Functional Programming Language Design

Christian J. Rudder

January 2025

Contents

This page is left intentionally blank.

Big thanks to **Professor Nathan Mull**
for teaching CS320: Concepts of Programming Languages
at Boston University [1].
Content in this document is based on content provided by Mull.

Disclaimer: These notes are my personal understanding and interpretation of the course material. They are not officially endorsed by the instructor or the university. Please use them as a supplementary resource and refer to the official course materials for accurate information.

Prerequisite Definitions

This text assumes that the reader has a basic understanding of programming languages and grade-school mathematics along with a fundamentals grasp of discrete mathematics. The following definitions are provided to ensure that the reader is familiar with the terminology used in this document.

Definition 0.1: Token

A **token** is a basic, indivisible unit of a programming language or formal grammar, representing a meaningful sequence of characters. Tokens are the smallest building blocks of syntax and are typically generated during the lexical analysis phase of a compiler or interpreter.

Examples of tokens include:

- `keywords`, such as `if`, `else`, and `while`.
- `identifiers`, such as `x`, `y`, and `myFunction`.
- `literals`, such as `42` or `"hello"`.
- `operators`, such as `+`, `-`, and `=`.
- `punctuation`, such as `(`, `)`, `{`, and `}`.

Tokens are distinct from characters, as they group characters into meaningful units based on the language's syntax.

Definition 0.2: Non-terminal and Terminal Symbols

Non-terminal symbols are placeholders used to represent abstract categories or structures in a language. They are expanded or replaced by other symbols (either terminal or non-terminal) as part of generating valid sentences in the language.

- **E.g.**, "Today is ⟨name⟩'s birthday!!!", where ⟨name⟩ is a non-terminal symbol, expected to be replaced by a terminal symbol (e.g., "Alice").

Terminal symbols are the basic, indivisible symbols in a formal grammar. They represent the actual characters or tokens that appear in the language and cannot be expanded further. For example:

- `+`, `1`, and `x` are terminal symbols in an arithmetic grammar.

Definition 0.3: Symbol ":="

The symbol := is used in programming and mathematics to denote "assignment" or "is assigned the value of". It represents the operation of giving a value to a variable or symbol.

For example:

$$x := 5$$

This means the variable x is assigned the value 5.

In some contexts, := is also used to indicate that a symbol is being defined, such as:

$$f(x) := x^2 + 1$$

This means the function $f(x)$ is defined as $x^2 + 1$.

Definition 0.4: Substitution: $[v/x]e$

Formally, $[v/x]e$ denotes the substitution of v for x in the expression e. For example:

$$[3/x](x + x) = 3 + 3$$

This means that every occurrence of x in e is replaced with v. We may string multiple substitutions together, such as:

$$[3/x][4/y](x + y) = 3 + 4$$

Where x is replaced with 3 and y is replaced with 4.

0.1 Semantic Evaluation

0.1.1 Small-step Semantics

In our previous derivations, we've been doing **Big-step semantics**:

Definition 1.1: Big-Step Semantics

Big-step semantics describes how well-formed expressions evaluate to a final value, without detailing each intermediate step.

Notation: We write $e \Downarrow v$ to mean that the expression e evaluates to the value v.
Example:

$$(\text{sub } 10 \ (\text{add } (\text{add } 1 \ 2) \ (\text{add } 2 \ 3))) \Downarrow 2$$

Here, we now introduce **Small-step semantics**:

Definition 1.2: Small-Step Semantics

Small-step semantics describes how an well-formed expressions reduce one step at a time until a final value is reached. **Notation:** $e \to e'$, means that e reduces to e' in a single step. These semantics are housed in a **configuration** with some state-structure and program. We write:

$$\underbrace{\langle S, p \rangle}_{\text{configuration}} \longrightarrow \underbrace{\langle S', p' \rangle}_{\text{transformation.}}$$

Where S is the state of the program and p is the program. The rightarrow shows the **transformation** or **reduction** of the program. For purposes in FP there is no state, hence:

$$\langle \varnothing, p \rangle \longrightarrow \langle \varnothing, p' \rangle$$

Moving forward we **shorthand** this to $p \to p'$ for brevity. Inference rules outline the semantics of the language:

$$\frac{e_1 \to e_1'}{e_1 + e_2 \longrightarrow e_1' + e_2} \ (\text{reduce-left-first})$$

This rule states that if the e_1 position is reducible ($e_1 \to e_1'$), reduce it first. **Additionally**, ϵ (epsilon), e.g., $\langle \varnothing, \epsilon \rangle$ is the empty program with an empty state.

Let's try defining some small-step semantics for a toy-language:

Example 1.1: Defining Grammars in Small-Step Semantics

Say we have some grammar that can only add from 0 to 9:

```
<expr> ::= "(" <op> <expr> <expr> ")"
         | <int>
<op>   ::= add
<int>  ::= 0-9
```

Let's assume our language reads from **left-to-right** and define the semantics of `add`:

- **Both arguments are expressions:**

$$\frac{e_1 \to e_1'}{(\text{add } e_1\ e_2) \to (\text{add } e_1'\ e_2)}\ \text{(add-left)}$$

- **Left argument is an integer:**

$$\frac{n \text{ is an int} \qquad e_2 \to e_2'}{(\text{add } n\ e_2) \to (\text{add } n\ e_2')}\ \text{(add-right)}$$

- **Both arguments are integers:**

$$\frac{n_1 \text{ and } n_2 \text{ are int} \qquad n_1 + n_2 = v \text{ (integer addition)}}{(\text{add } n_1\ n_2) \to v}\ \text{(add-ok)}$$

When writing semantics (in this case **add**) think, "What are all the possible argument states?" If we have `(add 5 e2)`, we must figure out what e2 represents in order to add.

We can almost think of these terminal-symbols as **base cases**. Additionally, since we read left-to-right, we do not need a rule for `(add e1 n)`, where $e1$ is an expression and n is an integer. This scenario is already covered by (add-left).

