The Formal Semantics of XACML

Polar Humenn

Syracuse University

Abstract

This paper presents the formal declarative semantics of the Extensible Access Control Markup Language (XACML) Version 1.1 using a declarative functional language. Having a formal semantics specified for this language leads to greater assurance that implementations will not misinterpret the XACML specification, and gives the community a formal language with which to discuss problems, ambiguities, or future features. This formal analysis has produced some ambiguities, or maybe some undesirable interpretations of the specification.

Introduction

The XACML language is an XML syntax based language that provides a common means to express access control policies. This language can be used to hold descriptions of access control policies, as well its standard format to be used to transport policies between disparate access control systems. In order to enable this interoperability, a mere syntax is not good enough to have a common understanding of the language and its semantics. The semantics of XACML are described, informally, in normative documents. However, using English paragraphs to explain the various components of the language can lead to misinterpretations, not only by developers, but in the document itself. It is desirable to have a formal description of the XACML semantics.

Formal semantics can be described in many different ways. Previously, a denotational semantics, using greek symbols, mathematical structures, and familiar paradigms, like set theory and its notation are the standard mechanism. This approach leads to problems in that it only served the people who understood such concepts with defacto standard representations only a defacto standard by tradition, and not specification. The notation must be defined and in some cases, redefined, in order to make the description complete.

In contrast to that approach, we take advantage of a declarative language, Haskell, with which to describe formal semantics of XACML. Using Haskell for this purpose has many advantages. First, the language is declarative. The description of functions are not stated in loops or repetitive side effecting statements, but are stated as re-writable transformations using recursive definitions. Second, the language is executable. The result of this document may be used as a reference implementation of XACML, which gives the formal semantics an air of correctness in more practical terms. Third, Haskell is strongly typed, which means that the functions must adhere to strict typing. This approach cuts down on errors, and provides proofs by types of the evaluative structure. Forth, Haskell contains higher order functions and lazy semantics. The higher order

functions allow functions to be first class objects that get passed to other functions, which offers greater flexibility and shortness in the expressions. The lazy semantics allows us to write declarative expressions where the evaluation order may not matter. One example, of this is passing a description of an infinite list of 1's to a “take” function that will only take the first three. In a pass by value system, the argument, which is the infinite list would have to be evaluated before being passed to the the take function. Evaluating the infinite list would run the computer out of memory and/or space. In lazy semantics the description of the expression will yield a list of three 1's, and it will evaluate to that much since the infinite list of 1's will only be evaluated to the length needed. Finally, the formal semantics of Haskell is defined, and the language has been shown to be sufficiently complete. Therefore, any semantic description done in Haskell is essentially a formal semantic description of which proofs can be made. Haskell is widely understood in semantic community.

As one chief benefit, this formal description will serve as a way for the XACML to discuss the concepts and semantics of XACML, especially ambiguities that arise, and possibly new features.

Haskell

Haskell is a functional programming language. The reader is encouraged to read various materials on Haskell and its declarative style. However, for a quick summary, in the following Haskell notation, a function definition takes the form of clauses that are applied to patterns of structures, namely lists. Function application is denoted by juxtaposition. The symbol “[]” denotes the empty list, whereas the expression “(x:xs)” matches against an argument of a non-empty list of which “x” represents the first element of the list, and “xs” is the rest of the list, which is a list itself, and it may be an empty list. A function that counts the number of elements of a list is defined with the two following clauses:

```
length []     = 0
length (x:xs) = add 1 (length xs)
```

We see the declarative style of Haskell. There are no loops counting the amount of elements in the list structure. Instead, the first declarative clause states, “The length of the empty list is 0.” The second clause states, “The length of a non empty list is one more than the length of the rest of the list.” Together these two statements create a formal description of the length function using the recursive definition, and it may be proved, using standard induction proof techniques, that the description denotes the number of elements in any list.

Haskell is strictly typed and polymorphic. Types are defined by either constants that are denoted by starting with a capital letter, and polymorphism is described using type variables, which are denoted by words starting with small letters. The type of the length function is denoted by the following type expression:

```
length :: [a] -> Int
```

The type of the function states that length is a function, denoted by the infix "->" operator. On the left side is the type of its argument and on the right side is the type of the result. Here "[a]" denotes a list of type "a", where "a" denotes a type variable. This declaration means that a function of this type may be applied to a list of *any* type. When given this list as an argument, this function return an "Int" which is the Haskell type for an integer.

Haskell has not only the ability to describe function definitions and types of functions but also can describe abstract data types other than lists. For instance, even in Haskell, the type representing boolean values, called Bool is defined as follows:

```
data Bool = True | False
```

Above, the key word "data" is used to define a type "Bool" and its data consists of two "constructors", which have zero-arity, "True" and "False". All constructors must start with capital letters. An example of a polymorphic data type would be:

```
data Tree a = Leaf a | Node a (Tree a) (Tree b)
```

This familiar data type describes a tree of type "a" in which one element is constructed by the constant "Leaf" followed by an element of type "a", and another element is constructed by the constant "Node" followed by an element of type "a" and a left and right "sub"-trees. An example of a constructed value in Haskell follows:

```
my_tree :: Tree Int
my_tree = (Node 1 (Leaf 2) (Node 3 (Leaf 3) (Leaf 4)))
```

In the above description the type of "my_tree" is defined to be a tree of "Int". Constructors work like functions in that they are applied via juxtaposition to their element values and return a value of their type. The value of "my_tree" is declared to be of type "Tree Int" containing a node containing 1, a left subtree containing a leaf containing 2, and a right subtree containing a node with 3, and left and right subtrees containing leaves containing 3 and 4 respectively.

