Cedar Language Spec v2.0

This document outlines the specification and architecture for the Cedar policy language. We describe the main design goals; the syntax and semantics of the policy language; the goals and interface for the policy validator; and some implementation notes.

Overview

Cedar is a language for writing authorization policies, together with an engine for evaluating those policies to make authorization decisions. An authorization policy governs which **principal** in a system can perform what **actions** on a given **resource** in a given **context**. For example, a policy might state that the employees of a company can access their own HR records only during regular business hours. Cedar is a new language for writing such policies, and includes a standalone engine for answering authorization queries such as "Can the principal *P* perform the action *A* on the resource *R* in the context X?" These queries are answered with respect to a backend that stores the data about the principals, actions, and resources used in a given application.

Language and evaluator

EXAMPLE AUTHORIZATION MODEL FOR A PHOTO SHARING SERVICE

To illustrate the core concepts of Cedar, consider building the authorization model for a hypothetical photo sharing application, Photoflash (Figure 1: Photoflash). The application provides users with the ability to store, organize, and share their photos. Users can upload photos to their Photoflash account and organize them into albums. This organization system is flexible: albums can be nested in (multiple) other albums and photos can belong to multiple albums. Users can also tag their photos with custom tags so that they can search for them later, in addition to searching for them based on the photo metadata (e.g., creation time).

Most interesting from an authorization point of view: Photoflash users can share their photos and albums with other Photoflash users or user groups. For example, a user Jane can create groups such as "family", "friends", or "coworkers", populate these groups with other Photoflash users, and specify (through a UI) how to share her photos and albums with these groups. User groups are as flexible as albums: they can be nested, and a given user can belong to multiple groups. Based on group membership, Jane can allow specific users to perform specific actions on her Photoflash resources; e.g., Jane's friends can view and comment on every photo in her "trips" album. Any action that is not explicitly permitted is denied. But some actions, even if explicitly permitted by a user, are always denied as a matter of service-wide security or design constraints. For example, regardless of how Jane sets up her permissions, nobody except Jane is allowed to view resources in her account that are tagged as "private". Cedar can capture all of these authorization constraints.

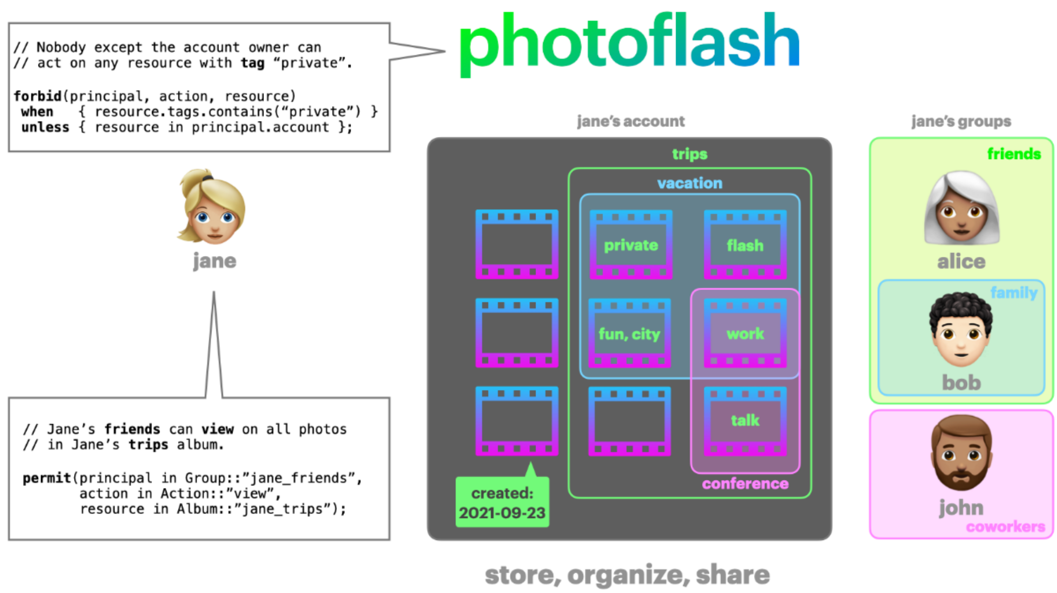

Figure 1: Photoflash

DATA MODEL, VALUES, AND OPERATIONS

Policies are written in the Cedar language, and policies are evaluated by the Cedar evaluator. Viewing a policy as a kind of program, the evaluator "runs" the policy to determine its impact on an authorization request. That is, evaluation takes place as part of *authorization*: the Cedar authorizer evaluates each relevant policy and combines the evaluation results to make the final decision.

Entities

The Cedar language is based on a data model that organizes **entities** into **hierarchies**. Entities serve as *principals*, *resources*, and *actions* in Cedar policies. An entity is a stored object with an *entity type*, an *entity identifier* (EID), zero or more *attributes* mapped to values, and zero or more *parent* entities. The entity type and entity ID are together referred to as the *entity unique ID* (UID), which uniquely identifies a stored object: no two objects have both the same entity type and same EID. The parent relation on entities forms a directed acyclic graph (DAG), which we call the *entity hierarchy*.

For example, Photoflash (Example authorization model for a photo sharing service) might store its entities in a custom graph database, partially visualized in Figure 2: Photoflash store. The entities in this store include the application's users (`jane`, `alice`, `bob`, and `john`), user groups (`jane_friends`, `jane_family`, and `jane_coworkers`), photos, albums (`jane_trips`, `jane_vacation`, `jane_conference`), user accounts, and operations (`view`, `comment`). Each of these entities has an entity type (`User`, `UserGroup`, `Photo`, `Album`, `Account`, `Action`, etc.), an EID, and, optionally, attributes and parents. The visualized part of the entity hierarchy reflects the membership relation between users and user groups. So, the user `bob` is a member of the group `jane_family` and, transitively, `jane_friends`. We can also see that every `User` has an `account` attribute, which stores a reference to the user's `Account` entity, identified by its type and EID.

Figure 2: Photoflash store

Entities are the only reference values in Cedar. Cedar supports three operations on entities: *equality, reachability,* and *attribute retrieval.* In particular, if `A`, `B`, and `C` are UIDs, then we express equality as `A == B`, reachability as `A in B` or `A in [B, C, ...]` and attribute retrieval *as* `A.f` where `f` is an attribute.