■

Tip: States can represent data structures like stacks, making them ideal for modeling stack-oriented languages. For example (ϵ is the empty program):

$$\begin{aligned} & (\varnothing,\ \texttt{push 2; push 3; add}) \\ \to\ & (2 :: \varnothing,\ \texttt{push 3; add}) \\ \to\ & (3 :: 2 :: \varnothing,\ \texttt{add}) \\ \to\ & (5 :: \varnothing,\ \epsilon) \end{aligned}$$

Definition 1.3: Multi-Step Semantics

Multi-step semantics captures the idea of reducing a configuration through **zero or more single-step reductions**. We write $C \to^\star D$ to mean that configuration C reduces to configuration D in zero or more steps. This relation is defined inductively with two rules:

$$\frac{}{C \to^\star C}\ \text{(reflexivity)} \qquad \frac{C \to C' \quad C' \to^\star D}{C \to^\star D}\ \text{(transitivity)}$$

These rules formalize:

- Every configuration reduces to itself *(reflexivity)*
- Multi-step reductions can be extended by single-step reductions *(transitivity)*
- If there are multiple ways to reduce $C \to^\star D$, then the semantics are **ambiguous**.

Example 1.2: Proving Multi-step Reductions

We show (add (add 3 4) 5)$\to^\star$ 14 based off semantics defined in Example (1.1). We will do multiple rounds of single-step reductions to yield a result. At each round, read bottom up:

1.
$$\frac{\dfrac{\dfrac{}{\text{add 3 4} \to 7}\ \text{(add-ok)}}{\text{add 5 (add 3 4)} \to \text{add 5 7}}\ \text{(add-right)}}{\text{(add (add 5 (add 3 4)) 2)} \to \text{(add (add 5 7) 2)}}\ \text{(add-left)}$$

2.
$$\frac{\dfrac{}{\text{(add 5 7)} \to 12}\ \text{(add-ok)}}{\text{(add (add 5 7) 2)} \to \text{(add 12 2)}}\ \text{(add-left)}$$

3.
$$\frac{}{\text{(add 12 2)} \to 14}\ \text{(add-ok)}$$

Thus, (add (add 3 4) 5) $\to^\star$ 14. Following rules, we match on the next outermost pattern. Our very first reduction matches with (add-left). In particular, e_1 := (add 5 (add 3 4)), and we see that that's our starting value the next layer up.

Moreover, the trailing 2 in (add (add 5 (add 3 4)) 2), is not evaluated until the very last step (3), as we read from left-to-right. Even though *we can see it*, our patterns do not.

■

Though there is a clear distinction between big-step and multi-step semantics:

Theorem 1.1: Multi-Step vs. Big-Step Semantics

Both Multi-step and Big-step semantics have the same outcome, but differ in meaning.

$$e \to^\star v \quad \approx \quad e \Downarrow v$$

Unlike big-step semantics, small-step semantics concerns all possible patterns, allowing us to choose how we reduce terms, and **in which order**. This eliminates possible ambiguity in in big-step pattern matching.

0.1.2 Lambda Calculus

We briefly touched on **Lambda Calculus** in a previous section when discussing the Anonymous Function Definition (**??**). Here we go more in-depth:

Definition 1.4: Lambda Calculus Syntax

Lambda calculus is a formal system for representing computations using only single-argument functions, avoiding the need for multiple parameters or state. Lambda calculus has three basic constructs:

- **Variables:** x, y, z, etc.
- **Abstraction:** $\lambda x.e$ (a function that takes an argument x and returns expression e).
- **Application:** $e_1\, e_2$ (applying function e_1 to argument e_2).

The OCaml translation:

$$\lambda x.e \quad \equiv \quad \texttt{fun}\ x\ \texttt{->}\ e$$

Replacing '`fun`' with 'λ', and '`->`' with '`.`'. In pure lambda calculus (no additional syntax), **the only value** is a lambda **abstraction**.

Definition 1.5: The Identity Function

The identity function is a function that returns its argument unchanged. Represented as: $\lambda x.x$. If we introduce numbers, $(\lambda x.x)\ 5 \to 5$ (application). In OCaml, this is represented as: `(fun x -> x) 5`. This is also know as the "`I`" combinator.

Before moving on we address a few notational conventions:

Definition 1.6: Symbols $\triangleq$ vs. :=

The symbol := is used to define a variable or expression. The symbol $\triangleq$ is used to state that two expressions are equal **by definition**. For example, we may write a paper which reuses some large specific configuration $\langle \{\dots\}, \dots \rangle$; Instead of writing it again and again, we assign one variable to represent such idea:

$$\Delta^{\star}_{\Pi} \triangleq \langle \{\dots\}, \dots \rangle$$

Now throughout our paper, $\Delta^{\star}_{\Pi}$ signals to the reader that we are using this configuration. As opposed to :=, where we might temporarily assign the variable a to some value multiple times over the course of a document.

Next we look at what happens when we apply the identity function to itself:

Definition 1.7: The Diverging Term Ω

Lambda Calculus is capable of recursion; The backbone of which is the **diverging term**, for which we define as Ω (Omega):

$$\Omega \triangleq (\lambda x.x\, x)(\lambda x.x\, x)$$

The inner function $\lambda x.x\, x$ is sometimes called the **mockingbird combinator**, as it applies its argument to itself:

$$M \triangleq \lambda x.x\, x$$

Thus, $\Omega = M\, M$ creates an infinite loop of self-application.

