now have a small language of types. Types in this language come in two forms: primitive and constructed. A *primitive type* is a type that is "built into" the language, and not built from any other types. Examples in real programming languages usually include types such as `int`, `float`, and `char`. When we introduce primitive values to our semantics, we will also include their types among the primitive types, but the simply-typed λ-calculus has no primitive values. So, we describe primitive types abstractly, using lowercase t, possibly subscripted. For instance, the identity function over the type t_1 is $\lambda x : t_1 . x$. To be clear, in the simply-typed λ-calculus, there are no *values* of the type t_1—it's a useless, empty type—but we define it so have a language for defining useful types, such as boolean, later.

A *constructed type* is a type that is built from other types. A constructed type also includes a *type constructor*, that indicates how it is built from its constituent type(s). For the simply-typed λ-calculus, there is only one kind of constructed type: the function type, whose type constructor is an arrow ($\rightarrow$). We will consider other kinds of constructed types later in this module, and in later modules. When we need to describe an arbitrary type, without knowing exactly which, we will use τ (tau), possibly subscripted. So, given types τ_1 and τ_2, the constructed type $\tau_1 \rightarrow \tau_2$ represents the type of functions with parameters of type τ_1 and results of type τ_2. t_1 is distinct from τ_1 because t_1 is a specific type, albeit a useless one, while τ_1 is the metavariable we use to describe any type, for instance in $\forall \tau_1$. τ is not a part of the simply-typed λ-calculus, but t is.

Although there are no actual primitive values in the simply-typed λ-calculus, it is still valid to define an abstraction as having a primitive type for its formal parameter. This abstraction will not be callable, because no value will ever be able to inhabit this variable, but abstractions *are* values, so it can still be used as a piece of data, even if it can't really be used as a function. This is important, because abstract types such as $\tau_1 \rightarrow \tau_2$ are not part of our language of types itself, so every type must be fully resolved to some construction of primitive types, and the a value of the type $t_1 \rightarrow t_2$ is an abstraction with a formal parameter type of t_1. Of course, this means that types are essentially a toy in the simply-typed λ-calculus, but this language will act as a baseline to build more interesting languages, with primitives. Thus, our previous example of $\lambda x : t_1 . x$ is by definition the identity function over t_1, but since no value can occupy t_1, it's not a usable function.

The core operations of substitution, α-conversion, β-reduction, and η-reduction from the untyped λ-calculus carry over to the simply-typed λ-calculus unchanged, simply ignoring the type on abstractions. Instead of changing these, we will be defining a new property: we are interested in determining whether an expression is *well-typed*, and if so, what its type is.

As we are discovering types to avoid the problem of getting stuck, what it means for an expression to be well-typed is that when evaluated (reduced), it will not get stuck. What it means for an expression to have a given type is that, when evaluated, if the evaluation reaches a terminal value, that value will be of the given type. That is, we can predict the type of the value that is the result of an expression prior to executing it. Note that it is possible for an expression to be well-typed but not reach a terminal value, by reducing infinitely. Generally, we will not attempt to guarantee termination with types, though there are type systems that do this as well.

To determine whether an expression is well-typed, we will define a set of type rules, formulated as a Post system, and use these rules to derive types syntactically. These rules are independent of the rules to reduce the expression; the goal is that we can determine an expressions type *without* reducing it.

To begin, we introduce the notion of a *type environment*, which is usually denoted as Γ (uppercase gamma), also called a set of *type assumptions*. Type environments are used to supply types for free variables, which rely on external context for their semantics. Specifically, for instance, a type environment for the body of an abstraction will associate the variable of the abstraction with the type it's been ascribed. A type environment is essentially a lookup table that, given a variable name, returns its type. We can model a type environment as a list of $\langle name, type \rangle$ pairs, or as a $name \to type$ map. As a list, it is written as an "ordered set", because the order of two elements relative to each other usually doesn't matter, but if they have the same name, then the first is preferred; this is because names can be reused, and we need to make sure to bind a variable to its innermost definition. We will describe type environments as ordered sets in this module, using set syntax.

Our goal is to use a type environment and an expression to make a *type judgment*. A type judgment will take the form $\Gamma \vdash E : \tau$. This judgment denotes the statement, "Under the type assumptions Γ, expression E has type τ". The symbol $\vdash$ is called a turnstile, and is used in logic to indicate that the right-hand side can be derived from the left-hand side. The following is an example of a type judgment, in a language with at least the primitive type int:

$$\{\langle \text{x}, \text{int} \rangle, \langle \text{y}, \text{int} \to \text{int} \rangle\} \vdash \text{y x} : \text{int}$$

This judgment means that applying the function y to the argument x (denoted by y x) under the given type environment (where x is of type int and y is of type int $\to$ int) will yield an expression of type int.

Now, let's define the set of Post rules that we will use to derive type judgments in the simply-typed λ-calculus. The first rule we consider determines the type of a single variable. Since a lone variable is necessarily free, its type information can only come from the type environment Γ. Thus, we find the type of a lone variable by looking it up in the environment. We'll start with an obvious definition, but we will have to slightly correct this definition later:

$$\text{T\_VARIABLEPRELIM} \quad \frac{\langle x, \tau \rangle \in \Gamma}{\Gamma \vdash x : \tau}$$

The "T_VARIABLEPRELIM" next to the rule is a name for the rule. It's common, but not universal, to name type rules and semantic rules with a distinct prefix or suffix—in this case, the T_ prefix for type rules—in order to easily distinguish them. Informally, this rule says "if the variable x is associated with the type τ in Γ, then the expression x has the type τ in the type environment Γ".

Next, we consider the type rule for abstractions. Since an abstraction denotes a function, it must have some type $\tau_1 \to \tau_2$, where τ_1 is the type of the parameter and τ_2 is the type of the result. To determine the type of the result, however, is to perform a type judgment on the body of the abstraction; just like many of our semantic rules required that a subexpression take a step, our type judgment may depend on a type judgment for a subexpression. The rule is as follows:

$$\text{T\_ABSTRACTION} \quad \frac{\{\langle x, \tau_1 \rangle\} + \Gamma \vdash E : \tau_2}{\Gamma \vdash (\lambda x : \tau_1. E) : \tau_1 \to \tau_2}$$

Informally, this rule says "if expression E has type τ_2 in the type environment formed by extending Γ with the pair $\langle x, \tau_1 \rangle$, then the expression $\lambda x : \tau_1. E$ has the type $\tau_1 \to \tau_2$ in the type environment Γ". That is, the type of a function is the constructed ($\to$) type in which the parameter type is the explicitly defined parameter type, and the result type is the result of a type judgment of the body, with the variable name of the argument having the parameter type. Note that we use + with ordered sets as an ordered union, with the left-hand side coming first.