- Equality `A == B` holds if and only if `A` and `B` are literally the same UID. So, `User::"alice" == User::"alice"` holds but `User::"john" == User::"alice"` does not.
  - It makes no difference whether different UIDs happen to refer to objects having the same attributes; they are still deemed unequal.
  - Equality holds even when a UID does not actually reference an entity object in the entity hierarchy. Thus, `User::"alice" == User::"alice"` holds regardless of whether `User::"alice"` is a "dangling reference" or not.
- Reachability `A in B` holds if and only if either `A` is equal to `B`, or `A` is a descendant of `B` in the entity hierarchy. For example, in the Photoflash hierarchy (Figure 2: Photoflash store), `bob in jane_friends` is true but `john in jane_friends` is not. The expression `A in [B,C,...]` is equivalent to `A in B || A in C || ...` but will yield an error if any element in the set is not an entity reference. `A in B` will return `false` if `A` and/or `B` do not exist in the entity hierarchy, except for the special case of `A in A` which returns `true` even if `A` does not exist.
- Finally, policies can retrieve the value of an entity's attribute using the dot operator; e.g., `A.account` retrieves the entity value that references the account object that belongs to the entity `A`. If `A` does not reference an entity in the hierarchy, or the referenced entity does not have an `account` attribute, an error is raised. Entity attributes can also be accessed with the `[]` operator, which takes a string literal representation of the attribute; so, `A.account` is equivalent to `A["account"]`.

Entities may be referenced using namespaces. For example, `Photoflash::Groups::Album::"vacation"` is a UID where the entity type is `Photoflash::Groups::Album`; which is to say that it is the type `Album` defined in namespace `Photoflash::Groups`.

Primitive and aggregate values

In addition to entities, Cedar's data model includes *primitive* and *aggregate values*. Such values can be stored in entities' attributes, and may appear in context information provided with an authorization request.

Primitive values include:

- Booleans `true` and `false`
- 64-bit signed integers (examples: `-8` or `772389`)
- strings (examples: `"hello world"` or `"q7+/%"`)
- IP addresses and ranges, both IPv4 and IPv6 (examples: `ip("222.222.222.222")`, `ip("::1")`, `ip("222.222.222.0/24")`)
- Decimal values with at most 4 digits after the decimal point (examples: `decimal("0.1")`, `decimal("123.4567")`)

Aggregate values include:

- Sets. Sets may be heterogeneous (the same set may contain elements of different dynamic types) and can be constructed with `[]` syntax (examples: `[2, 4, "hello"]`, `[-1]`, `[]`, `[3<5, ["nested", "set"], true]`).
- Anonymous records. Record attributes are valid identifiers or (arbitrary) strings, and values may be heterogeneous (the same record may contain values of different dynamic types). Records can be constructed with `{}` syntax (examples: `{ "key": "value" }`, `{id:"value"}`, `{"key": "value", id:"value" }`, `{}`, `{ "foo": 2, bar: [3, 4, -47], ham: "eggs", "hello": true }`). Record attribute values are accessed just as entity attributes are, using the `[]` indexing operator and a string literal, e.g., `record["key"]`, or the `.` operator, e.g., `record.foo`. Nested record access is as expected, e.g., `context.some.nested.attribute` or `context["some"].nested["attribute"]`.

Cedar values (that is, entities, primitive values, and aggregate values) are compared for equality in the usual way. Two sets `s1` and `s2` are equal, `==`, if they contain exactly the same elements, regardless of order. Two records are equal if they consist of the same set of key-value pairs. Values of different types are never equal — in particular, an entity is never equal to a record (even if the record happens to contain the same keys/values as the entity's attributes).

In addition to equality, Cedar values can be used with the small set of operators and functions listed in Appendix A: Cedar operators and functions. These include relational operators, and operations on strings, sets, and records. The `&&` and `||` operators perform short-circuiting. I.e., `false && ...` will evaluate to `false` without evaluating `...` and likewise `true || ...` will evaluate to `true` without doing so. This is true even when the `...` has type errors, e.g., `true || "a" < 3` evaluates to `true`.

TYPES

Cedar is dynamically typed, like Javascript or Python. As mentioned above, this means you can write expressions like `[3, 4, -47] == "hello"` and Cedar will happily accept them (here, evaluating to `false`). Similarly to many other dynamically typed languages, Cedar is *type safe*: the type of every value is known at runtime, and the operators and functions check that their arguments have the expected types, resulting in runtime errors if those expectations are violated.

POLICY SYNTAX

Cedar polices are written using the grammar in Appendix B: Policy grammar. A policy consists of three elements:

1. The *effect* the policy has on authorization, which is either `permit` or `forbid` (nonterminal `Effect` in the grammar)
2. The *scope*, which constrains which principal, action, and resource the policy applies to (nonterminal `Scope` in the grammar)
3. The *conditions*, which further refine the circumstances under which the policy applies (non-terminal `Conds` in the grammar).

Roughly speaking, the scope describes a *role-based access control* (RBAC)-style policy, while the conditions refine it to express an *attribute-based access control* (ABAC) policy. The effect and scope are mandatory, but the conditions are optional.

Static policies

Cedar supports two kinds of policies: *static policies* and *policy templates.* In what follows we focus on static policies, but all that we say applies to policy templates as well; the thing that distinguishes a static policy from a policy template is the latter's use of `?principal` and/or `?resource` as "slots" in the scope. We explain the purpose of these slots below.

Example 1

The following RBAC static policy for Photoflash permits Jane's friends to view or comment on all photos that are transitively contained in her `trips` album (i.e., in the album or any nested sub-albums):

```
// "Policy c1"
permit(principal in Group::"jane_friends",
       action in [Action::"view", Action::"comment"],
       resource in Album::"jane_trips");
```

The following ABAC static policy forbids any user other than the owner of a Photoflash account from performing any action on resources tagged as "private":

```
// "Policy c2"
forbid(principal, action, resource)
  when { resource.tags.contains("private") }  // assuming resource.tags is a set c
  unless { resource in principal.account };  // assuming the principal has an "acc
```

Every policy begins with either the `permit` or the `forbid` keyword. A `permit` policy grants access, while a `forbid` policy restricts access by overriding a `permit` policy. Example 1 shows both kinds of policies.

Next, each policy contains the keyword variables `principal`, `action`, and `resource`, possibly including constraints. The constraints determine which principals, actions, and resources the policy applies to, according to the underlying entity hierarchy. The hierarchy constraints for `principal` and `resource` take one of two forms: `var` or `var ('in' | '==') Entity` (an additional form, for policy templates, is discussed below). The `action` constraint can take either of those forms, or a third form `var in [Entity, Entity, ...]`. These are RBAC-style constraints; in Example 1, the policy c_1 uses these RBAC-style constraints, while c_2 uses ABAC (via `when` and `unless` clauses) to express constraints on which principals and resources the policy applies to.

