Operational semantics of a small functional programming language.

Johan Montelius

January 12, 2018

Introduction

The operational semantics of a language should answer the question what an expression will evaluate to. The description should not leave any room for interpretation since the description is what defines the language.

The operational semantics should preferably be described in a way that it also captures the time and memory complexity of an execution. It does not have to be a detailed description of how things are actually implemented but it should give an understanding of the execution to allow a programmer to reason about the efficiency of a particular program.

The operational semantics of a language can also serve as a architecture for an abstract machine for the language or as the design criteria for a compiler. The observable properties of a program execution should conform to the properties one can derive from the description of the operational semantics.

There are many ways to describe an operational semantics of a programming language and we will use a strategy called *big-step semantics*. We will describe the semantics as a set of rewrite rules or evaluation relation; given an expression in the language we ill describe the rule of how to evaluate the expression to obtain an answer.

This description of an operational semantics for our small functional programming language will serve our purposes in that we will be able to talk and reason about program execution. We will also be able to use it when we implement an interpreter for the language.

1 The language

Our language is a very small functional programming language that should behave similar to Elixir. We will in the beginning only have a handful of data types and program constructions but will the extend the language to look more like a proper language. To start with our language will only consist of the constructs to express a sequence of pattern matching expressions followed by a single expression.

```
x = :foo, y = :nil, {z,_} = {:bar,:grk}, {x,{z,y}}.
```

Figure 1: a simple sequence

The expressions that we allow are simple *term expressions* i.e. no function calls, *case* or *if expressions*. We define this language using a BNF grammar.

We assume that we have two lexical constructs, *atoms* and *variables* that consist of all words written in with an initial lowercase, atoms, and initial uppercase letters, variables. We also have one compound expression called a *cons*, a simple binary construct that holds two expressions.

⟨*expression*⟩ ::= ⟨*atom*⟩
  | ⟨*variable*⟩
  | '{' ⟨*expression*⟩ ',' ⟨*expression*⟩ '}'

If this was the whole language we would only be able to write expressions like `:foo, x, :zot`. This would not be that interesting so we add the the description of a *sequence*.

⟨*sequence*⟩ ::= ⟨*expression*⟩
  | ⟨*match*⟩ ';' ⟨*sequence*⟩

⟨*match*⟩ ::= ⟨*pattern*⟩ '=' ⟨*expression*⟩

A *pattern matching expression* consists of a *pattern* and an expressions, and the pattern will look almost exactly as an expression. The only difference is that we allow `'_'` which will be called *don't care*.

⟨*pattern*⟩ ::= ⟨*atom*⟩
  | ⟨*variable*⟩
  | '_'
  | '{' ⟨*pattern*⟩ ',' ⟨*pattern*⟩ '}'

An expression is just a syntactical construct, the grammar presented does not give any meaning to the expressions that we can write bit it defines which sequences of characters are legal sequences, patterns and expressions.

2 The domain

The *domain* is the set of data structures that will be the result of our computation. It is the set of all *Atoms* and *compound* data structures that we can generate from elements in the domain.

$$\text{Atoms} = \{a, b, c, ...\}$$

$$\text{Structures} = \text{Atoms} \cup \{\{d_1, d_2\} | d_i \in \text{Structures}\}$$

We have a one-to-one mapping from our atoms in the language and the atoms in the domain. You might wonder if they are not the same but

there is a difference between expressions in our language and elements in our domain. Think about the written number "12" or in Roman "XII" and the number twelve. We will have expression that look like {`:foo`, {`:bar`, `:zot`}} and data structures that we write {*foo*, {*bar*, *zot*}}. If everything works out fine, the *evaluation* of an expression will return the corresponding data structure.

3 An environment

In the description of the evaluation we will need an *environment*. This is a mapping from variables in expressions to elements in the domain. We will start with an empty environment and gradually add more information as we evaluate a sequence of pattern matching expressions.

An environment is represented as a set of bindings v/s. Here v is a variable (and we will typically use v or x, y etc when we talk about variables) and t is a data structure. An environment that binds X to a and Y to b would then be written:

$$\{x/a, y/b\}$$

4 The evaluation function

So we're now ready to describe the operational semantics of our programming language and we do this by describing how an expressions is *evaluated*. We will start with the simplest expressions and then work our way up to more complex expressions.

