Big-step semantik for JUNTA - fri projektrelateret opgave

d402f13

23. maj 2013

Dette er vores besvarelse på den frie projektrelaterede opgave, hvor vi har valgt en delmængde af vores sprog, JUNTA, som er et funktionelt objektorienteret programmeringssprog.

Tankerne bag sproget er, at man skal kunne programmere brætspil i sproget, der kan spilles ved hjælp af en simulator. Mange funktionelle principper ses i sproget, da der ingen sideeffekter er og ingen programtilstande findes.

Herunder defineres semantikken for følgende konstruktioner i sproget: Let-in-udtryk, metode-kald, member-tilgang, liste-tilgang, lambda-udtryk, set-konstruktionen og typedefinitioner (typer minder om klasser i andre objektorienterede sprog).

Contents

Definitionerne er på engelsk, da rapporten skrives på engelsk (og disse ses som et udtræk af rapporten).

1 Syntactic categories

Before we can describe the behaviour of programs written in JUNTA and their lexical structure, we must first present the syntax of programs. At this point, we are only interested in a notion of abstract syntax because we do not need to concern ourselves with operator precedence and so forth.

The different syntactic categories are seen in table 1.

2 Environments

In this section we present the definitions which will be used throughout the construction of semantics for JUNTA. In the following definitions we use the syntactic categories presented in table 1. For some of the definitions we define arbitrary members which will be used in the semantics of the constructs of the language.

$n \in$	**Integer**
$x \in$	**Variable**
$s \in$	**String**
$E \in$	**Expression**
$P \in$	**Pattern**
$L \in$	**List**
$X \in$	**VarList**
$Y \in$	**Coordinate**
$Z \in$	**Direction**
$C \in$	**ConstantNames**
$T \in$	**TypeNames**
$D_G \in$	**GlobalDef**
$D_M \in$	**MemberDef**

Table 1: *The syntactic categories of* JUNTA.

Definition (Type environment) The set of type environments is the set of partial functions from type names to type values:

$$\mathbf{EnvT} = \mathbf{TypeNames} \rightharpoonup \mathbf{TypeValues}$$

Definition (Constant environment) The set of constant environments is the set of partial functions from constant names to expressions and variable lists:

$$\mathbf{EnvC} = \mathbf{ConstantNames} \rightharpoonup \mathbf{Expression} \times \mathbf{VarList}$$

An arbitrary member is defined as $env_C \in \mathbf{EnvC}$.

Definition (Variable environment) The set of variable environments is the set of partial functions from variables to values:

$$\mathbf{EnvV} = \mathbf{Variable} \rightharpoonup \mathbf{Values}$$

An arbitrary member is defined as $env_V \in \mathbf{EnvV}$.

Definition (Values) The set of values can contain many different values, and is defined as follows:

$$\begin{aligned}\mathbf{Values} = {} & \mathbf{Integers} \cup \mathbf{Strings} \cup \mathbf{Lists} \cup \mathbf{Patterns} \cup \mathbf{Coordinates} \cup \mathbf{Directions} \\ & \cup \mathbf{TypeValues} \cup \mathbf{FunctionValues} \cup \mathbf{ObjectValues} \cup \mathbf{Booleans}\end{aligned}$$

Definition (List values) The values of lists is defined as follows:

$$\mathbf{ListValues} = \mathbf{Integers} \times \mathbf{Elements}$$

A length of an arbitrary list is defined as $l \in \mathbf{Integer}$.

Definition (List elements) The set of elements in lists is the set of partial functions from integers to values:

$$\mathbf{Elements} = \mathbf{Integers} \rightharpoonup \mathbf{Values}$$

An arbitrary member is defined as $elem \in \mathbf{Elements}$.

Definition (Function values) The set of function values is defined as follows:

$$\textbf{FunctionValues} = \textbf{VarLists} \times \textbf{Expressions} \times \textbf{EnvV} \times \textbf{EnvC}$$

A function value consists of a variable list, an expression and the set of variable and constant environments.

Definition (Type values) The set of type values is defined as follows:

$$\textbf{TypeValues} = \textbf{TypeNames} \times \textbf{VarLists} \times \textbf{D}_{\textbf{M}} \times \textbf{List} \times \textbf{TypeValues}$$

An arbitrary member is defined as $t \in \textbf{TypeValues}$.

A type value consists of a type name, a variable list, the member definitions, a list and the super type's type value. A type value is recursively defined since **TypeValues** is defined in terms of itself. This is not a problematic definition because these must be considered as values for two different types. The **TypeValues** on the right-hand side is in fact the super type's set of values whereas the **TypeValues** on the left-hand side is the current type's set of values. The list in the definition is in fact the super type's parameters whereas the variable list is the current type's formal parameters.