Example 1.3: Showing Ω Divergence

We can show that Ω diverges by repeated self-application:

$$\begin{aligned}
\Omega &\triangleq (\lambda x.x\, x)(\lambda x.x\, x) \\
&\to (\lambda x.(\lambda x.x\, x)\ (\lambda x.x\, x)) \\
&\to (\lambda x.(\lambda x.(\lambda x.x\, x)\, (\lambda x.x\, x))) \\
&\to (\lambda x.(\lambda x.(\lambda x.(\lambda x.x\, x)\, (\lambda x.x\, x)))) \\
&\to \dots
\end{aligned}$$

Hence, Ω diverges as it continues to apply itself indefinitely. ■

Application has a formal definition in lambda calculus:

Definition 1.8: Application & β-Reduction

A (beta) **β-reduction** is the process of applying a function as an argument in lambda calculus:

1. $$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2}\ \text{(beta-left)}$$
2. $$\frac{e_2 \to e_2'}{(\lambda x.e_1)\ e_2 \to (\lambda x.e_1)\ e_2'}\ \text{(beta-right)}$$
3. $$\frac{}{(\lambda x.e)\ (\lambda y.e') \to [(\lambda y.e')/x]e}\ \text{(beta-ok)}$$

Where (1) we reduce the left-hand side of the application, (2) we reduce the right-hand side of the application, and (3) we apply a function to another function by substitution.

Note: (3) is our base case where **only values** are allowed to be substituted. We can take a different approach and not **eagerly** evaluate expressions.

1. $$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2}\ \text{(beta-left)}$$
2. $$\frac{}{(\lambda x.e)\ e' \to [e'/x]e}\ \text{(beta-ok)}$$

Note: (2) though we don't evaluate e' immediately, we **still** expect it to be a value when evaluating the full expression. As when we evaluate, we are expecting a value, **not some unknown variable**.

Example 1.4: Simple β-Reduction

Consider the following example of β-reduction:

$$(\lambda x.x + 1)\ 2 \quad \to \quad [2/x](x + 1) \quad \to \quad 2 + 1 \to 3$$

Here, we supply the function with 2, substitute 2 for x in the body of the function, and then evaluate the expression, yielding 3. ■

Though we must be wary of what we are substituting for:

Definition 1.9: α-Equivalence

Two expressions are (alpha) **α-equivalent** if they differ only variable names. This formalizes the **principle of name irrelevance**: renaming bound variables does not change the meaning of an expression.

$$\lambda x.\lambda y.x \ =_\alpha \ \lambda v.\lambda w.v$$

In OCaml-like syntax:

$$\texttt{let x = 2 in x} \ =_\alpha \ \texttt{let z = 2 in z}$$

Substitution should preserve α-equivalence. If $e_1 =_\alpha e_2$, then for any term v, we have:

$$[v/x]e_1 =_\alpha [v/x]e_2$$

To continue we make the following distinction:

Definition 1.10: Free & Bound Variables

In lambda calculus, a variable in an expression can be either **free** or **bound**:

- A variable is **bound** if it is defined by a λ abstraction in the expression.
  - **E.g.,** $(\lambda x.x + 1)$, the variable x is bound.
- A variable is **free** if it is not bound by any enclosing λ abstraction.
  - **E.g.,** $(\lambda x.y + 1)$, the variable y is free, **but** x is still bound.

Formally, the set of free variables in an expression e, written $FV(e)$, is defined inductively as:

$$\begin{aligned} FV(x) &= \{x\} && \textit{(Alone variable)} \\ FV(\lambda x.e) &= FV(e) \setminus \{x\} && \textit{(Function body)} \\ FV(e_1\ e_2) &= FV(e_1) \cup FV(e_2) && \textit{(Application)} \end{aligned}$$

In just a moment, we will define substitution in a way that preserves α-equivalence. The high-level idea is that we should **avoid** substituting variables that are **bound** within an expression.

Now we define the semantics of substitution:

Definition 1.11: Substitution Semantics

Substitution replaces **free** occurrences of a variable with another expression. The rules are defined inductively as follows, where in $[v/x]e$, v is assumed to be a **value** (abstraction):

$$(1) \qquad [v/y]x = \begin{cases} v & \text{if } x = y \\ x & \text{otherwise} \end{cases}$$

$$(2) \qquad [v/y](\lambda x.e) = \begin{cases} \lambda x.e & \text{if } x = y \\ \lambda z.[v/y]([z/x]e) & \text{if } x \in \text{FV}(v), \text{ choose a fresh} z \\ \lambda x.[v/y]e & \text{otherwise} \end{cases}$$

$$(3) \qquad [v/y](e_1\ e_2) = ([v/y]e_1)\ ([v/y]e_2)$$

Moreover on (2)'s middle condition. $FV(v)$ means free variables in v, so if $v := (\lambda w.y)$ then $FV(v) = \{y\}$. Then $[v/x](\lambda\ y.x)$ would be a major problem:

$$\lambda y.\lambda w.y \equiv \lambda y.y \neq_\alpha \lambda y.x$$

The middle condition avoids this by swapping in a **fresh variable** z that does not have any conflicts in the body. **E.g.,** $[v/x](\lambda y.x(\lambda z.y))$, $y \in FV(v)$; Observe what happens when we swap out every y for z ignoring the freshness of z:

$$(\lambda z.v(\lambda z.z)) \neq_\alpha (\lambda y.x(\lambda z.y))$$

The abstraction $(\lambda z.y)$ **captures** the subbed z in $(\lambda z.z)$. Now we pick some arbitrary **fresh** variable z, such that no captures occur:

$$(\lambda u.y(\lambda z.u)) =_\alpha (\lambda y.x(\lambda z.y))$$

Here we chose the variable u as it does not conflict with the rest of the expression.

.

Example 1.5: Multi-step β-Reductions

Consider the following derivations of $(\lambda f.\lambda x.fx)(\lambda y.y)$ using our substitution semantics:

1. $$\frac{}{(\lambda f.\lambda x.fx)(\lambda y.y) \to [(\lambda y.y)/f](\lambda x.fx) = (\lambda x.(\lambda y.y)x)} \text{ (beta-ok)}$$

Yielding $(\lambda x.(\lambda y.y)x)$. We **cannot** evaluate inner-terms. I.e., let $w := (\lambda y.y)x$, then we have, $(\lambda x.w)$. Hence, there is no rule in Definition (1.8) that allows further reduction. ■

There are times where we can't evaluate a term

Definition 1.12: Stuck Terms

A **stuck term** is a well-formed expression that cannot be reduced, yet is not a value. Applying a non-function value to an argument often causes such issue:

$$((\lambda x.yx)(\lambda x.x)) \to y(\lambda x.x)$$

Here, the variable y is free, and thus, cannot proceed with application. Hence, we are "stuck" in the evaluation. We can avoid such scenarios via **typing systems** (discussed in Sec. (0.2.1)).

