Evaluation

Johan Montelius

KTH

VT23

semantics

We will define a small subset of the Elixir language and describe the *operational semantics*.

Warning - this is not exactly how Elixir works ... but it could have been.

expressions

The language is described using a BNF notation.

⟨*atom*⟩ ::= :a | :b | :c | . . .

⟨*variable*⟩ ::= x | y | z | . . .

⟨*literal*⟩ ::= <atom>

⟨*expression*⟩ ::= <literal> | <variable> | '{' <expression> ',' <expression> '}'

Examples: `{:a,:b}` , `{x,y}` , `{:a, {:b, z}}`

Simple expressions are also referred to as *terms*.

patterns

A *pattern* is a syntactical construct that uses almost the same syntax as terms.

⟨*pattern*⟩ ::= <literal>
| <variable>
| '_'
| '{' <pattern> ',' <pattern> '}'

The _ symbol can be read as “don't care”.

sequence

$\langle$*match*$\rangle$ ::= <pattern> '=' <expression>

$\langle$*sequence*$\rangle$ ::= <expression>
| <match> ';' <sequence>

examples:

- `x = :a; {:b, x}`
- `x = :a; y = {:b, x}; {:a, y}`

evaluation

When we *evaluate* sequences, the result will be a *data structure*.

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

$$\text{Structures} = \text{Atoms} \cup \{\{s_1, s_2\} | s_i \in \text{Structures}\}$$

An evaluation can also result in $\perp$, called "bottoms", this represents a failed evaluation.

injective mapping

For every atom a, there is a corresponding data structure s.

We write $\mathrm{a} \mapsto s$.

`:foo` $\mapsto$ *foo*
`:gurka` $\mapsto$ *gurka*

For every digit `1,2,3` (or `I, II, III`) there is a corresponding number $1, 2, 3$.

Our language could have data structures that do not have corresponding terms.

evaluation

A sequence is evaluated given an *environment*, written σ (sigma).

The environment holds a set of variable substitutions (bindings): $v/s \in \sigma$, v is a variable and s is a structure.

An evaluation of a sequence e given an environment σ is written $E\sigma(e)$.

We write:

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

where *result* is a *data structure*.

evaluation of expressions

We have the following rules for evaluation of expressions:

Evaluation of an atom:

$$\frac{a \mapsto s}{E\sigma(a) \rightarrow s}$$

Evaluation of a variable:

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

Evaluation of a compund structure:

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

evaluation tree

$$\dfrac{\dfrac{:\mathrm{foo} \mapsto foo}{E\{x/bar\}(:\mathrm{foo}) \rightarrow foo} \qquad \dfrac{\mathrm{x}/bar \in \{x/bar\}}{E\{x/bar\}(\mathrm{x}) \rightarrow bar}}{E\{x/bar\}(\{:\mathrm{foo}, \mathrm{x}\}) \rightarrow \{foo, bar\}}$$

evaluation of expressions

What if we have $E\sigma(v)$ and $v/s \notin \sigma$?

$$\frac{v/s \notin \sigma}{E\sigma(v) \rightarrow \perp}$$

evaluation of expressions

assume: $\sigma = \{x/\{a, b\}\}$

$E\sigma(:c) \quad \rightarrow \quad c$
$E\sigma(\mathtt{x}) \quad \rightarrow \quad \{a, b\}$

assume: $\sigma = \{x/a, y/b\}$

$E\sigma(\{x, y\}) \quad \rightarrow \{a, b\}$

pattern matching

The result of evaluating a *pattern matching* is a an extended environment. We write:

$$P\sigma(p, s) \rightarrow \theta$$

where θ (theta) is the extended environment.

Match an atom:

$$\frac{a \mapsto s}{P\sigma(a, s) \rightarrow \sigma}$$

Match an unbound variable:

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

Match a bound variable:

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

Match `ignore`:

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

matching failure

What do we do with $P\sigma(a, s)$ when $a \not\mapsto s$?

$$\frac{a \not\mapsto s}{P\sigma(a, s) \rightarrow \mathrm{fail}}$$

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

A fail is not the same as $\perp$.

matching compound strcutures

If the pattern is a compound pattern, the components of the pattern are matched to their corresponding sub structures.