Definition (Object values) The set of object values (an instantiated **TypeValue**) is defined as follows:

$$\textbf{ObjectValues} = \textbf{TypeValues} \times \textbf{EnvC} \times \textbf{EnvV} \times \textbf{ObjectValues}$$

An object value consists of the set of type values, the set of constant environment, the set of variable environments and the set of object values. This is yet another recursively defined definition, because an object value can have a parent object value, this is the case when an object value extends another object value.

2.1 Formulation rules

Each syntactic category is used in one or more of the formulation rules presented in figure 1. The formulation rules define the structure of the members of the syntactic categories.

Not all of the constituents of the formulation rules are syntactic categories. We for instance see different parentheses, forward slashes, different operators, and words like `this` and `define`. These are part of the construction of the given formulation rule. If they are omitted from a rule then it is not valid in JUNTA.

$$
\begin{array}{rcl}
E & ::= & n \mid x \mid s \mid Y \mid Z \mid T \mid C \mid L \mid \texttt{-}E \mid \texttt{(}\,E\,\texttt{)} \mid \texttt{/}\,P\,\texttt{/} \mid \\
 & & E_1\ L \mid E_1\texttt{.}C \mid \texttt{\#}\ X\ \texttt{=>}\ E \mid \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E3 \mid E_1\ \texttt{is}\ E_2 \mid \texttt{not}\ E \mid \\
 & & E_1\ \texttt{and}\ E_2 \mid E_1\ \texttt{or}\ E_2 \mid E_1\ \texttt{==}\ E_2 \mid E_1\ \texttt{!=}\ E_2 \mid E_1\ \texttt{<}\ E_2 \mid E_1\ \texttt{>}\ E_2 \mid \\
 & & E_1\ \texttt{<=}\ E_2 \mid E_1\ \texttt{>=}\ E_2 \mid E_1\ \texttt{+}\ E_2 \mid E_1\ \texttt{-}\ E_2 \mid E_1\ \texttt{*}\ E_2 \mid E_1\ \texttt{/}\ E_2 \mid \\
 & & E_1\ \texttt{\%}\ E_2 \mid \texttt{this} \mid \texttt{super} \mid \texttt{let}\ x_1\ \texttt{=}\ E_1,\ x_2\ \texttt{=}\ E_2,\ \cdots,\ x_k\ \texttt{=}\ E_k\ \texttt{in}\ E_{k+1} \mid \\
 & & \texttt{set}\ x_1\ \texttt{=}\ E_1,\ x_2\ \texttt{=}\ E_2,\ \cdots,\ x_k\ \texttt{=}\ E_k \\
L & ::= & \texttt{[}E_1,\ \cdots,E_k\texttt{]} \\
X & ::= & \texttt{[}x_1,\ \cdots,\ x_k\texttt{]} \mid \texttt{[}x_1,\ \cdots,\ \ldots x_k\texttt{]} \mid \texttt{[}\ldots x\texttt{]} \\
D_G & ::= & \texttt{define}\ C\ \texttt{=}\ E\ D_G \mid \texttt{define}\ C\ X\ \texttt{=}\ E\ D_G \mid \texttt{type}\ T\ X\ D_G \mid \\
 & & \texttt{type}\ T\ X\ \texttt{extends}\ T\ L\ D_G \mid \texttt{type}\ T\ X\ \{D_M\}\ D_G \mid \\
 & & \texttt{type}\ T\ X\ \texttt{extends}\ T\ L\ \{D_M\}\ D_G \mid \varepsilon \\
D_M & ::= & \texttt{define}\ C\ \texttt{=}\ E\ D_M \mid \texttt{define}\ C\ X\ \texttt{=}\ E\ D_M \mid \texttt{define abstract}\ C\ D_M \mid \\
 & & \texttt{define abstract}\ C\ X\ D_M \mid \texttt{data}\ x\ \texttt{=}\ E\ D_M \mid \varepsilon
\end{array}
$$

Figure 1: *The formulation rules for the syntactic categories of* JUNTA.

The $::=$ means that the left-hand side of the rule can be any one of the $|$-separated right-hand sides. Furthermore, we use "$\cdots$" to illustrate a repetition of some element in the rule. We have also used the slightly different "$\ldots$" to illustrate the three dots that precede a variable argument (vars) (this is explained in the report - and has been omitted from this document). It should be clear from the context which of the two is being used.

