Eidola Semantics

Paul Cantrell

[Development Version]
Revised April 9, 2002

Here follows, gentle reader, the formal standard for the Eidola language. The current version of this document, supplementary material, and discussion are available at http://eidola.org/semantics. If you are new to Eidola, you should read "About Eidola" (http://eidola.org/about.shtml), which explains the motivation and ideas of the Eidola project.

Your comments and questions are *very* valuable to the project, so please send them! See http://eidola.org/contact.shtml for contact information.

Contents

0 Background and Terminology

This standard serves purposes typical of formal semantics — to aid in the design of the language, provide criteria for implementations, allow formal verification of the language and programs written in it, and so forth. Eidola's semantics also holds an unusual central role. Since Eidola is representation-independent, the semantics not only represents the language, but actually *is* the language — implementations represent the semantics instead of the reverse. What a wild thought!

This semantics expresses Eidola in terms of sets and functions over those sets. Some of these, the *fundamental structures*, are what determine a program in Eidola — a representation specifies a particular program by specifying the values of the fundamental structures. These include, for example, the set of elements, the name of an element, or the parents of a class.

The *derived structures* are uniquely determined by the fundamental structures. They include the full name of an element (derived from the names of the chain of owners), or the generalizations of a class (derived from the parents).

An Eidola *representation* is a mapping from the fundamental structures to some target domain, such that every legal Eidola program in the semantic domain has a representation in the target domain. A binary file format, a database schema, or a piece of software could all be representations. Note, however, that the semantics does not specify any representation.

An Eidola *kernel* is a special kind of representation which

- enforces the rules of the semantics,
- allows interactive modification of the fundamental structures, and
- provides mechanisms for observers to track these modifications.

A kernel will also likely want adaptors for translating a program to other representations.

If an Eidola kernel always strictly enforced all the rules of the semantics, it would carry the unhappy burden of proving that every legal program is reachable through a legal sequence of modifications, and then require the hapless programmer to discover these sequences. To avoid this problem, some of the rules in this semantics are *lazy rules* (marked in this document with $\bigstar$). A representation may allow violations of these rules. A kernel should track and flag any lazy violations it allows. A runtime environment, however, should be strict even about lazy rules, and refuse to execute a program which contains any semantic violations (lazy or otherwise).

1 High-Level Structure

1.1 Containers, Namespaces and Elements ($\mathcal{U}$, $\mathcal{N}$, $\mathcal{E}$)

Eidola programs have a fundamentally hierarchical structure, whose basic building blocks are the set of *containers* ($\mathcal{U}$). There are two kinds of containers: *namespaces* ($\mathcal{N}$), which are the outermost containers in the hierarchy; and *elements* ($\mathcal{E}$), which sit inside other containers.

$$\mathcal{U} = \mathcal{N} \cup \mathcal{E}$$

Namespaces correspond to physical divisions of a program, such as projects or libraries. Elements describe the structure of the program itself, and include everything from packages to expressions.

Every element has an *owner*, which contains it.

$$\text{owner} \;:\; \mathcal{E} \to \mathcal{U}$$

An element's *indirect owners* include all the elements in the chain of ownership from the element itself up to its namespace.

$$\begin{aligned} \mathit{owner}^* &: \mathcal{E} \to 2^{\mathcal{U}} \\ \mathit{owner}^*(e) &= \{e, \mathit{owner}(e), \mathit{owner}(\mathit{owner}(e)), \ldots, n \in \mathcal{N}\} \end{aligned}$$

The graph of owners is acyclic.

$$\forall\ e \in \mathcal{E}, e \notin \mathit{owner}^*(\text{owner}(e))$$

Every container has a set of *contents*.

$$\mathit{contents} : \mathcal{U} \to 2^{\mathcal{E}}$$

A container owns its contents, and an element is one of its owner's contents.

$$\begin{aligned} &\forall\ c \in \mathcal{U}, \forall\ e \in \mathit{contents}(c), \text{owner}(e) = c && \bigstar \\ &\forall\ e \in \mathcal{E}, e \in \mathit{contents}(\mathit{owner}(e)) && \bigstar \end{aligned}$$

A namespace holds a single *root* element (typically a package). The root is the namespace's only content, and the namespace owns it.