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

Note that the second part is evaluated in σ'.

Example: $P\{\}(\{x, \{y, x\}\}, \{a, \{b, c\}\})$

Match a compund pattern with anyting but a compound structure will fail.

examples

assume: $\sigma = \{y/b\}$

- $P\sigma(\mathrm{x}, a) \rightarrow \{x/a\} \cup \sigma$
- $P\sigma(\mathrm{y}, b) \rightarrow \sigma$
- $P\sigma(\mathrm{y}, a) \rightarrow \mathrm{fail}$
- $P\sigma(\{y, y\}, \{a, b\}) \rightarrow \mathrm{fail}$

pattern matching

Pattern matching can *fail*.

fail is different from $\perp$
We will use failing to guide the program execution, more on this later.

evaluation of sequences

A new *scope* is created by removing variable bindings from an environmet.

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

A sequence is evaluated one pattern matching expression after the other.

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

Erlang and Elixir differ in how this rule is defined.

example

```
x = :a; y = :b; {x,y}
```


Where are we now

We have defined the semantics of a programming language (not a complete language) by defining how expressions are evaluated.

Important topics:

- set of data structures: atoms and compound structures
- environment: that binds variables to data structures
- expressions: term expressions, pattern matching expressions and sequences
- evaluation: from expressions to data structures $E\sigma(e) \rightarrow s$

Why

Why do we do this?

more

What is missing:

- evaluation of *case* (and *if* expressions)
- evaluation of function applications

case expression

```
case  x  do
    :a ->  :foo
    :b ->  :bar
end
```


case expression

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

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

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

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

evaluation of case expression

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

$C\sigma(s, \text{clauses})$ will select one of the clauses based on the patterns of the clauses and then continue the evaluation of the sequence of the selected clause.

selection of a clause

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

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

$$\frac{S\sigma(p) \rightarrow \sigma' \qquad P\sigma'(p, s) \rightarrow \text{fail}}{C\sigma(s, p -> \text{sequence}) \rightarrow \perp}$$

example

$E\{x/\{a, b\}\}(\texttt{case x do :a -> :a; \{\_,y\} -> y end}) \rightarrow$

$E\{X/\{a, b\}\}(\text{x}) \rightarrow \{a, b\}$

$C\{X/\{a, b\}\}(\{a, b\},\ \texttt{:a -> :a; \{\_,y\} -> y}) \rightarrow$

$P\{x/\{a, b\}\}\}(\text{:a}, \{a, b\}) \rightarrow \text{fail}$

$C\{x/\{a, b\}\}(\{a, b\},\ \texttt{\{\_,y\} -> y}) \rightarrow$

$P\{x/\{a, b\}\}(\texttt{\{\_,y\}}, \{a, b\}) \rightarrow \{y/b,\ x/\{a, b\}\}$

$E\{y/b, x/\{a, b\}\}(\text{y}) \rightarrow$

b

free variables

Are all syntactical correct sequences also valid sequences?

A sequence must not contain any *free variables*.

A free variable in a `<sequence>` is bound by the pattern matching expressions in the sequence `<patter> = <expression>, <sequence>` if the variable occurs in the `<pattern>`.

A free variable in a `<sequence>` is bound by the pattern matching expressions in the clause `<pattern> -> <sequence>` if the variable occurs in the `<pattern>`.

free variables

```
x = :a; {y,z} = {x,:b}; {x,y,z}
```

```
{y,z} = {x,:b}; {x,y,z}
```

```
x = {:a,:b}; case x do {:a,z} -> z end
```


much ado about nothing

A lot of work for something that simple - why bother, it could not have been done differently.

variable scope

```
x = {:a,:b};
y = case x do
      {:a, z} -> {:c, z}
    end;
{y, z}
```

This is not allowed in our language, z in {y,z} is a free variable. However .... is allowed in Erlang and was until changed allowed in Elixir (fixed in v1.5).

what's missing

Handle lambda expressions, closures and function application.