Advanced Notes on ML

v1.7

Prof. Alvise Spanò

February 23, 2024

1 Introduction

This document includes advanced notes for the course of *Functional Languages*, MD in Computer Science, University of Padova. Here you will find the formalization of an ML-like functional language, with details of the syntax, the type system and the semantics.

Table 1 shows the syntax of terms. Our calculus resembles the core ML language [1] with conditional, tuples and a sample binary operator added on top of it. Let-rec is syntactically restricted to lambda abstractions for enabling the definition of strict semantics in Section 3.

Table 1: Syntax of terms.

e	$::=$			**expressions**
			L	literal
		$\mid$	x	variable identifier
		$\mid$	$\lambda x.e$	lambda abstraction
		$\mid$	$e\ e$	application
		$\mid$	$\texttt{let}\ x = e\ \texttt{in}\ e$	let binding
		$\mid$	$\texttt{let rec}\ f = \lambda x.e\ \texttt{in}\ e$	recursive let binding
		$\mid$	$\texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$	conditional
		$\mid$	$(e,\ ..\ ,e)$	tuple
		$\mid$	$e + e$	plus binop
L	$::=$			**literals**
			n	integers
		$\mid$	m	floats
		$\mid$	`true` $\mid$ `false`	booleans
		$\mid$	`"abc"`	strings
		$\mid$	`'a'`	chars
		$\mid$	`()`	unit

where x and f are identifiers, $n \in \mathbb{Z}$, $m \in \mathbb{R}$

2 Type System

Type systems are used for verifying the correctness of programs. Type checking and other advanced forms of typing such as type inference happen at compile-time in strongly-typed programming languages. Table 2 shows the syntax of types, type schemes, typing environments and substitutions. Mind that c represent type names, a.k.a. type constructors, such as `int`, `float` etc [5]. Parametric types are not supported. Type variables have form of greek letters α, β, γ etc.

Table 2: Syntax of types and related..

τ	$::=$			**types**
			c	type constructor
		$\mid$	$\tau \rightarrow \tau$	arrow type
		$\mid$	$\alpha, \beta, \gamma, ..$	type variables
		$\mid$	$\tau * \; .. \; * \tau$	tuple type
σ	$::=$		$\forall \overline{\alpha}.\tau$	**type schemes**
Γ	$::=$			**typing environment**
			$\varnothing$	
		$\mid$	$\Gamma, (x : \sigma)$	
θ	$::=$			**substitutions**
			$\varnothing$	
		$\mid$	$\theta, [\alpha \mapsto \tau]$	
			where c are identifiers	

2.1 Preliminaries

Before delving into the details of the typing rules, a number of utility functions must be defined. First of all, the environment Γ can be used as a lookup function $\Gamma(x)$ returning the type scheme σ bound to the given variable x. This implies that $\mathrm{dom}(\Gamma)$ is the set of all variable identifiers bound in Γ. The lookup function is defined recursively by case and ensures that shadowing takes place consistently:

$$\begin{array}{lcl} \Gamma & : & x \rightarrow \tau \\ \\ \Gamma, (x : \tau)(x) & = & \tau \\ \Gamma, (y : \tau)(x) & = & \Gamma(x) \\ \varnothing(x) & = & \varnothing \end{array}$$

The following function `ftv`[1] calculates the free type variables occurring in a type τ, a type scheme σ or an environment Γ.

$$\begin{array}{lcl} \mathtt{ftv} & : & (\tau \cup \sigma \cup \Gamma) \rightarrow \mathscr{P}(\alpha) \\ \\ \mathtt{ftv}(c) & = & \varnothing \\ \mathtt{ftv}(\alpha) & = & \alpha \\ \mathtt{ftv}(\tau_1 \rightarrow \tau_2) & = & \mathtt{ftv}(\tau_1) \cup \mathtt{ftv}(\tau_2) \\ \mathtt{ftv}(\tau_1 * .. * \tau_n) & = & \bigcup_{i=1}^{n} \mathtt{ftv}(\tau_i) \\ \\ \mathtt{ftv}(\forall \overline{\alpha}.\tau) & = & \mathtt{ftv}(\tau) \setminus \{ \; \overline{\alpha} \; \} \\ \\ \mathtt{ftv}(\varnothing) & = & \varnothing \\ \mathtt{ftv}(\Gamma, (x : \sigma)) & = & \mathtt{ftv}(\sigma) \cup \mathtt{ftv}(\Gamma) \end{array}$$