$$\begin{aligned} \text{root} &: \mathcal{N} \to \mathcal{E} \\ \mathit{contents}(n \in \mathcal{N}) &= \{\text{root}(n)\} \\ \mathit{owner}(\text{root}(n \in \mathcal{N})) &= n && \bigstar \end{aligned}$$

The function $\mathit{uses}(e)$ gives the containers which the element e directly references. An element always uses itself, its owner, and its contents, but may use other elements as well.

$$\begin{aligned} \mathit{uses} &: \mathcal{E} \to 2^{\mathcal{U}} \\ \mathit{uses}(e \in \mathcal{E}) &\supseteq \{e, \mathit{owner}(e)\} \cup \mathit{contents}(e) \end{aligned}$$

1.2 Named Elements ($\mathcal{E}_\text{N}$)

Named elements include such creatures as classes and variables which other elements may reference across the container hierarchy. Named elements are a subset of all the elements, and share their properties.

$$\mathcal{E}_\text{N} \subseteq \mathcal{E}$$

Named elements have (surprise!) a *name*, which they pull from a set of *identifiers* yet to be determined.[1]

$$\begin{aligned} \text{Ident} &= \{\ldots ?? \ldots\} \\ \text{name} &: \mathcal{E}_\text{N} \to \text{Ident} \end{aligned}$$

A container's named contents must have unique names.

$$\begin{aligned} &\forall\ e \in \mathcal{U},\ \forall\ n, m \in (\mathit{contents}(e) \cap \mathcal{E}_\text{N}), \\ &\quad \text{name}(n) = \text{name}(m) \implies n = m && \bigstar \end{aligned}$$

A container's *full name* is a globally unique identifier within a namespace, formed from the list of the names of its owners. We need to allow named elements to be contents of

[1]This could be the traditional alphanumeric set, or all unicode strings, or perhaps something even wilder. Some argue that identifiers or ever names shouldn't be part of the semantics; I'm not sure about this. Names are conceptually important, and may become semantically important when we start considering linking library code. See http://eidola.org/semantics/questions/identifiers.shtml.

unnamed elements, as will be the case with local variables (which belong to a block of code). In order to keep full names unique without having to contrive artificial names for unnamed elements, the "name" of an unnamed element is, for the purposes of the semantics, the element itself.

$$\begin{aligned} \textit{full-name} &: \mathcal{U} \to (\mathrm{Ident} \cup \mathcal{E})^{+} \\ \textit{full-name}\,(e \in \mathcal{E}) &= \textit{full-name}\,(\textit{owner}\,(e)) + \begin{cases} \mathrm{name}\,(e)\,, & e \in \mathcal{E}_\mathrm{N} \\ [e], & e \notin \mathcal{E}_\mathrm{N} \end{cases} \\ \textit{full-name}\,(n \in \mathcal{N}) &= [\;] \end{aligned}$$

Note that a corollary of name uniqueness of contents is that full names are globally unique within a namespace.

$$\begin{aligned} &\forall\; n \in \mathcal{N},\; \forall\; d, e \;:\; n \in \textit{owner}^{*}(d)\,, n \in \textit{owner}^{*}(e)\,, \\ &\qquad \textit{full-name}\,(d) = \textit{full-name}\,(e) \implies d = e \qquad \bigstar \end{aligned}$$

The *specialization* relation, $a \lhd b$ (read "a specializes b") or $b \rhd a$ ("b generalizes a") indicates when one named element fits the signature of another. It means that in a situation which requires something which looks like b, the element a will work. Specialization is the core of the Big Important Rule of Subtypes (see §1.5), and specialization of classes is the basis of Eidola's type system.

$$\lhd : \mathcal{E}_\mathrm{N} \times \mathcal{E}_\mathrm{N} \to \mathrm{boolean}$$

This relation is defined differently for different element types. The full definition appears in summary in §1.9.