The evaluation of an expression e is written $E\sigma(e)$, meaning that we evaluate the expression in the context of the environment σ.

When we describe our rules for evaluation we will use a notation that is:

$$\frac{\text{prerequisite}}{E\sigma(\text{expression}) \rightarrow \text{result}}$$

In this description E is the evaluation function, and we will also have rules for other functions and σ an environment. The result of applying the function is a data structure.

In order to apply the rule the prerequisite must be met and this will in the end guide us in how we can implement a recursive evaluator. The so called *big-step* operational semantics is often used since it is easily turned into an implementation of the language.

4.1 atoms

The simplest rule is the one that describes how we evaluate an expression consisting of a simple atom.

$$\frac{a \equiv s}{E\sigma(a) \to s}$$

This mean that if we have an atom, for example `:foo` then this is evaluated to the corresponding data structure *foo*. The environment σ, is in this case not relevant.

A variable is of course different since we then need to consult the environment for a binding of the variable.

$$\frac{v/s \in \sigma}{E\sigma(v) \to s}$$

The rule for a compound expression {`:foo, :bar`} is straight forward Given that we can evaluate its two components we will of course be able to evaluate the compound expression.

$$\frac{E\sigma(e_1) \to s_1 \qquad E\sigma(e_2) \to s_2}{E\sigma(\{e_1, e_2\}) \to s_1, s_2}$$

4.2 a pattern matching expression

Slightly more complex is how to evaluate a pattern matching expression. What we need to do is to first evaluate the right hand side and then try to match the pattern of the left hand side to the data structure that we obtain. The result of a pattern matching is either an extended environment or a *failure*. This failure is important since we will later use it in our case statement.

We will start with one rule that might seam to be a bit redundant but it will save us some problems when we have to write up rules for failure. A pattern matching given a failed environment will result in a failure.

$$\frac{}{P \text{ fail } (e, s) \to \text{fail}}$$

We now proceed with the rules how to do pattern matching. The first two rules are simple; an atom will of course match its corresponding data structure and the don't care symbol will match anything.

$$\frac{a \equiv s}{P\sigma(a, s) \to \sigma}$$

$$\frac{}{P\sigma(\_, s) \to \sigma}$$

If we try to match an atom to a data structure that is not the corresponding data structure then we fail.

$$\frac{a \not\equiv s}{P\sigma(a, s) \to \text{fail}}$$

If we have a unbound variable as a pattern then the variable is bound to a structure in the environment.

$$\frac{v/t \notin \sigma}{P\sigma(v, s) \rightarrow \{v/s\} \cup \sigma}$$

If the variable is bound to the equivalent structure we proceed without extending the environment but if it is bound to something else we fail.

$$\frac{v/s \in \sigma}{P\sigma(v, s) \rightarrow \sigma}$$

$$\frac{v/t \in \sigma \wedge t \not\equiv s}{P\sigma(v, s) \rightarrow \text{fail}}$$

Matching a cons expression is quite simple but note what the rules says about the environment. We need to do the pattern matching of the expression e_1 and the data structure s_1 in σ but the o matching of e_2 and s_2 in σ'. We thus gain information in the first matching that must be consisting with the second matching.

$$\frac{P\sigma(p_1, s_1) \rightarrow \sigma' \wedge P\sigma'(p_2, s_2) \rightarrow \theta}{P\sigma(\{p_1, p_2\}, \{s_1, s_2\}) \rightarrow \theta}$$

As an exercise you should do the pattern matching of the expression {`x,` {`x, :c`}} and the data structure {*a,* {*b, c*}}. Note how we first add *x/a* as a binding and then *shadow* this binding when we match {`x, :c`} and {*b,c*}.

The remaining alternative, the case where we have a cons expression and we try to match this to an data structure that is not a compound data structure, will of course lead to a failure.

$$\frac{s \neq \{s_1, s_2\}}{P\sigma(\{p_1, p_2\}, s) \rightarrow \text{fail}}$$

That it is as far as pattern matching goes. Try to do some matching by hand and explain which rules you apply. Try these:

- $P(\{: b, : a\}, \{a, b\})$
- $P(\{x, : b\}, \{a, b\})$
- $P(\{x, x\}, \{a, a\})$
- $P(\{x, x\}, \{a, b\})$

4.3 a sequence

So now we're ready to evaluate a sequence; a sequence that always consist of zero or more pattern matching expressions followed by a single expression. The pattern matching expressions will, if they succeed, add more bindings to the environment as we proceed and the final expression is then evaluated given this environment.