There are two main evaluation strategies in program evaluation:

Definition 1.13: Call-by-Value vs. Call-by-Name

There are two main evaluation strategies:

- **Call-by-value (CBV)** evaluates the argument *before* substitution.
- **Call-by-name (CBN)** substitutes the argument expression *directly* without evaluating it first.

We may illustrate this with the following rules:

$$\text{(CBV)}\quad \frac{e_1 \Downarrow \lambda x.e_1' \quad e_2 \Downarrow v_2 \quad [v_2/x]e_1' \Downarrow v}{e_1\ e_2 \Downarrow v} \qquad \text{(CBN)}\quad \frac{e_1 \Downarrow \lambda x.e_1' \quad [e_2/x]e_1' \Downarrow v}{e_1\ e_2 \Downarrow v}$$

The benefit of CBV is that it **only evaluates an argument once** and is reused. With CBN, the argument is evaluated **every time** it needs to be computed. This is good if an expensive computation is passed around, but barely touched in execution (e.g., expensive edge-case).

We saw this before in Definition (1.8). The first semantics were CBV, while the latter was CBN.

Tip: There are many evaluation strategies optimizing different aspects of computation. In addition to **CBV** and **CBN** there are: **Call-by-need** (lazy eval)—like call-by-name, but avoids recomputation by memoizing results. Used in Haskell. **Call-by-reference**—used in languages with pointers (functions receive variable references). **Call-by-sharing**—also pointer focused langues (functions receive object references).

Definition 1.14: Well-Scopedness and Closedness

We expand on the idea of **scope** in respect to free and bound variables:

- An expression e is **closed** if it contains **no free variables**, also called a **combinator**.
- An expression e is **well-scoped** if the expression is closed or is bound under the configuration's state.

Every closed expression is well-scoped, **but** not every well-scoped expression is closed.

Note: The expression $x \mapsto y$ means that x is mapped to y. Hence, x is bound to y.

Example 1.6: Closed vs. Open Terms

Recall that abstractions bind to their argument variable:

- **Open Term:** $(\lambda x.y)$ is *not closed*, since y is free.
- **Closed Term & Well-scoped:** $(\lambda x.\lambda y.y)$ is *closed*, since both x and y are bound.
- **Well-scoped:** $\langle\{x \mapsto (\lambda y.y)\}, (\lambda w.x)\rangle$ is *well-scoped*, since x is bound in state, but not closed.

■

Definition 1.15: Lexical vs Dynamic Scope

A variable's **scope** determines where in the program the variable can be referenced.

- **Lexical (or static) scope:** Textual delimiters define the scope of a binding.
- **Dynamic scope:** Bindings are determined at runtime based on the call stack. I.e., the most recent binding in the call stack is used regardless was declared.

To understand the difference between lexical and dynamic scoping:

Example 1.7: Dynamic vs Lexical Scoping

Consider the following Bash code:

```
f() { x=23; g; }
g() {  y=$x; }
f
echo $y   # prints 23
```

In Bash, the variable `x` is not defined in `g`, but since `f` called `g` and `x` was defined in `f`, `g` sees it. This is **dynamic scoping**. In contrast, consider the following Python code:

```
x = 0
def f():
    x = 1
    return x

assert f() == 1
assert x == 0
```

Now consider the following OCaml code:

```
let x = 0
let f () =
    let x = 1 in
    x

let _ = assert (f () = 1)
let _ = assert (x = 0)
```

Both Python and OCaml use **lexical scoping**, meaning each use of `x` refers to the closest enclosing definition in the source code, not the caller's environment. ■

Lastly, we can model if statements in lambda calculus:

Theorem 1.2: Lambda Calculus – True-False Conditions

True,false, and if conditions are defined as:

$$\texttt{true} \triangleq \lambda x.\lambda y.x \qquad \texttt{false} \triangleq \lambda x.\lambda y.y$$

$$\texttt{IF} \triangleq \lambda cond.\lambda then.\lambda else.cond\ then\ else$$

E.g.,
(`IF true 1 2`) $\rightarrow [2/y][1/x](\lambda x.\lambda y.x) \rightarrow 1$.
(`IF false 1 2`) $\rightarrow [2/y][1/x](\lambda x.\lambda y.y) \rightarrow 2$.

0.2 Type Theory

0.2.1 Simply Typed Lambda Calculus

An additional way to protect and reduce ambiguity in programming languages is to use **types**:

Definition 2.1: A Type

A **type** is a syntactic object that describes the kind of values that an expression pattern is allowed to take. This happens before evaluation to safeguard unintended behavior.

Recall our work in Section (**??**). We add the following:

Definition 2.2: Contexts & Typing Judgments

Contexts: Γ is a finite mapping of variables to types. **Typing Judgments:** $\Gamma \vdash e : \tau$, reads "e has type τ in context Γ". It is said that e is **well-typed** if $\cdot \vdash e : \tau$ for some τ, where $(\cdot)$ is the **empty context**. Such types we may inductively define:

$$\Gamma ::= \cdot \mid \Gamma, x : \tau$$
$$x ::= vars$$
$$\tau ::= types$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \cdots \quad \Gamma \vdash e_k : \tau_k}{\Gamma \vdash e : \tau}$$

In practice, a context is a set (or ordered list) of variable declarations (variable-type pairs). Our inference rules operate with these contexts to determine the type of an expression:

This leads us to an extension of lambda calculus:

Definition 2.3: Simply Typed Lambda Calculus (STLC)