1.3 Capsules ($\mathcal{E}_\mathrm{C}$)

Capsules are Eidola's generic encapsulation structure, and capture the numerous common properties of packages and classes. All capsules are named elements.

$$\mathcal{E}_\mathrm{C} \subseteq \mathcal{E}_\mathrm{N}$$

Capsules have *public members* and *private members*. Only other named elements can be members. Note that members do not have a particular order. (Although a notation could assign one for display purposes, it would not exist in this semantic domain.)

$$\begin{aligned} \mathrm{public}, \mathrm{private} &: \mathcal{E}_\mathrm{C} \to 2^{\mathcal{E}_\mathrm{N}} \\ \textit{contents}\,(e \in \mathcal{E}_\mathrm{C}) &= \mathrm{public}\,(e) \cup \mathrm{private}\,(e) \end{aligned}$$

Note that all types of named elements can be members of a capsule. This means that packages can be members of classes.[2]

No element can be both a public and a private member.

$$\forall\; e \in \mathcal{E}_\mathrm{C}, \mathrm{public}\,(e) \cap \mathrm{private}\,(e) = \emptyset \qquad \bigstar$$

[2]The meaning of a package inside a class is an open design question, and we may disallow it; see http://eidola.org/semantics/questions/packagesinclasses.shtml.

1.4 Packages ($\mathcal{P}$)

A *package* is a logical division of the namespace. Packages are capsules.

$$\mathcal{P} \subseteq \mathcal{E}_\mathrm{C}$$

Packages do not participate in the frivolous game of specialization; they specialize only themselves.

$$\forall\ p_1, p_2 \in \mathcal{P},\ p_1 \lhd p_2 \iff p_1 = p_2$$

A package uses the minimal set of elements possible — itself, its owner and its contents.

$$\mathit{uses}\,(p \in \mathcal{P}) = \{p, \mathit{owner}\,(p)\} \cup \mathit{contents}\,(p)$$

1.5 Classes ($\mathcal{C}$)

A *class* is an Eidola type. Classes are a kind of capsule.

$$\mathcal{C} \subseteq \mathcal{E}_\mathrm{C}$$

A class has a set of *parents*, other classes which it explicitly specializes.[3]

$$\text{parents} : \mathcal{C} \to 2^{\mathcal{C}}$$

The *generalizations* of a class include the class itself, its parents, its parents' parents, and so on. Note that cycles in the generalization graph are just fine — as we will see shortly, the classes in the cycle are simply all equivalent types. Class generalization is the "is-a" relation forming the basis of Eidola's type system, and governs things like parameter passing and assignment.

$$b \lhd a \iff b = a \text{ or } (\exists\ b' \in \text{parents}\,(b)\ \ :\ \ b' \lhd a)$$

A class uses the minimal set of elements possible, plus its parents. (Note that it does not use all the generalizations — just the parents.)

$$\mathit{uses}\,(c \in \mathcal{C}) = \{c, \mathit{owner}\,(c)\} \cup \mathit{contents}\,(c) \cup \text{parents}\,(c)$$

When one class specializes another, it must have a matching subsignature. This means that for every public member of every parent class, there must be a matching member of the child class. This is the *Big Important Rule of Subtypes.*

$$\begin{gathered}\forall\ c \in \mathcal{C},\ \forall\ p \in \text{parents}\,(c)\,,\ \forall\ m_p \in \text{public}\,(p)\,,\\ \exists\ m_c \in \text{public}\,(c)\ \ :\ \ \mathit{name}\,(m_c) = \mathit{name}\,(m_p)\ \text{ and } m_c \lhd m_p \qquad \bigstar\end{gathered}$$

Future work in this section: constructors, abstract classes, maybe static members, maybe limited-access subinterfaces (a generalization of "protected").

[3]It may be wise to require add the restriction $\nexists\ x, y \in \text{parents}\,(c \in \mathcal{C})\ :\ x \lhd y$. This (1) keeps the set of parents well-defined, and (2) prevents borrowings from bypassing member overrides.

1.6 Borrowings ($\mathcal{B}$)

Eidola classes do not automatically inherit members from parent classes as in most OO languages; a class must explicitly either override or *borrow* the members of its parents.[4] Special elements called *borrowings* allow this to happen.

$$\mathcal{B} \subseteq \mathcal{E}_{\mathrm{N}}$$

The element which is borrowed may be either a function (defined in §1.8), a class, or another borrowing.

$$\text{borrowed} : \mathcal{B} \to \mathcal{F} \cup \mathcal{C} \cup \mathcal{B}$$

Borrowings must be public members of classes, and can borrow only from that class's parents.

$$\forall\ b \in \mathcal{B}, \begin{cases} \text{owner}\,(b) \in \mathcal{C} & \\ b \in \text{public}\,(\text{owner}\,(b)) & \star \\ \text{owner}\,(\text{borrowed}\,(b)) \in \text{parents}\,(\text{owner}\,(b)) & \star \end{cases}$$

