

Homework 11

The primary purpose of this assignment is to explore objects in object-oriented languages. Concretely, we will extend JAKARTA SCRIPT with object literals, field dereference expressions, and field assignments. We extend our type checker to support the new features, but we do not yet add support for subtyping. At this point, we have many of the key features of JavaScript/TypeScript.

Problem 1 Warm-Up on Objects (16 Points)

This is a warm-up exercise to get yourself familiar with objects (which you will implement in the second part of this homework).

For each of the following programs: (1) say whether the program can be safely evaluated or not, i.e., whether the evaluation will get stuck or produce a value. If it can be safely evaluated, provide the computed value; (2) determine whether the program is well-typed according to the typing rules given in figures 3 and 4. (Note that this rules do **not** yet allow for subtyping.) If the program is not well-typed, provide a brief explanation of the type error.

(i)

```
1  const x = {let f: 3};
2  const fun =
3    (y: {let f: Num, let g: Bool}) => (y.g = true, y);
4  fun(x).f
```

(ii)

```
1  const x = {let f: 3};
2  const fun =
3    (y: {let f: Num, const g: Bool}) => (y.f = 4, y);
4  fun(x).f
```

(iii)

```
1  const x = {let f: 3};
2  const fun = (y: {let f: Num}) => (y.f = 4, y);
3  fun(x).f
```

(iv)

```
1  const x = {let f: 1, const g: true};
2  const y = {const f: false, let g: 2};
3  const z = true ? x : y;
4  z.f
```

$n \in Num$	numbers (double)
$s \in Str$	strings
$a \in Addr$	addresses
$b \in Bool ::= \mathbf{true} \mid \mathbf{false}$	Booleans
$x \in Var$	variables
$f \in Fld$	field names
$\tau \in Typ ::= \mathbf{Bool} \mid \mathbf{Num} \mid \mathbf{String} \mid \mathbf{Undefined} \mid$ $(\bar{\tau}) \Rightarrow \tau_0 \mid \{\overline{mut\ f:\tau}\}$	types
$v \in Val ::= \mathbf{undefined} \mid n \mid b \mid s \mid a \mid \mathbf{function}\ p(\overline{x:\bar{\tau}})t\ e$	values
$e \in Expr ::= x \mid v \mid uop\ e \mid e_1\ bop\ e_2 \mid e_1\ ?\ e_2 : e_3 \mid e.f \mid \{\overline{mut\ f:e}\}$ $\mathbf{console.log}(e) \mid e_1(\bar{e}) \mid mut\ x = e_1; e_2$	expressions
$uop \in Uop ::= - \mid ! \mid *$	unary operators
$bop \in Bop ::= + \mid - \mid * \mid / \mid === \mid !== \mid < \mid > \mid$ $<= \mid >= \mid \&\& \mid \mid , \mid =$	binary operators
$p ::= x \mid \epsilon$	function names
$t ::= :\tau \mid \epsilon$	return types
$mut \in Mut ::= \mathbf{const} \mid \mathbf{let}$	mutability
$o \in Obj = Fld \rightarrow Val$	objects
$k \in Con ::= v \mid o$	memory contents
$M \in Mem = Addr \rightarrow Con$	memories

Figure 1: Abstract syntax of JAKARTAScript

Problem 2 JAKARTAScript Interpreter with Objects (24 Points)

We start from our language in Homework 10 and extend it with object literals and field dereference operations. The syntax of the new language is shown in Figure 1.

An object literal $\{\overline{mut\ f:e}\}$ is a sequence of field names associated with initialization expressions and mutabilities. The mutability of a field indicates whether the field can be reassigned a new value after the object literal has been evaluated. When an object literal is evaluated, we first evaluate the initialization expressions of the fields to obtain an object value $\{f:v\}$. Note that object values are not actually values. In JavaScript, objects are dynamically allocated on the heap and then referenced with an extra level of indirection through a heap address. To model object allocation, object literals in JAKARTAScript evaluate to an address a , which is a value that references a memory location. This memory location stores the object value obtained from the object literal.