The syntax of the Simply Typed Lambda Calculus (STLC) extends the lambda calculus by including types and a unit expression.

```
<e>   ::=  () | <v> | <e> <e>
       |   fun "("<v> : <ty>")" -> <e>
<ty>  ::=  unit | <ty> -> <ty>
<v>   ::=  [a-zA-Z]
```

We include the unit type (arbitrary value/void) and that functions are now typed. We transition into a more mathematical notation:

$$e ::= \quad \bullet \mid x \mid \lambda x^{\tau}.\, e \mid e\, e$$
$$\tau ::= \quad \top \mid \tau \to \tau$$
$$x ::= \quad variables$$

This brings us to the typing rules for STLC:

Definition 2.4: Typing Rules for STLC

Typing Rules: The typing rules for STLC are as follows:

$$\frac{}{\Gamma \vdash \bullet : \top}\ (\textsf{unit}) \qquad \frac{\Gamma, x{:}\tau \vdash e : \tau'}{\Gamma \vdash \lambda x^{\tau}.\, e : \tau \to \tau'}\ (\textsf{abstraction})$$

$$\frac{(x{:}\tau) \in \Gamma}{\Gamma \vdash x : \tau}\ (\textsf{variable}) \qquad \frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'}\ (\textsf{application})$$

Such rules enforce that application is only valid when the e_1 position is a function type and the e_2 position is a valid argument type.

When encountering notation, types are often omitted in some contexts:

Definition 2.5: Church vs. Curry Typing

There are two main styles of typing:

Curry-style typing: Typing is **implied** (extrinsic) via typing judgement:

```
fun x -> x
```

Church-style typing: Types are **explicitly** (intrinsic) annotated in the expression:

```
fun (x : unit) -> x
```

Important: Curry-style does not imply polymorphism, expressions are judgement-backed.

This leads us the an important lemma:

Definition 2.6: Lemma – Uniqueness of Types

Let Γ be a typing context and e a well-formed expression in STLC:

$$\text{If } \Gamma \vdash e : \tau_1 \text{ and } \Gamma \vdash e : \tau_2, \text{ then } \tau_1 = \tau_2.$$

I.e., typing in STLC is **deterministic** – a well-typed expression has a **unique type** under any fixed context.

To prove the above lemma we must recall structural induction:

Definition 2.7: Structural Induction

Structural induction is a proof technique used to prove properties of recursively defined structures. It consists of two parts:

- **Base case:** Prove the property for the simplest constructor (e.g., a variable or unit).
- **Inductive step:** Assume the property holds for immediate substructures, and prove it holds for the structure built from them.

This differs from standard mathematical induction over natural numbers, where the base case is typically $n = 0$ (or 1), and the inductive step proves $(n + 1)$ assuming (n).
In the context of **lambda calculus**, expressions are recursively defined and built from smaller expressions. Structural induction proceeds as:

- **Base case:** Prove the property for the simplest expressions (e.g., variables and units).
- **Inductive step:** Assume the property holds for sub-expressions $e_1, e_2, \ldots, e_k$, and prove it holds for a compound expression e (e.g., abstractions and application).

A proof of the previous lemma is as follows:

Proof 2.1: Lemma – Uniqueness of Types

We prove that if $\Gamma \vdash e : \tau_1$ and $\Gamma \vdash e : \tau_2$, then $\tau_1 = \tau_2$ (Γ is a fixed typing context, and e a well-formed expression) via structural-induction:

Case 1: $e = \bullet$ (unit value). We define a generation lemma (*): if $\Gamma \vdash \bullet : \tau \to \tau = \top$. Therefore, if $\Gamma \vdash \bullet : \tau_1$ and $\Gamma \vdash \bullet : \tau_2$, then $\tau_1 = \tau_2 = \top$, by lemma (*).

Case 2: $e = x$ (a variable). We define a generation lemma (**): $\Gamma, x \vdash \tau \to x : \tau \in \Gamma$ Since $(x : \tau_1), (x : \tau_2) \in \Gamma$ and Γ is fixed, x maps to a single type. Thus, $\tau_1 = \tau_2$, by lemma (**).

Case 3: $e = \lambda x^{\tau}.e'$. Abstraction typings require the form $\tau \to \tau'$. Both derivations, $\Gamma \vdash \lambda x^{\tau}.e' : \tau_1$ and $\Gamma \vdash \lambda x^{\tau}.e' : \tau_2$, must have such form. Hence, by inductive hypothesis on e' (under $\Gamma, x : \tau$), we conclude $\tau_1 = \tau_2$.

Case 4: $e = e_1\, e_2$ (application). Suppose $\Gamma \vdash e_1\, e_2 : \tau_1$ and $\Gamma \vdash e_1\, e_2 : \tau_2$. Then e_1 must have type $\tau' \to \tau_1$ and $\tau' \to \tau_2$ respectively, and e_2 type τ'. With likewise reasoning from Case 3, and through the inductive hypothesis on e_1 and e_2, we conclude $\tau_1 = \tau_2$.

Hence, by induction on the typing derivation, the type assigned to any expression is unique.

■

We continue with the following theorems:

Theorem 2.1: Well-Typed Implies Well-Scoped

If e is well-typed in Γ, then e is well-scoped.

Proof 2.2: Well-Typed Implies Well-Scoped

We prove this by induction on the structure of a well-formed expression e:

- Base cases: $e = \bullet$ or $e = x$ (variable), as based on Definition (2.4), maps to a single type. Hence, they are well-typed.
- Inductive cases:
  - $e = \lambda x^{\tau}.e'$: The abstraction argument x is bound and explicitly typed. By the inductive hypothesis, e' is well-typed in $\Gamma, x : \tau$.
  - $e = e_1\, e_2$: Expression e_1 must be a function type $\tau' \to \tau_1$ and e_2 must be of type τ'. By the inductive hypothesis, both e_1 and e_2 are well-typed in Γ.