Generalization promotes a type τ to a type scheme σ by quantifying type variables that represent polymorphic types through the universal quantifier *forall* ($\forall$):

$$\begin{array}{lcll} \mathtt{gen} & : & \Gamma \times \tau \rightarrow \sigma & \\ \mathtt{gen}^{\Gamma}(\tau) & = & \forall \overline{\alpha}.\tau & \quad \mathtt{where}\ \overline{\alpha} = \mathtt{ftv}(\tau) \setminus \mathtt{ftv}(\Gamma) \end{array}$$

[1]Mind that when the domain of a function includes a set-union, it means the function is defined on multiple domain sets. Also, in the codomain, a powerset appears: if A is a set, $\mathscr{P}(A)$ is the powerset of A, i.e. the set of sets of A.

Only type variables not occurring free in the environment can be quantified, hence the extra parameter Γ. Generalization takes place at let-binding time, as revealed by rules (Let) and (Let-Rec) in Tables 3 and 4, hence the name *let-polymorphism*.

Instantiation is the reverse operation of generalization, converting a type scheme into a type by refreshing its polymorphic type variables, i.e. those quantified by the forall. Instantiation takes place in rule (Var) at lookup time, i.e. when a variable identifier is encountered. As a matter of fact, function `inst` basically relies on another function, namely `re`, that refreshes type variables occurring in a type:

$$\begin{array}{lcll}
\texttt{inst} & : & \sigma \to \tau & \\
\texttt{inst}(\forall\overline{\alpha}.\tau) & = & \texttt{re}^{\overline{\alpha}}(\tau) & \\
 & & & \\
\texttt{re} & : & \mathscr{P}(\alpha) \times \tau \to \tau & \\
\texttt{re}^{\overline{\alpha}}(c) & = & c & \\
\texttt{re}^{\overline{\alpha}}(\alpha) & = & \alpha & \texttt{if } \alpha \notin \overline{\alpha} \\
\texttt{re}^{\overline{\alpha}}(\alpha) & = & \beta & \texttt{if } \alpha \in \overline{\alpha} \texttt{ and with } \beta \texttt{ fresh} \\
\texttt{re}^{\overline{\alpha}}(\tau_1 \to \tau_2) & = & \texttt{re}^{\overline{\alpha}}(\tau_1) \to \texttt{re}^{\overline{\alpha}}(\tau_2) & \\
\texttt{re}^{\overline{\alpha}}(\tau_1 * .. * \tau_n) & = & \texttt{re}^{\overline{\alpha}}(\tau_1) * \;..\; * \texttt{re}^{\overline{\alpha}}(\tau_n) & \texttt{with } n \geq 2
\end{array}$$

2.1.1 More on Type Variables

To refresh type variables means to replace, for instance, a type variable whose name is α with a *new* type variable β, where the name β has never been used before. For example, let $\tau = \alpha \to \beta \to \beta \to \texttt{int}*\alpha*\gamma$, then refreshing its type variables means to replace type variables occurring in τ with new ones having unused names, thus yielding to $\delta \to \epsilon \to \epsilon \to \texttt{int} * \delta * \zeta$. All occurrences of α has been replaced with δ; occurrences of β with ϵ; and occurrences of γ with ζ. This is equivalent to applying the substitution $[\alpha \mapsto \delta; \beta \mapsto \epsilon; \gamma \mapsto \zeta]$ to the type τ. More on substitutions in Section 2.3.1.

An implementation must produce new fresh names when refreshing, granting they haven't been used before in the typing context. Type variables must therefore be *unique identifiers*. A type $\alpha \to \texttt{int} * \alpha \to \beta$ would then be encoded as $1 \to \texttt{int} * 1 \to 2$, where α is actually encoded by the number 1 and β by the number 2. Refreshing such type is extremely simple: for each type variable a new fresh number must be produced, leading to $2 \to \texttt{int} * 2 \to 3$.

