LCI Project Report

Giulio Paparelli

Formal Definition

Syntax

We start by defining the formal syntax of our extension of untyped Lambda Calculus, which we call *Lambda+*.
The syntax is inspired by the one on Leroy's slides and *HOFL* (Higher Order Functional Language) seen in the course Principles for Software Composition:

$$\begin{aligned} t ::= \ & x \mid n \mid b \\ & \mid \text{aop}(t_0, t_1) \mid \text{bop}(t_0, t_1) \mid \text{not}(t) \\ & \mid \text{if } t \text{ then } t_0 \text{ else } t_1 \\ & \mid \lambda t \mid t_1 \ t_0 \end{aligned}$$

where:

- $x \in Var$
- n are integers
- b are booleans
- $\text{aop} \in \{+, -, \times\}$
- $\text{bop} \in \{\wedge, \vee, =, \neq, >, <\}$

Observations:

1. De Bruijn indices are used to represent variable bindings. Therefore the set Var is a set of De Bruijn indices.
2. We will define an "environment" as a sequence of values to store the bindings index-value.

Type System

The above syntax allows *pre-terms*, terms that do not have a valid semantic.
The (pre-)term

$$t = \text{if } 3 \text{ then } 1 \text{ else } 2$$

has not a valid semantic: the guard of the `if` has to be a boolean.

There is the need to introduce a **type system** to distinguish pre-terms from terms.
Let's define the types of our values as follows

$$\tau = \mathit{int} \mid \mathit{bool} \mid \tau_0 \to \tau_1$$

and then we denote with $\mathcal{T}$ the set of all possible types that can be made, e.g.,
$\mathcal{T} = \{\text{integer}, \text{ boolean}, \text{integer} \rightarrow \text{integer}, \text{integer} \rightarrow \text{boolean}, \ldots\}$

Then we assume that the variables are typed and that we have a *typing context* Γ:

- $Var = \{Var_\tau\}_{\tau \in \mathcal{T}}$
- Γ is a list or set of variable-type pairs. Each pair associates a variable with its type

The type system, a set of inference rules using structural induction of the Lambda+ syntax, assigns to each pre-term a type, if possible.
We use the type system to make type judgments, such as

$$t : \tau \iff \text{t has type } \tau$$

A pre-term t is **well-formed** if $\exists \tau \in \mathcal{T} . t : \tau$.
In other words: if we can assign a type to a pre-term t then t is well-formed.

The type system is made of the following inference rules:

$$\frac{}{\Gamma \vdash x : \tau_x} \quad \frac{}{\Gamma \vdash n : int} \quad \frac{}{\Gamma \vdash b : bool}$$

$$\frac{\Gamma \vdash t_0 : int \quad \Gamma \vdash t_1 : int}{\Gamma \vdash (t_0 \text{ aop } t_1) : int} \quad \frac{\Gamma \vdash t_0 : bool \quad \Gamma \vdash t_1 : bool}{\Gamma \vdash (t_0 \text{ bop } t_1) : bool} \quad \frac{\Gamma \vdash t : bool}{\Gamma \vdash \text{not}(t) : bool}$$

$$\frac{\Gamma \vdash t : bool \quad \Gamma \vdash t_0 : \tau \quad \Gamma \vdash t_1 : \tau}{\Gamma \vdash \text{if } t \text{ then } t_0 \text{ else } t_1 : \tau} \quad \frac{\Gamma, x : \tau_0 \vdash t : \tau_1}{\Gamma \vdash \lambda.t : \tau_0 \rightarrow \tau_1} \quad \frac{\Gamma \vdash t_1 : \tau_0 \rightarrow \tau_1 \quad \Gamma \vdash t_0 : \tau_0}{\Gamma \vdash t_1 \ t_0 : \tau_1}$$

The above type system will be "*embedded*" in the code of the interpreter.

Semantics

Small-Step Semantics

Now we define the small-step semantics of **well-formed** Lambda+ terms.
To simplify the notation we represent with $\mathbb{T}$ the set of all the possible well-formed terms.
We consider a structural operational semantics, i.e. specifying the execution recursively on syntax, in a machine-independent way.