Therefore by induction, if e is well-typed, then all sub-expressions must also be bound, and hence well-scoped. ■

We've been assuming the following properties of our evaluation relation:

Theorem 2.2: Big-Step Soundness

If $\cdot \vdash e : \tau$ then there is a value v such that $\langle \varnothing, e \rangle \Downarrow v$ and $\cdot \vdash v : \tau$.

Or to be more specific with small-step evaluation:

Theorem 2.3: Progress & Preservation

If $\cdot \vdash e : \tau$, then

- **(Progress)** Either e is a value or there is an e' such that $e \to e'$.
- **(Preservation)** If $\cdot \vdash e : \tau$ and $e \to e'$, then $\cdot \vdash e' : \tau$.

Proof 2.3: Progress & Preservation

We prove the progress and preservation theorem by induction on the structure of a well-formed expression e:

- **Base cases:** $e = \bullet$, $e = x$ (well-scoped variable), or $e = \lambda x^{\tau}.e'$ are values by Definition(2.4), so they satisfy progress and need not reduce.
- **Inductive case:** $e = e_1\, e_2$ (application):
  - *Progress.* By the inductive hypothesis, either e_1 and e_2 are values, or they can take a step. If both are values, then e_1 must be a lambda abstraction (i.e., $\lambda x^{\tau}.e'$), and $e_1\, e_2$ can step to $[e_2/x]e'$, given that e' is the body of e_1. Thus, progress holds.
  - *Preservation.* Suppose $\Gamma \vdash e_1\, e_2 : \tau$. Then by inversion, $\Gamma \vdash e_1 : \tau_1 \to \tau$ and $\Gamma \vdash e_2 : \tau_1$ for some τ_1. Assume $e_1 = \lambda x^{\tau_1}.e'$. Then $e_1\, e_2 \to [e_2/x]e'$. By the typing rule for abstractions, we have $\Gamma, x : \tau_1 \vdash e' : \tau$. Then by the *Substitution Lemma*:

    If $\Gamma \vdash e_2 : \tau_1$ and $\Gamma, x : \tau_1 \vdash e' : \tau$, then $\Gamma \vdash [e_2/x]e' : \tau$.

    Therefore, $\Gamma \vdash [e_2/x]e' : \tau$, which proves preservation for this case.

Hence by induction, if e is well-typed, then either e is a value or there exists e' such that $e \to e'$, and if $e \to e'$, then e' is of the same type as e. ■

Now for some practice:

Example 2.1: Determining Type of an Expression (Part-1)

We determine the smallest typing context Γ for the expression $(\lambda x^{(\top\to\top)\to\top}.\ x(\lambda z^{\top}.\ x(wz)))\ y$:

$$\varnothing \vdash (\lambda x^{(\top\to\top)\to\top}.\ x(\lambda z^{\top}.\ x(wz)))\ y\ :?\qquad \text{(Given)}$$

$$\{y : (\top \to \top) \to \top\} \vdash (\lambda x^{(\top\to\top)\to\top}.\ x(\lambda z^{\top}.\ x(wz)))\ y\ :?\qquad \text{(Application Arg.)}$$

$$\{y : (\top \to \top) \to \top, x : (\top \to \top) \to \top\} \vdash x(\lambda z^{\top}.\ x(wz))\ :?\qquad \text{(Abstraction Type Sub.)}$$

We note that $(\lambda z^{\top}.\ x(wz))$ must be of type $(\top \to \top)$ to satisfy x:

$$\{y : (\top \to \top) \to \top, x : (\top \to \top) \to \top, z : \top\} \vdash x(wz)\ :?\qquad \text{(Application Arg.)}$$

We see (wz) must be type $(\top \to \top)$ as well. We know z is of type $\top$, therefore w must accept such type. In addition, w must return type $(\top \to \top)$ for the application of x to be valid. Hence, we conclude:

$$\Gamma := \{y : (\top \to \top) \to \top, x : (\top \to \top) \to \top, z : \top, w : \top \to (\top \to \top)\}$$

Since x is the outermost abstraction, we can conclude that the output type is $\top$. ■

Example 2.2: Typing an Ocaml Expression

We find the typing context Γ for the following expressions:

```
(*1*) fun f -> fun x -> f (x + 1)      : ?
(*2*) let rec f x = f (f (x + 1)) in f : ?
```

1. We note the application of $f\ (x+1)$. Therefore, f must be an (`int`→`?`), as addition returns an `int`. Subsequently, the x used in such addition and the function argument, must also be an `int`. The rest of the expression (`f (x + 1)`) is some arbitrary type `'a`. Hence:

$$\Gamma := \{f : \texttt{int} \to \texttt{'a}, x : \texttt{int}\}$$

With a final type of `(int -> 'a) -> int -> 'a` for the entire expression.

2. Again, we note the application of $f\ (x+1)$. Therefore, f must be an (`int`→`?`). This function is enclosed within another f yielding $f(f(x+1))$, therefore, f must return an `int` to satisfy the outer f. Hence, we conclude Γ as:

$$\Gamma := \{f : \texttt{int} \to \texttt{int}, x : \texttt{int}\}$$

■

0.2.2 Polymorphism

There are moments when we might redundantly define functions such as:

```
let rec rev_int (l : int list) : int list =
    match l with
    | [] -> []
    | x :: l -> rev l @ [x]
let rec rev_string (l : string list) : string list =
    match l with
    | [] -> []
    | x :: l -> rev l @ [x]
```

Here we have two functions that are identical in structure, but differ in type.

Definition 2.8: Polymorphism

Polymorphism is the ability of a function to operate on values of different types while using a single uniform interface (signature). There are two types:

- **Ad hoc:** The ability to **overload** (redefine) a function name to accept different types.
- **Parametric:** The ability to define a function that can accept any type as an argument, and return a value of the same type.