Now, let's reexamine our variable rule. Consider the expression $\lambda x : t_1. \lambda x : t_2. x$. Assuming we start with an empty type environment, the inner x will be judged in the type environment $\{\langle x, t_2 \rangle, \langle x, t_1 \rangle\}$. The premise of T_VARIABLEPRELIM is $\langle x, \tau \rangle \in \Gamma$. In this example, there are two values of τ for which this is true: t_1 and t_2. Thus, both $\Gamma \vdash x : t_1$ and $\Gamma \vdash x : t_2$ would be true. In some contexts it can be correct for an expression to be judged to have multiple types, but in most, this is a mistake. Here, it's certainly a mistake, since the inner x is bound only to the declaration of x with type t_2, and not to the declaration of x with type t_1. We only wanted the *first* instance of x in our environment. We will write $\Gamma(x) = \tau$ as a shorthand for "the first entry of x in Γ is paired with τ". Now, we can rewrite our variable rule with this restriction:

$$\text{T\_VARIABLE} \quad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

And, we can now discard the T_VARIABLEPRELIM rule in preference of T_VARIABLE.

Exercise 1. Write the formal rules for $\Gamma(x) = \tau$.

Finally, we consider the rule for applications. A function should only accept arguments of the same type as its formal parameter. In other words, if we have a function with formal parameter x of type τ_1 and result type τ_2, and we want to apply this function to an argument y, then y should be required to have type τ_1 as well. The type of the application itself should then be the type of the result of the function, τ_2. Note that this means our type judgment is acting both as a way of discovering the type of the whole expression and as a way of restricting the type of certain subexpressions; it is in this way that we prevent programs with incorrect types from compiling, by refusing to give them types. The application rule is formalized as follows:

$$\text{T\_APPLICATION} \quad \frac{\Gamma \vdash E_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash E_2 : \tau_1}{\Gamma \vdash E_1 E_2 : \tau_2}$$

In this case, the rule is read, "If E_1 has type $\tau_1 \to \tau_2$ in the type environment Γ, and E_2 has type τ_1 in the type environment Γ, then the expression $E_1 E_2$ has type τ_2 in the type environment Γ".

We now have a system for deriving the type of an expression in the simply-typed λ-calculus. Expressions for which no type judgment can be derived from these Post rules are not well-typed—that is, they possess type errors—and are not semantically valid expressions in the simply-typed λ-calculus.

Although we won't prove it here, it is also important to note that our type judgment is *decidable*. By Rice's theorem, we cannot decide whether an expression will get stuck, because it could take an unlimited number of steps to get there; but, we *can* decide whether an expression is well-typed. So, we would like to guarantee that no well-typed program gets stuck.

> Video 4.1 (https://student.cs.uwaterloo.ca/~cs442/W25/videos/4.1/): Type judgment

9 Type Safety

A programming language is said to be *type safe* (or *type sound*) if a well-typed program can't "go wrong" due to types. Specifically, this means that the semantics won't get stuck for any well-typed program, and the type predicted by the type judgment reflects the actual type of the value produced. These ideas are formalized as *progress* and *preservation*. The precise statement of progress and preservation depends on the semantics under scrutiny, so we will describe progress and preservation for the simply-typed λ-calculus.

Theorem 1 (Progress). Let E be a closed, well-typed expression in the simply-typed λ-calculus. That is, for some τ, we have $\{\} \vdash E : \tau$. Then, either E is a value, or there is an expression E' such that $E \to_\beta E'$.

Theorem 2 (Preservation). Let Γ be a type environment and P and Q be λ-expressions such that $P \to_{\beta\eta}^* Q$ (that is, P reduces to Q by a sequence of zero or more β-reductions and η-reductions). If $\Gamma \vdash P : \tau$ for some type environment Γ and type τ, then $\Gamma \vdash Q : \tau$.

Progress guarantees that well-typed terms do not get stuck: if E is well-typed, then E cannot be, for example, $(+ \; (\text{cons } 1 \; \text{empty}) \; (\text{cons } 2 \; \text{empty}))$, for which there is no reduction rule. Progress for the simply-typed λ-calculus is trivial, since no closed λ-expressions get stuck without our primitive additions, but we will prove it anyway in a moment; it is the addition of primitives that makes progress interesting. Our original goal was to be able to know that a program will not get stuck, and progress alone isn't quite sufficient for this. While progress guarantees that any E that is well-typed is not stuck, it does *not* guarantee that the E' it reduces to is well-typed, and thus does not guarantee that E' is not stuck.

Preservation, also known as the Subject-Reduction Theorem, guarantees that our predicted type is correct and consistent; that is, if we predicted that an expression has type τ, and we take a step of evaluation, then it still has type τ.

With preservation and progress together, we can accomplish our original goal, to guarantee that a program will not get stuck. Progress says that a well-typed E is not stuck, and reduces to E'. Preservation says that E' is also well-typed. Progress says that E' is, therefore, also reducible, etc. Read together, this means that if we can make a type judgment for an expression, then that expression either reduces forever or reduces to a value of that type, and does not get stuck; that is type safety.

Although it's not obvious, we've actually excluded the possibility that it reduces forever in the simply-typed λ-calculus, but this is a terrible sacrifice! We will discuss that problem in the next section.

We will now prove progress and preservation for the simply-typed λ-calculus.

Proof of Progress. Let E be a closed λ-expression of type τ. The proof is by induction on the length of the type derivation for E. Since E is closed, E cannot be a variable. If E is an abstraction, then E is a value, and we are done. Thus, the only interesting case is when E is an application. Let $E = E_1 E_2$. By the T_APPLICATION rule, there is a type τ_1 such that $\{\} \vdash E_1 : \tau_1 \to \tau$, and $\{\} \vdash E_2 : \tau_1$. By induction, either E_1 is a value, or E_1 is reducible, and similarly for E_2. If E_1 is reducible, we have $E_1 \to_\beta E_1'$, and so $E = E_1 E_2 \to_\beta E_1' E_2$, and thus E is reducible. If E_2 is reducible, we have $E_2 \to_\beta E_2'$, and so $E = E_1 E_2 \to_\beta E_1 E_2'$, and thus E is reducible. Otherwise, both E_1 and E_2 are values, and thus abstractions. Thus, E_1 is an abstraction, and we have previously established that it is well-typed. By the T_ABSTRACTION rule, there is some E_3 such that $E_1 = \lambda x : \tau_1 . E_3$. By the unconditional application rule of β-reduction, $E = (\lambda x : \tau_1 . E_3) E_2 \to_\beta E_3[E_2/x]$, and thus E is reducible. Progress now follows by induction. $\square$