To make the Big Important Rule work, the specialization relation commutes over borrowing.[5]

$$\forall\ e \in \mathcal{E}, b \in \mathcal{B}, \begin{cases} e \lhd b \iff e \lhd \text{borrowed}\,(b) \\ e \rhd b \iff e \rhd \text{borrowed}\,(b) \end{cases}$$

A borrowing is a named element; it takes its name from the element which it borrows.[6]

$$\forall\ b \in \mathit{borrow}, \text{name}\,(b) = \text{name}\,(\text{borrowed}\,(b))$$

A borrowing uses itself, its owner, and the element is borrows. It has no contents.

$$\begin{aligned} \mathit{contents}\,(b \in \mathit{borrow}) &= \emptyset \\ \mathit{uses}\,(b \in \mathit{borrow}) &= \{b, \text{owner}\,(b)\,, \text{borrowed}\,(b)\} \end{aligned}$$

1.7 Variables ($\mathcal{V}$)

A *variable* is a reference to an object. (An object is an instance of a class — more on this later.) All variables are named elements.

$$\mathcal{V} \subseteq \mathcal{E}_{\mathrm{N}}$$

A variable has a *type*.[7] (In the runtime semantics, the variable can reference an object if the variable's type is a generalization of the object's type.)

$$\text{var-type} : \mathcal{V} \to \mathcal{C}$$

[4] This poses problems for deploying classes in varying runtime contexts, and more or less ruins backward compatibility for API extension. This a problem I'm ignoring for now, but we'll have to solve it before the language gets too far along.

[5] Surprisingly, this method allows a class which is a member of its own generalization to *inherit itself*. Although this seems counterintuitive, I haven't found that it leads to any semantic contradictions. The problem of whether to disallow this is currently up for discussion; see http://eidola.org/semantics/questions/memberofancestor.shtml.

[6] Should this rule be lazy?

[7] It might be spiffy to allow the type of a variable to be a function as well (and thus allow first-order functions) – or even allow any named element (though this would allow pass-by-reference, which I don't miss at all). Fascinating.

Variables have no contents.[8]

$$\mathit{contents}\,(v \in \mathcal{V}) = \emptyset$$

A variable uses the minimal set of elements possible, plus its type.

$$\mathit{uses}\,(v \in \mathcal{V}) = \{v, \mathit{owner}\,(v)\,, \text{var-type}\,(v)\}$$

Variables specialize other variables with matching types.

$$\forall\ v_1, v_2 \in \mathcal{V},\ \ v_2 \lhd v_1 \iff \text{var-type}\,(v_2) \lhd \text{var-type}\,(v_1)$$

1.8 Functions ($\mathcal{F}$)

A *function* contains an algorithm, which acts on inputs and produces outputs. Functions are named elements.

$$\mathcal{F} \subseteq \mathcal{E}_{\text{N}}$$

A function has *inputs*, *outputs*, and an *algorithm*. Its inputs and outputs are both lists of variables,[9] which allows the option of traditional pass-by-order presentation, as well as pass-by-name or intelligent combinations of the two. See §2 for a description of Stmt, the set of algorithms. A function's algorithm may simply be NULL, which signals that the function is abstract.

$$\begin{aligned} \text{inputs}, \text{outputs} &: \mathcal{F} \to \mathcal{V}^* \\ \text{algorithm} &: \mathcal{F} \to \text{Stmt} \cup \{\textsc{null}\} \end{aligned}$$

The inputs, outputs, and algorithm are the function's contents.

$$\mathit{contents}\,(f \in \mathcal{F}) = \text{inputs}\,(f) \cup \text{outputs}\,(f) \cup \{\text{algorithm}\,(f)\}$$

A function uses the minimal possible set of elements.

$$\mathit{uses}\,(f \in \mathcal{F}) = \{f, \mathit{owner}\,(f)\} \cup \mathit{contents}\,(f)$$

A variable cannot appear twice in either the input or output list. (Note, however, that a variable *can* appear once in both, signifying that the value is passed through by default.)

$$\forall\ f \in \mathcal{F}, p_i = p_j \implies i = j,\ p \in \{\text{inputs}\,(f)\,, \text{outputs}\,(f)\} \qquad \star$$