An RBAC-style equality constraint, `var == Entity`, says that the policy applies only when `var` is equal to `Entity` (meaning that the policy applies only to one specific entity, `Entity`). An RBAC-style membership constraint, `var in Entity`, says that the policy applies only when `var` is a descendant of `Entity` in the entity hierarchy. (`in` is reflexive, so any entity is implicitly a descendant of itself.) For example, the constraint `resource in Album::"jane_trips"` in the policy c_1 means that the policy applies only to resources that are transitively contained in Jane's "trips" album, including the album itself. Finally, the RBAC-style set form, `var in [Entity, Entity, ...]` (which is only allowed for `action`) says that `var` is either equal to or descendant of one (or more) of the entities specified in the set.

If you just write `var`, that imposes no constraints. For instance, in policy c_2, `forbid(principal, action, resource)` imposes no constraints on the principal, action, or resource, and thus the policy applies to all principals, actions, and resources in the system, subject to any `when` and `unless` clauses, if present.

Conditions start with `when` or `unless`, and are boolean expressions. The policy only applies if all `when` clauses evaluate to `true` and all `unless` clauses evaluate to `false`. Conditions are written in the *Cedar expression language*, defined by the non-terminal `Expr` in the grammar. We can think of the constraints on `principal`, `action`, and `resource` in the scope as expressions, too, as they are also described by the `Expr` grammar.

The Cedar expression language is a simple language whose syntax resembles that of JavaScript, Java, and C, while taking additional inspiration from Rust in order to minimize ambiguities in the grammar. The language has some desirable properties: Cedar expressions have no side effects; expression (and policy) evaluation is guaranteed to terminate; and we can bound the worst-case running time of each policy to be quadratic in policy and input size, but usually linear. Cedar has the familiar relational and logical binary operators (e.g., `x < 5` and `!(x && y)`). Expressions may contain conditionals `if E1 then E2 else E3` (like `E1 ? E2 : E3` in C). Expressions may also contain `in` expressions like `A in B` or `A in [B, C, ...]`, as discussed earlier (see Entities are the only reference values in Cedar. Cedar supports three operations on entities: equality, reachability, and attribute retrieval. In…). The language's data types are discussed more thoroughly in Data model, values, and operations, while a full list of operators and functions is provided in Appendix A: Cedar operators and functions.

Given a policy string, Cedar parses it to produce an *abstract policy tuple*, *c* = `<Effect, Principal, Action, Resource, Conds>`. The elements of a policy tuple correspond to the grammar productions in the obvious way. We define the following functions on policy tuples *c*:

- *Effect(c): {Allow, Deny}*;
- *Principal(c), Action(c), Resource(c): Expr*;
- *Conds(c): List<Expr>*

The function *Effect(c)* returns the value *Allow* for `permit` policies and *Deny* for `forbid` policies. The function *Conds(c)* returns the list of *Expr* clauses for *c*, which may be empty. These clauses come from `when` or `unless` clauses in the policy; `when` clauses are individual expressions, and `unless` clauses are *negated* expressions. The *Principal(c), Action(c), and Resource(c)* functions produce the expanded constraint expressions on the input variables `principal`, `action`, and `resource`; if there is no constraint expression, they simply produce `true`.

Example 1, parsed
For policy *c1* given in Example 1, above, parsing yields:

- *Effect*(*c1*) = *Allow*
- *Principal(c1)* = `principal in Group::"jane_friends"`
- *Action(c1)* = `action in [Action::"view", Action::"comment"]`
- *Resource(c1)* = `resource in Album::"jane_trips"`
- *Conds(c1)* = []

For policy *c2* given above, parsing yields:

- *Effect*(*c2*) = *Deny*
- *Principal(c2)* = `true`
- *Action(c2)* = `true`
- *Resource(c2)* = `true`
- *Conds(c2)* = [`resource.tags.contains("private")`, `!(resource in principal.account)`]

Notice how *Conds*(*c1*) is empty, since *c1* has no `when` or `unless` clauses, but *Conds(c2)* is a list of two, where *c2*'s `when` clause appears unchanged, and its `unless` clause is negated.

Policy templates

Policy templates are similar to prepared statements in SQL. They allow for creating policies programmatically in a safe and convenient way. A policy template has one or more *slot*s. There are currently two permitted slots, `?principal` and `?resource`. A slot may only appear in the policy scope constraint for its variable, and may only appear on the right-hand side of `==` or `in`.

Example 2

The following policy is a template with slots for both `?principal` and `?resource`.

```
permit(
  principal == ?principal,
  action in [Action::"view", Action::"comment"],
  resource in ?resource
)
unless {
  resource.tag =="private"
};
```

A template cannot be evaluated as part of an authorization request directly. It must first be *linked* by providing Entity UIDs as arguments for the slots. The number of arguments must match the number of slots in the policy template.

Example 2, linked
Linking the above policy template with `[{ "Principal" :"User::\"bob\"", "Resource":"Photo::\"trip\""},{"Principal" :"User::\"cat\"", "Resource":"Doc::\"sales\""}]` will yield *template-linked policies* equivalent to the following two static policies.

```
permit(
  principal == User::"bob",
  action in [Action::"view", Action::"comment"],
  resource in Photo::"trip"
)
unless{
  resource.tag == "private"
};
permit(
  principal == User::"cat",
  action in [Action::"view", Action::"comment"],
  resource in Doc::"sales"
)
unless{
  resource.tag == "private"
};
```

The Cedar *policy set* is the set of static policies together with the set of template-linked policies.

POLICY SEMANTICS

Static policies and template-linked policies have the same semantics. A policy *c* may refer to a static policy or a template-linked onef. An *authorization request* is defined as the tuple *<P, A, R, X>* where *P* is a principal, *A* is an action, *R* is a resource, and *X* is the context. *P*, *A*, and *R* are entity UIDs, while *X* is a record. (See Data model, values, and operations.) Cedar's authorizer grants the request — that principal *P* is allowed to perform the action *A* on the resource *R* in circumstances described by the context *X* — if that request is *satisfied* by the *authorization relation* for a given application, defined by that application's policy set. The authorization relation authorizes the request *<P, A, R, X>* if and only if it *satisfies* at least one permission (`permit`) policy and no restriction (`forbid`) policies. We define what it means for a request to satisfy a policy as follows.