Mind that this is *not* a plain increment: this is a full replacement with new numbers not occurring before. For example, consider the following scenario with multiple types in the typing context including a variety of type variables:

`original type`	`implementation`	`refreshed`
$\alpha \to \alpha$	$1 \to 1$	$7 \to 7$
$\beta * \beta \to \gamma$	$2 * 2 \to 3$	$8 * 8 \to 9$
$\delta \to \zeta \to \eta \to \eta$	$4 \to 6 \to 5 \to 5$	$10 \to 12 \to 11 \to 11$

A common encoding is through integer numbers produced by a global counter that is incremented each time a new fresh type variable is required. Obviously, such a counter would always provide numbers never used before.

2.2 Type rules

Type rules for expressions are shown in Table 3. Type judgements are logical formulas of form $\Gamma \vdash e : \tau$. The environment Γ stores bindings from variable identifiers to types τ rather than type schemes σ as originally defined in Table 1. This is enough for type rules and is acceptable from the point of view of the syntax, a type τ can be treated as a degenerate form of scheme $\forall\varnothing.\tau$. In rule (Let-Rec) the form of the recursive expression e_1 being bound is restricted to lambdas. This enforces only the let-rec to operate only on functions, disallowing recursive non-functional values. Rules for literals are trivial and only a sample rule (Lit-Int) for integers is shown. Rule (Plus) shows how to deal with binary operators.

Mind that the goal of type rules is not to tell how types are calculated or deduced. A type rule just predicates how expressions can get a type given a number of hypotheses, and can be read as: *if the type judgments in the hypotheses are true, then the type judgment in the thesis is true.* For example, in the (ABS) rule the domain τ_1 seems coming out of nowhere: how can we know the type of the lambda parameter x if it is unannotated? The point is that type rules do not answer this question. That type rule only says: *if the lambda body e has type τ_2 in an environment where the parameter x has type τ_1, then the whole lambda has type $\tau_1 \to \tau_2$*. That's all. It is just a logical implication.

Table 3: Type rules for expressions.

$$\textsc{Lit-Int}\ \frac{\diamond}{\Gamma \vdash n : \texttt{int}} \qquad \textsc{Var}\ \frac{x \in \mathrm{dom}(\Gamma) \qquad \Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \textsc{Abs}\ \frac{\Gamma, (x : \tau_1) \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

$$\textsc{App}\ \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1} \qquad \textsc{If}\ \frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}$$

$$\textsc{Tup}\ \frac{\Gamma \vdash e_i : \tau_i \qquad (\forall i \in [1, n])}{\Gamma \vdash (e_1,\ ..\ ,\ e_n) : \tau_1 *\ ..\ * \tau_n} \qquad \textsc{Let}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, (x : \tau_1) \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

$$\textsc{Plus}\ \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}} \qquad \textsc{Let-Rec}\ \frac{\Gamma, (f : \tau_1 \to \tau_2) \vdash \lambda x.e_1 : \tau_1 \to \tau_2 \qquad \Gamma, (f : \tau_1 \to \tau_2) \vdash e_2 : \tau_3}{\Gamma \vdash \texttt{let rec}\ f = \lambda x.e_1\ \texttt{in}\ e_2 : \tau_3}$$

2.3 Type Inference

Type inference, a.k.a. type reconstruction, is an advanced typing mechanism through which types are deduced from the code rather than being annotated by the programmer in the program text. Type inference rules are shown in Table 4. The original ML type inference algorithm [1][3] is here formulated in terms of syntax-directed rules [8]. Type judgements are logical formulas of form $\Gamma \vdash e : \tau \rhd \theta$, i.e. yielding two outputs, a type and a substitution, for which $\theta(\tau) \equiv \tau$ holds. This means that the output type τ is granted to be less general as possible, hence the output substitution θ does not need to be applied to it.

2.3.1 Substitutions

Table 2 defines substitutions θ syntactically as a map from type variables α to types τ. For the sake of brevity, we often use the compact notation $[\alpha_1 \mapsto \tau_1; ..; \alpha_n \mapsto \tau_n]$ with $n \geq 1$ in place of $\varnothing, [\alpha_1 \mapsto \tau_1], .., [\alpha_n \mapsto \tau_n]$.

