# A brief introduction to theoretical concepts in programming languages:formal syntax and evaluation

Brigitte Pientka

March 5, 2007

So far we have seen key concepts and programming paradigms such as higher-order functions, data-types types, type inference, modules etc. Mainly we have learned how these concepts are exemplified in the programming language SML, and we often introduced these concepts by showing different examples. This gives us a good intuition of why these concepts are valuable and how we can use them, but often we need a more solid and more precise theoretical foundation to answer questions such as: What is the scope of a variable? What are the legal expressions I can write? Understanding these questions will help us to better understand the language we use. More importantly, we can also ask questions such as: What are the expressions which will be well-typed? How does SML infer the type of a given expression? How is the program going to be executed? How can we in general reason about a program? These questions are extremely important to understand why some programs are not correct or are rejected as ill-typed. In other words they will help us to debug the program. Understanding these theoretical concepts, will also help us to write well-structured and better code which is more likely to be correct. These notes introduce the theoretical concepts behind programming languages.

In order to talk rigorously about how programs behave and and reason about programs, we begin with a concise and formal specification of a programming language. Such a specification consists of three steps:

1. Grammar of the language (i.e. What are the syntactically legal expressions?)
2. Operational dynamic semantics (i.e. How is a given program executed?)
3. Static semantics (= type system) (i.e. When is a given program well-typed? What can we statically say about the execution of a program without actually executing it?)

We will begin by considering a tiny functional language called Nano-ML, and introduce its grammar and operational semantics. Later, we will also consider its static semantics. We would like to emphasize that this language Nano-ML is not identical to the programming language SML, although it share many of the main ideas. We will use this tiny fictional language to explain some of the underlying theoretical principles and ideas which underly

language desing. The techniques are general enough that the apply to many real languages such as SML, OCaml, or Haskell, and even object-oriented languages such as Java as well.

# 1 Inductive definitions of expressions

First, we would like to describe the legal expressions our tiny functional language consists of. Our language will have numbers and some basic arithmetic operations, booleans and if-statements. We will define the syntactically well-formed expressions inductively.

**Definition 1.1** *The set of expressions is defined inductively by the following clauses*

1. *A number $n$ is an expression.*
2. *The booleans* $\mathsf{true}$ *and* $\mathsf{false}$ *are expressions.*
3. *If $e_1$ and $e_2$ are expressions, then $e_1$ $\mathsf{op}$ $e_2$ is an expression where $\mathsf{op} = \{+, =, -, *, <\}$.*
4. *If $e_1$, $e_2$ and $e_3$ are expressions, then* $\mathsf{if}$ $e$ $\mathsf{then}$ $e_1$ $\mathsf{else}$ $e_2$ *is an expression.*

This inductive definition is often considered too verbose, and a shorter way of describing what expressions are well-formed, is the Backus-Naur Form (BNF). The left side introduces the the symbol for operations $\mathsf{op}$ and expressions $e$, and the right side describes recursively the set of syntactically valid expressions. The vertical bar | indicates a choice.

$$\begin{array}{lll} \text{Operations } \mathsf{op} & ::= & + \mid - \mid * \mid < \mid = \\ \text{Expressions } e & ::= & n \mid e_1\ \mathsf{op}\ e_2 \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \end{array}$$

This grammar inductively specifies well-formed expressions. To illustrate, we consider some well-formed and some ill-formed expressions.

**Examples of well-formed expressions**

- $3 + (2 + 4)$
- $\mathsf{true} + (2 + 4)$
- $\mathsf{if}\ (2 = 0)\ \mathsf{then}\ 5 + 3\ \mathsf{else}\ 2$
- $\mathsf{if\ true\ then\ (if\ false\ then}\ 5\ \mathsf{else}\ 1 + 3)\ \mathsf{else}\ 2 = 5$
- $(\mathsf{if}\ 0\ \mathsf{then}\ 55\ \mathsf{else}\ 77 - 23) = 0$

**Examples of ill-formed expressions**

- if true then 2 else
- $-4$
- $+23$

Note that the grammar only tells us when expressions are syntactically well-formed. The grammar does not say anything whether a given expression is for example well-typed. In fact, you may notice that the syntactically well-formed expressions (3) and (4) are not well-typed. Similarly, you may argue that $-4$ is a perfectly sensible expression. This is of course correct, but our grammar does not accept it as a well-formed syntactically expression, since any operation requires two arguments.