A request *<P, A, R, X>* satisfies a policy *c* when evaluating *c* on the request produces the value *true*. More precisely, every policy *c* denotes a function $[\![c]\!]$ from entity hierarchies *H* and queries *<P, A, R, X>* to booleans. We say that *<P, A, R, X>* ***satisfies*** *c* with respect to the hierarchy *H* when $[\![c]\!]_H$ *(<P, A, R, X>)* is *true.*

We define the function $[\![c]\!]$ by evaluating the policy *c* with respect to *H* and the request *<P, A, R, X>*; the variables `principal`, `action`, `resource`, and `context` that appear in *c* are bound to the values *P*, *A*, *R*, and X, respectively. The result of the evaluation is *true* if *Principal(c), Action(c),* and *Resource(c)* all evaluate to *true*; every `when` expression in *Conds(c)* evaluates to *true*; and every `unless` expression in *Conds(c)* evaluates to *false*. Cedar policies are total functions, which means that they return *true* or *false* for every input. In particular, a policy returns *false* if its evaluation would error under the standard expression semantics, e.g., because the policy attempts to access an attribute that does not exist for a given entity.

> Another way to view evaluation of a policy *c* is that from *c* we can construct the Cedar *expression e* which has the form *Principal(c)* `&&` *Action(c)* `&&` *Resource(c)* `&&` *{x | x in Conds(c) }*. Then we evaluate this expression *e* for a particular request *<P,A,R,X>* and hierarchy *H*, resulting in either `true` or `false` . For example,
>
> - policy *c1* in Example 1 corresponds to the expression `principal in Group::"jane_friends" && action in [Action::"view", Action::"comment"] && resource in Album::"jane_trips"`. (There are no conditions in this policy.)
> - policy *c2* in Example 1 corresponds to the expression `true && true && true && resource.tags.contains("private") && !(resource in principal.account)` . (There are no scope constraints in this policy, so each is represented by `true` in the expression form.)

In addition to computing an authorization decision (*Allow* or *Deny*), an implementation of Cedar must also compute the reasons that accompany the decision. Specifically, the authorization output is a triple $\langle dec, reason, error \rangle$, consisting of a decision $dec \in \{Allow, Deny\}$, a set of reasons, and a set of errors. We consider the output $\langle dec, reason, error \rangle$ to be correct if it satisfies Definition 1. Authorization semantics. That is, if *dec* is *Allow* then *reason* consists of the policy IDs for all satisfied permissions. Otherwise, *dec* must be *Deny*, and *reason* consists of the policy IDs for all satisfied restrictions. *error* consists of the evaluation error messages. This semantics is deterministic: it is a function of *P*, *A*, *R*, *X*, the entity hierarchy *H*, and the application's policies C.

Definition 1. Authorization semantics

Let C be the set of an application's policies, including all the static policies and template-linked policies, and *H* its entity hierarchy. Let *I* = *<P, A, R, X>* be an authorization request and define two sets, $C_I^- \subseteq C$ and $C_I^+ \subseteq C$, with respect to *I* as follows. The set C_I^- consists of all restriction (`forbid`) policies $c_- \in C$ that are satisfied by *I*; i.e., $C_I^- = \{c_- \in C \mid Effect(c_-) = Deny \wedge [\![c_-]\!]_H(I)\}$. The set C_I^+ consists of all permission (`permit`) policies $c_+ \in C$ that are satisfied by *I*. Given these sets, the authorization output $\langle dec, reason, error \rangle \in \{Allow, Deny\}$ for *I* is determined as follows:

- If C_I^+ is empty or C_I^- is not empty, then $dec = Deny$ and $reason = \{PolicyID(c_-) \mid c_- \in C_I^-\}$.
- Otherwise, $dec = Allow$ and $reason = \{PolicyID(c_+) \mid c_+ \in C_I^+\}$.
- *error* = All the evaluation error messages

Example 1, semantics

> Consider policies *c1* and *c2* given in Example 1, above, and the entity hierarchy in Figure 2. Suppose that we extend this hierarchy with the following entities.

Type : Account
eid : "jane"
owner : User::"jane"

Type : Photo
eid : "receipt"
tags : ["private"]
"parents" : [Account::"jane",
Album::"jane_trips"]

Type : Photo
eid : "summer"
"parents" : [Account::"jane",
Album::"jane_trips"]

> If the user sends the request `<P=User::"alice", A=Action::"view", R=Photo::"summer", X={}>`, the policy *c1* is satisfied because `User::"alice"` belongs to the group `Group::"jane_friends"`, the resource `Photo::"summer"` belongs to the group `Album::"jane_trips"`, and the action `Action::"view"` appears in

the list `[Action::"view", Action::"comment"]`. Policy *c2* is *not* satisfied because `resource.tags.contains("private")`, i.e., the resource `Photo::"summer"`'s attribute `tags` does not contain `"private"`. Therefore, $c1 \in C_l^+$, while C_l^- is empty. The decision is `Allow`.

If the user sends the request `<P=User::“alice”, A=Action::"view", R=Photo::"receipt", X={}>`, the policy *c1* is satisfied for reasons similar to the above. However, policy *c2* is *also* satisfied because the `when` condition evaluates to `true`. This is because resource `Photo::"receipt"`'s attribute `tags` contains `"private"`, and *c2*'s `unless` condition evaluates to `false` because the photo is not a member of to `User::"alice"`'s `account`. Therefore, $c1 \in C_l^+$ and $c2 \in C_l^-$. Because C_l^- is nonempty, the decision is `Deny`, i.e., the `forbid` policy *c2* overrides the `permit` policy *c1.*

Policy Validation

As mentioned in Types, the Cedar language is dynamically typed, meaning that the evaluator will detect type errors as it evaluates (e.g., when it comes across an expression like `1 < "hello"`). As described in Policy semantics, if evaluation of a policy results in an error, then the evaluation result is *false*, meaning that the policy will not be considered in the authorization decision.

To avoid the possibility of an evaluation error, Cedar provides a *Schema-based Policy Validator*: Given a schema that describes the assumed structure of both entities and queries, the validator will flag those policies that may error during evaluation. If the validator flags no policies, then we can be sure that no type errors will error during policy evaluation for any entity hierarchy and request that adheres to the prescriptions of the schema.

The validator can operate in one of two modes: strict mode (the default), or permissive mode. The former mode is simpler and makes it easier to carry out additional automated reasoning.

Validation is optional: You can choose not to run the validator to check your policies. The evaluator will never run the validator; the two are independent. When running, the validator assumes the schema it is given is *complete*; i.e., it contains full information for every entity and action mentioned in the policies it considers.