Haskell also makes use of type synonyms. Sometimes it is desirable to describe a type by another name, which may be more descriptive and take advantage of type constructors. For example, the declaration,

```
type IntTree = Tree Int
```

formally describes the type "IntTree" type to be the same as the constructed type "Tree Int". Type variables may also be used. For example, the declaration,

```
type Transform a b = Tree a -> Tree b
```

formally describes a type of “Transform a b”. The type is described as a function that takes a Tree of type “a” and returns a tree of type “b”. This constructed type may be used to describe the type of some functions such as:

```
toDouble :: Transform Int Double
toDouble (Leaf x)            = (Leaf (int2double x))
toDouble (Node x left right) = (Node (int2double x)
                                     (toDouble left) (toDouble right))
```

The above clauses show that a Tree of type “Int” is deconstructed and reconstructed as a Tree of “Double” using a function called “ int2double ” which does the numeric type conversion. It's type is declared as follows:

```
int2double  :: Int -> Double
```

We do not show the description of the function of “int2double” because it is primitive in nature and it is not needed to define our understanding of the description as a whole. One advantage about using a Haskell based description is that it is sufficient to declare the type of the function in order to “use” the function in a declarative description. This approach allows us to describe the semantics of a function while delaying the actual description of the needed functions.

Haskell has many more features such as type classes and automatically derived functions based on declared data types. However, these mechanisms are beyond the needed features to describe the formal semantics of XACML, which is our main purpose. The reader is encouraged to read documents on the Haskell language to get a better understanding of the language as well as download Haskell processors, which are available for many different platforms, to exercise this description.

We now embark upon our formal semantics for XACML.

XACML

The XACML specifications describe a model whereby a Policy Enforcement Point (PEP) asks a Policy Decision Point to make an access control decision. The documents describe the inputs to this question, namely the XACML Request Context, and the response, the XACML Response Context. The model also allows for a Policy Administration Point to take a policy described in the XACML Policy Language and implement the policy as a PDP.

We start with defining a Haskell module to hold our declarations.

```
module XACML where

import Prelude
```

The “import” statement states to use the standard Haskell “Prelude”, which contains a library of important definitions.

All code surrounded in the blue shadow boxes is normative Haskell. One may take all, but not less than all, the definitions that are the blue shadow boxes in this document, concatenate them in order, and feed it to a Haskell 98 interpreter. The Haskell used here is sensitive to fixed with spacing. All leading spacing must be maintained as seen in this document for a Haskell interpreter to parse it.

The Policy Decision Point

The Policy Decision Point, or PDP for short, is the term used by XACML that abstractly denotes the point at which an access decision is calculated. In short, a Policy Enforcement Point (PEP) asks a PDP for an access decision, and the PEP enforces that decision. For XACML, the PDP takes an XACML Request Context and returns an XACML Response Context. We model the PDP as a function.

```
type PDP = RequestContextT -> ResponseContextT
```

The type PDP is a type synonym that represents a function whose argument is of type “RequestContextT” and its result is of type “ResponseContextT”. Since Haskell constructors and type names may have the same names, differentiated due to language scope, it may still be confusing. We use a “T” suffix to denote the type name of an abstract data type as to avoid confusion of the reader with the constructor name of that data type.

The XACML Request Context

The XACML Request Context is defined in XML. It is straight forward to model the XACML Request Context elements into Haskell data types so that we may indeed write Haskell functions that operate on the request context.

For simplicity and without loss of generality we will only concern ourselves with subjects, resources, actions, and environments that use the XACML attribute approach. Also, for simplicity with types and without loss of generality we assume that all attributes contain all its data elements. Every attribute has the Attributeid, Issuer, IssueInstant, and Attribute Value sub elements specified.

We do not model the ResourceContent element as it requires XML parsing. which we feel is unnecessary to express the semantics of it in Haskell. Furthermore, the attribute approach is sufficient to formally specify the semantics of the basic structure of XACML.

<RequestContext>

The XACML Request Context structure is modeled by the following data type:

```
data RequestContextT =
             RequestContext [SubjectT] ResourceT ActionT EnvironmentT
                     deriving Show
```

The above declaration straightforwardly follows the XACML Request Context structure. An XACML Request Context is defined as being a “RequestContext” constructor followed by a list of subjects, a resource, an action, and an environment.

<Subject>

The XACML Request Context's Subject structure is modeled by the following data type:

```
data SubjectT = Subject SubjectCategoryIdT [AttributeT] deriving Show
```

We define the XACML Request Context structure's Subject element using the “Subject” constructor followed by subject category identifier and a list of attributes. The subject category identifier is a string, and we will represent it in Haskell as the primitive string type.

```
type SubjectCategoryIdT  = String
```

<Attribute>

The XACML Request Context's Attribute structure is modeled by the following data type:

```
type AttributeT = (AttributeIdT, IssuerIdT, DateTimeT, AttributeValueT)
```

We take advantage of the “tuple” constructor in Haskell, using “(,)” to represent an XACML Attribute. An attribute contains all of its sub elements. We define it as a tuple of attribute id, issuer, issue instant, and finally, the attribute value. The attribute value contains the data type specification.

For our purposes, we restrict identifiers to primitive strings. We specify the type of attribute values at the end of this section, which will also describe the data type.

```
type AttributeIdT    = String
type DataTypeIdT     = String
```

```
type IssuerIdT       = String
```

<Resource>, <Action>, <Environment>

The XACML Request Context's Resource, Action, and Environment structures are modeled by separate data types.

We define the XACML Request Context structure's resources, actions, and environment elements similarly to the Subject element as a specific constructor followed by a list of relevant attributes.

```
data ResourceT    = Resource    [AttributeT] deriving Show
data ActionT      = Action      [AttributeT] deriving Show
data EnvironmentT = Environment [AttributeT] deriving Show
```

<AttributeValue>

Each AttributeValue may have different data types. We define an attribute value in Haskell using a abstract data type “AttributeValueT” tagging the specific data types with different constructors. It is straight forward to take advantage of the corresponding primitive types in Haskell.

```
data ValueT =
           IntAtom           Int
         | BoolAtom          Bool
         | DoubleAtom        Double
         | StringAtom        String
         | DateTimeAtom      DateTimeT
         | TimeAtom          TimeT
         | DateAtom          DateT
         | HexBinaryAtom     String
         | Base64BinaryAtom  String
         | AnyURIAtom        String
         | YearMonthDurationAtom Int Int
         | MonthDayDurationAtom  Int Int
         | Bag [ValueT]
         | IndeterminateVal
             deriving (Eq,Show)
```

The “deriving(Eq)” clause means that an equality predicate, corresponding to the “==” infix function will be derived for the type of “ValueT”. This clause means that it is fully specified such that two elements of the “ValueT” type may be compared to each other for equality by each of their constructor names and subelements.