Aliasing. The indirection introduced by dynamic allocation of objects means two program variables can reference the same object, which is called aliasing. With mutation, aliasing is

```

/** Types */
enum Typ:
  ...
  /** Object types */
  case TObj(fts: Map[Fld, (Mut, Typ)]) // <~ {mut1 f1: tau1, ..., mutk fk: tauk}

/** Fields */
type Fld = String

/* Unary operators */
enum Uop:
  ...
  /** Field dereference */
  case FldDeref(f: String) // <~ _.f

/** Expressions */
enum Expr:
  ...
  /** Object literals */
  case ObjLit(fes: Map[Fld, (Mut, Expr)]) // <~ {mut1 f1: e1, ..., mutk fk: ek}

/** Objects (to be stored in memory) */
enum ObjCon:
  case Obj(fs: Map[Fld, Val]) // <~ {f1: v1, ..., fk: vk}

/** Values */
type Val = ...

/** Memory contents */
type Con = Val | Obj.type

```

Figure 2: The abstract syntax of JAKARTAScript as represented in Scala. After each **case**, we show the corresponding JavaScript expression represented by that case.

now observable as demonstrated by the following example:

```

const x = { f : 1 };
const y = x;
x.f = 2;
console.log(y.f)

```

The code above should print 2 because *x* and *y* are aliases (i.e., reference to the same object). Aliasing makes programs more difficult to reason about and is often the source of subtle bugs.

In Figure 2, we show the updated and new AST nodes. Note that in our Scala representation of ASTs, we treat field dereference as a unary operator `FldDeref` that is parameterized by the dereferenced field name. That is, a field dereference expression $e.f$ in JAKARTAScript is represented in Scala by the expression `UnOp(FldDeref(f), e)`. This design choice simplifies some aspects of our implementation. In particular, we will not have to extend the substitution function `subst` of our interpreter with an extra case for field dereference expressions.

Type Checking. The inference rules defining the typing relation are given in figures 3 and 4. The only change compared to Homework 10 are the new rules for typing objects, which are summarized in Figure 4. A template for the new type inference function

```
def typeInfer(env: Map[String, (Mut, Typ)], e: Expr): Typ
```

has been provided for you. The only missing cases are for the new rules in Figure 4. Your first task is to implement those missing cases.

Evaluation. The new big-step operational semantics is given in figures 5, 6, and 7. The rules are identical to those given in Homework 10, except that we add new rules for evaluating object literals, field dereference operations, and field assignments. These new rules are summarized in Figure 7.

Your second task is to update the `subst` and `eval` functions of the interpreter to account for the new language primitives.

- The `subst` function has the same signature as last time

```
def subst(e: Expr, x: String, er: Expr): Expr
```

The only missing case is the case for object literals. For object literals of the form $\{mut_1 f:e_1, \dots, mut_k f_k:e_k\}$ the substitution function should simply recurse into the field initialization expressions. That is, we have

$$\{mut_1 f:e_1, \dots, mut_k f_k:e_k\}[e_r/x] = \{mut_1 f:e_1[e_r/x], \dots, mut_k f_k:e_k[e_r/x]\}$$

Hint: use the `mapValues` method of the `Map` data structure in the Scala standard library to implement the recursive substitution inside the field initialization expressions in `fes`.

- The `eval` function has the same signature as last time

```
def eval(e: Expr): State[Mem, Val]
```

That is, `eval` takes an expression and returns a state monad `State[Mem, Val]` that encapsulates a computation from an input memory state to an output memory state and a result value. We suggest to proceed as follows:

- Copy over your solution from Homework 10 for the missing non-object cases (or see the sample solution for Homework 10 once it is released).
- Then implement the new cases for the rules `EVALDEREF` and `EVALASSIGN`. **Hint:** The helper functions `eToAddr` and `readObj` will be useful here.
- Finally, implement the case for the rule `EVALOBJLIT`. The template that has been provided for this case splits the evaluation into two parts: (1) the construction of the map `o` for the object to be stored in memory, (2) the actual allocation and update of the memory. Part (1) should be implemented with a `foldLeft` over the map `fes` holding the fields and associated expressions of the object literal. This `foldLeft` constructs a state monad `sm` whose result is the map `o` for the object. Part (2) should then be implemented with a `flatMap` call on the state monad returned by `foldLeft`. **Hint:** in part (1) use a `for` expression in the body of the function to which `foldLeft` is applied in order to extract the current field/value map from the accumulated state monad `sm`. Then extend this map in the `yield` part with the current field `fi` and the value obtained from `fi`'s initialization expression `ei`.

