Functional Programming Language Design

Christian J. Rudder

January 2025

Contents

Contents 1
0.1 Formalizing Ocaml Expressions . . . 3
0.1.1 All Ocaml Formalizations . . . 9
0.1.2 Derivations . . . 10

1 Algebraic Data Types 11

Bibliography 12

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.

0.1 Formalizing Ocaml Expressions

Now we can begin to formalize expressions in OCaml. We again re-iterate what steps are needed to build expressions in our language, given that we have some *context* now.

Definition 1.1: Building Expressions

When creating new expressions we must follow these steps:

1. **Context:** Define variable-to-type mappings.
2. **Syntax:** Establish how the expression/operation should be written.
3. **Typing Rules:** Define the type of the whole expression and its sub-expressions.
4. **Semantics:** Clarify the resulting value/evaluation of the defined expression.

I.e., what are our types, how are they used, what type of data do they represent, and how does it evaluate?

Now we begin to formalize, though we will abstract the context to Γ, assuming all the types we've defined before (**??**).

Definition 1.2: Formalizing Let-Expressions

Let Γ be the OCaml context, and $=$ be mathematical equality, and `=` be an OCaml token:

- **Syntax:** $\langle\text{expr}\rangle ::= \texttt{let}\ \langle\text{var}\rangle\ \texttt{=}\ \langle\text{expr}\rangle\ \texttt{in}\ \langle\text{expr}\rangle$

  If x is a valid variable name, and e_1 is a well-formed expression and e_2 is a well-formed expression. Then `let` x `=` e_1 `in` e_2 is also a well-formed expression.

- **Typing-Rule:** $$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \tau}$$

Given context Γ, if there's some well-formed expression e_1 of type τ_1 and some well-formed expression e_2 of type τ, given a variable declaration of x of type τ_1, then within this context, the expression `let` x `=` e_1 `in` e_2 is of type τ.

- **Semantics:** $$\frac{e_1 \Downarrow v_1 \quad [v_1/x]e_2 \Downarrow v}{\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \Downarrow v}$$

  Following our context Γ, if a well-formed expression e_1 evaluates to v_1 and the substitution of v_1 for variable x in another well-formed expression e_2 evaluates to v, then the expression `let` x `=` e_1 `in` e_2 evaluates to v.

Thus, we have formalized the `let` expression in OCaml.

Before we continue, we introduce the concept of $\top$ and $\bot$.

Definition 1.3: Top and Bottom ($\top$, $\bot$)

In logic and computer science:

- $\top$ is used to represent *true*, *valid*.
- $\bot$ is used to represent *false* or *invalid*.

Specifically, they are the greatest and least element of a lattice/boolean algebra (hence top and bottom), which when it comes to logic means truthhood and falsehood.

We continue with the formalization of the `if` expression in OCaml.

Definition 1.4: Formalizing If-Expressions

Let Γ be the OCaml context, then:

- **Syntax:** $\langle\text{expr}\rangle ::= \texttt{if}\ \langle\text{expr}\rangle\ \texttt{then}\ \langle\text{expr}\rangle\ \texttt{else}\ \langle\text{expr}\rangle$

  If e_1 is a well-formed expression, e_2 is a well-formed expression, and e_3 is a well-formed expression, then `if` e_1 `then` e_2 `else` e_3 is also a well-formed expression.

- **Typing-Rule:** $$\frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}$$

  Given context Γ, let there be well-formed expressions, e_1 of type `bool`, e_2 of type τ, and e_3 of type τ. Then the expression `if` e_1 `then` e_2 `else` e_3 is of type τ.

- **Semantics:** $$\frac{e_1 \Downarrow \top \quad e_2 \Downarrow v}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow v}\ (\text{trueCond.})$$

  Following our context Γ, if a well-formed expression e_1 evaluates $\top$ and another well-formed expression e_2 evaluates to v, then the expression `if` e_1 `then` e_2 `else` e_3 evaluates to v (e_3 a well-formed expression).