We define the abstract data types to handle dates and times as constructed values of integers.

```
data TimeT = Time Int Int Int deriving (Eq,Show)
data DateT = Date Int Int Int deriving (Eq,Show)
type DateTimeT = (DateT,TimeT)
```

Finally, an attribute value is just a tagged value.

```
type AttributeValueT = ValueT
```

We use the value abstract data type for two purposes. Its representation specifies the values that attributes can take on, and also model the results of expressions in the policy. Attribute values may specify all but “Bag” and “IndeterminateVal” constructors. These restrictions cannot be enforced in Haskell. However, there is no need for enforcement as a straight forward translation from the XML to this abstract data type will not yield the use of “Bag” or “IndeterminateVal”. These particular constructors make specifying the semantics of the value representations and the their corresponding values as evaluated entities more unified.

Sample Request Context

This set of declarations fully describes an XACML Request Context that uses attributes.

```
<xacml-context:Request>
  <Subject SubjectCategory="access-subject">
     <Attribute AttributeId="subject-id" DataType="string" Issuer="Sam"
                IssueInstant="2003/10/15 12:32:54">
         <AttributeValue>Polar</AttributeValue>
     </Attribute>
     <Attribute AttributeId="weight" DataType="integer" Issuer="Sam"
                IssueInstant="2003/10/15 12:32:54">
         <AttributeValue>185</AttributeValue>
     </Attribute>
  </Subject>
  <Resource>
     <Attribute AttributeId="resource-id" DataType="string" Issuer="Bob"
                IssueInstant="2003/10/15 14:23:32">
         <AttributeValue>xacml-document</AttributeValue>
     </Attribute>
  </Resource>
  <Action>
     <Attribute AttributeId="action-id" DataType="string" Issuer="Bob"
                IssueInstant="2003/10/15 14:23:32">
         <AttributeValue>nodify</AttributeValue>
     </Attribute>
  </Action>
  <Environment>
     <Attribute AttributeId="weather" DataType="string" Issuer="Bob"
                IssueInstant="2003/10/15 14:23:32">
         <AttributeValue>fair</AttributeValue>
     </Attribute>
  </Environment>
</xacml-context:Request>
```

In our Haskell model of the request context, this sample XACML Request Context corresponds to the following definition:

```
xacml_req :: RequestContextT
xacml_req = (RequestContext
               [(Subject "access-subject"
                  [("subject-id", "Sam", (Date 2003 10 15, Time 12 23 54),
                                          StringAtom "Polar"),
                   (“weight”, "Sam", (Date 2003 10 15, Time 12 23 54),
                                          IntAtom 185)])]
               (Resource
                  [("resource-id","Bob", (Date 2003 10 15, Time 14 23 22),
                                          StringAtom "xacml-document")])
               (Action
                  [("action-id","Bob", (Date 2003 10 15, Time 14 23 22),
                                          StringAtom "modify")])
               (Environment
                  [(“weather”,"Bob", (Date 2003 10 15, Time 14 23 22),
                                          StringAtom ”fair”)]))
```

The XACML Response Context

The XACML Response Context has the ability to return multiple results, each having other elements such as a decision, a status, and a list of obligations.

```
data ResponseContextT = ResponseContext [ResultT] deriving Show
data ResultT          = Result DecisionT StatusT [ObligationT]
                                              deriving Show
```

We model the XACML decision as zero-ary constructors for each of the four decision values we have in XACML.

```
data DecisionT = Permit
               | Deny
               | Indeterminate
               | NotApplicable
                      deriving (Eq,Show)
```

We model Status as a constructed value of strings for status code, status message, and status detail. We model obligations as a constructed value with a string identifier.

```
data StatusT     = Status String String String
                                 deriving Show

type EffectT     = DecisionT

type ObligationT = (EffectT,String)
```

An “EffectT” is synonymous with a “DecisionT” except that we restrict its use to only the “Permit” and “Deny” constructors. An “ObligationT” is a tuple of an effect and a string, where the first element, which is the effect, models the “FullfilOn” XML attribute

of the XACML Obligation element.

A sample Response Context, may look as follows:

```
xacml_resp :: ResponseContextT
xacml_resp = (ResponseContext [Result Permit (Status “ok” “” “”)  []])
```

The above declarations fully describe an XACML Request Context, Response Context, and the type of a PDP.

XACML Policy

We have formally specified the contents of a XACML request context and XACML response context, but we have not yet described how one gets from one to the other. To describe this functionality we could easily write any function in Haskell that can satisfy the type PDP. Alternatively, we should be able to describe the process of following an XACML Policy in when evaluated against a Request Context to produce the corresponding Response Context. We will use the term policy to mean both XACML Policy or XACML Policy Set in places where it is meant to be either.

The Policy Administration Point

In XACML it can be said that a PDP may follow a single policy. Although that is not a strict requirement, we can indeed follow this approach without loss of generality.

The Policy Administration Point (PAP) is an abstract entity in XACML that is said to administer policies for a PDP . We model a PAP as a function that takes a single policy and produces a PDP function from it. Therefore, an instance of a PDP may be modeled as the function returned from applying a single policy and produces a function that is equivalent to the PDP function.

```
type PAP = PolicyT -> PDP
```

We will not address the concern of PAPs that take multiple policies or perform static updates to their policy databases, as these constructions may be modeled as one policy.

<Policy> and <PolicySet>

We describe XACML Policy and Policy Set elements as constructed elements of the PolicyT type.

```
data PolicyT =
    Policy    String TargetT RuleCombinatorIdT   [RuleT]   [ObligationT]
  | PolicySet String TargetT PolicyCombinatorIdT [PolicyT] [ObligationT]
        deriving Show
```

```
type RuleCombinatorIdT   = String
type PolicyCombinatorIdT = String
```

Straightforwardly from the XML, the XACML Policy is modeled by a data structure tagged with the “Policy” constructor followed by a string for the policy identifier, a target specification, a rule combinator identifier, a list of rules, and a list of obligations. Similarly, the XACML Policy Set is the same except that the constructor is “PolicySet”, and it holds a list of policies (and policy sets) instead of rules.