One function specializes another when its inputs and outputs specialize the other's.[10]

$$\forall\ f_1, f_2 \in \mathcal{F},\ \ f_1 \lhd f_2 \iff \begin{pmatrix} \text{inputs}\,(f_1) \lhd \text{inputs}\,(f_2) \\ \text{and outputs}\,(f_1) \lhd \text{outputs}\,(f_2) \end{pmatrix}$$

In turn, one list of variables specializes another when they have the same length, and variables in corresponding positions specialize and have matching names.

$$\forall\ d, e \in \mathcal{V}^*, d \lhd e \iff \mathit{size}\,(d) = \mathit{size}\,(e) \ \text{ and } \begin{cases} d_i \lhd e_i \\ \mathit{name}\,(d_i) = \mathit{name}\,(e_i) \end{cases}$$

Future work in this section: is-abstract function, overloading.

[8] This could change if we allow a special syntax for initializers, but this will probably not be necessary.

[9] Some worry that allowing multiple function outputs dilute the traditional OO idea of message passing, and will lead to bloated, multi-purpose functions. I'm skeptical, but it's a reasonable concern; see http://eidola.org/semantics/questions/multipleoutputs.shtml.

[10] As several readers have pointed out, this creates the potential for runtime type errors. Yes, it does, and I'm not thrilled with that. On the other hand, this covariant return type pattern crops up a lot. For pondering.

1.9 Specialization ($\lhd$)

The *specialization* relation is a reflexive, transitive relation on the set of named elements. Its notation is $a \lhd b$, read "a specializes b." Note that it is *not* symmetric.

$$\begin{gathered} \lhd \ : \mathcal{E}_\mathrm{N} \times \mathcal{E}_\mathrm{N} \to \text{boolean} \\ \forall\ a \in \mathcal{E}_\mathrm{N},\ a \lhd a \\ \forall\ a, b, c \in \mathcal{E}_\mathrm{N},\ a \lhd b \text{ and } b \lhd c \implies a \lhd c \end{gathered}$$

Specializing indicates when one element's signature is compatible with another's. If $a \lhd b$, then it is OK to use a in a context that expects something with the signature of b. What constitutes a "signature" varies between different element types. Explanations of the specializing rules for the different kinds of named elements appear in their respective sections. The full specializing relation is as follows:

$$d \lhd e \iff \begin{cases} d = e & d, e \in \mathcal{P} \\ d = e \text{ or } (\exists\ d' \in \text{parents}(d) \ : \ d' \lhd e) & d, e \in \mathcal{C} \\ \text{var-type}(d) \lhd \text{var-type}(e) & d, e \in \mathcal{V} \\ \text{inputs}(d) \lhd \text{inputs}(e) \text{ and outputs}(d) \lhd \text{outputs}(e) & d, e \in \mathcal{F} \\ \mathit{size}(d) = \mathit{size}(e) \text{ and } d_i \lhd e_i \text{ and } \mathit{name}(d_i) = \mathit{name}(e_i) & d, e \in \mathcal{V}^* \\ \text{false} & \text{otherwise.} \end{cases}$$

1.10 Access Rules *(draft)*

The *access* function limits when one element is allowed to use another. This is what creates the notions of " public" and " private" contents.

$$\mathit{access} : \mathcal{E} \to 2^{\mathcal{U}}$$

The elements of $\mathit{access}(x)$ are the roots of trees of containers which are allowed to use x — y can use x only if some element of $\mathit{access}(x)$ indirectly owns y.

$$\forall\ y \in \mathcal{E}, \forall\ x \in \mathit{uses}(y), \exists\ s_x \in \mathit{access}(x) \ : \ s_x \in \mathit{owner}^*(y) \qquad \bigstar$$

Note that (1) because an element always uses itself, it must always be able to access itself; and (2) because it is always used by its owner, it must be accessible to its owner and thus all its sibling containers. An element's access is not determined by the type of the element itself, but rather the type of its owner.