A BASIC EXAMPLE

Here is a simple example of policy validation at work. Here is our example schema:

```
{
  "My::Namespace": {
    "entityTypes": {
      "Employee": {
        "shape": {
          "type": "Record",
          "attributes": {
            "jobLevel": {
              "type": "Long"
            },
            "numberOfLaptops": {
              "required": false,
              "type": "Long"
            },
            "isAdmin": {
              "type": "Bool"
            }
          }
```

```
          }
        },
        "Admin": {
          "shape": {
            "type": "Record",
            "attributes": {
              "isAdmin": {
                "type": "Bool"
              }
            }
          }
        }
      },
      "actions": {
        "remoteAccess": {
          "appliesTo": {
            "principalTypes": [
              "Employee"
            ],
            "context": {
              "type": "Record",
              "attributes": {
                "ip": {
                  "type": "String"
                }
              }
            }
          }
        }
      }
    }
}
```

This schema specifies that:

1. Every entity of type `My::Namespace::Employee` in the store has an attribute `jobLevel` whose value is a Cedar `long`, and another optional attribute `numberOfLaptops` which is also a Cedar `long`.
2. In a request with action `My::Namespace::Action::"remoteAccess"`, the principal should always be an entity of type `Employee`.

Now consider validation of the `when` clause of the following policy:

```
permit(principal, action == My::Namespace::Action::"remoteAccess", resource)
when {
   principal.numberOfLaptops < 5 &&   // (a)
   principal.jobLevel > "foo" &&      // (b)
   principal.jobLevel == "foo"        // (c)
}
```

For evaluation of a given request to reach the `when` clause, the request must satisfy the policy scope constraints. Thus, the action must be `My::Namespace::Action::"remoteAccess"`. Based on the schema, we can assume that the principal is an `Employee` and thus that it has a `jobLevel` attribute. With this knowledge, validation will report an error or warning on each of the comparisons (a)-(c) for the following reasons:

- (a): Validation error. The `principal` is not guaranteed to have the optional `numberOfLaptops` attribute, so the attribute access may raise a runtime error. The reasoning would be the same if the policy contained an attribute that wasn't present in the schema (e.g., `age`) or just had a typo such as `principal.jobbLevel`.
- (b): Validation error. The right operand of `>` is a string, but `>` only accepts `long`s, so the `>` operator will always raise a runtime error.
- (c): The left operand of `==` is always a `long`, the right operand is always a `string`, and the `==` operator returns `false` if its operands have different runtime types, so this comparison will always return `false`. While this won't raise a runtime error, it probably isn't what the policy author intended.

As in most programming languages, the main principle of type checking is that each Cedar operator has requirements on the types of its operands and returns a result of a given type. For example, `x > y` requires that `x` and `y` both have type `long`, and it returns a `boolean`. The validator reports an error if an operand does not have the required type: either `x > y` where the type of `y` is not `long`, or `x.jobLevel` where the type of `x` does not have an attribute named `jobLevel`. For the `==` case, it should be possible that the two operands have the same type. Optional attribute accesses, as in part (a), should be preceded with a `has` check, e.g., as follows:

```
principal has numberOfLaptops && principal.numberOfLaptops < 5
```

The `has` expression in the left operand of the `&&` is used to determine that the access to the optional `numberOfLaptops` attribute will not raise a runtime error. The `&&` expression short circuits, so the whole expression evaluates to `false` without evaluating the right operand when the attribute is not present. This could be equivalently written with the `has` in the condition of and `if` expression and the attribute access in the `then` branch.

The example policy up to this point does not contain any errors that are detected specifically by strict validation, so resolving the errors mentioned above would result in a policy passing both permissive and strict validation. The following policy contains errors which are reported by strict validation while being accepted by permissive validation without any changes.

```
permit(principal, action, resource) {
    (if principal.jobLevel < 8 then principal else Admin::"admin").isAdmin && // (d)
    ip(context.ip).isIpv6 // (e)
}
```

- (d) Strict validation error. The `then` and the `else` branch must have exactly the same type, one branch is an `Employee` entity (the assumed type of `principal`, per the `actions` part of the schema) while the other is an `Admin`. This same errors can occur for any `==`, `contains` , `containsAny`, `containAll` or set literals expression where the types of operands do not match exactly.
- (e) Strict validation error. The extension function `ip` is called on `context.ip`. The argument to the function call is a `String` which matches the required argument type for the `ip` constructor, but the strict validator forbids calling extension constructors with anything other than a literal value.

VALIDATION CHECKS SUPPORTED

The validator compares the policy set with the schema to look for inconsistencies. From these inconsistencies, the validator will be able to do the following

- Detect unrecognized Entity Types
    - e.g., Misspelling `"Album"` as `"Albom"`
- Detect unrecognized Action
    - e.g., Misspelling `Action::"viewPhoto"` as `Action::"viewPhoot"`
- Detect Action applied to unsupported Principal/Resource

- e.g., saying a `Photo` can view a `User`
- Detect improper use of `in` or `==` (provide a hint about proper use)
    - e.g., writing principal in `Album::“trip”` but principal cannot be a `Photo`
- Detect unrecognized attributes
    - e.g., `principal.jobbLevel` ← Typo, should be `"jobLevel"`
- Detect unsafe access to optional attributes
    - e.g., `principal.numberOfLaptops` where `numberOfLaptops` is an optional attribute (declared with `"required": false`). These should be guarded by `has` checks as in `if principal has numberOfLaptops then principal.numberOfLaptops else 0`. or `principal has numberOfLaptops && principal.numberOfLaptops < 2`
- Detect type mismatch in operators
    - e.g., `principal.jobLevel > "14"` ← Illegal comparison of a Long with a String
- Detect policies that will always evaluate to false, and thus never apply
    - e.g., ABAC part is `when { ["hello"].contains(1) }` always evaluates to `false` so the policy never applies

Strict validation additionally supports the following

- Detect type mismatch in certain expressions
    - e.g., `if principal.jobLevel > 2 the principal else Admin::"admin"`
- Detect empty sets occurring in policies
    - e.g., `action in []`
- Detect extension constructors applied to non-literal expressions
    - e.g., `ip(context.ip_string)`

Actions in the `actions` part of the schema may specify the expected format of the `context` (see schema format description below), so the above-listed errors can be flagged on references to `context` in the ABAC portion of rules, too.

SCHEMA FORMAT

The validator schema is written in JSON. It bears some resemblance to JSON Schema but unique aspects of Cedar’s design, such as its use of entity types, mean that there are some differences.