We will focus on **parametric polymorphism**, as simply overloading in OCaml redefines the function name. An example of a parametric polymorphic function in OCaml is the identity function:

```
let id = fun x -> x (* 'a -> 'a *)
let a = id 0        (* int *)
let b = id (0 = 0)  (* bool *)
let c = id id       (* (int -> int) *)
```

Definition 2.9: Polymorphism vs Type Inference

Polymorphism and type inference are distinct concepts: **polymorphism** allows a function to work uniformly over many types, while **type inference** is the compiler's ability to deduce types automatically. Polymorphism does not require inference, and **inference does not imply polymorphism**.

Additionally, Parametric Polymorphism **cannot be used for dispatch** (inspecting types at runtime).

To implement such, there are two main systems:

Definition 2.10: Implementing Polymorphism

Parametric polymorphism can be implemented in two main ways:

- **Hindley-Milner (OCaml):** Automatically infer the most general polymorphic type for an expression, without requiring explicit type annotations.
- **System F (Second-Order λ-Calculus):** Extend the language to take types as explicit arguments in functions.

Both approaches introduce the concept of a **type variable**, representing an unknown or arbitrary type. For example:

```
let id : 'a -> 'a = fun x -> x
```

Here `'a` is a type variable, and the function `id` can accept any type as an argument and return a value of the same type.

Though we will focus on OCaml, we discuss System F briefly.

Definition 2.11: Quantification

A polymorphic type like `'a -> 'a` is read as:

"for any type `'a`, this function has type `'a -> 'a`."

Also notated as: `'a . 'a -> 'a`, or, $\forall\alpha.\alpha \to \alpha$

System F expands on this idea, providing extended syntax:

Definition 2.12: System F Syntax

The following is System F syntax:

```
e ::= • | x | λx^τ.e | e e | Λα.e | e τ
τ ::= ⊤ | τ → τ | α | ∀α.τ
x ::= variables
α ::= type variables
```

Notably: Λ (capital lambda) refers to type variables in the same way λ refers to expression variables. Moreover, e and τ are expressions and types respectively.

Definition 2.13: Polymorphic Identity Function

The identity function $\lambda x.x$ can be expressed in System F as a polymorphic function:

$$id \triangleq \Lambda\alpha.\lambda x^{\alpha}.x$$

This motivates application: $(id\ \tau) \to^{\star} (\lambda x^{\tau}.x) : \tau \to \tau$. Note $\Lambda\alpha$ is dropped after substitution.

Definition 2.14: System F Typing Rules

The typing rules for System F are as follows:

$$\frac{}{\Gamma \vdash \bullet : \top} \qquad \frac{(x:\tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x:\tau \vdash e : \tau'}{\Gamma \vdash \lambda x^{\tau}.e : \tau \to \tau'} \qquad \frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'}$$

$$\frac{\Gamma \vdash e : \tau \quad \alpha \text{ not free in } \Gamma}{\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau} \text{ (var abstr.)} \qquad \frac{\Gamma \vdash e : \forall\alpha.\tau \quad \tau' \text{ is a type}}{\Gamma \vdash e\tau' : [\tau'/\alpha]\tau} \text{ (type app.)}$$

Unit, variable, abstraction, application, type abstraction, and type application respectively.

Now to define how we handle our substitution:

Definition 2.15: System F Substitution

The rules for substitution in System F are as follows:

$$[\tau/\alpha]\,\top = \top$$

$$[\tau/\alpha]\,\alpha' = \begin{cases} \tau & \alpha' = \alpha \\ \alpha' & \texttt{else} \end{cases}$$

$$[\tau/\alpha](\tau_1 \to \tau_2) = [\tau/\alpha]\tau_1 \to [\tau/\alpha]\tau_2$$

$$[\tau/\alpha](\forall\alpha'.\tau') = \begin{cases} \forall\alpha'.\tau' & \alpha' = \alpha \\ \forall\beta.[\tau/\alpha][\beta/\alpha']\tau' & \texttt{else}\ (\beta \text{ is fresh}) \end{cases}$$

Example 2.3: Typing a System F Expression

We derive the type of $(\Lambda\alpha.\lambda x^{\alpha}.x)\ (\top \to \top)\ \lambda x^{\top}.x$ in System F (read from bottom to top):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{x:\alpha\} \vdash x:\alpha}}{\cdot \vdash \lambda x^{\alpha}.x : \alpha \to \alpha}}{\cdot \vdash \Lambda\alpha.\lambda x^{\alpha}.x : \forall\alpha.\alpha \to \alpha}}{\cdot \vdash (\Lambda\alpha.\lambda x^{\alpha}.x)\ (\top \to \top) : (\top \to \top) \to (\top \to \top)} \qquad \dfrac{\dfrac{}{\{x:\top\} \vdash x:\top}}{\cdot \vdash \lambda x^{\top}.x : \top \to \top}}{\cdot \vdash (\Lambda\alpha.\lambda x^{\alpha}.x)\ (\top \to \top)\ \lambda x^{\top}.x : \top \to \top}$$

■

We switch from doing bottom up proof trees, to a top down file tree structure to save on space:

Definition 2.16: File Tree Derivations

Given the above Example (2.3), we represent it as a file tree:

$$
\begin{array}{l}
\cdot \vdash (\Lambda\alpha.\lambda x^{\alpha}.x)\ \ (\top \to \top)\ \ \lambda x^{\top}.x : \top \to \top \\
\quad\vdash \cdot \vdash \lambda x^{\top}.x : \top \to \top \\
\quad\quad\vdash \{x : \top\} \vdash x : \top \\
\quad\vdash \cdot \vdash (\Lambda\alpha.\lambda x^{\alpha}.x)\ \ (\top \to \top) : (\top \to \top) \to (\top \to \top) \\
\quad\quad\vdash \cdot \vdash \Lambda\alpha.\lambda x^{\alpha}.x : \forall\alpha.\alpha \to \alpha \\
\quad\quad\quad\vdash \cdot \vdash \lambda x^{\alpha}.x : \alpha \to \alpha \\
\quad\quad\quad\quad\vdash \{x : \alpha\} \vdash x : \alpha
\end{array}
$$

Where the conclusion is the root node, each directory level defines the premises for the parent node, and the leaf nodes are the base cases.