The access rules are as follows for namespaces and contents of named elements:

$$\mathit{access}(e) = \begin{cases} \mathcal{N} & e \in \mathcal{N} \text{ or } d \in \mathcal{N} \\ \mathit{access}(d) & d \in \mathcal{E}_\mathrm{C}, e \in \mathit{public}(d) \\ \{d\} & d \in \mathcal{E}_\mathrm{C}, e \in \mathit{private}(d) \\ \mathit{access}(d) & d \in \mathcal{F}, e \in \mathit{inputs}(d), \mathit{outputs}(d) \\ \{d\} & d \in \mathcal{F}, e = \mathit{algorithm}(d) \end{cases}$$
$$\text{...where } d = \mathit{owner}(e).$$

Access rules for other elements appear in the algorithmic semantics.

Astute observers will note that, under these definitions, *access* always returns single-element sets for named elements — in other words, Eidola has very simple access control. In the future, however, Eidola may support more sophisticated access constructs such as protected members.

2 Algorithmic Semantics *(draft)*

2.1 Blocks and Statements

Eidola is a procedural language — its *algorithms* are made of *statements*, which may be grouped into *blocks*. Blocks are themselves a kind of statement, and statements are, naturally, a kind of element.

$$\text{Block} \subset \text{Stmt} \subset \mathcal{E}$$

A block groups two statments together. (We are used to thinking of blocks as lists of statements, not merely pairings, and this is how notations will likely represent them. The chain of second children corresponds to the list of statements in a conventional language's block.)

$$\text{first}, \text{second} : \text{Block} \rightarrow \text{Stmt} \cup \{\textsc{null}\}$$

A block's contents are its statements; it uses the minimal set of elements.

$$\begin{aligned} \mathit{contents}\,(b \in \text{Block}) &= \text{block-stmts}\,(b) \\ \mathit{uses}\,(b \in \text{Block}) &= \{b, \mathit{owner}\,(b)\} \cup \mathit{contents}\,(b) \end{aligned}$$

Block contents are private. All other statement contents are public.[11]

$$\forall\ e \in \mathit{contents}\,(d \in \text{Stmt})\,,\ \ \mathit{access}\,(e) = \begin{cases} \{d\} & d \in \text{Block} \\ \mathit{access}\,(d) & \text{otherwise.} \end{cases}$$

2.2 Local Declarations

A *local declaration* holds a named element for use only within a certain block. Local variables should be familiar to most readers, but note that it is also possible to declare local classes and functions.

$$\begin{aligned} \text{Local-decl} &\subset \text{Stmt} \\ \text{declared} \ &: \ \text{Local-decl} \rightarrow \mathcal{E}_{\text{N}} \end{aligned}$$

Local declarations contain only the element declared.

$$\begin{aligned} \mathit{contents}\,(l \in \text{Local-decl}) &= \{\text{declared}\,(l)\} \\ \mathit{uses}\,(l \in \text{Local-decl}) &= \{l, \mathit{owner}\,(l)\,, \text{declared}\,(l)\} \end{aligned}$$

2.3 Expressions and Instances

To make sense of this section, it's helpful to glimpse ahead to the runtime semantics. At runtime, every container can have an instance (possible many), and the state of a program while it is running is a big web of such instances. Instances of classes (a.k.a. " objects") are a familiar idea, but note that *any* container can have instances — an instance of a variable, for example, is a pointer. The idea of "instances" is a very general one, and covers almost anything that will exist in memory at runtime. More information on instances appears in §3 (or will, when it exists).

An *expression* is a statement which returns an instance of a named element at runtime.

$$\text{Expr} \subset \text{Stmt}$$

[11] When we get to loops, this may change. I like C++/Java's local iterator declarations in for loops. However, that may be best handled as a notation pattern and not a semantic construct.

Every expression has a *type*. When evaluated, the expression returns an instance of this type.

$$\textit{expr-type} : \text{Expr} \to \mathcal{E}_{\text{N}}$$

Since we frequently are interested in instances of specific subsets of the named elements, there is a shorthand notation, Expr_T, for expressions of a particular type T.

$$\forall\ s \in \text{Expr}_T,\ \textit{expr-type}\,(s) \in T$$

2.3.1 Context

Some containers (such as namespaces) will always have exactly one instance, which any element could access directly. Others, however, may potentially have zero or multiple instances, and in this case, you need a *context* determine which instance you're accessing. For example, to get at a member of a class, you must have an instance of that class — class members are associated with objects.