The ε represents an empty definition.

3 Big-step semantics

3.1 Lists

The semantics presented in table 2 are the transition rules for lists.

$$[\text{LIST}_{\text{ACCESS-1}}] \quad \frac{env_T, env_C, env_V \vdash \langle E\ \rangle \rightarrow v_2 \qquad env_T, env_C, env_V \vdash \langle L \rangle \rightarrow v_3}{env_T, env_C, env_V \vdash \langle E\ L \rangle \rightarrow v_1}$$

where $v_2 = (l_1, elem_1)$
and $v_3 = (l_2, elem_2)$
and $l_2 = 1$
and $i = elem_2 = 0$
and $v_1 = \begin{cases} elem_1\ i & \text{if } i \geq 0 \\ elem_1\ (l_1 + i) & \text{if } i \geq 0 \end{cases}$

$$[\text{LIST}_{\text{ACCESS-2}}] \quad \frac{env_T, env_C, env_V \vdash \langle E \rangle \rightarrow v_2 \qquad env_T, env_C, env_V \vdash \langle L \rangle \rightarrow v_3}{env_T, env_C, env_V \vdash \langle E\ L \rangle \rightarrow v_1}$$

where $v_2 = (l_1, elem_1)$
and $v_3 = (l_2, elem_2)$
and $l_2 = 2$
and $i = \begin{cases} elem_2\ 0 & \text{if } elem_2\ 0 \geq 0 \\ l_1 + elem_2\ 0 & \text{if } elem_2\ 0 < 0 \end{cases}$
and $j = \begin{cases} elem_2\ 1 & \text{if } elem_2\ 1 \geq 0 \\ l_1 + elem_2\ 1 & \text{if } elem_2\ 1 < 0 \end{cases}$
and $elem_3\ z = \begin{cases} elem_1\ i + 1 & \text{if } z = 0 \\ \vdots & \\ elem_1\ i + n - 1 & \text{if } z = n - 1 \end{cases}$
and $v_1 = (n = j - i + 1, elem_3)$

Table 2: *Transition rules for let expressions.*

3.2 Let-expressions

The semantics presented in table 3 are the transition rules for the let expression. This is transition rule is defined recursively to best illustrate the functionality of the expression.

The transition rules for [LET-1] is recursively because we must evaluate each expression ($x_1 = E_1$) before we move on to the next one. This is a must because of the fact that the next expressions can in fact make use of the previous expressions value. As an example take a look at the following code sample:

```
let $a = 2, $b = $a * 2
  in $b + $a
```
(1)

$$\text{[LET-1]} \quad \frac{env_T, env_C, env_V[x_1 \mapsto v_1] \vdash \langle \texttt{let}\ x_2 = E_2, \cdots,\ x_k = E_k\ \texttt{in}\ E_{k+1}\rangle \rightarrow v_{k+1}}{env_T, env_C, env_V \vdash \langle \texttt{let}\ x_1 = E_1,\ x_2 = E_2, \cdots, x_k = E_k\ \texttt{in}\ E_{k+1}\rangle \rightarrow v_{k+1}}$$

where $env_T, env_C, env_V \vdash E_1 \rightarrow v_1$
and $k \geq 2$

$$\text{[LET-2]} \quad \frac{env_T, env_C, env_V[x_1 \mapsto v_1] \vdash \langle E_2\rangle \rightarrow v_2}{env_T, env_C, env_V \vdash \langle \texttt{let}\ x_1 = E_1\ \texttt{in}\ E_2\rangle \rightarrow v_2}$$

where $env_T, env_C, env_V \vdash E_1 \rightarrow v_1$
and $k < 2$

Table 3: *Transition rules for let expressions.*

So, each call where there are more than one expression to be evaluated we call the transition rule [LET-1] where $k \geq 2$. Here the expression first in line to be evaluated will be evaluated before a new call to one of the two transition rules is made. When we reach a let expression with only one expression then we call the transition rule [LET-2] where $k < 2$.

3.3 Lambda expressions

The semantics presented in table 4 is the transition rule for the lambda expression.

$$\text{[LAMBDA]} \quad env_T, env_C, env_V \vdash \langle \texttt{\#}\ X\ \texttt{=>}\ E\rangle \rightarrow v \qquad \text{where } v = (X, E, env_V, env_C)$$

Table 4: *Transition rules for lambda expressions.*

The three environments (env_T, env_C, env_V) must be known before it is possible to execute a lambda expression. We need to know which types, constants and different variables are given in the specific scope.