Proof of Preservation. We prove the result in the case of a single reduction step $P \to_{\beta\eta} Q$. The stated result then follows by iteration. We prove the result by induction on the structure of P. Note first that P cannot be a variable, for then P would not be reducible. There are thus five cases to consider, which arise from the productions of β- and η-reduction, each rewritten here for easy recollection:

$$ 1. \quad \frac{M \to_\beta P}{MN \to_\beta PN} $$

There exist some P_1, P_2, P_1' such that $P = P_1 P_2$, $P_1 \to_{\beta\eta} P_1'$, $Q = P_1' P_2$.
Then by T_APPLICATION there is a type τ_1 such that $\Gamma \vdash P_1 : \tau_1 \to \tau$ and $\Gamma \vdash P_2 : \tau_1$. By induction, since $P_1 \to_{\beta\eta} P_1'$, we have $\Gamma \vdash P_1' : \tau_1 \to \tau$. Thus, by T_APPLICATION, $\Gamma \vdash P_1' P_2 : \tau$, i.e., $\Gamma \vdash Q : \tau$.

$$ 2. \quad \frac{N \to_\beta P}{MN \to_\beta MP} $$

There exist some P_1, P_2, P_2' such that $P = P_1 P_2$, $P_2 \to_{\beta\eta} P_2'$, $Q = P_1 P_2'$.
Similar to case 1, but by induction, since $P_2 \to_{\beta\eta} P_2'$, we have $\Gamma \vdash P_2' : \tau_1$. Thus, by T_APPLICATION, $\Gamma \vdash P_1 P_2' : \tau$, i.e., $\Gamma \vdash Q : \tau$

$$ 3. \quad \frac{M \to_\beta P}{\lambda x.\, M \to_\beta \lambda x.\, P} $$

There exist some x, τ_1, E, E' such that $P = \lambda x : \tau_1 . E$, $E \to_{\beta\eta} E'$, $Q = \lambda x : \tau_1 . E'$.
Then there is some type τ_2 such that $\tau = \tau_1 \to \tau_2$. By T_ABSTRACTION, we have $\langle x, \tau_1 \rangle + \Gamma \vdash E : \tau_2$. By induction, since $E \to_{\beta\eta} E'$, we have $\langle x, \tau_1 \rangle + \Gamma \vdash E' : \tau_2$. Then, by T_ABSTRACTION, we obtain $\Gamma \vdash (\lambda x : \tau_1 . E') : \tau_1 \to \tau_2$, i.e., $\Gamma \vdash Q : \tau$.

4. $\lambda x.\, Mx \to_\eta M$ (if $x \notin FV[M]$)

There exist some x, τ_1, E such that $P = \lambda x : \tau_1 . Ex$, $Q = E$, $x \notin FV[E]$.
Then there is some type τ_2 such that $\tau = \tau_1 \to \tau_2$. By T_ABSTRACTION, we have $\langle x, \tau_1 \rangle + \Gamma \vdash (Ex) : \tau_2$. Since x has type τ_1 and Ex has type τ_2, by T_APPLICATION, it must be true that $\langle x, \tau_1 \rangle + \Gamma \vdash E : \tau_1 \to \tau_2$. Finally, since $x \notin FV[E]$ (i.e., x does not occur (free) in E), the type of E is not dependent on the type of x, so removing $\langle x, \tau_1 \rangle$ from the type environment cannot affect our type judgment of E. Thus, we obtain $\Gamma \vdash E : \tau_1 \to \tau_2$, i.e., $\Gamma \vdash Q : \tau$.

$$ 5. \quad \frac{}{(\lambda x.\, M)N \to_\beta M[N/x]} $$

There exist some x, τ_1, M, N such that $P = (\lambda x : \tau_1 . M)N$, $Q = M[N/x]$. Then, by T_APPLICATION, $\Gamma \vdash N : \tau_1$, and further, by T_ABSTRACTION, $\langle x, \tau_1 \rangle + \Gamma \vdash M : \tau$. We prove $\Gamma \vdash Q : \tau$ by induction on the structure of M. A λ-expression can be a variable, an abstraction, or an application, so there are three cases:

(a) M is a variable. If $M = x$, then by T_VARIABLE, $\langle x, \tau_1 \rangle + \Gamma \vdash x : \tau_1$. As M was previously shown to be of type τ in this environment, $\tau = \tau_1$. Then, since $Q = M[N/x] = N$ and $\tau_1 = \tau$, $\Gamma \vdash N : \tau_1$ is equivalent to $\Gamma \vdash Q : \tau$. If $M = z \neq x$, then $\langle x, \tau_1 \rangle + \Gamma \vdash M : \tau$ is equivalent to $\Gamma \vdash z : \tau$, since the type of z is not dependent upon the type of x. Then, since $z[N/x] = z$, we have $\Gamma \vdash M[N/x] : \tau$, i.e., $\Gamma \vdash Q : \tau$.

(b) M is an abstraction. Then there exist some y, τ_2, E such that $M = \lambda y : \tau_2 . E$. By performing an α-conversion, we can arrange that $y \neq x$ and $y \notin FV[N]$. From $\langle x, \tau_1 \rangle + \Gamma \vdash M : \tau$ and $M = \lambda y : \tau_2 . E$, we see that there exists some τ_3 such that $\tau = \tau_2 \to \tau_3$, and we have $\langle y, \tau_2 \rangle + \langle x, \tau_1 \rangle + \Gamma \vdash E : \tau_3$. Since $y \notin FV[N]$, we can augment the type judgment for N and obtain $\langle y, \tau_2 \rangle + \Gamma \vdash N : \tau_1$. We can now apply the induction hypothesis and conclude that $\langle y, \tau_2 \rangle + \Gamma \vdash E[N/x] : \tau_3$. By T_ABSTRACTION, $\Gamma \vdash (\lambda y : \tau_2 . E[N/x]) : \tau_2 \to \tau_3$. But since $y \neq x$, $(\lambda y : \tau_2 . E[N/x]) = M[N/x]$. Thus, $\Gamma \vdash (M[N/x]) : \tau_2 \to \tau_3$, and since $\tau = \tau_2 \to \tau_3$, we obtain $\Gamma \vdash Q : \tau$.

(c) M is an application. Then there exist some E_1, E_2 such that $M = E_1 E_2$. Then, by T_APPLICATION, there exists a type τ_2 such that $\langle x, \tau_1 \rangle + \Gamma \vdash E_1 : \tau_2 \to \tau$ and $\langle x, \tau_1 \rangle + \Gamma \vdash E_2 : \tau_2$. Then, by induction, we have $\Gamma \vdash (E_1[N/x]) : \tau_2 \to \tau$ and $\Gamma \vdash (E_2[N/x]) : \tau_2$. Now, by T_APPLICATION, $\Gamma \vdash ((E_1[N/x])(E_2[N/x])) : \tau$, i.e., $\Gamma \vdash ((E_1 E_2)[N/x]) : \tau$. But this is $\Gamma \vdash (M[N/x]) : \tau$, or simply, $\Gamma \vdash Q : \tau$.