To help us determine the semantics of a term we define a **store** function $\sigma : Var \rightarrow \mathbb{N} \cup \mathbb{B}$, and with $\mathbb{M}$ we represent the set of all possible stores σ.
We also define a function bind, that adds (on top) to the current environment (store) a new value.

A program written in Lambda+ that terminates returns a value
$v \in V = \{\text{int}, \text{bool}, \text{closure}(t, \rho)\}$.
We will define a **bounded interpreter** with a fuel parameter that is decreased at each step, hence every program technically "terminates".

Nonetheless, we have to consider the case where the fuel reaches 0 before the program has produced a valid value.
We then define the set of the possible values returned by a Lambda+ program as

$$\bar{V} = V \cup \bot$$

where $\bot$ is called **bottom**, the value of non-terminating programs (as diverging programs that never terminate).
Abusing the notation, we also use $\bot$ to represent undefined variables and type errors.

We represent intermediate steps of the computation as pairs $\langle t, \sigma \rangle \in \mathbb{T} \times \mathbb{M}$.
Then we define a transition system $(S, \rightarrow)$ where the first is a set of states and the second is a transition system.
The transitions have the following form:

$$\langle t, \sigma \rangle \rightarrow \langle t', \sigma' \rangle$$

Notation: We distinguish between the syntactic operations and their actual semantic operations using an over-line:

- aop is syntax
- $\overline{\mathrm{aop}}$ is semantic

The small-step semantics of Lambda+ is defined by the following inference rules:

$$\frac{\langle t_0, \sigma \rangle \rightarrow \langle t_0', \sigma \rangle}{\langle t_0 \text{ aop } t_1, \sigma \rangle \rightarrow \langle t_0' \text{ aop } t_1, \sigma \rangle} \qquad \frac{\langle t_1, \sigma \rangle \rightarrow \langle t_1', \sigma \rangle}{\langle n_0 \text{ aop } t_1, \sigma \rangle \rightarrow \langle n_0 \text{ aop } t_1', \sigma \rangle} \qquad \frac{}{\langle n_0 \text{ aop } n_1, \sigma \rangle \rightarrow \langle n_0 \ \overline{\text{aop}}\ n_1, \sigma \rangle}$$

$$\frac{\langle t, \sigma \rangle \rightarrow \langle t', \sigma \rangle}{\langle \text{if } t \text{ then } t_0 \text{ else } t_1, \sigma \rangle \rightarrow \langle \text{if } t' \text{ then } t_0 \text{ else } t_1, \sigma \rangle}$$

$$\frac{}{\langle \text{if } true \text{ then } t_0 \text{ else } t_0, \sigma \rangle \rightarrow \langle t_0, \sigma \rangle} \qquad \frac{}{\langle \text{if } false \text{ then } t_0 \text{ else } t_1, \sigma \rangle \rightarrow \langle t_1, \sigma \rangle}$$

$$\frac{}{\langle \lambda t, \sigma \rangle \rightarrow \langle closure(t, \sigma), \sigma \rangle}$$

$$\frac{\langle t_1, \sigma \rangle \rightarrow \langle t_1', \sigma \rangle}{\langle t_1\ t_0, \sigma \rangle \rightarrow \langle t_1'\ t_0, \sigma \rangle} \qquad \frac{\langle t_0, \sigma \rangle \rightarrow \langle t_0', \sigma \rangle}{\langle closure(t, \sigma')\ t_0, \sigma \rangle \rightarrow \langle closure(t, \sigma')\ t_0', \sigma \rangle}$$

$$\frac{}{\langle closure(t, \sigma')\ v, \sigma \rangle \rightarrow \langle t, \text{bind}(v, \sigma') \rangle}$$

Note: For brevity not all the rules have been listed:

- The rules above are under the assumption that the program terminates and that every variable is always defined in σ. In reality, there will be rules to cover the scenario in which a part of the program diverges or a lookup error, giving as result $\bot$
- The rules for boolean expressions are identical to the ones for arithmetic expressions, hence are not shown

Big-Step Semantics

To define the big-step semantics we build upon the small-step semantics.
A big-step computation is denoted with $\longrightarrow$ instead of the $\to$ used for the small-step.