- **Semantics:** $$\frac{e_1 \Downarrow \bot \quad e_3 \Downarrow v}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow v}\ (\text{falseCond.})$$

  Following our context Γ, if a well-formed expression e_1 evaluates $\bot$ and another well-formed expression e_3 evaluates to v, then the expression `if` e_1 `then` e_2 `else` e_3 evaluates to v (e_2 a well-formed expression).

Take note that we must write two semantics rules for the `if` expression, one for when the condition evaluates to $\top$ and one for when it evaluates to $\bot$.

We continue with the formalization of the `function` expression in OCaml.

Definition 1.5: Formalizing Functions

Let Γ be the OCaml context, then:

- **Syntax:** $\langle\text{expr}\rangle ::= \texttt{fun}\ \langle\text{var}\rangle\ \texttt{->}\ \langle\text{expr}\rangle$

  If x is a valid variable name and e is a well-formed expression, then $\texttt{fun}\ x\ \texttt{->}\ e$ is also a well-formed expression.

- **Typing-Rule:** $\dfrac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\ x\ \texttt{->}\ e : \tau_1 \to \tau_2}$

  Given context Γ with a variable declaration of $(x : \tau_1)$ added, if there's a well-formed expression e of type τ_2 and, then the expression $\texttt{fun}\ x\ \texttt{->}\ e$ is of type $\tau_1 \to \tau_2$.

- **Semantics:** $\dfrac{}{\texttt{fun}\ x\ \texttt{->}\ e \Downarrow \lambda x.e}$

  Under no premises, the expression $\texttt{fun}\ x\ \texttt{->}\ e$ evaluates to the lambda function $\lambda x.e$.

Definition 1.6: Formalizing Application

Let Γ be the OCaml context, then:

- **Syntax:** $\langle\text{expr}\rangle ::= \langle\text{expr}\rangle\ \langle\text{expr}\rangle$

  If e_1 is a well-formed expression and e_2 is a well-formed expression, then $e_1\ e_2$ is also a well-formed expression.

- **Typing-Rule:** $\dfrac{\Gamma \vdash e_1 : \tau_1\ \texttt{->}\ \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$

  Given context Γ, if there's a well-formed expression e_1 of type $\tau_1\ \texttt{->}\ \tau_2$ (Functions.1.5) and a well-formed expression e_2 of type τ_1, then the expression $e_1\ e_2$ is of type τ_2.

- **Semantics:** $\dfrac{e_1 \Downarrow \lambda x.e \quad e_2 \Downarrow v \quad [v/x]e \Downarrow v'}{e_1\ e_2 \Downarrow v'}$

  Following our context Γ, if a well-formed expression e_1 evaluates to a lambda function $\lambda x.e$, another well-formed expression e_2 evaluates to v, and the substitution of v for x in e evaluates to v', then the expression $e_1\ e_2$ evaluates to v'.

Onto tuples and matching:

Definition 1.7: Formalizing Tuples

Let Γ be the OCaml context, then:

- **Syntax:** $\langle\text{expr}\rangle$ ::= ($\langle\text{expr}\rangle$, . . . , $\langle\text{expr}\rangle$)

  If e_1 is a well-formed expression and e_2 is a well-formed expression, then (e_1, e_2) is also a well-formed expression.

- **Typing-Rule:** $$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash (e_1, e_2, \ldots, e_n) : \tau_1 * \tau_2 * \cdots * \tau_n}$$

  Given context Γ, if there are well-formed expressions e_1 of type τ_1, e_2 of type τ_2, and e_n of type τ_n, then the expression $(e_1, e_2, \ldots, e_n)$ is of type $\tau_1 * \tau_2 * \cdots * \tau_n$.

- **Semantics:** $$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad \ldots \quad e_n \Downarrow v_n}{(e_1, e_2, \ldots, e_n) \Downarrow (v_1, v_2, \ldots, v_n)}$$

  Following our context Γ, if well-formed expressions e_1 evaluates to v_1, e_2 evaluates to v_2, and e_n evaluates to v_n, then the expression $(e_1, e_2, \ldots, e_n)$ evaluates to $(v_1, v_2, \ldots, v_n)$.

