Definitions

For the purpose of this definition, consider an Oso Cloud environment $E = \{\Pi ; \Delta\}$ to be a persistent object comprising a policy $\Pi$ and a facts DB $\Delta$.

We consider the behavior of an environment in terms of how it responds to requests to its query endpoint. A query of expression $\pi$ with context facts $\delta$ against environment $E$ which responds with results $\rho$ is denoted as $E \vdash [\pi ; \delta] \Rightarrow \rho$.

Two environments $E$ and $F$ which exhibit the same query input/output behavior are said to be extensionally equivalent. Precisely:

$$\forall \delta, \pi. (E \vdash [\pi ; \delta] \Rightarrow \rho \land F \vdash [\pi ; \delta] \Rightarrow \rho) \quad E \simeq F$$

Extensional equivalence is a rather strict notion. Two Oso environments may exhibit effectively the same logic without being extensionally equivalent to one another; for example, two environments with identical policies except that they use different names for their roles and permissions. Relatedly, one environment may implement the exact same logic as another but also implement some additional logic; for example, the same policy but with an added rule.

To have a more lenient notion of similarity between environments we define what it means for an environment $E$ to reduce to an environment $\{\Pi ; \Delta\}$, denoted $E \leq \{\Pi ; \Delta\}$. If there exists a mapping $\mu$ of query inputs/outputs and a set of facts $\Delta'$ such that such that $\{\Pi ; \Delta \cup \Delta'\} \cdot \mu \simeq E$ then $E \leq F$. Precisely,

$$\frac{\{\Pi ; \Delta \cup \Delta'\} \cdot \mu \simeq E}{E \leq \{\Pi ; \Delta\}}$$

Two environments whose policies/facts are identical except for differing role and permission names would reduce to one another, witnessed by a mapping $\mu$ which rewrites these role and permission names. Relatedly, if two environments are the same except that one’s policy is missing a few rules that the other’s has, then that environment reduces to the other as witnessed by a no-op mapping. An environment with a policy that hardcodes role and permission implications reduces to an environment that implements custom roles, where $\Delta'$ is a set of grants_permission facts.

A policy $\Pi$ is universal if all environments reduce to $\{\Pi ; \emptyset\}$. 
A Universal Policy

This section presents an almost universal policy. The full policy may be found in the appendix. Note that our current implementation of Polar has shortcomings that preclude this particular version of the policy. The approach taken in designing this policy was to ensure supporting the entirety of Polar by structuring the policy in a way that's very similar to the structure of Polar's syntax.

Polar Fragment

This section presents a universal policy which can directly express the following fragment of Polar:

\[
\begin{align*}
    p &: (x_0, x_1, x_2) \text{ if } e \\
    e &: := \\
    & \quad p(x_i, x_j, x_k) \quad \text{call with variables} \\
    & \quad e_1 \land e_2 \quad \text{conjunction} \\
    & \quad x = v \quad \text{unify with literal}
\end{align*}
\]

Only rules and calls of arity three are allowed, and all parameters/arguments to rules/calls are variables. The most notable exclusion is matches. I conjecture that all Polar policies can be reduced to this fragment.

Encoding

This universal policy is designed to evaluate rules of the above fragment which are supplied as facts. The entry point to querying this policy is the query predicate, which is implemented in terms of the fact, rule, and eval predicates:

\[
\begin{align*}
    \text{query}(\text{predicate}: \text{String}, a: \text{Value}, b: \text{Value}, c: \text{Value}) \text{ if } \\
    & \quad \text{fact}(\text{predicate}, a, b, c); \\
    \text{query}(\text{predicate}: \text{String}, a: \text{Value}, b: \text{Value}, c: \text{Value}) \text{ if } \\
    & \quad \text{body matches Expr} \text{ and } \\
    & \quad \text{rule}(\text{predicate}, \text{body}) \text{ and } \\
    & \quad d \text{ matches Value} \text{ and } e \text{ matches Value} \text{ and } \\
    & \quad \text{eval}(\text{body}, a, b, c, d, e);
\end{align*}
\]

To query with predicate \(p\) and arguments \(v_1\), \(v_2\), and \(v_3\), you would query the universal policy with the expression query\((p, v_1, v_2, v_3)\). Similarly, a fact with predicate \(p\) and arguments \(v_1\), \(v_2\), and \(v_3\) is stored as fact\((p, v_1, v_2, v_3)\).

A rule with predicate \(p\) and body expression \(e\) has arguments \(x_0\), \(x_1\), and \(x_2\) and is stored as rule\((p, [e])\), where \([e]\) is a value with type Expr and an arbitrary identifier from "compiling" the body expression of the rule into facts. Expressions are represented as facts by effectively encoding each node of the expression’s syntax tree as a fact. The precise process of compiling an expression to facts is as follows:

\[
\begin{align*}
    \text{e\_call}([p(x_i, x_j, x_k)], \text{Var}\{i\}, \text{Var}\{j\}, \text{Var}\{k\}) \\
    \text{e\_and}([e_1 \land e_2], [e_1], [e_2]) \\
    \text{e\_unify}([x_i = v], \text{Var}\{i\}, v)
\end{align*}
\]
The following sections go into how each of these is implemented in the policy.