Substitutions can also be seen as functions from types to types [6]: a substitution application $\theta(\tau)$ consists in producing a new type τ' where all occurrences of each type variable $\alpha_i \in \mathrm{dom}(\theta)$ are replaced with the mapped type $\tau_i \in \mathrm{codom}(\theta)$, such that $\alpha_i \notin \tau'$. Substitutions can also be applied to environments and type schemes, leading to the following overall definition of application:

$$\begin{array}{lcll}
\theta & : & (\tau \to \tau) \cup (\sigma \to \sigma) \cup (\Gamma \to \Gamma) & \\
 & & & \\
\theta(c) & = & c & \\
\theta(\alpha) & = & \tau & \texttt{if}\ [\alpha \mapsto \tau] \in \theta \\
\theta(\alpha) & = & \alpha & \texttt{if}\ \alpha \notin \mathrm{dom}(\theta) \\
\theta(\tau_1 \to \tau_2) & = & \theta(\tau_1) \to \theta(\tau_2) & \\
\theta(\tau_1 * .. * \tau_n) & = & \theta(\tau_1) * .. * \theta(\tau_n) & \texttt{with}\ n \geq 2 \\
 & & & \\
\theta(\forall \overline{\alpha}.\tau) & = & \forall \overline{\alpha}.\theta'(\tau) & \texttt{with}\ \theta' = \theta \setminus \{\alpha_i \mapsto \tau_i \mid \alpha_i \in \overline{\alpha}\} \\
 & & & \\
\theta(\varnothing) & = & \varnothing & \\
\theta(\Gamma, (x : \sigma)) & = & \theta(\Gamma), (x : \theta(\sigma)) &
\end{array}$$

Table 4: Type inference algorithm in form of syntax-directed rules.

$$\textsc{I-Lit-Int}\quad \frac{\diamond}{\Gamma \vdash n : \mathtt{int} \rhd \varnothing}$$

$$\textsc{I-Var}\quad \frac{x \in \mathrm{dom}(\Gamma) \qquad \Gamma(x) = \sigma \qquad \tau = \mathtt{inst}(\sigma)}{\Gamma \vdash x : \tau \rhd \varnothing}$$

$$\textsc{I-App}\quad \frac{\begin{array}{l} \Gamma \vdash e_1 : \tau_1 \rhd \theta_1 \qquad \theta_1(\Gamma) \vdash e_2 : \tau_2 \rhd \theta_2 \\ \mathbf{U}(\tau_1; \tau_2 \to \alpha) = \theta_3 \qquad (\alpha\ \mathtt{fresh}) \\ \tau = \theta_3(\alpha) \qquad \theta_4 = \theta_3 \circ \theta_2 \end{array}}{\Gamma \vdash e_1\ e_2 : \tau \rhd \theta_4}$$

$$\textsc{I-Abs}\quad \frac{\Gamma, (x : \forall\varnothing.\alpha) \vdash e : \tau_2 \rhd \theta_1 \qquad \tau_1 = \theta_1(\alpha)}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2 \rhd \theta_1}$$

$$\textsc{I-If}\quad \frac{\begin{array}{l} \Gamma \vdash e_1 : \tau_1 \rhd \theta_1 \qquad \mathbf{U}(\tau_1;\ \mathtt{bool}) = \theta_2 \\ \theta_3 = \theta_2 \circ \theta_1 \qquad \theta_3(\Gamma) \vdash e_2 : \tau_2 \rhd \theta_4 \\ \theta_5 = \theta_4 \circ \theta_3 \qquad \theta_5(\Gamma) \vdash e_3 : \tau_3 \rhd \theta_6 \\ \theta_7 = \theta_6 \circ \theta_5 \qquad \mathbf{U}(\theta_7(\tau_2);\ \theta_7(\tau_3)) = \theta_8 \\ \tau = \theta_8(\tau_2) \qquad \theta_9 = \theta_8 \circ \theta_7 \end{array}}{\Gamma \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \tau \rhd \theta_9}$$

$$\textsc{I-Tup}\quad \frac{\begin{array}{l} \theta_0 = \varnothing \\ \theta_{i-1}(\Gamma) \vdash e_i : \tau_i \rhd \theta_i \qquad (\forall i \in [1, n]) \end{array}}{\Gamma \vdash (e_1,\ ..\ ,\ e_n) : \tau_1 *\ ..\ * \tau_n \rhd \theta_n}$$