The big-step semantics of Lambda+ is defined by the following inference rules:

$$\frac{\langle t, \sigma\rangle \longrightarrow \langle true, \sigma\rangle}{\langle \text{if } t \text{ then } t_0 \text{ else } t_1, \sigma\rangle \longrightarrow \langle t_0, \sigma\rangle} \quad \frac{\langle t, \sigma\rangle \longrightarrow \langle false, \sigma\rangle}{\langle \text{if } t \text{ then } t_0 \text{ else } t_1, \sigma\rangle \longrightarrow \langle t_1, \sigma\rangle}$$

$$\frac{\langle t_0, \sigma\rangle \longrightarrow \langle n_0, \sigma\rangle \quad \langle t_1, \sigma\rangle \longrightarrow \langle n_1, \sigma\rangle}{\langle t_0 \text{ aop } t_1, \sigma\rangle \longrightarrow \langle n_0 \; \overline{\text{aop}} \; n_1, \sigma\rangle}$$

$$\frac{\langle t_0, \sigma\rangle \longrightarrow \langle closure(t, \sigma'), \sigma\rangle \quad \langle t_1, \sigma\rangle \longrightarrow \langle v, \sigma\rangle \quad \langle t, \text{bind}(v, \sigma') = \sigma''\rangle \to \langle v', \sigma''\rangle}{\langle t_0 \; t_1, \sigma\rangle \longrightarrow \langle v', \sigma\rangle}$$

Note: For brevity not all the rules have been listed:

- Some rules (e.g., the rule for the lambda abstraction) remain the same and not listed
- The rules above are under the assumption that the program terminates, i.e., we have enough fuel. In reality there will be rules to cover the scenario in which a part of the program diverges, giving as a result $\perp$
- The rules for boolean expressions are identical to the ones for arithmetic expressions, hence are not shown

Properties of the Semantics

Determinism

Is it true that two different "computations", on the same term and environment, always produce the same value?

We can express this property in symbols:

$$\forall \sigma \in \mathbb{M}. \forall v, v' \in V. \langle t, \sigma\rangle \longrightarrow v \wedge \langle t, \sigma\rangle \longrightarrow v' \implies v = v'$$

Assuming that the term is not diverging (e.g., it can be computed in a finite fuel) we could provide a formal proof using the Rule Induction Principle, expressing the predicate of formulas

$$P(\langle t, \sigma\rangle \longrightarrow v) = \forall v'. \langle t, \sigma\rangle \longrightarrow v' \implies v = v'$$

and proving that it holds for all the conclusions of the inference rules.

Termination

Is it true that every term terminates? This is immediately not true as we can define a term that always diverges, no matter the amount of fuel we provide.

The lambda expression

$$e := (\lambda x.\, x\; x)(\lambda x.\, x\; x)$$

is the infinitely self-applying function that never terminates, and it can be defined in Lambda+.
The term obtained from e will always consume all the fuel.

Definitional Interpreter

We can represent the provided definitional interpreter as a function, in a way that resembles what we would do for defining denotational semantics.

We introduce the function $[[\cdot]]_I : (\mathbb{Z}, \mathbb{M}, \mathbb{T}) \to (V \cup \{\bot\})$ as the *interpreter function*, where:

- the first argument is the remaining fuel
- the second argument is the current store
- the third argument is the term currently being interpreted

The interpreter function is the mathematical representation of the `eval` function defined in the attached source code, and it is defined as follows:

