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.

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 Type Theory

0.1.1 Simply Typed Lambda Calculus

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

Definition 1.1: 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 1.2: 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:

$$\begin{aligned} \Gamma &::= \cdot \mid \Gamma, x : \tau \\ x &::= \mathit{vars} \\ \tau &::= \mathit{types} \end{aligned} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \cdots \quad \Gamma \vdash e_k : \tau_k}{\Gamma \vdash e : \tau}$$

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 1.3: 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:

$$\begin{aligned} e &::= \quad \bullet \mid x \mid \lambda x^{\tau}.\, e \mid e\, e \\ \tau &::= \quad \top \mid \tau \to \tau \\ x &::= \quad \mathit{variables} \end{aligned}$$

This brings us to the typing rules for STLC:

Definition 1.4: Typing Rules for STLC

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

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

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

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 1.5: 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 1.6: 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 1.7: 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 1.1: 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 1.1: Well-Typed Implies Well-Scoped

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

Proof 1.2: 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 (1.4), 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 1.2: 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 1.3: 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 1.3: 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(1.4), 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 1.1: Determining Type of an Expression (Part-1)

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

$$\begin{aligned}
\varnothing &\vdash (\lambda x^{(\top\to\top)\to\top}.\, x(\lambda z^{\top}.\, x(wz)))\; y \;:\, ? && \text{(Given)}\\
\{y : (\top \to \top) \to \top\} &\vdash (\lambda x^{(\top\to\top)\to\top}.\, x(\lambda z^{\top}.\, x(wz)))\; y \;:\, ? && \text{(Application Arg.)}\\
\{y : (\top \to \top) \to \top, x : (\top \to \top) \to \top\} &\vdash x(\lambda z^{\top}.\, x(wz)) \;:\, ? && \text{(Abstraction Type Sub.)}
\end{aligned}$$

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 1.2: 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 ($\texttt{int}\rightarrow$`?`), 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:

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

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 ($\texttt{int}\rightarrow$`?`). 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:

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

■

0.1.2 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 1.8: 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 1.9: 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 1.10: 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 1.11: 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 1.12: 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 1.13: 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 1.14: 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 1.15: 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 1.3: 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 1.16: File Tree Derivations

Given the above Example (1.3), we represent it as a file tree:

```
$\cdot \vdash (\Lambda\alpha.\lambda x^{\alpha}.x)\ (\top \to \top)\ \lambda x^{\top}.x : \top \to \top$
├─$\cdot \vdash \lambda x^{\top}.x : \top \to \top$
│  └─$\{x : \top\} \vdash x : \top$
└─$\cdot \vdash (\Lambda\alpha.\lambda x^{\alpha}.x)\ (\top \to \top) : (\top \to \top) \to (\top \to \top)$
   └─$\cdot \vdash \Lambda\alpha.\lambda x^{\alpha}.x : \forall\alpha.\alpha \to \alpha$
      └─$\cdot \vdash \lambda x^{\alpha}.x : \alpha \to \alpha$
         └─$\{x : \alpha\} \vdash x : \alpha$
```

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 1.17: 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 1.18: 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 1.19: 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 1.20: 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 1.4: 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 1.21: Hindly-Milner Light

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

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

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

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

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

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

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

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

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

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., $\tau_1 \doteq \texttt{int}$ and $\tau_2 \doteq \texttt{int}$), 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.