<Rule>

A rule is a specification for a decision predicated on two conditions, one being the target, which is somewhat simple in order to facilitate indexing, and the other being a possibly more complex predicate called the “condition”.

```
data RuleT = Rule TargetT ConditionT DecisionT  deriving Show
type ConditionT = ExpressionT
```

One will note how we are reusing data type definitions from the Response Context, namely “DecisionT”. Using common definitions between the policy, request context, and response context allow us to unify the semantics without having to specify trivial translations.

Expressions and Conditions

The condition is an expression such that its evaluation results in a boolean value. We define the elements of an expression:

```
data ExpressionT =
      Value AttributeValueT
    | Apply FunctionIdT [ExpressionT]
    | SubjectAttributeDesignator
         SubjectCategoryIdT AttributeIdT DataTypeIdT IssuerSpecT
         MustBePresentT
    | ResourceAttributeDesignator
         AttributeIdT DataTypeIdT IssuerSpecT MustBePresentT
    | ActionAttributeDesignator
         AttributeIdT DataTypeIdT IssuerSpecT MustBePresentT
    | EnvironmentAttributeDesignator
         AttributeIdT DataTypeIdT IssuerSpecT MustBePresentT
             deriving Show
type FunctionIdT      = String
type MatchIdT         = String
type MustBePresentT   = Bool
```

The expression is a data type that specifies any value may be specified (excluding Bag

and Indeterminate). Also, the Apply constructor applies a function named by the function identifier which is applied to a list of expressions, which are the arguments to the function.

The attribute designators are straight forward translation from the XML elements. Each attribute designator names its attribute identifier, its data type identifier, an optional issuer id, and a boolean specifying if the attribute must be present.

We use an abstract data type to model the "optional" trait of the issuer element:

```
data IssuerSpecT = AnyIssuer | Issuer IssuerIdT
                      deriving (Eq,Show)
```

The constructor "AnyIssuer" is used when an argument is not supplied, and the constructor "Issuer" followed by the issuer identifier is used to make the element explicit.

<Target>

A target consists of a structured set representative of matching predicates on the Request Context.

```
data TargetT = EmptyTarget
             | Target SubjectsT ResourcesT ActionsT
                    deriving Show
```

For some rules, the target is optional. For this purpose we tag the target with two different constructors to model the difference.

A target is either empty signified by the "EmptyTarget" constructor, or it is constructed with the "Target" constructor followed by a specification for subjects, resources, and actions. Modeling the XML of XACML in Haskell almost directly we get the following:

```
data SubjectsT  = AnySubject  | Subjects  [SubjectTargetT]  deriving Show
data ResourcesT = AnyResource | Resources [ResourceTargetT] deriving Show
data ActionsT   = AnyAction   | Actions   [ActionTargetT]   deriving Show
```

Each of the subjects, resources, and actions sections consist of separate target specifications.

```
data SubjectTargetT  = SubjectTarget [SubjectMatchT]   deriving Show
data ResourceTargetT = ResourceTarget [ResourceMatchT] deriving Show
data ActionTargetT   = ActionTarget [ActionMatchT]     deriving Show
```

We map the targets "Subject", "Resource", and "Action" to "SubjectTarget", "ResourceTarget", and "ActionTarget" respectively since Haskell prohibits using the

same constructor to represent elements of different data types. The matching functions are straight forward. The are differentiated by their tag, and the contain the id of the matching function, an attribute value, and an attribute designator.

```
data SubjectMatchT =
       SubjectMatch  MatchIdT AttributeValueT SubjectAttributeDesignatorT
                     deriving Show

data ResourceMatchT =
       ResourceMatch MatchIdT AttributeValueT ResourceAttributeDesignatorT
                     deriving Show

data ActionMatchT =
       ActionMatch   MatchIdT AttributeValueT ActionAttributeDesignatorT
                     deriving Show
```

The attribute designators are modeled as expressions, as they can appear anywhere in an XACML Apply construct. However, we restrict to the particular attribute designator constructors.

```
type SubjectAttributeDesignatorT     = ExpressionT
type ResourceAttributeDesignatorT    = ExpressionT
type ActionAttributeDesignatorT      = ExpressionT
```

A sample Target follows:

```
target1 :: TargetT
target1 = (Target
             (AnySubject)
             (Resources
                [(ResourceTarget
                     [(ResourceMatch "string-equal"
                          (StringAtom "xacml-document")
                          (ResourceAttributeDesignator "resource-id"
                                 "string" AnyIssuer False))])])
           (Actions
              [(ActionTarget
                   [(ActionMatch "string-equal"
                        (StringAtom "modify")
                        (ActionAttributeDesignator "action-id"
                                 "string" AnyIssuer False))])]))
```

A sample rule follows:

```
rule1 :: RuleT
rule1 = (Rule EmptyTarget
             (Apply "integer-greaterthan"
```

```
                [(Value (IntAtom 200)),
                 (Apply "integer-one-and-only"
                        [(SubjectAttributeDesignator "access-subject"
                            "weight" "integer" (Issuer "Sam") False)])])
            Permit)
```

The above declaration specifies a rule with an empty target, a condition, and a decision.

<Policy>

Having described the structure of rules, just as in XACML, we can reuse our definitions of the target that are used in rules to apply to Policy, as well. This approach fully specifies the structure of a “Policy” quite easily.

```
policy1 :: PolicyT
policy1 = Policy “id-1” target1 “first-applicable” [rule1] []
```

The above specifies a policy with the target defined above, the rule combining algorithm, and a list of one rule, which was also defined above, and an empty list of obligations.

In the above sample of a policy, we use the variable “rule1” to specify a rule. Since Haskell is a declarative language, the expression of applying a constructor to the names of declarations is as if the resulting structure is created itself using the declaration's definition.