At the top level, the schema is a JSON map where the keys are **namespaces** and the values are JSON objects containing the **entity types specification** and the **actions specification** as maps from names to JSON objects containing the type or action defintion. The present implementation does not support multiple namespaces, so the namespaces JSON map must have exactly one entry defining a single namespace. The entity types and actions for a namespace are identified via keywords `entityTypes` and `actions`, respectively. The entity types map describes the type of each entity that may appear in the entity hierarchy, including the entity type’s attributes and the parent/child relationship that entities of this type can have to other entities in the hierarchy, if any. The actions map contains the entity IDs of entity type `Action` that may be used as actions in authorization requests, as well as assumptions on the `principal`, `resource`, and `context` parts of the request submitted with that action. Since actions are also entities, this part of the schema lists hierarchy information too.

Each entry in the `entityTypes` map has a key, the name of the entity type as a string, and a value which is a JSON object containing the definition of the entity type. This name must be an identifier, which is defined in the Cedar grammar as a sequence of alphanumeric character, omitting any Cedar reserved words. This type name is qualified by the enclosing namespace to form a fully qualified entity type which must be used when referencing this type in a policy. The JSON object for the value must have the following properties:

- `memberOfTypes`: A list containing strings which are the entity types that can be direct parents of an entity with this entity type. Such entity types must be valid entity type identifiers declared in the schema. If the `memberOfTypes` list is empty, or the property is not defined, then the entity type cannot have any ancestors in the entity hierarchy.

- `shape`: A JSON object following the JSON Schema-style format with custom `type` property values for Cedar types. The top level of this object must have the property `"type": "Record"`, as we treat entity attributes as a kind of record in this schema. Entity attributes may be declared as optional using the `required` property described for `Record` types below.

Each entry in the `actions` map has a key, the name of the action type as a string, and a value which is a JSON object containing the definition of the action. The name is an entity identifier rather than an entity type, so it can contain anything that would be valid inside a Cedar string. When combined with the entity type `Action`, this forms the complete entity UID for the action entity. Then the entity type `Action` is qualified by the enclosing namespace to obtain the fully qualfiied identifier for the declared action.The JSON object for the value must have the following properties:

- `appliesTo`: A JSON object containing two lists, `principalTypes` and `resourceTypes`, which contain the principal and resources entity types that the action can accompany in an authorization request.
    - If the `appliesTo` property or either of the component lists are absent from the `actions` element object, then it is assumed the action could appear in an authorization request with an entity of *any* type, or with an unspecified entity. It is therefore not possible to access any attributes on the principal or resource.
    - Both the `principalTypes` and `resourceTypes` could be empty lists, but this would represent an action that cannot be used in an authorization request with any entity types.
    - `context`: A JSON object in the same format as entity `shape` property which defines the attributes that must be present in the context record in authorization requests made with this action.

The schema format uses a JSON-Schema-like structure for declaring entity attributes and contexts. Different values for the `type` property are used to support Cedar types.

- `String`, `Long`, and `Boolean` types are used to encode the primitive Cedar types.
- `Set` encodes the Cedar set type. Used together with a property `element` to hold the type of elements in the set
- `Record` encodes Cedar record types. The `attributes` property is a map from record attribute names to their type. The type of each attribute is structured using this JSON format, but with an additional property `required`. This attribute specifies if the attribute is always present in the record. This property is `true` by default. Setting to `false` means the attribute can be absent from the record, so specific checks will be required before safely accessing the attribute.
- `Entity` encodes Cedar entity reference types. This is used together with a property `name` which specifies the type of the referenced entity. The value of `name` is again a Cedar `Name`.

A full example schema is given in Appendix D: Sample Schema.

Implementation Notes

FORMAL SPECIFICATION

This specification is formalized as a Dafny model. We are using this formal model to prove properties of the Cedar language model, and to use it as a basis against which to test the production implementation.

PRODUCTION IMPLEMENTATION

Our production implementation of this specification is written in Rust. We chose Rust to balance performance and safety. We have also written Java bindings for the Rust engine.

ERROR REPORTING

Cedar’s implementation aims to report errors early when possible, and should clearly describe what went wrong. However, there is currently no semantics for the behavior of errors, and these are subject to change. Cedar does attempt to follow some basic guidelines:

- Some error is always reported when processing cannot continue
- Multiple errors are returned when possible and convenient for the user
- Error types and codes will remain consistent when possible, but may change without a major version change
- Error text is subject to change at any time and will not be reliable

It may also be useful to note here that if evaluating a policy during authorization results in an error, that policy will simply be ignored, and processing of other policies will continue. These errors are reported as diagnostics along with the main results.

Appendix A: Cedar operators and functions

Table 1. Built-in operators and functions shows the built-in operators and functions for the Cedar language. The table lists the available overloads for each operator. In addition to the operators and functions in the table, Cedar also supports:

- if-then-else ternary expressions (like the C `? :` operator), with the syntax `if expr1 then expr2 else expr3`. The condition `expr1` must evaluate to a boolean.

See Data model, values, and operations which discusses operators `.` (attribute access) and `[]` (record element access), not shown.

Table 1. Built-in operators and functions

Symbol	Types and overloads	Description
==	any → any	equality. Works on arguments of any type, even if the types don't match. (Values of different types are never equal to each other.)
!=	any → any	inequality; the exact inverse of equality (see above)
<	(long, long) → bool	long integer less-than
<=	(long, long) → bool	long integer less-than-or-equal-to
>	(long, long) → bool	long integer greater-than
>=	(long, long) → bool	long integer greater-than-or-equal-to
+	(long, long) → long	long integer addition
-	long → long	long integer unary negation (e.g., -1)
	(long, long) → long	long integer subtraction (e.g., 2-1)
*	(long, long) → long	long integer multiplication. At least one argument must be a constant.
in	(entity, entity) → bool	Hierarchy membership (reflexive: A in A is always true)
	(entity, set(entity)) → bool	Hierarchy membership: A in [B, C, ...] is true iff (A in B) || (A in C) || ... error if the set contains a non-entity
!	bool → bool	logical not
&&	(bool, bool) → bool	Logical *and* (short-circuiting)
||	(bool, bool) → bool	Logical *or* (short-circuiting)
has	(entity, attribute) → bool	infix operator. `e has f` tests if the record or entity `e` has a binding for the attribute `f` . returns `false` if `e` does not exist or if `e` does exist but doesn't have the attribute `f`. Attributes can be expressed as identifiers or string literals.
like	(string, string) → bool	infix operator. `t like p` checks if the text `t` matches the pattern `p`, which may include wildcard characters `*` that match 0 or more of any character. In order to match a literal star character in `t`, users can use the special escaped character sequence `\*` in `p`.
.contains()	(set, any) → bool	Set membership (is B an element of A)
.containsAll()	(set, set) → bool	Tests if set A contains all of the elements in set B
.containsAny()	(set, set) → bool	Tests if set A contains any of the elements in set B

In the table, a function name that is prefixed with a dot and ends with parentheses (e.g., `.contains()`) should be called with method-style syntax (e.g., `a.contains(b)`). All other symbols denote operators prefix or infix operators, and should be used in the usual way (e.g., `!b` or `a in b`). In the description when we reference `A` and `B` for a method-style operator, `A` is the receiver, i.e., `A.contains(B)`. The integer arithmetic operators (`+`, `-`, `*`) will error on overflow (range: min: -9223372036854775808, max:9223372036854775807). This semantics may change in future versions.