Definition 1.8: Formalizing Lists

Let Γ be the OCaml context, then:

- **Syntax:** $\langle\text{expr}\rangle$ ::= `[]` | $\langle\text{expr}\rangle$ `::` $\langle\text{expr}\rangle$

  The empty list `[]` is a well-formed expression. If e_1 is a well-formed expression and e_2 is a well-formed list, then e_1 `::` e_2 is also a well-formed expression.

- **Typing-Rule:** $$\frac{}{\Gamma \vdash [] : \tau\ \texttt{list}}\ (\text{nil}) \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau\ \texttt{list}}{\Gamma \vdash e_1\ \texttt{::}\ e_2 : \tau\ \texttt{list}}\ (\text{cons})$$

  Given context Γ, the empty list `[]` has type τ `list` for any type τ (nil). If e_1 is of type τ and e_2 is of type τ `list`, then the expression e_1 `::` e_2 has type τ `list` (cons).

- **Semantics:** $$\frac{}{\texttt{[]} \Downarrow \varnothing}\ (\text{nilEval}) \qquad \frac{e_2 \Downarrow [v_2, ..., v_k] \quad e_1 \Downarrow v_1}{e_1\ \texttt{::}\ e_2 \Downarrow [v_1, v_2, ..., v_k]}\ (\text{consEval})$$

  The empty list `[]` evaluates to the empty list as a value (nilEval). If e_2 evaluates to the list $[v_2, ..., v_k]$ and e_1 evaluates to v_1, then e_1 `::` e_2 evaluates to the list $[v_1, v_2, ..., v_k]$ (consEval).

Matching in a general sense is complex (deep matching), we can simplify with weak matching:

Definition 1.9: Weak Matching

Weak matching is a form of pattern matching that is for specific cases.

Additionally, we introduce the concept of *side conditions* before we jump into weak matching on lists.

Definition 1.10: Side Conditions in Formal Semantics

A **side condition** is an additional constraint that must be satisfied before applying a rule. Side conditions are used to prevent undefined behavior and ensure correctness in evaluation.

Example 1: Integer Division Rule

- Consider the evaluation rule for integer division:

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v_2 \neq 0}{e_1 \div e_2 \Downarrow v_1 \div v_2}$$

- The side condition $v_2 \neq 0$ ensures that:
  - The denominator v_2 is not zero before performing division.
  - If $v_2 = 0$, the rule cannot be applied to avoid division by zero.
- Without this side condition, the expression could cause an error or undefined behavior.

Example 2: Exponentiation with Non-Negative Exponents

- Consider the evaluation rule for exponentiation:

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v_2 \geq 0}{e_1^{e_2} \Downarrow v_1^{v_2}}$$

- The side condition $v_2 \geq 0$ ensures that:
  - The exponent v_2 is non-negative before applying the exponentiation operation.
  - If $v_2 < 0$, the rule cannot be applied to avoid undefined results in integer arithmetic.
- Without this side condition, expressions like 2^{-3} would be invalid in integer arithmetic.

Side conditions help enforce correctness by restricting operations to only valid inputs.

Definition 1.11: Weak Matching on Lists

Let Γ be the OCaml context, then:

- **Syntax:** $\langle\text{expr}\rangle ::=$ `match` $\langle\text{expr}\rangle$ `with`
  `| [] ->` $\langle\text{expr}\rangle$
  `|` $\langle\text{var}\rangle$ `::` $\langle\text{var}\rangle$ `->` $\langle\text{expr}\rangle$

  If e, e_1, e_2 are well-formed expressions and x, y are valid variable names, then `match` e `with | [] ->` e_1`|` x `::` y `->` e_2 is a well-formed expression.