Similarly we may specify a policy set much the same way:

```
policyset1 :: PolicyT
policyset1 = PolicySet “id-2” target1 “first-applicable” [policy1] []
```

The above element specifies a policy set with the id “id-2”, with the same target, the policy combining algorithm, a list of one policy, which is also defined above, and an empty list of obligations.

This specification completes the representation of XACML Policy and Policy Sets into a Haskell representation.

Evaluation Semantics

We show the transformations of the Haskell representation of an XACML policy to Haskell functions that operate on the request context, which is the job of the PAP as we described it.

Attribute Designators

The very core of XACML is being able to retrieve the data from the request context so that it may be compared or matched to elements in the policy. In general, we describe any attribute designator, whether it be subject, resource, action, or environment, as a function retrieves a value, namely a Bag of attribute values, or possibly Indeterminate, from the request context.

```
type AttributeDesignatorF = RequestContextT -> ValueT
```

Of an attribute designator the XACML 1.1 specification section 5.27 Complex Type AttributeDesignatorType says:

> A named ***attribute*** SHALL match an ***attribute*** if the values of their respective `AttributeId`, `DataType` and `Issuer` attributes match. The ***attribute*** designator's `AttributeId` MUST match, by URI equality, the `AttributeId` of the ***attribute***. The ***attribute*** designator's `DataType` MUST match, by URI equality, the `DataType` of the same ***attribute***.
>
> If the `Issuer` attribute is present in the ***attribute*** designator, then it MUST match, by URI equality, the `Issuer` of the same ***attribute***. If the `Issuer` is not present in the ***attribute*** designator, then the matching of the ***attribute*** to the named ***attribute*** SHALL be governed by `AttributeId` and `DataType` attributes alone.

We model this semantic by creating function type AttributeRetrieverF, of which functions must take an attribute identifier, a data type identifier, an optional issuer identifier, a boolean value that specifies mustBePresent, applied to a list of attributes and returns a value. That value is restricted to either a "Bag" constructor or Indeterminate (in the case that mustBePresent is True and there is no value).

We define the attribute designator function type by first describing a basic functionality, retrieving attribute values from a particular of attributes.

```
type AttributeRetrieverF =
            AttributeIdT -> DataTypeIdT -> IssuerSpecT -> Bool ->
            [AttributeT] -> ValueT
```

We will suffix types with F that describe functionality as opposed to abstract data types, which we suffixed with T. Different attribute designators will have different specifications for subject, resource, action, and environment.

The semantics of attribute retrieval against a list of attributes is described as follows with the function "attributeRetriever":

```
attributeRetriever :: AttributeRetrieverF
attributeRetriever attributeid datatype issuer mustBePresent attributes =
    if (empty values) && mustBePresent then IndeterminateVal
    else Bag values
        where
            values = getattrs attributeid datatype issuer attributes
```

The above definition says that if we find no attributes in the list and “mustBePresent” is true, then we must return “IndeterminateVal” as a value. The “empty” function is a boolean function that returns True if and only if the list is empty.

```
empty :: [a] -> Bool
empty []     = True
empty (v:vs) = False
```

Technically, the XACML 1.1 specification states that the “mustBePresent” element, may be optional. However, if not present, its value is specified to be False. Rather than dealing with writing two different functions that differ only in arity, we just specify that any application of the “attributeRetriever” function must place False for “mustBePresent” when it said to be absent.

We formally define the semantics of the “getattrs” function. According to the above specification, this function specifies retrieving an attribute value of the correct data type and matching issuer, if supplied, from the list of attributes.

```
getattrs :: AttributeIdT -> DataTypeIdT -> IssuerSpecT ->
            [AttributeT] -> [ValueT]
getattrs aid dt issuer [] = []
getattrs aid dt issuer ((id,issuer_id,instant,val):as) =
      if aid == id
         && isDataType dt val
         && (issuer == AnyIssuer || issuer == Issuer issuer_id)
      then val:( getattrs aid dt issuer as)
      else getattrs aid dt issuer as
```

The first clause declaratively states that if there are no attributes then there are no corresponding values, and therefore the value is the empty list of values. The second clause states that if that attribute identifiers match, the value is of the correct type, and if there is an issuer specification it equals the attributes issuer, then the result is the value of that attribute constructed as a list with the finding of any other matching attributes from the rest of the attributes. Otherwise, it does not match, and the result is the same as finding the matching attributes within the rest of the attributes.

The isDataType function specifies a boolean predicate that checks if the value matches

the data type specification. XACML states that a data type of the attribute value must match for it to be returned. Note, that we are using shortened names for data types, i.e. ones without the URN prefixes, for brevity.

```
isDataType :: DataTypeIdT -> ValueT -> Bool

isDataType "string"     (StringAtom _)      = True
isDataType "boolean"    (BoolAtom _ )       = True
isDataType "integer"    (IntAtom _ )        = True
isDataType "double"     (DoubleAtom _ )     = True
isDataType "datetime"   (DateTimeAtom _  )  = True
isDataType "date"       (DateAtom _ )       = True
isDataType "time"       (TimeAtom _ )       = True
isDataType "hex-binary" (HexBinaryAtom _)   = True
isDataType "base64-binary"       (Base64BinaryAtom _)        = True
isDataType "anyURI"              (AnyURIAtom _)              = True
isDataType "year-month-duration" (YearMonthDurationAtom _ _) = True
isDataType "month-day-duration"  (MonthDayDurationAtom _ _)  = True
isDataType _ _ = False
```

Each clause states a true condition. The last clause, by virtue of being last, states that if no other of the above clauses match the arguments, the result of this predicate False, which specifies that the value is not of the correct type.

The “attributeRetriver” function completely formalizes the semantics for retrieving attributes from a given list of attributes as we have defined them for the Request Context. We use this common definition to specify the matching of attributes for each the Subject, Resource, Action, and Environment.