$$\begin{aligned}
&[[(0,\ _,\ _)]]_I\ =\ \ \bot \\
&[[(n+1,\ \sigma,\ x)]]_I\ =\ \sigma(x) \\
&[[(n+1,\ \sigma,\ m)]]_I\ =\ m \\
&[[(n+1,\ \sigma,\ b)]]_I\ =\ b \\
&[[(n+1,\ \sigma,\ t_0 \operatorname{aop} t_1)]]_I\ =\ \begin{cases} m_0 \,\overline{\operatorname{aop}}\, m_1 \text{ if } [[(n,\ \sigma,\ t_0)]]_I = m_0 \in \mathbb{Z}\ \wedge [[(n,\ \sigma,\ t_1)]]_I = m_1 \in \mathbb{Z} \\ \bot\ \ \text{otherwise} \end{cases} \\
&[[(n+1,\ \sigma,\ t_0 \operatorname{bop} t_1)]]_I\ =\ \begin{cases} b_0 \,\overline{\operatorname{bop}}\, b_1 \text{ if } [[(n,\ \sigma,\ t_0)]]_I = b_0 \in \mathbb{B}\ \wedge [[(n,\ \sigma,\ t_1)]]_I = b_1 \in \mathbb{B} \\ \bot\ \ \text{otherwise} \end{cases} \\
&[[(n+1,\ \sigma,\ \operatorname{not} t)]]_I\ =\ \begin{cases} \neg b \text{ if } [[(n,\ \sigma,\ t)]]_I = b \in \mathbb{B} \\ \bot\ \ \text{otherwise} \end{cases} \\
&[[(n+1,\ \sigma,\ \text{if } t \text{ then } t_0 \text{ else } t_1]]_I\ =\ \begin{cases} [[(n, \sigma, t_0)]]_I \text{ if } [[(n,\ \sigma,\ t)]]_I = \mathit{true} \\ [[(n, \sigma, t_1)]]_I \text{ if } [[(n,\ \sigma,\ t)]]_I = \mathit{false} \\ \bot\ \ \text{otherwise} \end{cases} \\
&[[(n+1,\ \sigma,\ \lambda t)]]_I\ =\ \operatorname{closure}(t, \sigma) \\
&[[(n+1,\ \sigma,\ t_0\, t_1)]]_I\ =\ \begin{cases} [[(n, \sigma', t)]]_I \text{ if } \begin{cases} [[(n,\ \sigma,\ t_0)]] = \mathit{closure}(t, \sigma'') \\ [[(n, \sigma, t_1)]]_I\ =\ v \\ \sigma' = \operatorname{bind}(\sigma'', v) \end{cases} \end{cases}
\end{aligned}$$

Soundness

Now we want to prove that the interpreter is **sound:**

$$\forall n, t, v, \sigma.\, ([[(n, \sigma, t)]]_I = v \implies \langle t, \sigma \rangle \longrightarrow v)$$

Note: forcing the notation, we now leave out the σ produced when the computation gives a value v.

Let's try to prove that statement by induction on t.

The base cases, where t is either a variable, an integer, or a boolean, are immediate to see.

Let's now consider some complex cases.

If Than Else:

$$\text{if } t \text{ then } t_0 \text{ else } t_1$$

Note: we consider only the case where t evaluates to $true$, as the other case is analogous.

We want to prove that

$$\forall n, t, t_0, t_1 v, \sigma. ([[(n, \sigma, \text{if } t \text{ then } t_0 \text{ else } t_1)]] = v \implies \langle \text{if } t \text{ then } t_0 \text{ else } t_1, \sigma \rangle \longrightarrow v)$$

We can use the following inductive hypothesis:

1. $\forall n, t, v, \sigma. ([[(n, \sigma, t)]] = true \implies \langle t, \sigma \rangle \longrightarrow true)$
2. $\forall n, t_0, v, \sigma. ([[(n, \sigma, t_0)]] = v \implies \langle t_0, \sigma \rangle \longrightarrow v)$

Thanks to the first inductive hypothesis we only have to prove that

$$\forall n, t_0, v, \sigma. ([[(n, \sigma, t_0)]] = v \implies \langle t_0, \sigma \rangle \longrightarrow v)$$

which is true by the second inductive hypothesis.