$$\begin{array}{c}
\frac{}{\Gamma \vdash b : \mathbf{Bool}} \text{TYPEBOOL} \quad \frac{}{\Gamma \vdash n : \mathbf{Num}} \text{TYPERNUM} \quad \frac{}{\Gamma \vdash s : \mathbf{String}} \text{TYPESTR} \\
\\
\frac{}{\Gamma \vdash \mathbf{undefined} : \mathbf{Undefined}} \text{TYPEUNDEFINED} \\
\\
\frac{\Gamma \vdash e : \mathbf{Num}}{\Gamma \vdash -e : \mathbf{Num}} \text{TYPEUMINUS} \quad \frac{\Gamma \vdash e : \mathbf{Bool}}{\Gamma \vdash !e : \mathbf{Bool}} \text{TYPENOT} \\
\\
\frac{\Gamma \vdash e_1 : \mathbf{Bool} \quad \Gamma \vdash e_2 : \mathbf{Bool} \quad bop \in \{\&\&, ||\}}{\Gamma \vdash e_1 \text{ } bop \text{ } e_2 : \mathbf{Bool}} \text{TYPEANDOR} \\
\\
\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1, e_2 : \tau_2} \text{TYPESEQ} \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathbf{console.log}(e) : \mathbf{Undefined}} \text{TYPEPRINT} \\
\\
\frac{\Gamma \vdash e_1 : \mathbf{String} \quad \Gamma \vdash e_2 : \mathbf{String}}{\Gamma \vdash e_1 + e_2 : \mathbf{String}} \text{TYPEPLUSSTR} \\
\\
\frac{\Gamma \vdash e_1 : \mathbf{Num} \quad \Gamma \vdash e_2 : \mathbf{Num} \quad bop \in \{+, *, /, -\}}{\Gamma \vdash e_1 \text{ } bop \text{ } e_2 : \mathbf{Num}} \text{TYPEARITH} \\
\\
\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \tau \in \{\mathbf{Num}, \mathbf{String}\} \quad bop \in \{>, >=, <, <=\}}{\Gamma \vdash e_1 \text{ } bop \text{ } e_2 : \mathbf{Bool}} \text{TYPEINEQUAL} \\
\\
\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \tau \text{ has no function types} \quad bop \in \{===, !==\}}{\Gamma \vdash e_1 \text{ } bop \text{ } e_2 : \mathbf{Bool}} \text{TYPEEQUAL} \\
\\
\frac{\Gamma \vdash e_1 : \mathbf{Bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash e_1 ? e_2 : e_3 : \tau} \text{TYPEIF} \\
\\
\frac{\Gamma \vdash e_d : \tau_d \quad \Gamma' = \Gamma[x \mapsto (mut, \tau_d)] \quad \Gamma' \vdash e_b : \tau_b}{\Gamma \vdash mut \ x = e_d ; e_b : \tau_b} \text{TYPEDECL} \\
\\
\frac{x \in \text{dom}(\Gamma) \quad \Gamma(x) = (mut, \tau)}{\Gamma \vdash x : \tau} \text{TYPEVAR} \quad \frac{\Gamma(x) = (\mathbf{var}, \tau) \quad \Gamma \vdash e : \tau}{\Gamma \vdash x = e : \tau} \text{TYPEASSIGNVAR} \\
\\
\frac{\begin{array}{l} \Gamma' = \Gamma[x_1 \mapsto (\mathbf{const}, \tau_1)] \dots [x_k \mapsto (\mathbf{const}, \tau_k)] \\ \Gamma' \vdash e : \tau \quad \tau' = (\tau_1, \dots, \tau_k) \Rightarrow \tau \end{array}}{\Gamma \vdash (x_1 : \tau_1, \dots, x_k : \tau_k) \Rightarrow e : \tau'} \text{TYPEFUN} \\
\\
\frac{\begin{array}{l} \Gamma \vdash e : (\tau_1, \dots, \tau_k) \Rightarrow \tau \\ \text{for all } i: \quad \Gamma \vdash e_i : \tau_i \end{array}}{\Gamma \vdash e(e_1, \dots, e_k) : \tau} \text{TYPECALL} \quad \frac{\begin{array}{l} \Gamma' = \Gamma[x_1 \mapsto (\mathbf{const}, \tau_1)] \dots [x_k \mapsto (\mathbf{const}, \tau_k)] \\ \Gamma' \vdash e : \tau \quad \tau' = (\tau_1, \dots, \tau_k) \Rightarrow \tau \end{array}}{\Gamma \vdash (x_1 : \tau_1, \dots, x_k : \tau_k) \Rightarrow : \tau \ e : \tau'} \text{TYPEFUNANN} \\
\\
\frac{\begin{array}{l} \Gamma' = \Gamma[x \mapsto \tau'] [x_1 \mapsto (\mathbf{const}, \tau_1)] \dots [x_k \mapsto (\mathbf{const}, \tau_k)] \\ \Gamma' \vdash e : \tau \quad \tau' = (\tau_1, \dots, \tau_k) \Rightarrow \tau \end{array}}{\Gamma \vdash \mathbf{function} \ x(x_1 : \tau_1, \dots, x_k : \tau_k) : \tau \ e : \tau'} \text{TYPEFUNREC}
\end{array}$$