In order to describe this we introduce a new rule, a rule that describes how a new scope is created.

$$\frac{\sigma' = \sigma \setminus \{v/t \quad | \quad v/t \in \sigma \quad \wedge \quad v \quad \text{in} \quad p\}}{S(\sigma, p) \rightarrow \sigma'}$$

The new environment will not have any bindings for the variables that occur in the pattern. If we have a variable `x` in σ it will simply be *shadowed* by the pattern matching expression. This is quite differently from how things are handled in Erlang.

The rule that describes the evaluation of a sequence is now quite straight forward. We first evaluate the expression, e, of the pattern matching expression, then evaluate the matching giving us an updated environment, θ, that is used to evaluate the remaining sequence.

$$\frac{E\sigma(e) \rightarrow t \qquad \sigma' = S(\sigma, p) \qquad P\sigma'(p, t) \rightarrow \theta \qquad E\theta(\text{sequence}) \rightarrow s}{E\sigma(p = e, \text{sequence}) \rightarrow s}$$

A sequence consist of one or more patter matching expressions followed by an expression, so the rule will terminate once we reach the final expression.

4.4 that's it

That is it, we now have all the rules to evaluate a sequence on the form shown in fig:1. Make sure that you understand the rules and how they are applied by evaluating the sequence by hand. If you get it right the result will be {*foo,* {*bar, nil*}}. When you get it right you're ready to continue.

5 Adding a case expression

The expressions that we have seen so far are rather boring. In order to write a program that is at lest marginally interesting we need a construct that evaluates to different data structures depending on the state of the execution. We could have introduced a *if-then-else* expression but we choose to introduce a so called *case expression*.

We first need to extend the grammar so that we have a syntax to express our new construct. We choose a syntax that is similar to the case expression in Erlang.

⟨*expression*⟩ ::= ⟨*case expression*⟩ | ...

⟨*case expression*⟩ ::= 'case' ⟨*expression*⟩ 'do' ⟨*clauses*⟩ 'end'

⟨*clauses*⟩ ::= ⟨*clause*⟩ | ⟨*clause*⟩ ';' ⟨*clauses*⟩

⟨*clause*⟩ ::= ⟨*pattern*⟩ '->' ⟨*sequence*⟩

We then extend the rules for evaluation an expression, hopefully capturing our intended meaning. We will use a new set of rules, C, that will describe how the right clause is selected.

$$\frac{E\sigma(e) \to t \qquad C\sigma(t, \text{clauses}) \to s}{E\sigma(\texttt{case}\ e\ \texttt{do}\ \text{clauses}\ \texttt{end}) \to s}$$

The right clause is selected by trying to match the pattern of the first clause with the data structure obtained by evaluating the expression. If the pattern matching succeeds we will continue to evaluate the sequence of the clause, in the extended environment. Note that we also here create new scopes for the variables in the pattern.

$$\frac{\sigma' = S(\sigma, p) \qquad P\sigma'(p, s) \to \theta \qquad \theta \neq \text{fail} \qquad E\theta(\text{sequence}) \to s}{C\sigma(s, p \text{ -> sequence; clauses}) \to s}$$

This rule could of course also be used even if *clauses* is empty i.e. we match the last or only clause in a sequence.

If the pattern matching fails we will simply try the next clause in the sequence of clauses.

$$\frac{\sigma' = S(\sigma, p) \qquad P\sigma'(p, s) \to \text{fail} \qquad C\sigma(s, \text{clauses}) \to s}{C\sigma(s, p \text{ -> sequence; clauses}) \to s}$$

If no pattern matches the evaluation will simply terminate unsuccessfully. In real life this would mean that we receive a *CaseClauseError* or similarly.

We now have everything we need to handle case expressions in our language, this is starting to look like something.

6 Adding lambda expressions

The real task is when we want to add lambda expressions, or unnamed functions. To do this wee need to do several things. We need to extend the syntax to represent a function and to apply a function to a sequence of arguments. We also need to add a new data structure to represent a function and, extend the rules of evaluation to give everything a meaning.