The *context type* function governs this world of context: to get an instance of container x, you must have an instance of $\textit{context-type}\,(x)$.

$$\textit{context-type} : \mathcal{U} \to \mathcal{U}$$

In most cases, there is exactly one instance of an element for each instance of its owner; such elements need no more context than their owners do. Classes and functions, however, can be instantiated many times (or not at all), so it's not possible to simply pull an instance of one out of some context — there are special expressions (New and Func-call) which create them. The chain of context types thus crawls up the owner hierarchy, stopping at at the first function, class, or namespace it reaches.

$$\textit{context-type}\,(t) = \begin{cases} t & t \in \mathcal{N}, \mathcal{F}, \mathcal{C} \\ \textit{context-type}\,(\textit{owner}\,(t)) & \text{otherwise} \end{cases}$$

Namespaces are a special case, and are an unresolved problem here: how does Eidola resolve element accesses across physical boundaries (e.g. library calls)? Gobs of further work needed here.

In general, when an expression gives an instance of some target element t, it will require a context expression c. The context c must provide appropriate context for t, according to the relation $c \circledcirc t$ (read "c provides context for t"):

$$\circledcirc\ : \text{Expr} \times \mathcal{U} \to \text{boolean}$$

$$c \circledcirc t \iff \begin{cases} \textit{expr-type}\,(c) \lhd \textit{context-type}\,(t) & t \in \mathcal{C} \\ \textit{expr-type}\,(c) = \textit{context-type}\,(t) & \text{otherwise} \end{cases}$$

Note that Eidola allows polymorphism: when the necessary context is a class, the context expression can provide an instance of a specialization.

2.3.2 References

A *reference* is an expression which extracts an instance of a named element from an appropriate context.

$$\begin{aligned} \text{Ref} &\subset \text{Expr} \\ \text{target} &: \text{Ref} \to \mathcal{E}_{\text{N}} \\ \text{context} &: \text{Ref} \to \text{Expr} \end{aligned}$$

Since the reference returns an instance of the target element, that element is the expression's *type*.

$$\textit{expr-type}\,(r \in \text{Ref}) = \text{target}\,(r)$$

The context expression must provide the right context for the target.

$$\forall\ r \in \text{Ref},\ \text{context}\,(r) \circledcirc \text{target}\,(r) \quad \bigstar$$

Need expr for accessing current context instances ("this"). Beware classes inside functions — this is a special case!

2.3.3 This

Every element has access to an instance of each of its indirect owners.[12] Object-oriented programmers are familiar with the special variable *this*, which refers to the instance of the class in which the variable lives. Eidola extends this concept to elements of all kinds: an expression accesses members of a class by saying "this object", and accesses parameters of a function by saying "this function invocation".

"This" expressions are similar to references, but need no context expression.

$$\begin{aligned} \text{This} &\subset \text{Expr} \\ \text{target}\ &:\ \text{This} \to \mathcal{E}_\text{N} \end{aligned}$$

The context comes from the chain of ownership.

$$\forall\ t \in \text{This},\ \text{context}\,(t) \in \textit{owner}^*(t) \quad \bigstar$$

Like references, the *type* of a "this" expression is simply its target.

$$\textit{expr-type}\,(t \in \text{This}) = \text{target}\,(t)$$

2.3.4 Variable Values

A *variable value* expression extracts a variable's runtime value from the variable itself. All variables in Eidola have the semantic behavior of references, so this expression is anaologous to a pointer dereference in C.

$$\begin{aligned} \text{Var-val} &\subset \text{Expr} \\ \text{var-expr}\ &:\ \text{Var-val} \to \text{Expr}_\mathcal{V} \end{aligned}$$

A variable's type describes its runtime value. The type of a variable value expression is, therefore, simply the type of the variable itself.

$$\textit{expr-type}\,(v \in \text{Var-val}) = \text{var-type}\,(\textit{expr-type}\,(\text{var-expr}\,(v)))$$

2.3.5 Assignments

$$\begin{gathered} \text{Assign} \subset \text{Expr} \\ \text{left} : \text{Assign} \to \text{Expr}_\mathcal{V} \\ \text{right} : \text{Assign} \to \text{Expr}_\mathcal{C} \\ \textit{expr-type}\,(a \in \text{Assign}) = \text{var-type}\,(\textit{expr-type}\,(\text{left}\,(a))) \\ \forall\ a \in \text{Assign},\ \textit{expr-type}\,(a) \rhd \textit{expr-type}\,(\text{right}\,(a)) \quad \bigstar \end{gathered}$$

[12]In the absence of further restrictions, this formulation allows a local class's members to access the local variables of the function in which it is declared. In that circumstances, stack frames would actually have to be garbage collected like objects. Weird? Oh so very. Let's let it play out and see where it breaks.