Figure 3: Type checking rules for non-object primitives of JAKARTAScript (no changes compared to Homework 10)

$$\begin{array}{c}
\frac{\Gamma \vdash e_1 : \tau_1 \quad \dots \quad \Gamma \vdash e_k : \tau_k}{\Gamma \vdash \{mut_1 f_1 : e_1, \dots, mut_k f_k : e_k\} : \{mut_1 f_1 : \tau_1, \dots, mut_k f_k : \tau_k\}} \text{TYPEOBJLIT} \\
\\
\frac{\Gamma \vdash e : \{mut_1 f_1 : \tau_1, \dots, mut_k f_k : \tau_k\} \quad f = f_i \quad \tau = \tau_i \quad i \in [1, k]}{\Gamma \vdash e.f : \tau} \text{TYPEDEREFELD} \\
\\
\frac{\Gamma \vdash e_1 : \{mut_1 f_1 : \tau_1, \dots, mut_k f_k : \tau_k\} \quad f = f_i \quad \tau = \tau_i \quad mut_i = \mathbf{let} \quad i \in [1, k] \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1.f = e_2 : \tau} \text{TYPEASSIGNELD}
\end{array}$$

Figure 4: Type checking rules for objects of JAKARTA SCRIPT

$$\begin{array}{c}
\frac{}{\langle M, v \rangle \Downarrow \langle M, v \rangle} \text{ EVALVAL} \quad \frac{\langle M, e \rangle \Downarrow \langle M', n \rangle}{\langle M, -e \rangle \Downarrow \langle M', -n \rangle} \text{ EVALUMINUS} \quad \frac{\langle M, e \rangle \Downarrow \langle M', b \rangle}{\langle M, !e \rangle \Downarrow \langle M', !b \rangle} \text{ EVALNOT} \\
\\
\frac{\langle M, e_1 \rangle \Downarrow \langle M', \mathbf{true} \rangle \quad \langle M', e_2 \rangle \Downarrow \langle M'', v_2 \rangle}{\langle M, e_1 \ \&\& \ e_2 \rangle \Downarrow \langle M'', v_2 \rangle} \text{ EVALANDTRUE} \\
\\
\frac{\langle M, e_1 \rangle \Downarrow \langle M', \mathbf{false} \rangle \quad \langle M', e_2 \rangle \Downarrow \langle M'', v_2 \rangle}{\langle M, e_1 \ || \ e_2 \rangle \Downarrow \langle M'', v_2 \rangle} \text{ EVALORFALSE} \\
\\
\frac{\langle M, e_1 \rangle \Downarrow \langle M', \mathbf{false} \rangle}{\langle M, e_1 \ \&\& \ e_2 \rangle \Downarrow \langle M', \mathbf{false} \rangle} \text{ EVALANDFALSE} \quad \frac{\langle M, e_1 \rangle \Downarrow \langle M', \mathbf{true} \rangle}{\langle M, e_1 \ || \ e_2 \rangle \Downarrow \langle M', \mathbf{true} \rangle} \text{ EVALORTTRUE} \\
\\
\frac{\langle M, e_1 \rangle \Downarrow \langle M', v_1 \rangle \quad \langle M', e_2 \rangle \Downarrow \langle M'', v_2 \rangle}{\langle M, e_1, e_2 \rangle \Downarrow \langle M'', v_2 \rangle} \text{ EVALSEQ} \\
\\
\frac{\langle M, e \rangle \Downarrow \langle M', v \rangle \quad v \text{ printed}}{\langle M, \mathbf{console.log}(e) \rangle \Downarrow \langle M', \mathbf{undefined} \rangle} \text{ EVALPRINT} \\
