Operational Semantics of Kleis (Core Fragment)

In this appendix we present a big–step operational semantics for a core fragment of Kleis expressions. The judgment

$$\rho \vdash e \Downarrow v$$

is read: *in environment* ρ*, expression* e *evaluates to value* v*.*

Syntax (Core Fragment)

We consider the following informal grammar for expressions:

$$\begin{array}{rcl} e & ::= & x \mid v \mid \lambda x.e \mid e_1\, e_2 \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \\ & & \mid \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\ & & \mid \texttt{match}\ e\{p_1 \Rightarrow e_1 \ \mid\ \cdots\ \mid\ p_n \Rightarrow e_n\} \end{array}$$

Values:

$$v \quad ::= \quad c \mid \lambda x.e \mid C(v_1, \ldots, v_k)$$

where c is a primitive literal (number, boolean, string, etc.) and C ranges over data constructors of algebraic data types.

Patterns:

$$p \quad ::= \quad \_ \mid x \mid C(p_1, \ldots, p_k) \mid c$$

We assume a fixed set of primitive operations (+, -, etc.) given by a meta–level function $\mathrm{op}(v_1, \ldots, v_n)$.

Environments

An *environment* ρ is a finite mapping from variables to values. We write $\rho[x \mapsto v]$ for the environment that maps x to v and agrees with ρ on all other variables.

The lookup of a variable is written $\rho(x)$.

Values and Literals

Literals evaluate to themselves:

$$[\text{E-Const}]\rho \vdash c \Downarrow c$$

Lambda abstractions are values:

$$[\text{E-Lam}]\rho \vdash \lambda x.e \Downarrow \lambda x.e$$

Data constructors applied to values evaluate directly to a constructor value:

$$[\text{E-Constr}]\rho \vdash C(v_1, \ldots, v_k) \Downarrow C(v_1, \ldots, v_k)$$

Variables

Variables are looked up in the environment:

$$[\text{E-V}\textsc{ar}]\rho(x) = v\rho \vdash x \Downarrow v$$

Function Application

We use call–by–value semantics. To evaluate $e_1\, e_2$ we first evaluate e_1 to a function value, then e_2 to an argument value, then evaluate the body in an extended environment.

$$[\text{E-A}\textsc{pp}]\rho \vdash e_1 \Downarrow \lambda x.e\rho \vdash e_2 \Downarrow v_2\rho[x \mapsto v_2] \vdash e \Downarrow v\rho \vdash e_1\, e_2 \Downarrow v$$

If e_1 evaluates to a non-lambda value, the semantics is undefined (statically ruled out by typing).

`let`-Bindings

A `let`–binding evaluates its right-hand side and then evaluates the body under an extended environment:

$$[\text{E-L}\textsc{et}]\rho \vdash e_1 \Downarrow v_1\rho[x \mapsto v_1] \vdash e_2 \Downarrow v\rho \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow v$$

Conditionals

Booleans are assumed to be primitive values `True`, `False`.

$$[\text{E-I}\textsc{f}\text{T}\textsc{rue}]\rho \vdash e_0 \Downarrow \texttt{True}\rho \vdash e_1 \Downarrow v\rho \vdash \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow v$$

$$[\text{E-I}\textsc{f}\text{F}\textsc{alse}]\rho \vdash e_0 \Downarrow \texttt{False}\rho \vdash e_2 \Downarrow v\rho \vdash \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow v$$

If e_0 evaluates to a non-boolean, the semantics is undefined (again, ruled out statically).

Primitive Operations

For a primitive n-ary operator $\mathrm{op}(e_1, \ldots, e_n)$:

$$[\text{E-P}\textsc{rim}]\rho \vdash e_1 \Downarrow v_1 \quad \cdots \quad \rho \vdash e_n \Downarrow v_n\mathrm{op}(v_1, \ldots, v_n) = v\rho \vdash \mathrm{op}(e_1, \ldots, e_n) \Downarrow v$$

Here op is a meta–level interpretation of Kleis's built-in arithmetic, logical, or matrix operations.

Pattern Matching

We give the semantics of:

$$\texttt{match}\ e\{p_1 \Rightarrow e_1 \mid \ldots \mid p_n \Rightarrow e_n\}.$$

First, e is evaluated to a value v; then the patterns are tried in order until one matches. Pattern matching produces a *binding environment* θ (from pattern variables to subvalues), which is combined with ρ to evaluate the chosen branch.

We write:

$$\mathsf{match}(p, v) = \theta$$

to mean: pattern p matches value v and returns bindings θ, or is undefined if p does not match v.

Pattern Matching Auxiliary Rules We define $\mathsf{match}(p, v)$ by cases:

$$[\text{M-W}\textsc{ild}]\mathsf{match}(\_, v) = \emptyset$$

$$[\text{M-V}\textsc{ar}]\mathsf{match}(x, v) = [x \mapsto v]$$

$$[\text{M-C}\textsc{onst}]c = v\mathsf{match}(c, v) = \emptyset$$

$$[\text{M-C}\textsc{onstr}]v = C(v_1, \ldots, v_k)\mathsf{match}(p_1, v_1) = \theta_1 \quad \cdots \quad \mathsf{match}(p_k, v_k) = \theta_k \mathrm{dom}(\theta_i) \cap \mathrm{dom}(\theta_j) = \emptyset \text{ for } i \neq j$$

If no rule applies, $\mathsf{match}(p, v)$ is undefined.

Evaluation of `match` We write the branches as a list:

$$\mathcal{B} = (p_1 \Rightarrow e_1, \ldots, p_n \Rightarrow e_n).$$

We define:

$$[\text{E-M}\textsc{atch}]\rho \vdash e \Downarrow v\mathsf{firstMatch}(\rho, v, \mathcal{B}) = (\theta, e_k)(\rho \cup \theta) \vdash e_k \Downarrow v'\rho \vdash \texttt{match}\ e\{\mathcal{B}\} \Downarrow v'$$

where $\mathsf{firstMatch}$ is a meta–level function that searches the branch list from left to right:

$$\mathsf{firstMatch}(\rho, v, p_i \Rightarrow e_i, \ldots) = \begin{cases} (\theta, e_i) & \text{if } \mathsf{match}(p_i, v) = \theta, \\ \mathsf{firstMatch}(\rho, v, \mathrm{rest}) & \text{otherwise.} \end{cases}$$

If no branch matches, the semantics is undefined (a static exhaustiveness check in Kleis prevents this in well-typed programs).

Determinism

For this core fragment the semantics is deterministic:

$$\text{If } \rho \vdash e \Downarrow v_1 \text{ and } \rho \vdash e \Downarrow v_2, \text{ then } v_1 = v_2.$$

A proof proceeds by induction on derivations of $\rho \vdash e \Downarrow v$.

Extensions

Additional Kleis constructs (summations, integrals, derivatives, structure operations, etc.) may be given semantics by extending:

- the space of values (e.g. functionals, tensors, matrices),
- the primitive interpretation function op,
- the environment to include structure instances and built-in operators.

These extensions are conservative over the rules given above.