Thus, $\Gamma \vdash Q : \tau$.

Preservation now follows by induction. □

10   The Strong Normalization Theorem

The untyped λ-calculus was useful because it is simultaneously very simple and very powerful, able to represent any computation in spite of only having lambdas. We observed that some programs in untyped λ-calculus "got stuck", so we aimed to restrict "acceptable" λ-expressions to those that don't, and defined types and a type judgment to do so. However, the safety that our type system provides us comes at a severe cost in expressive power, as the following theorem, known as the Strong Normalization Theorem, shows:

Theorem 3 (Strong Normalization). Given any type environment Γ, the set of well-typed terms in the simply-typed λ-calculus is strongly normalizing, i.e., given a well-typed expression E, every sequence of reductions starting from E has a finite number of steps.

In other words, it is impossible for well-typed terms in the simply-typed λ-calculus to reduce infinitely. If we cannot construct infinitely reducing expressions, then we have lost computational power. We can no longer simulate an arbitrary Turing machine in the λ-calculus; the typed λ-calculus is not Turing-complete!

We won't prove the Strong Normalization Theorem here, as the proof is quite intricate, but we can intuit about it by considering what type we would give to the simplest infinitely reducing expression, $(\lambda x. xx)(\lambda x. xx)$. In fact, we need only to consider the problem of assigning a type to x in $\lambda x. xx$. In the expression xx, the x in the rator position is being treated as a function, so it must have a type of the form $\tau_1 \to \tau_2$, for some types τ_1 and τ_2. Then, for the expression to be well-typed, the argument to which x is applied must have type τ_1. But, the argument is x itself, and x has type $\tau_1 \to \tau_2$! We cannot express any type τ_1 in the simply-typed λ-calculus such that $\tau_1 \to \tau_2 = \tau_1$. Thus, the type derivation fails. We conclude that self-application cannot occur in the simply-typed λ-calculus. Other expressions with infinite reduction sequences fail to type-check in similar ways.

Because of the Strong Normalization Theorem, languages that are based on the simply-typed λ-calculus (statically typed functional languages), if they are to be Turing-complete, must include built-in facilities for constructing recursive definitions without violating the type system; directly implementing a recursion combinator (in the absence of additional primitives, at least) in these languages would be impossible. However, remember that we only needed the odd Y-combinator because a function could not refer to itself; we had to "solve for" the recursive call

and use the Y-combinator to reach a fixed point. In the λ-calculus, the only name binding is the formal parameter to λ functions. The standard workaround for the Strong Normalization Theorem is to allow a special kind of name binding by which a function may refer to itself, but where the binding is not itself the parameter of a λ function, as that would be the problematic combinator. For instance, OCaml's `let rec` construction creates a recursive (self-referential) binding. Judging types with `let rec` is complicated, because an inner expression can depend on the judged outer type; we will discuss how it's done in the next module. The simpler technique is to require functions to explicitly declare both their parameter and result types, so that the inner judgment doesn't depend on the outer judgment.

11 Polymorphism

Let's consider Church numerals in the simply-typed λ-calculus. Recall that a Church numeral is a two-argument lambda function (that is, a nested abstraction) in which the first argument is a function and the second argument is the value to apply that function to n times, where n is the value of the Church numeral. What type can be ascribed to the Church numeral, and to its arguments?

Well, the f argument needs to be of a function type, and because it can be called on x or on the result of fx, it needs to be of some type $\tau_1 \to \tau_1$, and x needs to be of type τ_1. We can give this the concrete type t_1, making the Church numeral for two, for example, $\lambda f : t_1 \to t_1. \lambda x : t_1. f(fx)$, with type $(t_1 \to t_1) \to t_1 \to t_1$.

Now, what types can we give to, for instance, $[\![*]\!]$? Recall that $[\![*]\!] = \lambda m. \lambda n. \lambda f. \lambda x. m(nf)x$. Let's particularly focus on nf. The type of f has to be $t_1 \to t_1$, because that's how we've just defined Church numerals. That's fine, since that's the type that n expects, so the expression nf is of type $t_1 \to t_1$. m is also a Church numeral, so nf is typed correctly, and so is $m(nf)x$. $[\![*]\!]$ is typable.

But now, let's consider $[\![\hat{\;}]\!]$ Recall that $[\![\hat{\;}]\!] = \lambda m. \lambda n. \lambda f. \lambda x. nmfx$. Let's focus on nm. n is a Church numeral, so the argument type it expects is $t_1 \to t_1$. But, m is also a Church numeral, so it is of type $(t_1 \to t_1) \to t_1 \to t_1$. These types don't match, so $[\![\hat{\;}]\!]$ is untypable!

The problem occurred when we moved from abstract types (some τ_1) to concrete types (specifically t_1). Everything would have type checked fine if we could have said that a Church numeral is ambivalent to exactly what types you pass in, so long as f is a function with domain and range equal to the type of x. So, let's imagine that abstract types are actually part of the language, and work out Church numerals with abstract types.

Now we say that f is of type $\tau_1 \to \tau_1$, and x is of type τ_1. A Church numeral is, therefore, of the type $\forall \tau_1.(\tau_1 \to \tau_1) \to \tau_1 \to \tau_1$. What this means is that x can be of any type at all, so long as it is the *same* type that is both the parameter and result type of f. Importantly, we've put the universal quantifier ($\forall$) on the Church numeral itself, so each Church numeral can have its own τ_1. $[\![*]\!]$ works exactly as it did before, substituting t_1 for τ_1. But now, let's type $[\![\hat{\;}]\!]$.

τ_1 only needs to be consistent *within* any Church numeral; different Church numerals in the same program can have different concrete versions of τ_1. So, for clarity, while examining $[\![\hat{\;}]\!]$, we will say that m is of type $\forall \tau_2.(\tau_2 \to \tau_2) \to \tau_2 \to \tau_2$, and n is of type $\forall \tau_3.(\tau_3 \to \tau_3) \to \tau_3 \to \tau_3$.

Let's start by examining the expression nm. n expects an argument of type $\tau_3 \to \tau_3$, and m is of type $(\tau_2 \to \tau_2) \to \tau_2 \to \tau_2$. These are both abstract types, so can we somehow relate τ_3 to τ_2 to make this possible? Yes! $\tau_3 = \tau_2 \to \tau_2$. Taking this assumption, the result of nm is of type $\tau_3 \to \tau_3$, or equivalently, $(\tau_2 \to \tau_2) \to \tau_2 \to \tau_2$.

nm expects an argument of type $\tau_3 = \tau_2 \to \tau_2$, and f is of type $\tau_1 \to \tau_1$. Can we somehow relate τ_3 to τ_1 to make this possible? Again, yes. $\tau_3 = \tau_1 \to \tau_1$, and thus, $\tau_1 = \tau_2$. Now, nmf is of type τ_3, or equivalently, $\tau_1 \to \tau_1$. x is of type τ_1, so everything is typable.

