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.

— 1 —

The Interpretation Pipeline

1.1 Type Theory

1.1.1 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:

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

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

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 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:

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

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 structure (e.g., base case of recursion).
- **Inductive step:** Assume the property holds for a structure and prove it holds for a more complex structure built from it.

This differs from mathematical induction over natural numbers, where the base case may be 0 and the inductive step is $n \to n+1$. For use in **lambda calculus**, a well-formed expression e may comprise of multiple sub-expressions $e_1, e_2, \ldots, e_k$:

- **Base case:** Prove the property for the simplest expression (e.g., variable or unit value).
- **Inductive step:** Assume the property holds for sub-expressions $e_1, e_2, \ldots, e_k$ and prove it holds for a more complex patterns e built from them (e.g., an application or abstraction).

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). The $\bullet$ is assigned to $\top$ under one rule. Therefore, $\tau_1 = \tau_2 = \top$.

Case 2: $e = x$ (a variable). Variable typings depend on the context Γ. Since $(x : \tau_1), (x : \tau_2) \in \Gamma$ and Γ is fixed, x maps to a single type. Hence, $\tau_1 = \tau_2$.

Case 3: $e = \lambda x^{\tau}.e'$. Abstraction typings require the form $\tau \to \tau'$. Given $\Gamma \vdash \lambda x^{\tau}.e' : \tau_1$ and $\Gamma \vdash \lambda x^{\tau}.e' : \tau_2$, x is a fixed variable in the context Γ. Given abstraction are deterministic the input must give the same output. By the induction hypothesis applied to 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 (case 3) and e_2 type τ'. By the inductive hypothesis on e_1 and e_2, we conclude $\tau_1 = \tau_2$.

Hence, by typing derivation structural induction, assigned types of any expression is unique.
■

Bibliography

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