2.3.6 New Objects

$$
\begin{gathered}
\text{New} \subset \text{Expr} \\
\text{context} : \text{New} \to \text{Expr} \\
\text{target} : \text{New} \to \mathcal{C} \\
\textit{expr-type}\,(n \in \text{New}) = \text{class}\,(n) \\
\forall\ n \in \text{New},\ \text{context}\,(n) \circledcirc \text{owner}\,(\text{target}\,(n)) \qquad \bigstar
\end{gathered}
$$

2.3.7 Function Calls

$$\text{Func-call} \subset ?????$$

This section will present three sketches of three alternative ways of handling things, based on different interpretations of a function's outputs. The first alternative uses traditional single-output functions. The second creates a special class of expression which handles multiple-output functions. The third establishes the List class as a language primitive and uses that to handle multiple-output functions (actually, I may scrap this one — it's going to get ugly).

Sketch 1: Single-output functions

$$
\begin{aligned}
\text{inputs} &: \mathcal{F} \to \mathcal{V}^* \\
\text{output} &: \mathcal{F} \to \{\mathcal{C} \cup \{\emptyset\}\} \\
\text{algorithm} &: \mathcal{F} \to \text{Stmt} \cup \{\emptyset\} \\
\text{Func-call} &\subset \text{Expr}
\end{aligned}
$$

Sketch 2: Multiple outputs through special expressions

$$
\begin{aligned}
\text{inputs}, \text{outputs} &: \mathcal{F} \to \mathcal{V}^* \\
\text{algorithm} &: \mathcal{F} \to \text{Stmt} \cup \{\emptyset\} \\
\text{Func-call} &\not\subset \text{Expr}
\end{aligned}
$$

Sketch 3: Multiple outputs through List class primitive

$$
\begin{aligned}
\text{inputs}, \text{outputs} &: \mathcal{F} \to \mathcal{V}^* \\
\text{algorithm} &: \mathcal{F} \to \text{Stmt} \cup \{\emptyset\} \\
\text{Func-call} &\subset \text{Expr}_{\{\texttt{List}\}}
\end{aligned}
$$

2.4 Branching and Looping

3 Runtime Semantics

This section will describe the execution of an Eidola program. The extent to which this is actually formalizable remains to be seen. At the very least, this section should include descriptions of runtime typing and access, class instantiation, and private member inheritance.

It's possible that we'll be able to also get a description of the state of a program, composed of the set of all instantiated objects and the states of their member variables, and then describe the effects of different language constructs as a set of functions from a state onto a new state. It probably won't be possible to produce a complete model of multithreading, however.

A Notation Conventions

Notations for sets, functions, and logic are the standard mathematical variety.

Fundamental sets and function appear in calligraphic capitals (e.g. $\mathcal{E}_\mathrm{N}$) or roman type (e.g. name). Derived sets and functions appear in italics (e.g. *contents*).

Lazy rules are marked with $\star$.

Lists use the following notation:

$[a, b, \ldots]$	An ordered list containing a, then b, ...
S^*	The set of ordered lists of zero or more items from S
S^+	The set of ordered lists of one or more items from S
$\mathit{size}(l)$	The number of elements in the list l
l_i	The ith element of the list l for $1 \leq i \leq \mathit{size}(l)$
$l + x$	The list l with x appended

B What's Missing

The following language constructs will be part of the semantics, but aren't yet:

- null
- super
- return, break, continue
- type casts
- constructors
- overloading
- abstract functions and classes
- generic types
- protected (possibly in a generalized form)

These constructs may be handled through standard libraries with well-formalize contracts, instead of though language primitives, though a shift to primitive types is still possible:

- boolean logic
- numerical constants and arithmetic
- strings
- arrays, lists, etc.
- threads
- reflection

These constructs could be part of the language as notational features, but will likely not be part of the semantics:

- for, do, switch
- operator overloading

If you don't see it in one of these lists, I have either forgotten about it, ruled it out, or simply chosen to the whole question for a while. The last is the case with all things runtime.

Index