\\
\frac{\langle M, e_1 \rangle \Downarrow \langle M', n_1 \rangle \quad \langle M, e_2 \rangle \Downarrow \langle M'', n_2 \rangle \quad n = n_1 + n_2}{\langle M, e_1 + e_2 \rangle \Downarrow \langle M'', n \rangle} \text{ EVALPLUSNUM} \\
\\
\frac{\langle M, e_1 \rangle \Downarrow \langle M', s_1 \rangle \quad \langle M', e_2 \rangle \Downarrow \langle M'', s_2 \rangle \quad s = s_1 + s_2}{\langle M, e_1 + e_2 \rangle \Downarrow \langle M'', s \rangle} \text{ EVALPLUSSTR} \\
\\
\frac{\langle M, e_1 \rangle \Downarrow \langle M', n_1 \rangle \quad \langle M', e_2 \rangle \Downarrow \langle M'', n_2 \rangle \quad n = n_1 \text{ bop } n_2 \quad \text{bop} \in \{*, /, -\}}{\langle M, e_1 \text{ bop } e_2 \rangle \Downarrow \langle M'', n \rangle} \text{ EVALARITH} \\
\\
\frac{\langle M, e_d \rangle \Downarrow \langle M', v_d \rangle \quad \langle M', e_b[v_d/x] \rangle \Downarrow \langle M'', v_b \rangle}{\langle M, \mathbf{const } x = e_d; e_b \rangle \Downarrow \langle M'', v_b \rangle} \text{ EVALCONSTDECL} \\
\\
\frac{\langle M, e_1 \rangle \Downarrow \langle M', n_1 \rangle \quad \langle M', e_2 \rangle \Downarrow \langle M'', n_2 \rangle \quad b = n_1 \text{ bop } n_2 \quad \text{bop} \in \{>, >=, <, <=\}}{\langle M, e_1 \text{ bop } e_2 \rangle \Downarrow \langle M'', b \rangle} \text{ EVALINEQUALNUM} \\
\\
\frac{\langle M, e_1 \rangle \Downarrow \langle M', s_1 \rangle \quad \langle M', e_2 \rangle \Downarrow \langle M'', s_2 \rangle \quad b = s_1 \text{ bop } s_2 \quad \text{bop} \in \{>, >=, <, <=\}}{\langle M, e_1 \text{ bop } e_2 \rangle \Downarrow \langle M', b \rangle} \text{ EVALINEQUALSTR} \\
\\
\frac{\langle M, e_1 \rangle \Downarrow \langle M', v_1 \rangle \quad \langle M', e_2 \rangle \Downarrow \langle M'', v_2 \rangle \quad b = (v_1 \text{ bop } v_2)}{\langle M, e_1 \text{ bop } e_2 \rangle \Downarrow \langle M'', b \rangle} \text{ EVALEQUAL} \\
\\
\frac{\langle M, e_1 \rangle \Downarrow \langle M', \mathbf{true} \rangle \quad \langle M', e_2 \rangle \Downarrow \langle M'', v_2 \rangle}{\langle M, e_1 ? e_2 : e_3 \rangle \Downarrow \langle M'', v_2 \rangle} \text{ EVALIFTHEN} \\
\\
\frac{\langle M, e_1 \rangle \Downarrow \langle M', \mathbf{false} \rangle \quad \langle M', e_3 \rangle \Downarrow \langle M'', v_3 \rangle}{\langle M, e_1 ? e_2 : e_3 \rangle \Downarrow \langle M'', v_3 \rangle} \text{ EVALIFELSE}
\end{array}$$