We see that even for Church numerals to work correctly, we need abstract types to be a part of our *language*, and a part of our type judgment. To type expressions, we need to be able to relate abstract types to each other. Congratulations, we've just invented polymorphism.

The abstraction of code to work over numerous types of data is known as polymorphism, and code that works on data of several types is called *polymorphic*. By contrast, code that is *monomorphic* is not abstracted over types
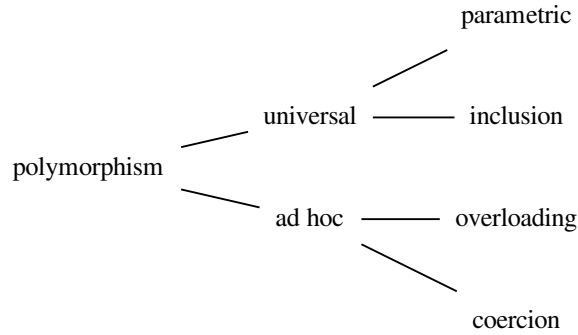
Figure 7: Cardelli-Wegner polymorphism hierarchy

and can only work on data of a single type. For instance, Church numerals must be polymorphic in order to easily be able to perform many kinds of arithmetic with them, since their behavior is how they relate f to x abstractly, without caring precisely what f or x are. A much simpler example of polymorphism is the identity function, $\lambda x.\, x$, which simply evaluates to its argument. Without polymorphism, we would need to write a new version of the identity function for every type in the system, which would be tricky, since there are infinitely many types. Other examples are often similar to Church numerals, in that they need to be able to apply a function to a value without caring about the type of the value.

We didn't need to discuss polymorphism in the context of the untyped λ-calculus, because the problem only arose when we tried to check types. It is not that dynamically typed code is all polymorphic; rather, dynamically typed code is neither polymorphic nor monomorphic, as those are terms from the domain of typing. The kind of code reuse afforded by dynamic typing is qualitatively different from polymorphism. Dynamic type systems, like that of Smalltalk, allow *all* functions to accept arguments of *all* types, but may fail when the values cannot actually behave as the functions expect them to. A polymorphic function may accept arguments of many different types, but it need not accept all types. Polymorphism does not preclude static type checking; rather, it *generalizes* static type checking. Statically typed, polymorphic code can work on a variety of data types, but still offers the guarantee of type safety at run-time; dynamically typed code can produce run-time type errors.

Appropriate to its name[2], polymorphism comes in several forms. In Cardelli and Wegner's paper "On understanding types, data abstraction, and polymorphism", they categorized polymorphism in a hierarchy, illustrated in Figure 7. There are two major varieties of polymorphism: *ad hoc polymorphism*, and *universal polymorphism*. Ad hoc polymorphism is based on constructing multiple implementations of the entity being coded, one for each specific type that it can be used with. As a result, ad hoc polymorphism only permits code to run on a finite and bounded number of data types. By contrast, universal polymorphism, considered by many to be the only true form of polymorphism, is based on constructing a single implementation that is generalized over types in some way. Universal polymorphism allows code to work on an unbounded number of types, possibly including types which may not even have existed when the polymorphic code was written.

Each of these types of polymorphism has two subvariants. Ad hoc polymorphism is further divided into *overloading* and *coercion*. Overloading occurs when the same name is used for different entities (usually functions) inside the same scope. For example, an addition function might be required to work on both integers and real numbers. References to overloaded names are disambiguated at compile-time, via a process known as overload resolution. To determine the exact implementation of an overloaded function to which a given reference is bound, the compiler generally examines the number and type of the function's arguments (as in C++), and, optionally, the return type required by the function's context (as in Ada). The compiler then chooses the implementation that matches these types. Examples should be fairly clear, as most languages call overloading polymorphism "overloading". Overloading in C++ and Java are forms of overloading polymorphism.

Coercion occurs when values of one type are converted to another type, so that an expression makes sense in context. Coercion can be either explicit or implicit. Explicit coercion, otherwise known as casting, occurs when a

programmer forces the type of an expression to change, either through a conversion function or through a built-in casting operator. Explicit coercion is not really a form of polymorphism, since the conversion function or casting operator is equivalent to calling a function with the appropriate argument and return type. Implicit coercion occurs when the compiler changes the type of an expression without any action on the part of the programmer. For example, compilers often coerce integers into floating-point or real numbers so that an addition of the form 3.5 + 4, which attempts to add an integer and a real number, will satisfy the type system. As we expect the result to be the real number 7.5, a compiler could implicitly (i.e., invisibly to the programmer) change the type of 4 to real so that the computation makes sense. Depending on the semantics of the language, coercion can lead to loss of information. For example, in C, if you call a function expecting a `char` with an argument of type `int`, it will happily (and silently) chop off the extra bits. In general, strongly typed languages limit implicit coercion to a few limited, safe cases, or exclude it entirely. but this isn't a fundamental part of strong typing, just a trend.

Universal polymorphism can be divided into two kinds as well: *parametric polymorphism* and *inclusion polymorphism*. Generally considered to be the most powerful and useful form of polymorphism, parametric polymorphism refers to the ability to build abstractions over all types, by constructing objects (generally, functions) that are parameterized (often implicitly) by types. We stumbled upon parametric polymorphism while giving types to Church numerals: τ_1 is, in essence, a type parameter to a Church numeral, and resolving the relationships between abstract types was how we found the argument for that parameter. A parametrically polymorphic static type system can guarantee that, no matter what types are supplied as type parameters, the result will be type safe. We achieved this with Church numerals by assuring that whatever type we related τ_1 to, the relationship between f and x's types was correct. OCaml has parametric polymorphism, and you've probably encountered type errors specifying types such as `'a -> 'a`. Those `'a`s are OCaml's τs! Parametric polymorphism is closely related to the concept of generic programming, which appears in many languages, including Ada, Modula-3 and EL1, and in a more limited form, in Java. We will discuss parametric polymorphism further, and far more formally, in the next module.

Inclusion polymorphism, also called subtyping, is based on the arrangement of types into a lattice, known as a subtype hierarchy. Type τ_1 is a subtype of type τ_2 if values of type τ_1 are valid in every context in which values of type τ_2 can occur. Thus, any function that works on values of type τ_2 can also work on values of type τ_1. Inclusion polymorphism is universal, rather than ad hoc, because new subtypes of τ_2 can be created, without code expecting τ_2 being affected. Functions accepting arguments of type τ_2 are guaranteed to work on these new subtypes. Inclusion polymorphism forms the basis for the kind of polymorphism observed in object-oriented programming. For instance, subclassing in C++ and Java are forms of inclusion polymorphism. We will discuss inclusion polymorphism further as part of our discussion on object orientation, in Module 8.