Definition 2.17: Hindley-Milner Type Systems Corollary

A Hindley-Milner (HM) enables automatic type inference of polymorphic types of non-explicitly typed expressions. It supports a limited form of polymorphism where type variables are always quantified at the outermost level (e.g., $\forall\alpha.\forall\beta.\alpha \to \beta$, not $\forall\alpha.\alpha. \to \forall\beta.\beta \to \alpha$).

These systems power languages like OCaml and Haskell, and make type inference both **decidable** and fairly **efficient**.

HM does this by employing a constraint-based approach to type inference:

Definition 2.18: Type Inference with Constraints

In Hindley-Milner type inference, we aim to assign the most general type τ to an expression e, while collecting a set of constraints $\mathcal{C}$ that must hold for τ to be valid. If the type of a subexpression is unknown, we generate a fresh type variable to stand in for it.

$$\Gamma \vdash e : \tau \dashv \mathcal{C}$$

Meaning, under context Γ, expression e has type τ if constraints $\mathcal{C}$ are satisfied.

What are constraints?

Definition 2.19: Type Constraint and Unification

A **type constraint** is a requirement that two types must be equal. We write this as:

$$\tau_1 \doteq \tau_2$$

This means "τ_1 should be the same as τ_2." Solving such a constraint—i.e., making τ_1 and τ_2 equal—is called **unification**. In particular, we are unifying τ_1 and τ_2.

Next, to introduce Hindley-Milner syntax:

Definition 2.20: Expressions and Types in Hindley-Milner

We define the syntax of expressions and types under the Hindley-Milner type system:

```
e ::= λx.e | e e
    | let x = e in e
    | if e then e else e
    | e + e | e = e
    | n
    | x

σ ::= int | bool | α | σ → σ  (monotypes)

τ ::= σ | ∀α.τ                (type schemes)
```

Monotypes (σ), are types without any quantification. A type is called **monomorphic** if it is a monotype with no type variables.

Type schemes (τ), allow quantification over type variables via the $\forall$ operator. These represent **polymorphic types**, and a type is polymorphic if it is a closed type scheme; Meaning, it contains no free type variables.

Example 2.4: OCaml Quantification

OCaml is a Hindley-Milner type system, and it allows for polymorphic types. For example, the identity function can be expressed as:

```
let id : 'a . 'a -> 'a = fun x -> x
```

When placed in utop it returns:

```
val id : 'a -> 'a = <fun>
```

■

For now we introduce a reduced form of the Hindley-Milner typing rules:

Definition 2.21: Hindly-Milner Light

Hindley-Milner Light ($\mathbf{HM}^-$) contains the following typing rules:

$$\frac{n \text{ is an integer}}{\Gamma \vdash n : \texttt{int} \dashv \varnothing} \text{ (int)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \quad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash e_1 + e_2 : \texttt{int} \dashv \tau_1 \doteq \texttt{int},\ \tau_2 \doteq \texttt{int},\ \mathcal{C}_1,\ \mathcal{C}_2} \text{ (add)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \quad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash e_1 = e_2 : \texttt{bool} \dashv \tau_1 \doteq \tau_2,\ \mathcal{C}_1,\ \mathcal{C}_2} \text{ (eq)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \quad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2 \quad \Gamma \vdash e_3 : \tau_3 \dashv \mathcal{C}_3}{\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \tau_3 \dashv \tau_1 \doteq \texttt{bool},\ \tau_2 \doteq \tau_3,\ \mathcal{C}_1,\ \mathcal{C}_2,\ \mathcal{C}_3} \text{ (if)}$$

$$\frac{\alpha \text{ is fresh} \quad \Gamma, x : \alpha \vdash e : \tau \dashv \mathcal{C}}{\Gamma \vdash \lambda x.e : \alpha \to \tau \dashv \mathcal{C}} \text{ (fun)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \quad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2 \quad \alpha \text{ is fresh}}{\Gamma \vdash e_1\ e_2 : \alpha \dashv \tau_1 \doteq \tau_2 \to \alpha,\ \mathcal{C}_1,\ \mathcal{C}_2} \text{ (app)}$$

$$\frac{(x : \forall\alpha_1 \ldots \forall\alpha_k.\tau) \in \Gamma \quad \beta_1, \ldots, \beta_k \text{ are fresh}}{\Gamma \vdash x : [\beta_1/\alpha_1] \ldots [\beta_k/\alpha_k]\tau \dashv \varnothing} \text{ (var)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash \texttt{let } x = e_1 \texttt{ in } e_2 : \tau_2 \dashv \mathcal{C}_1, \mathcal{C}_2} \text{ (let)}$$

This differs from the actual Hindley-Milner typing rules as our (let) is not polymorphic.

The (add) rule reads, given a fixed-context Γ, if we have integer addition of two expressions e_1 and e_2, we inspect their types left to right; e_1 is of type τ_1 under constraints $\mathcal{C}_1$. Such constraints may arise if e_1 is complex (e.g., it's another addition which returns its own constraints; Otherwise, if it is an monotype (base-case), the constraint is empty). Then we do the same for e_2.Finally, me make the declaration that τ_1 and τ_2 must be of type `int` (i.e., $\tau_1 \doteq$ `int` and $\tau_2 \doteq$ `int`), and then union any accumulated constraints from both expressions.

The (var) case reads that for any arbitrary number of quantifiers (typing-scheme, $\forall\alpha_1 \ldots \forall\alpha_k.\tau$), we substitute a fresh type variable β_i for each α_i. Therefore for expressions such as, "`if` e_1 `then` e_2 `else` e_3," our place-holder τ_1, τ_2, τ_3 types do not conflict with each other.

Bibliography

[1] Nathan Mull. Cs320: Concepts of programming languages. Lecture notes, Boston University, Spring Semester, 2025. Boston University, CS Department.