6.1 free variables

We want to know which variables in the function expression that are *free*. To see the problem let's look at the function expression that has a match expression in the sequence.

```
fn (x) -> y = 5; x + y + z end
```

Which variables are *free* in this expression? The variable `x` is not free since it is in the *scope* of the function parameter. Nor is the local variable `y` free since it is in the scope of the pattern matching expression. If you would translate this to lambda calculus the expression would look like follows:

$$\lambda x \rightarrow \text{let} \quad y = 5 \quad \text{in} \quad x + y + z$$

The variable `z` is however free and in order to make use of this function expression one would have to do it in an environment where `z` has a value. A function and the needed environment, i.e. values for all free variables, is called a *closure*. We need to introduce new constructs in our language to create closures and apply them to arguments.

6.2 function expression and application

So we introduce two new constructs in our language, one to express a function and one to apply a function to a sequence of arguments. Different from Elixir we don't allow patterns in function parameters; this is only to make the rules of the evaluation easier to describe.

A function consist of the keyword `fn` followed by a, possibly empty, sequence of parameters (all unique variables). After the arrow we have a regular sequence as we have defined before and everything is finished by the keyword `end`.

⟨*function*⟩ ::= 'fn' '(' ⟨*parameters*⟩ ')' '->' ⟨*sequence*⟩ 'end'

⟨*parameters*⟩ ::= ' ' | ⟨*variables*⟩

⟨*variables*⟩ ::= ⟨*variable*⟩ | ⟨*variable*⟩ ',' ⟨*variables*⟩

A function application is simply any expression (that hopefully will be evaluated to a *closure* and a sequence of arguments enclosed in parentheses. We here follow the Elixir syntax that requires a '.' between the name less function and the sequence of arguments. The arguments can of course be arbitrary expressions.

⟨*expression*⟩ ::= ⟨*expression*⟩'.(' ⟨*arguments*⟩ ') | ...

⟨*arguments*⟩ ::= ' ' | ⟨*expressions*⟩

⟨*expressions*⟩ ::= ⟨*expression*⟩ | ⟨*expression*⟩ ',' ⟨*expressions*⟩

6.3 closures

The next thing we need to do is to extend our set of data structures. When we evaluate a function expression we will construct the closure. The closure is a triplet: the parameters of the function, the sequence to evaluate and an environment.

$$\text{Closures} = \{\langle p : s : e\rangle \quad | \quad p \in \text{Par} \wedge s \in \text{Seq} \wedge e \in \text{Env}\}$$

$$\text{Structures} = \text{Closures} \cup ..$$

We have not formally defined what the set of paramters, sequences nor environmens are but the informal description will work for our purposes.

The environment of a closure is constructed by taking the bindings of all free variables in the sequence, not including the variables that are bound by the parameters.

$$\frac{\theta = \{v/s \mid v/s \in \sigma \wedge v \text{ free in sequence}\}}{E\sigma(\text{fn(parameters) } -> \text{ sequence } end) \rightarrow \quad \langle \text{parameters} : \text{sequence} : \theta\rangle}$$

6.4 applying a closure

So now to the interesting part where we apply a closure to a sequence of arguments. We first need to evaluate the arguments and then add a set of bindings to the environment of the closure. We then evaluate the sequence in the updated environment.

$$\frac{E\sigma(e) \rightarrow \langle v_1, \ldots : \text{seq} : \theta\rangle \qquad E\sigma(e_i) \rightarrow s_i \qquad E\{v_1/s_1, \ldots\} \cup \theta(\text{seq}) \rightarrow s}{E\sigma(e.(e_1, \ldots)) \rightarrow s}$$

Note that the closure that we get, $\langle v_1, \ldots : \text{seq} : \theta\rangle$ consist of a sequence of variables $v_1, \ldots$, that are all distinct. This is why we can simply add the bindings v_i/s_i to θ, there will not be any duplicate bindings.

Looks complicated but it's quite straight forward. Go through the evaluation of sequence 2. If everything works out fine the result should be {*foo, bar*}.

```
x = :foo; f = fn (y) -> x,y end; f.(:bar)
```

Figure 2: a sequence with a function

7 An interpreter

The big-step operational semantics that we have used in describing the language is very useful when one wants to implement an interpreter for the language. If we can only come up with a scheme to represent expressions, data structures and environments then the rules will give us a recursive interpreter with very little effort.