12 Types in Practice

A type judgment is a mathematical formulation of a *type checker*, which is implemented in a real programming language compiler or interpreter. Generally speaking, type checkers simply do a depth-first search over the code, carrying a stack-like type environment as they go, and determine types from the inside out. Most type judgments work in that way: if the subexpressions have some types, then the whole expression has some type. Parametric polymorphism complicates type checking, and we will investigate the algorithm for type checking in a parametrically polymorphic language in the next module.

More importantly, types are often manifest in how the code is compiled. For instance, on many systems, a C `int` and a C `double` must be stored in different banks of registers to use them, and probably have different sizes as well. In a garbage-collected language, you must communicate to the garbage collector where reference-typed values are stored. And, at the most basic level, if C code accesses a certain field of a `struct`, or code in an object-oriented language calls a particular method of an object, then the compiler must know where and how that is stored. This aspect of types is examined further in CS444, and we will mostly not address it in this course, but we will discuss the peculiarities of type implementations in different paradigms.

13 Semantics Redux

In the previous module, we introduced semantics using universal quantifiers to guarantee ordering ($\forall E_1'.E_1 \nrightarrow E_1'$), which was a bit unsatisfactory, since it makes proofs more difficult. With types, we can now have a more concrete sense of terminal values, so let's redefine our semantics in terms of terminal values. We will denote terminal values syntactically:

$$\langle Term \rangle ::= \langle Abs \rangle$$

Of course, we will add to this definition later. Note that this definition is correct for AOE; NOR's definition would be much more complicated.

We may now rewrite AOE in terms of terminal values:

Definition 1. (Small-Step Operational Semantics of the Simply-Typed λ-Calculus, AOE)

Let the metavariable M range over λ-expressions, and V range over terminal values.

APPLICATION

$$\frac{}{(\lambda x : \tau.\, M)V \to M[V/x]}$$

REDUCELEFT

$$\frac{M_1 \to M_1'}{M_1 M_2 \to M_1' M_2}$$

REDUCERIGHT

$$\frac{M \to M'}{V M \to V M'}$$

Note that we can avoid all universal quantifiers simply by restricting subexpressions which need to have been fully evaluated to V. Since REDUCERIGHT only applies if the rator has been fully evaluated, the rator will always be fully evaluated before the rand. In turn, since abstractions are values and APPLICATION requires the rand to be a value, both REDUCELEFT and REDUCERIGHT will occur before APPLICATION.

Exercise 2. Prove that these semantics are deterministic. That is, for all λ-expressions E_1 and E_2 such that $E_1 \to E_2$, and for all λ-expressions E_3, show that either $E_1 \nrightarrow E_3$ or $E_2 = E_3$ up to α-renaming. The $\to$ here is AOE's $\to$ above, *not* $\to_\beta$ (which is non-deterministic).

14 Adding More Types

In the remainder of this module, we will extend the simply-typed λ-calculus with some commonly seen primitives, to give you an idea of the kinds of type rules you are likely to see in other languages. First, we need to contend with the fact that the simply-typed λ-calculus isn't even Turing-complete, by making recursion possible again.

14.1 Let Bindings

We will add two new constructs to our language: the `let` binding and the `let rec` binding. A `let` binding allows us to define a variable without an actual application. A `let rec` binding allows us to define a variable which is usable within its own definition, thus allowing us to write a recursive function. To simplify typing, our `let rec` bindings will only allow us to bind abstractions, and will require explicitly specifying the return type of the abstraction. Since we didn't discuss `let` bindings in Module 3, we will also give formal semantics for their behavior.

We extend the syntax of the simply-typed λ-calculus as follows:

$$\langle Expr \rangle ::= \cdots$$
$$| \; \text{let} \, \langle Var \rangle = \langle Abs \rangle \, \text{in} \, \langle Expr \rangle$$
$$| \; \text{let rec} : \langle Type \rangle \, \langle Var \rangle = \langle Abs \rangle \, \text{in} \, \langle Expr \rangle$$

In practical languages, the value bound to a variable by a `let` can be any expression, and will be reduced. To simplify our language, we demand that it be a value, and specifically an abstraction, and thus don't need to reduce it.

While λ-calculus has always had variables, variables were resolved by substitution. This is not how most real programming languages work, and will not work for recursive functions. Instead, we need to *store* variables somewhere. Because of this, the way that we do reductions must change. Previously, our "step" morphism ($\rightarrow$) related a λ-expression to a λ-expression: a reduction could always be performed on an expression with no further context. We must add some context, in the form of the variables currently defined. We will do this by changing out step morphism to instead relate pairs, in which a pair is a *store* and a *program*.

Our store will be a partial map, denoted σ (sigma). $\sigma(x)$ is the value of x in the map, and $\sigma[x \mapsto v]$ denotes a new map in which x maps to v, and all other values map as they did in σ. `empty` denotes the empty map. The store will store only our `let` bindings, not other variables, as other variables will continue to operate by substitution.

Our first step to describe a step in the whole program, therefore, is to describe how it relates to stepping with a store:

$$\frac{\langle \text{empty}, E \rangle \rightarrow \langle \text{empty}, E' \rangle}{E \rightarrow E'}$$

That is, we can take a step in E if we can take a step in E with an empty store. Note that the store does not change "across" $\rightarrow$ (i.e., our step starts and ends with an empty store). The purpose of this store is not to describe changes over time (across the arrow), but to describe nested bindings without needing substitution, so we will only see different stores as we go into bindings, not as we progress through the program. That is, we will fill the store in subexpressions of E, not by moving from E to E'.

The semantic rules of AOE can be used verbatim, adjusting for our new syntax:

$$\text{APPLICATION} \quad \frac{}{\langle \sigma, (\lambda x : \tau . M)V \rangle \rightarrow \langle \sigma, M[V/x] \rangle} \qquad\qquad \text{REDUCELEFT} \quad \frac{\langle \sigma, M_1 \rangle \rightarrow \langle \sigma, M_1' \rangle}{\langle \sigma, M_1 M_2 \rangle \rightarrow \langle \sigma, M_1' M_2 \rangle}$$

$$\text{REDUCERIGHT} \quad \frac{\langle \sigma, M \rangle \rightarrow \langle \sigma, M' \rangle}{\langle \sigma, V M \rangle \rightarrow \langle \sigma, V M' \rangle}$$

Note that none of our existing semantic steps use the store. Now, let's define semantics for `let`.