- **Typing Rule:**

$$\frac{\Gamma \vdash e : \tau' \texttt{ list} \quad \Gamma \vdash e_1 : \tau \quad \Gamma, x : \tau', y : \tau' \texttt{ list} \vdash e_2 : \tau}{\Gamma \vdash \texttt{match } e \texttt{ with | [] -> } e_1 \texttt{| } x \texttt{ :: } y \texttt{ -> } e_2 : \tau}$$

  If e is of type τ' `list` in the context Γ, and e_1 is of type τ in the context Γ, and e_2 is of type τ in the context Γ with $(x : \tau')$ and $(y : \tau'$ `list`$)$ added, then the entire match expression is of type τ.

- **Semantics:**

$$\frac{e \Downarrow \varnothing \quad e_1 \Downarrow v}{\texttt{match } e \texttt{ with [] -> } e_1 \texttt{| } x \texttt{:: } y \texttt{ -> } e_2 \Downarrow v} \text{ (nil)}$$

  If e evaluates to the empty list $\varnothing$ and e_1 evaluates to v, then the entire match expression evaluates to v.

$$\frac{e \Downarrow h \texttt{ :: } t \quad e_2' = [t/y][h/x]e_2 \quad e_2' \Downarrow v}{\texttt{match } e \texttt{ with | [] -> } e_1 \texttt{| } x \texttt{ :: } y \texttt{ -> } e_2 \Downarrow v} \text{ (cons)}$$

  If e evaluates to a nonempty list $h :: t$ with first element h and remainder t, and the expression e_2 with h substituted for x and t substituted for y evaluates to v, then the entire match expression evaluates to v.

0.1.1 All Ocaml Formalizations

Below is the full list from which we will reference throughout the text.

Full Specifications: For a full list of all the formalized expressions we'll be using. This list also includes the next topic we'll discuss **Derivations**:
https://nmmull.github.io/PL-at-BU/320Caml/notes.html

0.1.2 Derivations

Derivations allow us to unpack the formal expressions to prove their validity.

Definition 1.12: Tree Derivations

Tree derivations are a structured way of representing step-by-step reasoning in formal systems. They are often used in type systems, operational semantics, and logic proofs to show how conclusions follow from premises.

Each derivation is represented as a **tree**, where:

- **Leaves** represent axioms or base cases.
- **Internal nodes** apply inference rules to derive new conclusions.
- **The root** represents the final conclusion of the derivation, i.e., the starting point.

Example 1.1: Typing Derivation

Say we wanted to prove the typing derivation: `let y = 2 in y + y` : `int`

$$\dfrac{\dfrac{}{\{\} \vdash 2 : \texttt{int}}\,(\text{intLit}) \qquad \dfrac{\dfrac{}{\{y : \texttt{int}\} \vdash y : \texttt{int}}\,(\text{var}) \qquad \dfrac{}{\{y : \texttt{int}\} \vdash y : \texttt{int}}\,(\text{var})}{\{y : \texttt{int}\} \vdash y + y : \texttt{int}}\,(\text{intAdd})}{\{\} \vdash \texttt{let y = 2 in y + y} : \texttt{int}}\,(\text{let})$$

Here the bottom of the tree is the final conclusion. We now unpack the highest level wrapper expression. The first expression we encounter is the `let` expression. The syntax of which are ($e_3 ::= \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2$). Hence we split into two branches to examine e_1 and e_2. The left branch examines the integer literal. The right branch looks at `(y + y)`. Since `(y + y)` is an addition operation, we must unravel once more into two branches examining both sides of the expression. The left and right branch examines the variable as an integer. Tunnelling from the leaf axioms to the root conclusion justifies the typing derivation as valid.

■

Example 1.2: Semantic Derivations

Continuing the same example, now with the semantics derivation: `let y = 2 in y + y` $\Downarrow 4$

$$\dfrac{\dfrac{}{2 \Downarrow 2}\,(\text{intLit}) \qquad \dfrac{\dfrac{}{y \Downarrow 2}\,(\text{var}) \qquad \dfrac{}{y \Downarrow 2}\,(\text{var})}{y + y \Downarrow 4}\,(\text{intAdd})}{\texttt{let y = 2 in y + y} \Downarrow 4}\,(\text{let})$$

■

— 1 —

Algebraic Data Types

Bibliography

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