$$\textsc{I-Plus}\quad \frac{\begin{array}{l} \Gamma \vdash e_1 : \tau_1 \rhd \theta_1 \qquad \mathbf{U}(\tau_1;\ \mathtt{int}) = \theta_2 \qquad \theta_3 = \theta_2 \circ \theta_1 \\ \theta_3(\Gamma) \vdash e_1 : \tau_2 \rhd \theta_4 \qquad \mathbf{U}(\tau_2;\ \mathtt{int}) = \theta_5 \qquad \theta_6 = \theta_5 \circ \theta_4 \end{array}}{\Gamma \vdash e_1 + e_2 : \mathtt{int} \rhd \theta_6}$$

$$\textsc{I-Let}\quad \frac{\begin{array}{l} \Gamma \vdash e_1 : \tau_1 \rhd \theta_1 \qquad \sigma_1 = \mathtt{gen}^{\theta_1(\Gamma)}(\tau_1) \\ \theta_1(\Gamma), (x : \sigma_1) \vdash e_2 : \tau_2 \rhd \theta_2 \qquad \theta_3 = \theta_2 \circ \theta_1 \end{array}}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau_2 \rhd \theta_3}$$

$$\textsc{I-Let-Rec}\quad \frac{\begin{array}{l} \Gamma, (f : \forall\varnothing.\alpha) \vdash \lambda x.e_1 : \tau_1 \rhd \theta_1 \qquad \Gamma_1 = \theta_1(\Gamma) \qquad \sigma_1 = \mathtt{gen}^{\Gamma_1}(\tau_1) \qquad (\alpha\ \mathtt{fresh}) \\ \Gamma_1, (f : \sigma_1) \vdash e_2 : \tau_2 \rhd \theta_2 \qquad \mathbf{U}(\alpha; \theta_1(\tau_1)) = \theta_3 \qquad \theta_4 = \theta_3 \circ \theta_2 \circ \theta_1 \end{array}}{\Gamma \vdash \mathtt{let\ rec}\ f = \lambda x.e_1\ \mathtt{in}\ e_2 : \tau_2 \rhd \theta_4}$$

When applying a substitution θ to a type scheme $\sigma \equiv \forall\overline{\alpha}.\tau$, only the type component τ must be affected. This is necessary because type variables quantified by the $\forall$ must not be touched, as they are meant to represent polymorphic types. Substitution θ must thefore be restricted to a smaller substitution θ' whose domain does not include the quantified type variables $\overline{\alpha}$.

Substitution composition [7] is defined like function composition: let θ_1 and θ_2 be two substitutions, then the composition $\theta_2 \circ \theta_1$ yields a new substitution θ' such that $\theta'(\tau) = \theta_2(\theta_1(\tau))$. Composition can also be defined constructively: let $\theta_1 = [\alpha_1 \mapsto \tau_1\ ..\ \alpha_n \mapsto \tau_n]$ and $\theta_2 = [\beta_1 \mapsto \tau'_1\ ..\ \beta_m \mapsto \tau'_m]$ for $n \geq 1$ and $m \geq 1$, then $\theta_2 \circ \theta_1 = [\beta_1 \mapsto \theta_1(\tau'_1)\ ..\ \beta_m \mapsto \theta_1(\tau'_m); \alpha_1 \mapsto \tau_1\ ..\ \alpha_n \mapsto \tau_n]$ where if $\alpha_i = \beta_j$ for some $i \in [1, n]$ and $j \in [1, m]$ then $\tau_i \equiv \tau'_j$. The last constraint means that the domains of the two substitutions must be disjoint, unless if the same type variable appears in both domains then they must map into the same type, otherwise composition leads to an error state[2].

Finally, circularity is not allowed: given a substitution $\theta = [\alpha_1 \mapsto \tau_1\ ..\ \alpha_n \mapsto \tau_n]$ for $n \geq 1$, then $\alpha_i \notin \tau_i$ forall $i \in [1, n]$.

[2]Mind that an error state is something related to computer implementations, not to mathematics or logic. It means that if a certain state occurs in an implementation, then an error must be raised. A compiler, for example, would fail in that case.