Application of a function:

We want to prove that:

$$\forall n, t_0, t_1, \overline{v}, \sigma. ([[(n, \sigma, t_0\, t_1)]] = \overline{v} \implies \langle t_0\, t_1, \sigma \rangle \longrightarrow \overline{v})$$

Let's assume the premise. It means that:

- $[[(n,\ \sigma,\ t_0)]] = \text{closure}(t, \sigma')$
- $[[(n, \sigma, t_1)]]\ =\ v$
- $\sigma'' = \text{bind}(\sigma', v)$

Then we use the following inductive hypotheses:

1. $\forall n, t_0, \text{closure}(t, \sigma'), \sigma. ([[(n, \sigma, t_0)]] = \text{closure}(t, \sigma') \implies \langle t_0, \sigma \rangle \longrightarrow \text{closure}(t, \sigma'))$
2. $\forall n, t_1, v, \sigma. ([[(n, \sigma, t_1)]] = v \implies \langle t_1, \sigma \rangle \longrightarrow v)$

And we are left with the burden of of proving:

$$\forall n, t, \overline{v}, \sigma''. ([[(n, \sigma'', t)]] = \overline{v} \implies \langle t, \sigma'' \rangle \longrightarrow \overline{v})$$

We can't use an inductive hypothesis as proving the above statement is as hard as proving the soundness for a generic term.

The proof is left unfinished but it can be completed using Rule Induction (Induction on Derivations, using formulas instead of terms).

Completeness

Completeness is defined as follows:

$$\forall t, \sigma, v. \exists n. \langle t, \sigma \rangle \longrightarrow v \implies [[(n, \sigma, t)]] = v$$

The proof is omitted.

Compilation and Virtual Machine

To compile and execute our code in the virtual machine we need to:

1. define the virtual machine that execute the code.
   - define the syntax of the language of the VM
   - define the semantics of that language
   - define an interpreter for that language
2. define a compilation step for our code that transforms a program written in Lambda+ in the code that the VM executes, similar to the transformation of Java to Bytecode

Continuing the similarities with Java, our virtual machine will be stack-based, its computation will be performed on an operand stack, in a way that resembles the JVM.
To keep things manageable the stack is used as an operand stack and as a call stack, but in our case, the "activation records" only store the "return address" and the environment.

Virtual Machine

We start by defining the syntax for the language of our virtual machine:

$$\begin{aligned} C :=\ & \mathrm{PVar}(Var) \mid \mathrm{PClosure}(C) \mid \mathrm{PValue}(Val) \\ & \mid \mathrm{Apply} \mid \mathrm{Return} \mid \mathrm{If}(C_0, C_1) \mid \mathrm{AOP} \mid \mathrm{BOP} \mid C; C \mid \mathrm{Skip} \end{aligned}$$

$$\begin{aligned} Val :=\ & b \in \mathbb{B} \mid n \in \mathbb{Z} \mid \mathrm{Closure}(C, \sigma) \mid \mathrm{Record}(C, \sigma) \\ OP :=\ & \ \mathrm{Add} \mid \mathrm{Sub} \mid \mathrm{Mul} \\ BOP :=\ & \ \mathrm{And} \mid \mathrm{Or} \mid \mathrm{Not} \mid \mathrm{Eq} \mid \mathrm{Less} \mid \mathrm{Greater} \end{aligned}$$

Now we have to define the semantics of the language.
We provide a small-step semantic where the intermediate step is the triple $\langle c, \sigma, \gamma \rangle$, where:

- c is the command
- σ is the environment
- γ is the operand/call stack

Notation:

1. A program is seen as a sequence of commands. We decide that the last command before the termination will always be a Skip. This is to make it easier to identify the end of a program: a program has reached its end when it is no more a sequence of

commands, and the remaining command is a Skip. We will present an inference rule for enforcing every non-skip command into a sequence where the last command Skip

2. We represent the final state of the VM as a single value.
3. We use $v :: \sigma$ to represent the insertion/presence of v on top of the stack.
4. We use the bottom symbol $\perp$ for representing errors.