The lambda expressions evaluates to a value v. The side condition of the transition rule explains that v is assigned the 4-tuple.

3.4 Set expressions

The semantics presented in table 5 is the transition rules for set expressions.

$$\text{[SET]} \quad \frac{env_C, env_V, env_T \vdash \langle E_1\rangle \rightarrow u_1 \quad \ldots \quad env_C, env_V, env_T \vdash \langle E_k\rangle \rightarrow u_k}{env_C, env_V, env_T \vdash \langle \texttt{set}\ x_1 = E_1, \ldots, x_k = E_k\rangle \rightarrow v_1}$$

where $env_v\ \texttt{this} = (t, env'_c, env'_v, v_2)$
and $v_1 = (t, env'_c, env'_v, v_2)$
and $env''_v = env'_v\,[x_1 \mapsto u_1, \ldots, x_k \mapsto u_k]$

Table 5: *Transition rules for set expressions.*

3.5 Function calls

The semantics presented in table 6 is transition rules for function calls.

$$[\text{CALL}_{\text{FUN}}] \quad \frac{\begin{array}{c} env_C, env_V, env_T \vdash \langle E \rangle \rightarrow v_2 \\ env_C, env_V, env_T \vdash \langle L \rangle \rightarrow v_3 \\ env'_C, env''_V, env_T \vdash \langle E' \rangle \rightarrow v_1 \end{array}}{env_C, env_V, env_T \vdash \langle E\ L\ \rangle \rightarrow v_1}$$

where $v_2 = (X, E', env'_V, env'_C)$
and $v_3 = (l, elem)$
and $env''_V = [x_1 \mapsto elem\ 1, \ldots, x_n \mapsto elem\ n]$

Table 6: *Transition rules for set expressions.*

The semantics presented in table 7 is the transition rule for member access.

$$[\text{MEMBER}_{\text{ACCESS}}] \quad \frac{env_C, env_V, env_T \vdash \langle E \rangle \rightarrow v_1}{env_C, env_V, env_T \vdash \langle E.C \rangle \rightarrow v_3}$$

where $v_2 = (t, env'_C, env'_V, v_2)$
and $env'_C\ C = v_3$

Table 7: *Transition rules for set expressions.*

3.6 Type definitions

The semantics presented in table 8 are the transition rules for type definitions. These type definitions have some optional arguments which correspond with the written grammar for these definitions, and this is why there are four transition rules described.

In the premises of the rules we present a 5-tuple where env_T is updated according to the rule. In three of the four 5-tuples we include the symbol ε, which denotes that the given position of the symbol is an empty slot. This is again due to the fact that we have some optional arguments.

The 5-tuple is ordered as follows:

1. $\mathbf{T_1}$ - current type
2. **X** - current type's formal parameters
3. $\mathbf{D_M}$ - member definitions
4. **L** - super type's parameters
5. $\mathbf{T_2}$ - super type

Throughout the transition rules we use the 5-tuple to update the type environment.

[TYPEDEF]	$\dfrac{env_C \vdash \langle D_G, env_T[T \mapsto (T, X, \varepsilon, \varepsilon, \varepsilon)]\rangle \rightarrow env'_T}{env_C \vdash \langle \texttt{type}\ T\ X\ D_G,\ env_T\rangle \rightarrow env'_T}$
[TYPEDEF$_{\text{BODY}}$]	$\dfrac{env_C \vdash \langle D_G, env_T[T \mapsto (T, X, D_M, \varepsilon, \varepsilon)]\rangle \rightarrow env'_T}{env_C \vdash \langle \texttt{type}\ T\ X\ \{D_M\}\ D_G,\ env_T\rangle \rightarrow env'_T}$
[TYPEDEF$_{\text{EXTEND}}$]	$\dfrac{env_C \vdash \langle D_G, env_T[T_1 \mapsto (T_1, X, \varepsilon, L, T_2)]\rangle \rightarrow env'_T}{env_C \vdash \langle \texttt{type}\ T_1\ X\ \texttt{extends}\ T_2\ L\ D_G,\ env_T\rangle \rightarrow env'_T}$
[TYPEDEF$_{\text{EXTEND-BODY}}$]	$\dfrac{env_C \vdash \langle D_G, env_T[T_1 \mapsto (T_1, X, D_M, L, T_2)]\rangle \rightarrow env'_T}{env_C \vdash \langle \texttt{type}\ T_1\ X\ \texttt{extends}\ T_2\ L\ \{D_M\}\ D_G,\ env_T\rangle \rightarrow env'_T}$

Table 8: *Transition rules for type definitions.*