EXTENSION TYPES

Cedar currently supports two extension types: IP addresses (**IPAddr**), and decimal numbers (**Decimal**). The functions associated with these types are given in Table 2 and Table 3, respectively. Cedar extension functions are either called function-style (e.g., `func(a, b)`) or method-style (e.g. `a.func(b)`). In tables below, function-style is indicated by `func(.)` while method-style is indicated by `.func()`. Note that we do not specifically define an equality operator for extension types: extension-type values can also be compared with `==`, just like all Cedar values. (See Data model, values, and operations.)

Table 2. IPAddr extension functions

Function	Type	Description
ip(.)	string → ipaddr	Parse a string representing an IP address or range. Supports both IPv4 and IPv6. Ranges are indicated with CIDR notation (e.g. /24).
.isIpv4()	ipaddr → bool	Tests whether an IP address is an IPv4 address
.isIpv6()	ipaddr → bool	Tests whether an IP address is an IPv6 address
.isLoopback()	ipaddr → bool	Tests whether an IP address is a loopback address
.isMulticast()	ipaddr → bool	Tests whether an IP address is a multicast address
.isInRange()	(ipaddr, ipaddr) → bool	Tests if ipaddr A is in the range indicated by ipaddr B. (If A is a range, tests whether A is a subrange of B. If B is a single address rather than a range, B is treated as a range containing a single address.)

Table 3. Decimal extension functions

Function	Type	Description
decimal(.)	string → decimal	Parse a string representing a decimal value. Matches against the regular expression `-?[0-9]+.[0-9]+`, allowing at most 4 digits after the decimal point.
.lessThan()	(decimal, decimal) → bool	Tests whether the first decimal value is less than the second
.lessThanOrEqual()	(decimal, decimal) → bool	Tests whether the first decimal value is less than or equal to the second
.greaterThan()	(decimal, decimal) → bool	Tests whether the first decimal value is greater than the second
.greaterThanOrEqual()	(decimal, decimal) → bool	Tests whether the first decimal value is greater than or equal to the second

Appendix B: Policy grammar

The grammar specification applies the following the conventions. It uses `|` for alternatives, `[]` for optional productions, `()` for grouping, and `{}` for repetition of a form zero or more times. Capitalized words stand for grammar productions, and lexical tokens are given in all-caps. Tokens are defined using regular expressions, where `[]` stands for character ranges; `|` stands for alternation; `*` , `+` , and `?` stand for zero or more, one or more, and zero or one occurrences, respectively; `~` stands for complement; and `-` stands for difference. The grammar ignores whitespace and comments.

```
Policies  := {Policy}
Policy    := Effect '(' Scope ')' {Conds} ';'
Effect    := 'permit' | 'forbid'
Scope     := Principal ',' Action ',' Resource
Principal := 'principal' [('in' | '==') (Entity | '?principal')]
Action    := 'action' [('in' '[' EntList ']' | '==' Entity)]
Resource  := 'resource' [('in' | '==') (Entity | '?resource')]
Conds     := ('when' | 'unless') '{' Expr '}'

Expr      := Or | 'if' Expr 'then' Expr 'else' Expr
Or        := And {'||' And}
And       := Relation {'&&' Relation}
Relation  := Add [RELOP Add]
           | Add 'has' (IDENT | STR)
           | Add 'like' PAT
Add       := Mult {('+' | '-') Mult}
Mult†     := Unary { '*' Unary}
Unary     := ['!' | '-']x4 Member
```

```
Member     := Primary {Access}
Access     := '.' IDENT ['(' [ExprList] ')'] | '[' STR ']'
Primary    := LITERAL
            | VAR
            | Entity
            | ExtFun '(' [ExprList] ')'
            | '(' Expr ')'
            | '[' [ExprList] ']'
            | '{' [RecInits] '}'

Path       := IDENT {'::' IDENT}
Entity     := Path '::' STR
EntList    := Entity {',' Entity}
ExprList   := Expr {',' Expr}
ExtFun     := [Path '::'] IDENT
RecInits   := (IDENT | STR) ':' Expr {',' (IDENT | STR) ':' Expr}

RELOP      := '<' | '<=' | '>=' | '>' | '!=' | '==' | 'in'
IDENT      := ['_''a'-'z''A'-'Z']['_''a'-'z''A'-'Z''0'-'9']* - RESERVED
STR        := Fully-escaped Unicode surrounded by '"'s
PAT        := STR with `\*` allowed as an escape
LITERAL    := BOOL | INT | STR
BOOL       := 'true' | 'false'
INT        := '-'? ['0'-'9']+
RESERVED   := BOOL | 'if' | 'then' | 'else' | 'in' | 'like' | 'has'
VAR        := 'principal' | 'action' | 'resource' | 'context'

WHITESPC   := Unicode whitespace
COMMENT    := '//' ~NEWLINE* NEWLINE
```

† We place a syntactic constraint on the multiplication operation: at most one of the operands can be something other than an integer literal. For instance, `1 * 2 * context.value * 3` is allowed while `context.laptopValue * principal.numOfLaptops` is not.

Appendix C: Cedar entity JSON format

The Cedar public API provides several functions to construct Cedar entities from a JSON representation. (One such function is `Entities::from_json_value()`.) We describe that JSON representation in this appendix.

At the top level, Cedar expects a JSON list (array) of objects. Each object represents a single entity, and should have three attributes, `uid`, `attrs`, and `parents`. We discuss these three fields in turn, focusing on `attrs` first.

ATTRS

`attrs` is a JSON object, whose keys and values are interpreted as Cedar attribute values. For example:

```
// Example 1
"attrs": {
    "department": "HardwareEngineering",
    "jobLevel": 5
},
```

Notice that the `department` attribute has a string value, and the `jobLevel` attribute has an integer value, but this is not explicitly marked in the JSON format. Instead, JSON strings and JSON integers are automatically converted into Cedar strings and Cedar integers. The same happens for JSON booleans (to Cedar booleans), JSON lists/arrays (to Cedar sets), and JSON objects (to Cedar records).