Subject Attribute Designator

The Subject Attribute Designator is a special case in that it must retrieve attributes from blocks of attribute lists categorized by the subject category identifier. XACML states that if different subject elements in the request context have the same subject category identifier, their collective attribute lists are considered from one subject. From the XACML 1.1 specification, section 5.28 Element <SubjectAttributeDesignator>:

> If there are multiple ***subjects*** with the same `SubjectCategory` xml attribute, then they SHALL be treated as if they were one *categorized* ***subject***.

```
type SubjectAttributeDesignatorF = SubjectCategoryIdT -> AttributeIdT ->
                                   DataTypeIdT -> IssuerSpecT ->
                                   Bool ->
                                   AttributeDesignatorF
```

To capture this semantic we describe the consolidation of attributes across subjects as follows:

```
consolidate :: SubjectCategoryIdT -> [SubjectT] -> [AttributeT]
```

```
consolidate cat [] = []
consolidate cat ((Subject cat' attrs):ss) =
          if (cat == cat') then attrs ++ (consolidate cat ss)
          else consolidate cat ss
```

The first clause states that if there are no subject sections then there are no relevant attribute values. The second clause states that if the subject category identifiers match then the result is the same as appending (infix ++ operator) its list of attributes with the consolidation of attributes with respect to the same subject category identifier of the rest of the subjects.

Finally, the following construction formally describes the semantics of a subject attribute designator using our generalized attribute designator function, which operates on the subjects of the Request Context.

```
subjectAttributeDesignator  :: SubjectCategoryIdT -> AttributeIdT ->
                               DataTypeIdT -> IssuerSpecT -> Bool ->
                               AttributeDesignatorF
subjectAttributeDesignator cat aid dt issuer mbp
        (RequestContext subjects resource action env) =
          attributeRetriever aid dt issuer mbp (consolidate cat subjects)
```

We define the semantics of the Resource, Action, and Environment attribute designators similarly using the the generalized attribute retriever on the respective members of the Request Context.

Resource Attribute Designator

The Resource Attribute Designator has the following semantics:

```
resourceAttributeDesignator :: AttributeIdT ->
                               DataTypeIdT -> IssuerSpecT -> Bool ->
                               AttributeDesignatorF
resourceAttributeDesignator aid dt issuer mbp
        (RequestContext subjects (Resource attrs) action env) =
                               attributeRetriever aid dt issuer mbp attrs
```

Action Attribute Designator

The Action Attribute Designator has the following semantics:

```
actionAttributeDesignator :: AttributeIdT ->
                             DataTypeIdT -> IssuerSpecT -> Bool ->
                             AttributeDesignatorF

actionAttributeDesignator aid dt issuer mbp
        (RequestContext subjects resource (Action attrs) env) =
                           attributeRetriever aid dt issuer mbp attrs
```

Environment Attribute Designator

The Environment Attribute Designator has the following semantics:

```
environmentAttributeDesignator :: AttributeIdT ->
                                  DataTypeIdT -> IssuerSpecT -> Bool ->
                                  AttributeDesignatorF

environmentAttributeDesignator aid dt issuer mbp
        (RequestContext subjects resource action (Environment attrs)) =
                                 attributeRetriever aid dt issuer mbp attrs
```

Request Predicates

The semantics of an XACML target or condition is to provide a predicate on the request context. We describe the type of a predicate function on the request as follows:

```
type PredicateF = RequestContextT -> BooleanT
type BooleanT = ValueT
```

We specify the type “BooleanT” is synonymous with “ValueT” only to state that the evaluation of the predicate may return a value of “IndeterminateVal”. When we use this type, we are expecting a boolean value with the “BoolAtom” constructor, or “IndeterminateVal”.

Target

A target is a predicate on the request, but it is made up of several parts, which are the predicates containing matches on the Subjects, Resource, and Action of the request context.

Each of these match sections contain lists of elements that specify a disjunctive sequence of conjunctive sequences of matches.

We describe the functions that specify the semantics of a disjunctive or conjunctive sequence by type, and leave the formal definition of its semantics until later. We model the semantics of a conjunctive sequence and disjunctive sequence using the ConjunctionF and DisjunctionF types respectively.

```
type ConjunctionF = [BooleanT] -> BooleanT
type DisjunctionF = [BooleanT] -> BooleanT
```

These types state that conjunction and disjunction functions will take a list of boolean values and produce a boolean value possibly with the introduction of Indeterminate not only as a result but as an actual argument to the function. XACML has specific semantics when dealing with Indeterminate as an argument in resolving a conjunctive or disjunctive sequence. We describe those semantics when we define the “conjunction” and “disjunction” functions later in this document.

Target Semantics

The XACML 1.1 section 5.5 Element <Target> states the semantics of the target as follows:.

> The `<Target>` element SHALL contain a ***conjunctive sequence*** of `<Subjects>`, `<Resources>` and `<Actions>` elements. For the parent of the `<Target>` element to be applicable to the ***decision request***, there MUST be at least one positive match between each section of the `<Target>` element and the corresponding section of the `<xacml-context:Request>` element.

Modeling each of the Subjects, Resources, and Actions sections of a target as a predicate on the request context, we formally construct a function from our definition of conjunction.

We describe the function of a target as follows:

```
type TargetF = SubjectsF -> ResourcesF -> ActionsF -> PredicateF
```

The formal semantics of this target function is specified by the “targetSemantics” function as follows:

```
targetSemantics :: TargetF
targetSemantics subf resf actf req =
                     conjunction [(subf req),(resf req),(actf req)]
```

Formally, we apply each of the predicates that represent each of the Subjects, Resources, and Actions sections to the request context, and combine the results as a conjunctive sequence using the “conjunction” function.

Target Subjects

A SubjectsF is a predicate on the request. The XACML 1.1 Section 5.6 Element <Subjects> states the following:

> The <Subjects> element SHALL contains a ***disjunctive sequence*** of <Subject> elements.

We formally define a function called “subjects” that functionally constructs the predicate as a disjunctive sequence of “SubjectF” predicates.

```
type SubjectsF = PredicateF

subjects :: [SubjectF] -> SubjectsF
subjects subs req = disjunction (map (\s -> s req) subs)
```

To explain the semantics, the “subjects” function takes a list of “subject” predicates and applies each one to the request. The result is processed as a disjunctive sequence, using the “disjunction” function.