Figure 5: Big-step operational semantics of non-imperative non-object primitives of JAKAR-TAScript (no changes compared to Homework 10).

$$\begin{array}{c}
\frac{\langle M, e \rangle \Downarrow \langle M', v \rangle \quad a \in \text{dom}(M')}{\langle M, * a = e \rangle \Downarrow \langle M'[a \mapsto v], v \rangle} \text{ EVALASSIGNVAR} \\
\\
\frac{a \in \text{dom}(M)}{\langle M, * a \rangle \Downarrow \langle M, M(a) \rangle} \text{ EVALDEREFVAR} \\
\\
\frac{\langle M, e_d \rangle \Downarrow \langle M_d, v_d \rangle \quad a \notin \text{dom}(M_d) \quad M' = M_d[a \mapsto v_d] \quad \langle M', e_b[* a/x] \rangle \Downarrow \langle M'', v_b \rangle}{\langle M, \text{let } x = v_d; e_b \rangle \Downarrow \langle M'', v_b \rangle} \text{ EVALLETDECL} \\
\\
\frac{\langle M, e_0 \rangle \Downarrow \langle M_0, v_0 \rangle \quad v_0 = \mathbf{function } x_0(\overline{x_i:\tau_i}): \tau \ e \quad v'_0 = (\overline{x_i:\tau_i}) \Rightarrow (e[v_0/x_0]) \quad \langle M_0, v'_0(\overline{e_i}) \rangle \Downarrow \langle M', v \rangle}{\langle M, e_0(\overline{e_i}) \rangle \Downarrow \langle M', v \rangle} \text{ EVALCALLREC} \\
\\
\frac{\langle M, e_0 \rangle \Downarrow \langle M_0, v_0 \rangle \quad v_0 = (x_1:\tau_1, \overline{x_i:\tau_i}) \Rightarrow e \quad \langle M_0, e_1 \rangle \Downarrow \langle M_1, v_1 \rangle \quad v'_0 = (\overline{x_i:\tau_i}) \Rightarrow (e[v_1/x_1]) \quad \langle M_1, v'_0(\overline{e_i}) \rangle \Downarrow \langle M', v \rangle}{\langle M, e_0(e_1, \overline{e_i}) \rangle \Downarrow \langle M', v \rangle} \text{ EVALCALLCONST} \\
\\
\frac{\langle M, e_0 \rangle \Downarrow \langle M_0, () \Rightarrow e \rangle \quad \langle M_0, e \rangle \Downarrow \langle M', v \rangle}{\langle M, e_0() \rangle \Downarrow \langle M', v \rangle} \text{ EVALCALL}
\end{array}$$

Figure 6: Big-step operational semantics of imperative non-object primitives and functions in JAKARTAScript (no changes compared to Homework 10).

$$\begin{array}{c}
\frac{M_0 = M \quad \text{for all } i \in [1, k] : \langle M_{i-1}, e_i \rangle \Downarrow \langle M_i, v_i \rangle \quad a \notin \text{dom}(M_k) \quad o = \{f_1 \mapsto v_1, \dots, f_k \mapsto v_k\} \quad M' = M_k[a \mapsto o]}{\langle M, \{\text{mut}_1 f_1 : e_1, \dots, \text{mut}_k f_k : e_k\} \rangle \Downarrow \langle M', a \rangle} \text{ EVALOBJLIT} \\
\\
\frac{\langle M, e \rangle \Downarrow \langle M', a \rangle \quad M'(a) = o}{\langle M, e.f \rangle \Downarrow \langle M', o(f) \rangle} \text{ EVALDEREFFLD} \\
\\
\frac{\langle M, e_1 \rangle \Downarrow \langle M_1, a \rangle \quad \langle M_2, e_2 \rangle \Downarrow \langle M_2, v_2 \rangle \quad M_2(a) = o \quad M' = M_2[a \mapsto o[f \mapsto v_2]]}{\langle M, e_1.f = e_2 \rangle \Downarrow \langle M', v_2 \rangle} \text{ EVALASSIGNFLD}
\end{array}$$

Figure 7: Big-step operational semantics of objects in JAKARTAScript.