2.3.2 Unification

Unification is crucial for type inference and appears every time a type of some form is required, e.g. in rules (I-App) and (I-If) in Table 4. Given two types τ_1 and τ_2, unification calculates a substitution that makes the two types equal. More formally, $\mathbf{U}(\tau_1; \tau_2)$ calculates a substitution θ such that $\theta(\tau_1) \equiv \theta(\tau_2)$. Such substituion θ is called the *most greater unifier* (MGU) [9]. The Martelli-Montanari unification algorithm [4] efficiently calculated the MGU:

$$\begin{array}{lcll}
\mathbf{U} & : & \tau \times \tau \to \theta & \\
\mathbf{U}(c_1;\ c_2) & = & \varnothing & \texttt{if}\ c_1 \equiv c_2 \\
\mathbf{U}(\alpha;\ \tau) = \mathbf{U}(\tau;\ \alpha) & = & [\alpha \mapsto \tau] & \texttt{iff}\ \alpha \notin \tau \\
\mathbf{U}(\tau_1 \to \tau_2;\ \tau_3 \to \tau_4) & = & \mathbf{U}(\tau_1; \tau_3) \circ \mathbf{U}(\tau_2; \tau_4) & \\
\mathbf{U}(\tau_1 * .. * \tau_n;\ \tau'_1 * .. * \tau'_n) & = & \mathbf{U}(\tau_1;\ \tau'_1) \circ\ ..\ \circ \mathbf{U}(\tau_n;\ \tau'_n) & \texttt{with}\ n \geq 2
\end{array}$$

Not all combinations of cases are defined: undefined cases would lead to an error state in an implementation. Notably, unification between a type variable α and a type τ is symmetric and cannot occur α appears in τ. This is called *circularity check*.

3 Operational Semantics

Semantics represent the behaviour of programs. Program evaluation happens at run-time, either by running the machine code produced by a compiler or by evaluating the code in case the language is interpreted. Semantics can be expressed in a number of ways: the kind of semantics defined in this document is called *operational* semantics [2].

Table 5 shows the syntax of values and evaluation environments, while Table 6 shows the evaluation rules for expressions, which are formulas $\Delta \vdash e \rightsquigarrow v$. Notably, environments for values are called Δ and contain bindings between identifiers x and values v. They are essentially equivalent to type environments Γ, except containing values v rather than types τ. Lookup in Δ is equivalent to that of Γ defined in Section 2.1.

Table 5: Syntax of values and related definitions for literals L and expressions e come from Table 1.

v	$::=$		**values**
	$\mid$	L	literal
	$\mid$	$\langle \lambda x.e; \Delta \rangle$	closure
	$\mid$	$\langle \lambda x.e; f; \Delta \rangle$	rec-closure
	$\mid$	$(v, .., v)$	tuple of values
Δ	$::=$		**evaluation environment**
		$\varnothing$	
	$\mid$	$\Delta, (x \rightsquigarrow v)$	
		x and f are identifiers	

Closures and rec-closures are special values that represent a frozen lambda in its original scope. The information stored in a closure $\langle \lambda x.e; \Delta \rangle$ is the lambda $\lambda x.e$ itself, consisting of the parameter x and the body expression e, plus an environment Δ. This grants *lexical scoping* when β-reduction takes place in the rule for application. The term β-reduction represents the evaluation of the left side of an application by substituting the occurrences of the lambda parameter with the argument standing at the right side of the application. In rule (E-App) this is achieved by extending the Δ_0 environment stored in the closure with a binding between the lambda parameter x and the argument value v_2.

Rec-closures are like ordinary closures with one additional information: the name of the recursive function f. This allows rule (E-App-Rec) to bind the rec-closure $\langle \lambda x.e_0; f; \Delta_0 \rangle$ to the identifier f, evaluating the lambda body e_0 in a context where the f itself exists, thus it is recursively callable.

Notably, rule (E-App) holds when the left side of the application e_1 evaluates to a closures, whereas rule (E-App-Rec) holds when e_1 evaluates to a rec-closure. Analogously, rule (E-If-True) holds when e_1 evaluates to `false` in the hypothesis, whereas rule (E-If-False) holds when e_1 evaluates to `false`. Rule (E-Plus) shows how to deal with binary arithmetic operators for intergers: evaluation of operands produces an integer literal of form n, with $n \in \mathbb{Z}$, as of Table 1. Special operator $\oplus$ stands for the actual addition between two integer numbers. An implementation would invoke the plus operator in the host language in this case, truly producing the sum between the two values. Other rules are rather straightforward.