# 2 Operational semantics

Next, we would like to formally describe how a given expression is going to be executed. In this section, we will define a high-level operational semantics for the tiny language we have seen so far. We begin by defining the evaluation judgment:

$$e \Downarrow v \quad \text{Expression } e \text{ evaluates to a final value } v$$

The first question which comes to mind is: What are final values $v$? – In the tiny language we have introduced so far, we expect a final value to be either a number $n$ or a boolean, true or false. More formally, we can write:

$$\text{Values } v \quad ::= \quad n \mid \mathsf{true} \mid \mathsf{false}$$

Now we can define recursively how to evaluate an expression by analyzing its structure. If we have already have a value (i.e. a number $n$ or a boolean true or false), then there is nothing to evaluate and we will simply return this value. To evaluate the expression $e_1 + e_2$ we evaluate the sub-expression $e_1$ to some value $v_1$ and the sub-expression $e_2$ to some value $v_2$. The final value of $e_1 + e_2$ is then obtained by adding the two values $v_1$ and $v_2$. In other words, once we have values $v_1$ and $v_2$ we rely on our basic primitive operations of arithmetic and add the two values $v_1$ and $v_2$. This is written as $\overline{v_1 \; op \; v_2}$. Similarly, to evaluate the expression $e_1 * e_2$, we evaluate the sub-expression $e_1$ to some value $v_1$ and the sub-expression $e_2$ to some value $v_2$. The final value of $e_1 * e_2$ is then obtained by multiplying the two values $v_1$ and $v_2$. More generally, we evaluate the expression $e_1 \; op \; e_2$ as follows:

- Evaluate sub-expression $e_1$ to some value $v_1$
- Evaluate sub-expression $e_2$ to some value $v_2$
- Compute the final value $\overline{v_1 \; op \; v_2}$ by appealing to the corresponding primitive operation.

We will now turn the informal description given above into a formal one using inference rules. In general, an inference rule has the following shape:

$$\frac{\text{premise}_1 \quad \ldots \quad \text{premise}_n}{\text{conclusion}} \text{ name}$$

The part below the line is called conclusion while the parts above the line are called premises. To the right, we often write the name of the rule. We can read an inference rule as follows: To achieve the conclusion, we must satisfy each of the premises. In other words, if the premises are satisfied, we can conclude the conclusion.

Let's look how we can use this formal notation to describe the evaluation of expressions.

We start by defining the evaluation rules for numbers and booleans. Numbers and booleans are already values, and hence there is nothing to evaluate and we simply return this number. This is described by the rule B-NUM, B-TRUE and B-FALSE. Note that these rules do not have any premises because any value evaluates to itself unconditionally.

$$\frac{}{n \Downarrow n} \text{ B-NUM} \qquad \frac{}{\mathsf{true} \Downarrow \mathsf{true}} \text{ B-TRUE} \qquad \frac{}{\mathsf{false} \Downarrow \mathsf{false}} \text{ B-FALSE} \Downarrow$$

Next, let us reconsider the evaluation of expression $e_1$ $\mathsf{op}$ $e_2$ and cast the informal recipe given above in a more formal description.

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1 \ \mathsf{op} \ e_2 \Downarrow \overline{v_1 \ \mathsf{op} \ v_2}} \text{ B-OP}$$

The rule B-OP says the following: If $e_1 \Downarrow v_1$ (i.e. expression $e_1$ evaluates to some value $v_1$") and $e_2 \Downarrow v_2$ (i.e. "expression $e_2$ evaluates to some value $v_2$), then $e_1$ $\mathsf{op}$ $e_2 \Downarrow \overline{v_1 \ \mathsf{op} \ v_2}$ (i.e. "expression $e_1$ $\mathsf{op}$ $e_2$ evaluates to $\overline{v_1 \ \mathsf{op} \ v_2}$).