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

$$\text{LETBODY} \quad \frac{x \notin FV[V] \qquad \sigma' = \sigma[x \mapsto V] \qquad \langle \sigma', M \rangle \rightarrow \langle \sigma', M' \rangle}{\langle \sigma, \text{let } x = V \text{ in } M \rangle \rightarrow \langle \sigma, \text{let } x = V \text{ in } M' \rangle}$$

$$\text{VARIABLE} \quad \frac{\sigma[x] = V}{\langle \sigma, x \rangle \rightarrow \langle \sigma, V \rangle}$$

$$\text{LETRESOLUTION} \quad \frac{x \notin FV[V_1]}{\langle \sigma, \text{let } x = V_1 \text{ in } V_2 \rangle \rightarrow \langle \sigma, V_2[V_1/x] \rangle}$$

The most important rule here is LETBODY, which defines how a variable enters σ, as σ'. If a `let` binding maps x to some value V, then a mapping from x to V is added to the store of variables during the reduction of M. M is evaluated with σ' as its store, so it can find x. x cannot be in the free variables of V; if it were, then we would find the wrong x when we use V in M.

The VARIABLE rule describes using variables from the store. Note that there is no VARIABLE rule in normal AOE, because variables are only resolved via substitution; with a store, we need an explicit rule to replace a variable with its value. The two do not conflict in this case because of how AOE evaluates. The VARIABLE rule will only be reached if it is not inside an abstraction, so there cannot be another same-named variable to contend with. However, substitution rules for `let` must be made with similar caveats to those for abstractions.

A `let` is not itself a value, so we also have a LETRESOLUTION rule to allow a fully-resolved `let` binding become a value. Note that because of how variables work in the λ-calculus, the final step to resolve a `let` binding to a value uses substitution; otherwise, any uses of the defined variable in the body would become free. This is just an unusual result of combining substitution with `let` binding, and not a universal feature of languages, and we'll have to use a different approach for `let rec`.

The LETDISTINCT rule is simply to avoid ambiguity between surrounding xs and xs used in V itself. V is defined in the *surrounding* environment (σ), which may very well already have the variable x defined, and so that x may appear in V. When x gets replaced with V in the body of the `let`, we need to make sure that we don't rebind x within V to the wrong binding of x. So, if V uses the surrounding x ($x \in FV[V]$), we change the x that this `let` defines to a fresh variable (z) to assure that x and z are distinct.

Now, let's look at the semantics for `let rec`, which will be mostly similar to `let`.

$$\text{LETRECBODY} \quad \frac{\sigma' = \sigma[x \mapsto V] \qquad \langle \sigma', M \rangle \to \langle \sigma', M' \rangle}{\langle \sigma, \text{let rec} : \tau \; x = V \text{ in } M \rangle \to \langle \sigma, \text{let rec} : \tau \; x = V \text{ in } M' \rangle}$$

$$\text{LETRECINVERT} \quad \frac{x \neq y \qquad x \in FV[M]}{\langle \sigma, \text{let rec} : \tau_1 \; x = V \text{ in } \lambda y : \tau_2. \, M \rangle \to \langle \sigma, \lambda y : \tau_2. \, \text{let rec} : \tau_1 \; x = V \text{ in } M \rangle}$$

$$\text{LETRECRESOLVE} \quad \frac{x \notin FV[V_2]}{\langle \sigma, \text{let rec} : \tau \; x = V_1 \text{ in } V_2 \rangle \to \langle \sigma, V_2 \rangle}$$

The LETRECBODY rule is identical to LETBODY, except that the version of V which is added to σ *may* still refer to x. This is how recursion is achieved. In a future reduction of the `let` binding, the x in the body of V can expand to V again.

There is no VARIABLE rule specific to `let rec`. We can use the VARIABLE rule already defined for `let`. Indeed, there couldn't be a `let rec`–specific VARIABLE rule, since the `let` syntax is not part of the rule for variables anyway.

The LETRECRESOLVE rule is similar to LETRESOLVE, but rather than using substitution to deal with any lingering x's in V_2, they are simply disallowed. We cannot simply remove the `let rec` if the recursive definition is still used.

What we can do in that case is use LETRECINVERT, which applies in all the cases that LETRECRESOLVE does not; if you're not convinced of this, think of the syntax of values and the definition of FV. In order to make the abstraction that this `let rec` resolves to be usable for further applications, while still allowing the recursive function to be used, we can put the `let rec` *inside* the abstraction. This sort of inversion is a common way to resolve semantic corners like this one.

Note that the right-hand side of our new $\to$ never actually changed σ; σ was changed only within subexpressions, relative to the outer expression. In fact, it would have been perfectly possible to define the entire language with $\langle \sigma, E \rangle \to E'$ instead of $\langle \sigma, E \rangle \to \langle \sigma', E' \rangle$. However, this would make the definition of $\to^*$ more complicated, so instead, we've defined $\to$ in a somewhat quirky way such that it never actually changes σ. This sort of quirk arises frequently in formal semantics.

Note also that with `let rec`, we never needed a rule similar to LETDISTINCT. This is simply because in `let rec x = V`, any xs in V *are* bound to this x, not a surrounding x; that is the point, after all!

Now that we've created syntax and semantic rules for `let` bindings, it's time to add type judgments. We can use all previous type judgments as they are; we only need to add new judgments for our new syntax.

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

$$\text{T\_LETREC} \quad \frac{\Gamma' = \{\langle x, \tau_1 \rangle\} + \Gamma \qquad \Gamma' \vdash V : \tau_1 \qquad \Gamma' \vdash E : \tau_2}{\Gamma \vdash (\text{let rec} : \tau_1 \; x = V \text{ in } E) : \tau_2}$$

The T_LET rule is similar to T_ABSTRACTION, but instead of the type being written, it is judged from the value bound to x.

The T_LETREC rules takes that one step further, by using the environment in which x has already been defined to judge the type of V itself. Thus, V can refer to x and still type-check. To do this without an explicit specification of τ_1 is complex, so for this simple calculus, we simply demanded that τ_1 be written, and that it match the actual type of V.

We will not prove progress and preservation for our new let-rec-calculus, but the proof follows a similar line of reasoning as for the simply-typed λ-calculus. More importantly, the let-rec-calculus is Turing-complete again, albeit far more cumbersome to use even than the untyped λ-calculus.

14.2 Booleans and Conditionals

Recall that our boolean and conditional extension in Module 3 added syntax for "true" and "false", "not", "and", "or", and "if" conditionals. The semantic rules can be used verbatim; we only need to add type and value syntax, and type rules.

First, we extend our definition of terminal values and our type language to add a new boolean type:

$$\langle Term \rangle ::= \cdots \mid \text{true} \mid \text{false}$$
$$\langle PrimType \rangle ::= \cdots \mid \text{boolean}$$

To add a specific type for "true" and "false" would require inclusion polymorphism, so we will stick to a single boolean type.