Table 6: Operational semantics as evaluation rules.

$$\textsc{E-Lit-Int}\quad \frac{\diamond}{\Delta \vdash n \rightsquigarrow n} \qquad \textsc{E-Var}\quad \frac{x \in \text{dom}(\Delta) \qquad \Delta(x) = v}{\Delta \vdash x \rightsquigarrow v} \qquad \textsc{E-Abs}\quad \frac{\diamond}{\Delta \vdash \lambda x.e \rightsquigarrow \langle \lambda x.e; \Delta \rangle}$$

$$\textsc{E-App}\quad \frac{\begin{array}{l}\Delta \vdash e_1 \rightsquigarrow \langle \lambda x.e_0; \Delta_0 \rangle \qquad \Delta \vdash e_2 \rightsquigarrow v_2 \\ \Delta_0, (x \rightsquigarrow v_2) \vdash e_0 \rightsquigarrow v\end{array}}{\Delta \vdash e_1\ e_2 \rightsquigarrow v} \qquad \textsc{E-App-Rec}\quad \frac{\begin{array}{l}\Delta \vdash e_1 \rightsquigarrow \langle \lambda x.e_0; f; \Delta_0 \rangle \qquad \Delta \vdash e_2 \rightsquigarrow v_2 \\ \Delta_0, (f \rightsquigarrow \langle \lambda x.e_0; f; \Delta_0 \rangle), (x \rightsquigarrow v_2) \vdash e_0 \rightsquigarrow v\end{array}}{\Delta \vdash e_1\ e_2 \rightsquigarrow v}$$

$$\textsc{E-If-True}\quad \frac{\Delta \vdash e_1 \rightsquigarrow \texttt{true} \qquad \Delta \vdash e_2 \rightsquigarrow v_2}{\Delta \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rightsquigarrow v_2} \qquad \textsc{E-If-False}\quad \frac{\Delta \vdash e_1 \rightsquigarrow \texttt{false} \qquad \Delta \vdash e_3 \rightsquigarrow v_3}{\Delta \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rightsquigarrow v_3}$$

$$\textsc{E-Plus}\quad \frac{\Delta \vdash e_1 \rightsquigarrow n_1 \qquad \Delta \vdash e_2 \rightsquigarrow n_2 \qquad n = n_1 \oplus n_2}{\Delta \vdash e_1 + e_2 \rightsquigarrow n} \qquad \textsc{E-Tup}\quad \frac{\Delta \vdash e_i \rightsquigarrow v_i \qquad (\forall i \in [1, n])}{\Delta \vdash (e_1, .., e_n) \rightsquigarrow (v_1, .., v_n)}$$

$$\textsc{E-Let}\quad \frac{\Delta \vdash e_1 \rightsquigarrow v_1 \qquad \Delta, (x \rightsquigarrow v_1) \vdash e_2 \rightsquigarrow v_2}{\Delta \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \rightsquigarrow v_2} \qquad \textsc{E-Let-Rec}\quad \frac{\Delta, (f \rightsquigarrow \langle \lambda x.e_1; f; \Delta \rangle) \vdash e_2 \rightsquigarrow v_2}{\Delta \vdash \texttt{let rec}\ f = \lambda x.e_1\ \texttt{in}\ e_2 \rightsquigarrow v_2}$$

References

[1] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, 1982.

[2] Matthew Hennessy. *The semantics of programming languages: an elementary introduction using structural operational semantics*. John Wiley & Sons, Inc., 1990.

[3] Daniel Leivant. Polymorphic type inference. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 88–98, 1983.

[4] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(2):258–282, 1982.

[5] Robin Milner. A proposal for standard ml. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, pages 184–197, 1984.

[6] Wikipedia. Substitution (logic). https://en.wikipedia.org/wiki/Substitution_(logic). [Online; accessed 19-January-2023].

[7] Wikipedia. Substitution (logic). https://en.wikipedia.org/wiki/Substitution_(logic)#:~:text=Composition%20is%20an%20associative%20operation,neutral%20element%20of%20substitution%20composition. [Online; accessed 19-January-2023].

[8] Wikipedia. Syntax-directed translation. https://en.wikipedia.org/wiki/Syntax-directed_translation. [Online; accessed 19-January-2023].

[9] Wikipedia. Unification (logic). https://en.wikipedia.org/wiki/Unification_(computer_science). [Online; accessed 19-January-2023].