The function “map” takes as its first argument, a function, and applies it to each element in the second argument, which is a list. The result is a one to one list of the results of applying the function.

The notation, “(\s -> s req)”, is Haskel notation for a function specification as an expression. For those familiar with the “Lambda calculus” the backslash is the “lambda”. It means that the letter after the lambda is a formal parameter, and after the arrow, i.e. “->”, is the expression that may use that parameter. This specific function expression takes a subject predicate applies it to the request. The result, of course, will be of the “BooleanT” type. The map function applies this function specification to each element of the list of subject predicates, and therefore, results in a list of “BooleanT” elements.

Target Subject

A function of type “SubjectF” is a predicate on the request. The XACML 1.1 Specification Section 5.7 Element <Subject> states:

> The <Subject> element SHALL contain a ***conjunctive sequence*** of <SubjectMatch> elements.

We define a function called “subject” that functionally constructs the predicate as a conjunctive sequence of matching functions.

```
type SubjectF = PredicateF


subject :: [PredicateF] -> SubjectF
subject matches req = conjunction (map (\m -> m req) matches)
```

To explain the semantics, the “subject” function takes a list of “subject match” predicates and applies each one to the request. The result is processed as a conjunctive sequence, using the “conjunction” function.

To explain the notation and semantics of the “map” function, please see the above section on “Subjects”.

Target Resources

A function of type “ResourcesF” is a predicate on the request. The XACML 1.1 Specification section 5.10 Element <Resources> states:

> The `<Resources>` element SHALL contain a ***disjunctive sequence*** of `<Resource>` elements.

We define a function called “resources” that functionally constructs the predicate as a disjunctive sequence of “ResourceF” functions.

```
type ResourcesF = PredicateF

resources :: [ResourceF] -> ResourcesF
resources res req = disjunction (map (\s -> s req) res)
```

To explain the semantics, the “resources” function takes a list of “resource” predicates and applies each one to the request. The result is processed as a disjunctive sequence, using the “disjunction” function.

To explain the notation and semantics of the “map” function, please see the above section on “Subjects”.

Target Resource

A function of type “ResourceF” is a predicate on the request. The XACML 1.1 Specification Section 5.11 Element <Resource> states the semantics of the resource element as follows:

> The `<Resource>` element SHALL contain a ***conjunctive sequence*** of `<ResourceMatch>` elements.

We define a function called “resource” that functionally constructs the predicate as a conjunctive sequence of matching functions.

```
type ResourceF = PredicateF

resource :: [PredicateF] -> ResourceF
resource matches req = conjunction (map (\m -> m req) matches)
```

To explain the semantics, the “resource” function takes a list of “resource match” predicates and applies each one to the request. The result is processed as a conjunctive sequence, using the “conjunction” function.

To explain the notation and semantics of the “map” function, please see the above section on “Subjects”.

Target Actions

A function of type “ActionF” is a predicate on the request. The XACML 1.1 Specification Section 5.14 Element <Actions> states the semantics of the Actions element as follows:

> The `<Actions>` element SHALL contain a ***disjunctive sequence*** of `<Action>` elements.

We define a function called “actions” that functionally constructs the predicate as a disjunctive sequence of “ActionF” functions.

```
type ActionsF = PredicateF

actions :: [ActionF] -> ActionsF
actions subs req = disjunction (map (\s -> s req) subs)
```

To explain the semantics, the “actions” function takes a list of “action” predicates and applies each one to the request. The result is processed as a disjunctive sequence, using the “disjunction” function.

To explain the notation and semantics of the “map” function, please see the above section on “Subjects”.

Target Action

A function of type “ActionF” is a predicate on the request. The XACML 1.1 Specification Section 5.15 Element <Action> states the semantics of the Action element as follows:

> The `<Action>` element SHALL contain a ***conjunctive sequence*** of `<ActionMatch>` elements.

We define a function called “action” that functionally constructs the predicate as a conjunctive sequence of matching functions.

```
type ActionF = PredicateF

action :: [PredicateF] -> ActionF
action matches req = conjunction (map (\m -> m req) matches)
```

To explain the semantics, the “action” function takes a list of “action match” predicates and applies each one to the request. The result is processed as a conjunctive sequence,

using the “conjunction” function.

To explain the notation and semantics of the “map” function, please see the above section on “Subjects”.

Conjunctive and Disjunctive Sequences

To complete semantics for target we need to define the semantics for the “conjunction” and “disjunction” functions that we have introduced by type above giving formal semantics to “conjunctive sequence” and “disjunctive sequence” respectively.

XACML defines “conjunctive sequence” and “disjunctive sequence” only in Section 1.1.1 Preferred Terms of the Glossary, and they are specified as follows:

> ***Conjunctive sequence*** - a sequence of boolean elements combined using the logical ‘AND’ operation.
>
> ***Disjunctive sequence*** - a sequence of boolean elements combined using the logical ‘OR’ operation

This part of the XACML specification is non-normative. However, we will take it as face value and specify “conjunction” as the “logical_and” function, and “disjunction” as the “logical_or” function.

```
conjunction :: [BooleanT] -> BooleanT
conjunction = logical_and

disjunction :: [BooleanT] -> BooleanT
disjunction = logical_or
```

Note: The semantics of “logical_and” and “logical_or” functions are defined in the Appendix under “Logical Functions”. However, taking tact presents a problem in defining the semantics of handling IndeterminateVal, which has explicit concerns on strict evaluation of its elements and forcing full evaluation of every element in the target. Currently, the XACML specification is ambiguous or unclear on which approach to take. For now, it seems that all elements of the Target must be evaluated to see if there is an IndeterminateVal returned (perhaps from a missing attribute). Please see the Appendix for details.

Matching Functions

The matching function is a predicate function that takes into account the ability to select attribute values from the request context and compare them to a specific value by some named function. For example, the XACML 1.1. Specification Section 5.13 Element <ResourceMatch> states the semantics of the ResourceMatch element as follows:

> The `<ResourceMatch>` element SHALL identify a set of ***resource***-related

entities by matching ***attribute*** values in the `<xacml-context:Resource>` element of the ***context*** with the embedded ***attribute*** value.