**Calls**

The policy for processing calls is:

```plaintext
resolve_var(Var{"0"}, a, _, _, _, _, out) if a = out;
resolve_var(Var{"1"}, _, b, _, _, _, out) if b = out;
resolve_var(Var{"2"}, _, _, c, _, _, out) if c = out;
resolve_var(Var{"3"}, _, _, _, d, _, out) if d = out;
resolve_var(Var{"4"}, _, _, _, _, e, out) if e = out;
```

```plaintext
eval(expr: Expr, a: Value, b: Value, c: Value, d: Value, e: Value) if
  predicate matches String and
  arg1 matches Var and arg2 matches Var and arg3 matches Var and
  e_call(expr, predicate, arg1, arg2, arg3) and
  out1 matches Value and out2 matches Value and out3 matches Value and
  resolve_var(arg1, a, b, c, d, e, out1) and
  resolve_var(arg2, a, b, c, d, e, out2) and
  resolve_var(arg3, a, b, c, d, e, out3) and
  query(predicate, out1, out2, out3);
```

Each rule in this formulation can use at most five\(^1\) unique variables, which are passed into every `eval` call as `a`, `b`, `c`, `d`, and `e`. In the above `eval` rule, the `resolve_var` helper rule is used to unify the correct local variables into the correct call arguments.

**Conjunction**

The policy for processing conjunctions is:

```plaintext
eval(expr: Expr, a: Value, b: Value, c: Value, d: Value, e: Value) if
  left matches Expr and right matches Expr and
  e_and(expr, left, right) and
  eval(left, a, b, c, d, e) and
  eval(right, a, b, c, d, e);
```

It simply passes the evaluation along to Polar's built-in conjunction.

**Literal Unification**

The policy for processing unifications between literals and variables is:

```plaintext
eval(expr: Expr, a: Value, b: Value, c: Value, d: Value, e: Value) if
  var matches Var and value matches Value and
  e_unify(expr, var, value) and
  out matches Value and
  resolve_var(var, a, b, c, d, e, out) and
  out = value;
```

\(^1\)This is an arbitrary limit unrelated to Oso Cloud’s 5-arity limit on facts; it can be made arbitrarily high.
It uses the `resolve_var` helper rule to unify with the specified variable, and then passes the unification along to Polar’s built-in unification.

**Examples**

The rule

```plaintext
foo(x, y, z) if
   bar(z, x, y) and
   baz(y, z, x);
```

is expressed in this universal policy as:

```plaintext
test "foo if bar and baz" {
    setup {
        rule("foo", Expr"0");
        e_and(Expr"0", Expr"1", Expr"2");
            e_call(Expr"1", "bar", Var"2", Var"0", Var"1");
            e_call(Expr"2", "baz", Var"1", Var"2", Var"0");
    }
    fact("bar", Value"C", Value"A", Value"B");
    fact("baz", Value"B", Value"C", Value"A");
    assert query("foo", Value"A", Value"B", Value"C");
    assert_not query("foo", Value"B", Value"B", Value"B");
}
```

The rule

```plaintext
has_permission(actor, "read", resource) if
    has_role(actor, "reader", resource);
```

is expressed in this universal policy as:

```plaintext
test "foo if bar and baz" {
    setup {
        rule("has_permission", Expr"0");
        e_and(Expr"0", Expr"1", Expr"2");
            e_unify(Expr"1", Var"1", Value"read");
            e_and(Expr"2", Expr"3", Expr"4");
                e_unify(Expr"3", Var"3", Value"reader");
                e_call(Expr"4", "has_role", Var"0", Var"3", Var"2");
    }
    fact("has_role", Value"alice", Value"reader", Value"anvils");
    assert query("has_permission", Value"alice", Value"read", Value"anvils");
    assert_not query("has_permission", Value"bob", Value"read", Value"anvils");
}
```
Appendix

resource Expr {}
resource Value {}
resource Var {}

query(predicate: String, a: Value, b: Value, c: Value) if
  fact(predicate, a, b, c);

query(predicate: String, a: Value, b: Value, c: Value) if
  body matches Expr and
  rule(predicate, body) and
  d matches Value and e matches Value and
  eval(body, a, b, c, d, e);

# Call expressions

resolve_var(Var{"0"}, a, _, _, _, _, out) if a = out;
resolve_var(Var{"1"}, _, b, _, _, _, out) if b = out;
resolve_var(Var{"2"}, _, c, _, _, _, out) if c = out;
resolve_var(Var{"3"}, _, _, d, _, _, out) if d = out;
resolve_var(Var{"4"}, _, _, _, e, out) if e = out;

eval(expr: Expr, a: Value, b: Value, c: Value, d: Value, e: Value) if
  predicate matches String and
  arg1 matches Var and arg2 matches Var and arg3 matches Var and
  e_call(expr, predicate, arg1, arg2, arg3) and
  out1 matches Value and out2 matches Value and out3 matches Value and
  resolve_var(arg1, a, b, c, d, e, out1) and
  resolve_var(arg2, a, b, c, d, e, out2) and
  resolve_var(arg3, a, b, c, d, e, out3) and
  query(predicate, out1, out2, out3);

# Unify literal expressions

eval(expr: Expr, a: Value, b: Value, c: Value, d: Value, e: Value) if
  var matches Var and value matches Value and
  e_unify(expr, var, value) and
  out matches Value and
  resolve_var(var, a, b, c, d, e, out) and
  out = value;

# And expressions

eval(expr: Expr, a: Value, b: Value, c: Value, d: Value, e: Value) if
  left matches Expr and right matches Expr and
  e_and(expr, left, right) and
  eval(left, a, b, c, d, e) and
  eval(right, a, b, c, d, e);