The small-step semantics of the VM is defined by the following inference rules:

$$\frac{C \neq C_1; \text{Skip}}{\langle C, \sigma, \gamma \rangle \to \langle C; \text{Skip}, \sigma, \gamma \rangle} \qquad \frac{}{\langle C_1; (C_2; C_3), \sigma, \gamma \rangle \to \langle (C_1; C_2); C_3, \sigma, \gamma \rangle}$$

$$\frac{v \neq \text{Record}(\_, \_)}{\langle \text{Skip}, \sigma, v :: \gamma \rangle \to v} \qquad \frac{\sigma[n] = v}{\langle \text{PVar}(n); C_2, \sigma, \gamma \rangle \to \langle C_2, \sigma, v :: \gamma \rangle} \qquad \frac{}{\langle \text{PVal}(v); C_2, \sigma, \gamma \rangle \to \langle C_2, \sigma, v :: \gamma \rangle}$$

$$\frac{}{\langle \text{PClosure}(C); C_2, \sigma, \gamma \rangle \to \langle C_2, \sigma, \text{Closure}(C, \sigma) :: \gamma \rangle}$$

$$\frac{v_1 = \text{Closure}(C, \sigma')}{\langle \text{Apply}; C_2, \sigma, v_2 :: v_1 :: \gamma \rangle \to \langle C, \text{bind}(v_2, \sigma'), \text{Record}(C_2, \sigma) :: \gamma \rangle} \qquad \frac{v_1 = \text{Record}(C', \sigma')}{\langle \text{Return}; C_2, \sigma, v_2 :: v_1 :: \gamma \rangle \to \langle C', \sigma', v_2 :: \gamma \rangle}$$

$$\frac{v = \text{true}}{\langle \text{If}(C_0; C_1); C_2, \sigma, v :: \gamma \rangle \to \langle C_0; C_2, \sigma, \gamma \rangle} \qquad \frac{v = \text{false}}{\langle \text{If}(C_0; C_1); C_2, \sigma, v :: \gamma \rangle \to \langle C_1; C_2, \sigma, \gamma \rangle}$$

$$\frac{v_1 \in Z \quad v_2 \in Z}{\langle \text{AOP}; C_2, \sigma, v_2 :: v_1 :: \gamma \rangle \to \langle C_2, \sigma, (v_2 \ \overline{\text{AOP}} \ v_1) :: \gamma \rangle}$$

Note: For brevity not all the rules have been listed:

- The rules for BOP are basically identical to the ones for arithmetic operations, hence are not shown
- The rule PClosure is not showed but is defined in the source code and behaves as suggested by its name.
- As we did for Lambda+ we here present only the rules for "correct" programs (rules that do not give $\perp$)
- The $\langle \text{Skip}; C, \sigma, \gamma \rangle$ is not listed as the semantics of a "non-final" Skip is trivial

Compilation of Lambda+

We now present the compilation step that transforms a program written in *Lambda+* to the language of the virtual machine.

We proceed in a similar way to the "interpretation function" and define the "compilation function" as a function that goes from well-formed Lambda+ terms to C:

$$[[\cdot]]_C : \mathbb{T} \to (C \cup \{\perp\})$$

The function $[[\cdot]]_C$ is defined by cases as follows:

$$[[v]]_C = \text{PVal(v)}$$

$$[[x]]_C = \begin{cases} \text{PVar(x) if } x \in Var \\ \perp \text{ otherwise} \end{cases}$$

$$[[t_0 \text{ aop } t_1]]_C = \begin{cases} ([[t_0]]_C; [[t_1]]_C); \text{AOP if } [[t_0]]_C, [[t_1]]_C \neq \perp \\ \perp \;\text{ otherwise} \end{cases}$$

$$[[\text{if } t \text{ then } t_0 \text{ else } t_1]]_C = \begin{cases} [[t]]_C; \text{If}([[t_0]]_C, [[t_1]]_C) \text{ if } [[t]]_C, [[t_0]]_C, [[t_1]]_C \neq \perp \\ \perp \text{ otherwise} \end{cases}$$

$$[[\lambda t]]_C = \begin{cases} (\text{PClosure}([[t]]_C; \text{Return}) \text{ if } [t]]_C \neq \perp \\ \perp \;\text{ otherwise} \end{cases}$$

$$[[t_0\ t_1]]_C = \begin{cases} ((([[t_0]]_C; [[t_1]]_C); \text{Apply}) \text{ if } [[t_0]]_C, [[t_1]]_C \neq \perp \\ \perp \text{ otherwise} \end{cases}$$

Project Delivery

The file `lambda_plus.v` implements all the content described in this report.
Note: We advise to use CoqIDE to read the source code as other editors might have problems with the indentation.