Reading the rule B-OP from the bottom to the top, this rules gives a recipe on how to evaluate an expression $e_1$ $\mathsf{op}$ $e_2$, by recursively evaluate its sub-expressions. The first premise $e_1 \Downarrow v_1$ says "evaluate expression $e_1$ to some value $v_1$ and the second premise $e_2 \Downarrow v_2$ says "evaluate expression $e_2$ to some value $v_2$. Then compute the final value $\overline{v_1 \ \mathsf{op} \ v_2}$ by appealing to the corresponding primitive operation.

Finally, let us consider the if-statement.

$$\frac{e \Downarrow \mathsf{true} \quad e_1 \Downarrow v}{\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Downarrow v} \text{ B-IFT} \qquad \frac{e \Downarrow \mathsf{false} \quad e_2 \Downarrow v}{\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Downarrow v} \text{ B-IFF}$$

There are two rules for evaluating if-expressions. If the guard $e$ evaluates to $\mathsf{true}$, we will evaluate the first branch (rule B-IFT), and if the guard $e$ evaluates to $\mathsf{false}$, we will evaluate the second one (rule B-IFF).

**Remark** Note that the evaluation rules do not impose an order in which premises need to be evaluated. For example, when evaluating $e_1 \mathsf{op} e_2$, we can first evaluate $e_1$ and then $e_2$ or the other way round. The rule just specifies that both subexpressions need to be evaluated. Big-step evaluation provides a high-level description of the operational semantics, and abstracts over the order in which we evaluate sub-expressions.

**Evaluation** How does evaluation work with the inference rules given? – We can evaluate an expression by applying the inference rules and constructing a derivation. Next, we give an example of an evaluation derivation to illustrate the use of the evaluation rules.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{4 \Downarrow 4}\,\text{B-NUM} \quad \dfrac{}{1 \Downarrow 1}\,\text{B-NUM}}{(4-1) \Downarrow 3}\,\text{B-OP} \quad \dfrac{}{6 \Downarrow 6}\,\text{B-NUM}}{((4-1) < 6) \Downarrow \mathsf{true}}\,\text{B-OP} \quad \dfrac{\dfrac{}{3 \Downarrow 3}\,\text{B-NUM} \quad \dfrac{}{2 \Downarrow 2}\,\text{B-NUM}}{3+2 \Downarrow 5}\,\text{B-OP}}{\mathsf{if}\ ((4-1) < 6)\ \mathsf{then}\ 3+2\ \mathsf{else}\ 4 \Downarrow 5}\,\text{B-IFT}$$

Evaluating an expression essentially means to construct such a derivation. However, there are many expressions which do not have a value, for example $\mathsf{true} + 3$ or $\mathsf{if}\ 0\ \mathsf{then}\ 3\ \mathsf{else}\ 4$ do not have a value. So how do we know that some expressions do not yield a value? – The answer is that there exists not derivation tree for such expressions, since there are no rules for evaluating these expressions. For example the derivation for $\mathsf{if}\ 0\ \mathsf{then}\ 3\ \mathsf{else}\ 4$ is stuck because $0 \Downarrow 0$. But the rules for if-expressions require that the guard either evaluates to $\mathsf{true}$ or to $\mathsf{false}$. Since there is no rule which specifies what to do in the case where the guard evaluates to a number, evaluation fails. Failure is handled implicitly.

# 3 Properties

Surprisingly, our tiny language already has some interesting properties. For example, we know that any successful evaluation of an expression will indeed yield a value, i.e. either a number $n$ or a boolean $\mathsf{true}$ or $\mathsf{false}$. This property is called *value soundness*. Moreover, we know the valuation is deterministic and yields a unique value.

**Value soundness** If $e \Downarrow v_1$ then $v_1$ is a value.

**Determinacy** If $e \Downarrow v_1$ and $e \Downarrow v_2$ then $v_1 = v_2$.

Both these properties can be proven by structural induction, but this is beyond this course.