Now, we add type rules. We will start with the booleans themselves, "not", "and", and "or".

$$\text{T\_TRUE} \quad \frac{}{\Gamma \vdash \text{true} \; : \; \text{boolean}} \qquad\qquad \text{T\_NOT} \quad \frac{\Gamma \vdash E \; : \; \text{boolean}}{\Gamma \vdash \text{not } E \; : \; \text{boolean}}$$

$$\text{T\_FALSE} \quad \frac{}{\Gamma \vdash \text{false} \; : \; \text{boolean}} \qquad\qquad \text{T\_AND} \quad \frac{\Gamma \vdash E_1 \; : \; \text{boolean} \qquad \Gamma \vdash E_2 \; : \; \text{boolean}}{\Gamma \vdash \text{and } E_1 E_2 \; : \; \text{boolean}}$$

$$\text{T\_OR} \quad \frac{\Gamma \vdash E_1 \; : \; \text{boolean} \qquad \Gamma \vdash E_2 \; : \; \text{boolean}}{\Gamma \vdash \text{or } E_1 E_2 \; : \; \text{boolean}}$$

The true and false literals are, of course, booleans. As well, "not", "and", and "or" are always of boolean type, but their operands must also be of boolean types for type judgment to succeed.

Now, let's examine "if".

$$\text{T\_IF} \quad \frac{\Gamma \vdash E_1 \; : \; \text{boolean} \qquad \Gamma \vdash E_2 : \tau \qquad \Gamma \vdash E_3 : \tau}{\Gamma \vdash \text{if } E_1 \text{ then } E_2 \text{ else } E_3 : \tau}$$

For "if" to work at all, the condition must be boolean. More importantly, the "then" and "else" branches must have the same type (τ). Note that we don't care what the type of τ *is*: we can simply judge the type of E_2 and E_3, and then make sure they're the same.

14.3 Numbers

We saw previously that it's hard to use Church numerals in the simply-typed λ-calculus without polymorphism. The better solution, of course, is to simply add numbers. Again, the semantics from Module 3 are sufficient, so we only need to add numbers to the terminals and type language, and add type judgments for the new operators.

First, we extend our terminals and type language to add a new natural number type:

$$\langle Term \rangle ::= \cdots \mid 0 \mid 1 \mid \cdots$$
$$\langle PrimType \rangle ::= \cdots \mid \text{nat}$$

And now, we only need two new type judgments:

$$\text{T\_Num} \quad \overline{\Gamma \vdash a : \text{ nat}}$$

$$\text{T\_NumOp} \quad \frac{O \in \langle NumBinOps \rangle \qquad \Gamma \vdash E_1 : \text{ nat} \qquad \Gamma \vdash E_2 : \text{ nat}}{\Gamma \vdash (O \ E_1 \ E_2) : \text{ nat}}$$

Recalling that a is a metavariable over natural numbers, T_Num quite simply says that all natural numbers are nats; this is an axiom. T_NumOp specifies that the result of a numeric binary operation is always a nat (as we have no other kind of number), and its operands must also be nats.

If we added a type for integers, for example, we would need rules for when we switch from natural numbers to integers. For instance, the sum of two nats is a nat, but the difference between two nats is an int.

Exercise 3. Write the syntax, semantics, and type rules for numeric comparison operations, such as $<$, $<=$, $>$, etc. Then, simple equality of $=$ or $==$. What if you want to be able to compare abstractions? What if you want to be able to compare anything to anything?

Video 4.2 (https://student.cs.uwaterloo.ca/~cs442/W25/videos/4.2/): Recursive function type judgment

14.4   Lists

The most critical change we will need for lists is a *constructed* list type: we must be able to distinguish a list of t_1s from a list of t_2s. In many languages, there would also be a simple "list" type, where all lists are of that type, but this would require subtyping, a form of polymorphism, so we'll exclude it. However, this presents a problem: what is the type of "empty"? Our solution will be a bit dubious, and we'll explain why after presenting the types.

First, our extended type language.

$$\langle Type \rangle ::= \cdots \mid \text{list } \langle Type \rangle$$

We may now declare something as, for instance, a "list t_1".

Now, let's extend the terminals so that fully-resolved lists are terminal.

$$\langle Term \rangle ::= \cdots \mid \text{empty} \mid [\langle Term \rangle \langle TermListRest \rangle]$$
$$\langle TermListRest \rangle ::= \varepsilon \mid, \langle Term \rangle \langle TermListRest \rangle$$

Now, the new type judgments. Let's start with cons and empty.

$$\text{T\_Cons} \quad \frac{\Gamma \vdash E_1 : \tau \qquad \Gamma \vdash E_2 : \text{ list } \tau}{\Gamma \vdash (\text{cons } E_1 \ E_2) : \text{ list } \tau} \qquad\qquad \text{T\_Empty} \quad \overline{\Gamma \vdash \text{empty } : \text{ list } \tau}$$

The T_Cons rule should be fairly clear: you can cons an element of type τ to a list of type list τ. But, consider the T_Empty rule carefully: τ is unrestricted! This means that for any type τ, it is true that $\Gamma \vdash \text{empty } : \text{ list } \tau$! This sort of non-deterministic type judgment is a particularly ad hoc form of polymorphism, and is usually not considered acceptable, since type judgments being deterministic makes them far easier to implement in a real language. We'll have to wait for better polymorphism to have a better option, though.

Exercise 4. Correct the semantics for lists from Module 3 to use terminal values instead of universal quantifiers to enforce ordering.

14.5 Sets

The rules for sets are extremely similar to the rules for lists, so we will simply present them:

$$\langle Term \rangle ::= \cdots \mid \text{empty} \mid \{\langle Term \rangle \, \langle TermSetRest \rangle\}$$
$$\langle TermSetRest \rangle ::= \varepsilon \mid , \langle Term \rangle \, \langle TermSetRest \rangle$$
$$\langle Type \rangle ::= \cdots \mid \text{set } \langle Type \rangle$$

$$\text{T\_INSERT} \quad \frac{\Gamma \vdash E_1 : \tau \qquad \Gamma \vdash E_2 : \text{ set } \tau}{\Gamma \vdash (\text{insert } E_1 \ E_2) : \text{ set } \tau} \qquad\qquad \text{T\_REMOVE} \quad \frac{\Gamma \vdash E_1 : \tau \qquad \Gamma \vdash E_2 : \text{ set } \tau}{\Gamma \vdash (\text{remove } E_1 \ E_2) : \text{ set } \tau}$$

$$\text{T\_EMPTYSET} \quad \frac{}{\Gamma \vdash \text{empty } : \text{ set } \tau}$$

15 Fin

In the next module, we will begin our examination of programming language paradigms with functional programming.

References

[1] Harold Abelson and Gerald Jay Sussman. *Structure and interpretation of computer programs.* The MIT Press, 1996.

[2] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)*, 17(4):471–523, 1985.

Rights