These implicit conversions produce 5 of the 7 possible types of Cedar values: bools, integers, strings, sets, and records. What remains is entity references (like `User::"12UA45"`) and extension values (like IP addresses). For entity references, the Cedar JSON format supports an `__entity` escape, whose value is a JSON object with attributes `type` and `id`:

```
// Example 2
"attrs": {
    "department": "HardwareEngineering",
    "jobLevel": 5,
    "manager": {
       "__entity": {
           "type": "User",
           "id": "78EF12"
       }
    }
},
```

Likewise, for extension values, the Cedar JSON format supports an `__extn` escape, whose value is a JSON object with attributes `fn` and `arg`:

```
// Example 3
"attrs": {
    "department": "HardwareEngineering",
    "jobLevel": 5,
    "home_ip": {
       "__extn": {
           "fn": "ip",
           "arg": "222.222.222.101"
       }
    }
},
```

The `fn` attribute names a specific Cedar extension function, which will be called with the `arg` value to produce the final attribute value.

SCHEMA-BASED PARSING

Cedar supports “schema-based parsing” for entity data and contexts. This allows customers to omit the `__entity` and `__extn` escapes when the schema indicates that the corresponding attribute values are entity references or extension values, respectively. For example:

```
// Example 4
"attrs": {
    "department": "HardwareEngineering",
    "jobLevel": 5,
    "manager": {
       "type": "User",
       "id": "78EF12"
    },
    "home_ip": {
```

```
        "fn": "ip",
        "arg": "222.222.222.101"
    }
},
```

For extension values, the `fn` can be implicit as well. If the schema indicates that `home_ip` is an IP address, the following is also accepted:

```
// Example 5
"attrs": {
    "department": "HardwareEngineering",
    "jobLevel": 5,
    "home_ip": "222.222.222.101"
},
```

`UID` AND `PARENTS`

Other than `attrs`, the other two fields expected for each entity are `uid` and `parents`. `uid` is expected to be a single entity reference. Customers can explicitly add the `__entity` escape, or leave it implicit. That is, both of the following are valid:

```
"uid": { "__entity": { "type": "User", "id": "12UA45" } }
```

```
"uid": { "type": "User", "id": "12UA45" }
```

Similarly, `parents` is expected to be a JSON list/array containing entity references. Those entity references can also take either of the two forms above, with `__entity` either explicit or implicit.

EXAMPLE EXCERPT FROM A JSON FILE, DESCRIBING TWO ENTITIES

This example pulls together many of the features discussed in the previous sections. It uses `uid`, `attrs`, and `parents`. For `uid` and `parents` it uses the implicit `__entity` escape rather than explicitly adding it. Of course, a full entities file would also need to include entries for the two `UserGroup` entities which are referenced but not defined in this example.

This example also demonstrates attribute values with entity types (`User`) and extension types (IP address and decimal), and uses the explicit `__entity` and `__extn` escapes for those. It does not rely on Schema-based parsing.

```
[
    {
        "uid": { "type": "User", "id": "alice"},
        "attrs": {
            "department": "HardwareEngineering",
            "jobLevel": 5,
            "homeIp": { "__extn": { "fn": "ip", "arg": "222.222.222.7" } },
            "confidenceScore": { "__extn": { "fn": "decimal", "arg": "33.57" } }
        },
        "parents": [{ "type": "UserGroup", "id": "alice_friends" }, { "type": "UserGrou
    },
    {
        "uid": { "type": "User", "id": "ahmad"},
        "attrs" : {
            "department": "HardwareEngineering",
            "jobLevel": 4,
```

```
                "manager": { "__entity": { "type": "User", "id": "alice" } }
            },
            "parents": []
        }
    ]
```

FINAL NOTES

- The portion of the just-described format for entity attribute values is also used by some functions for constructing the authorization request `context` from JSON (e.g., `Context::from_json_value()`).
- Alternatives to JSON exist for both `Entities` and `Context`, e.g., constructing entities/context programmatically using functions such as `Entities::from_entities()`.
- In any case, the `Entities` and `Context` (via `Request`) are ultimately passed to `Authorizer::is_authorized()` for an authorization request.

Appendix D: Sample Schema

The following schema is an example in the format described above.

```
{
  "My::Name::Space": {
    "entityTypes": {
      "User": {
        "shape": {
          "type": "Record",
          "attributes": {
            "department": { "type": "String" },
            "jobLevel": { "type": "Long" }
          }
        },
        "memberOfTypes": [ "UserGroup" ]
      },
      "UserGroup": {},
      "Photo": {
        "shape": {
          "type": "Record",
          "attributes": {
            "private": { "type": "Boolean" },
            "account": { "type": "Entity", "name": "Account" }
          }
        },
        "memberOfTypes": [ "Album" ]
      },
      "Album": {
        "shape": {
          "type": "Record",
          "attributes": {
            "private": { "type": "Boolean" },
            "account": { "type": "Entity", "name": "Account" }
          }
        },
        "memberOfTypes": [ "Album" ]
      },
```

```
      "Account": {
        "shape": {
          "type": "Record",
          "attributes": {
            "owner": { "type": "Entity", "name": "User" },
            "admins": {
              "required": false,
              "type": "Set",
              "element": { "type": "Entity", "name": "User" }
            }
          }
        }
      }
    },
    "actions": {
      "photoAction": {
        "appliesTo": {
          "principalTypes": [],
          "resourceTypes": []
        }
      },
      "viewPhoto": {
        "appliesTo": {
          "principalTypes": [ "User" ],
          "resourceTypes": [ "Photo" ],
          "context": {
            "type": "Record",
            "attributes": {
              "authenticated": { "type": "Boolean" }
            }
          }
        }
      },
      "listAlbums": {
        "appliesTo": {
          "principalTypes": [ "User" ],
          "resourceTypes": [ "Account" ],
          "context": {
            "type": "Record",
            "attributes": {
              "authenticated": { "type": "Boolean" }
            }
          }
        }
      },
      "uploadPhoto": {
        "appliesTo": {
          "principalTypes": [ "User" ],
          "resourceTypes": [ "Album" ],
          "context": {
            "type": "Record",
            "attributes": {
              "authenticated": { "type": "Boolean" },
              "photo": {
                "type": "Record",
```

```
                "attributes": {
                  "file_size": { "type": "Long" },
                  "file_type": { "type": "String" }
                }
              }
            }
          }
        }
      }
    }
  }
}
```