The `<ResourceMatch>` element contains the following attributes and elements:

`MatchId` [Required]

Specifies a matching function. Values of this attribute MUST be of type **xs:anyURI**, with legal values documented in Section .

`<AttributeValue>` [Required]

Embedded ***attribute*** value.

`<ResourceAttributeDesignator>` [Required Choice]

Identifies one or more ***attribute*** values in the `<Resource>` element of the ***context***.

In general, whether we are taking about Subject, Resources, or Actions, the match functions all have similar semantics. The common matching function is a function that takes a value matching function, an explicit attribute value, and an attribute designator, applies it to the request, and returns a boolean value.

```
type MatchF =
     ValueMatchF  -> AttributeValueT -> AttributeDesignatorF -> PredicateF
```

The "ValueMatchF" function that is applied for the match can be any function that takes two arguments and returns a boolean. To be consistent with functions of arbitrary arity, although that list will only contain two elements, we use the following definition:

```
type ValueMatchF = [ValueT] -> BooleanT
```

Note: The XACML 1.1 Specification only implies by Section 5.5 Element Target that a match function returns a positive or negative result. The specification does not normatively state the requirements for a positive or negative Subject, Resource, or Action match. One can only imply that the evaluation semantics will be True if an only if one match one is found to be true. This situation may yield ambiguous behavior from differing implementations with different evaluation strategies.

In general, we describe the semantics of the matching function is described as follows:

```
match :: MatchF
match valmatchf val designator request = traverse (designator request)
```

```
    where traverse (Bag [])         = BoolAtom False
          traverse (Bag (v:vs))     = check (valmatchf [val,v])
                                            (traverse (Bag vs))
          traverse IndeterminateVal = IndeterminateVal

          check (BoolAtom True)  _                  = BoolAtom True
          check _                (BoolAtom True)    = BoolAtom True
          check (BoolAtom False) IndeterminateVal   = IndeterminateVal
          check IndeterminateVal (BoolAtom False)   = IndeterminateVal
```

Since the bag is not supposed to have an order, the check function result should be unified in all evaluation orders, i.e. permutations of list. This requirement means one of two things for evaluation specification.

Since matches only occur in the target, they can be thought of as positive seeking. One positive match in a disjunction with an indeterminate takes precedence over the indeterminate. However, a negative match does not.

Alternatively, one can state that an indeterminate takes precedence over all values, and the result is indeterminate. This approach forces full evaluation of all matches against the values in the bag.

Above we have chosen the first approach, where a positive match takes precedence.

The traverse function traverse across the elements of the bag checking to see if the "valmatchf" function is true. To be consistent, once a true is found,

Note: This semantics is consistent with respect to True taking precedence; however, there may be some ambiguous requirements that if indeterminate arises from a single application of the match function an indeterminate should result.

The definition of the semantics of each match, whether it be a SubjectMatch, ResourceMatch, ActionMatch, are the same. The kind of match is only dependent on the supplied designator.

Expressions

Expressions are formalized as functions on the request that return a value consistent with the type of their function.

```
type ExpressionF = RequestContextT -> ValueT
```

Each explicit value is represented as a constant function against the request context.

```
constVal :: ValueT -> ExpressionF
constVal v req = v
```

An XACML function is formalized as:

```
type FunctionF = [ValueT] -> ValueT
```

A function of the type “FunctionF” takes a list of values and produces a single value . An example of such a function would be on that adds two numbers:

```
integer_subtract [(IntAtom x),(IntAtom y)] = IntAtom (x-y)
integer_subtract  _                        = IndeterminateVal
```

To explain the above Haskell notation, the first clause states that the argument to this function shall be a list of two integer values, no more, no less, and the result will be expression containing the integer value where the argument values are subtracted. The second clause is for when the arguments are not what it expected, and XACML states that the result shall be Indeterminate in that case for this particular function.

Note: That if strict checking of an XACML policy and evaluation model is enforced upon the request context the second clause may be proved never to be needed, and the seemingly “dynamic” argument and type checking can be factored away. However, for XACML interpreters that do not type check their policy, this semantic specification is supplied bringing them runtime value type checking.

An application of a function is defined as a function that takes a function that takes a list of expressions for arguments and produces a result by applying each of those expressions to the request, and then applying the given function to those results.

```
apply :: FunctionF -> [ExpressionF] -> ExpressionF
apply f vs req = f (map (\v -> v req) vs)
```

To explain the notation and semantics of the “map” function, please see the above section on “Subjects”.

Given this approach, and definitions for all XACML functions in Haskell, which is in the appendix, one can formally specify the semantics of any expression in XACML. Please see the section on compiling XACML expressions below.

Rule Function

A rule function is a function that models an XACML rule. It has the following type:

```
type RuleF   = RequestContextT -> DecisionT
```

A rule constructor may take a target predicate, a condition expression, a decision data types and specify the semantics of an XACML rule. The XACML 1.1 Specification Section 5.22 <Rule> states the semantics of a rule as follows:

> The <Rule> element SHALL define the individual ***rules*** in the ***policy***. The main components of this element are the <Target> and <Condition> elements and the Effect attribute.
>
> The <Rule> element contains the following attributes and elements:
>
> Effect [Required]
> ***Rule effect***. Values of this attribute are either “Permit” or “Deny”.
>
> <Target> [Optional]
> Identifies the set of ***decision requests*** that the <Rule> element is intended to evaluate. If this element is omitted, then the ***target*** for the <Rule> SHALL be defined by the <Target> element of the enclosing <Policy> element. See Section for details.
>
> <Condition> [Optional]
> A ***predicate*** that MUST be satisfied for the ***rule*** to be assigned its Effect value. A ***condition*** is a boolean function over a combination of ***subject***, ***resource, action*** and ***environment attributes*** or other functions.

The semantics of the evaluation of a rule is to evaluate the target predicate and the condition to determine if the effect.

The XACML 1.1 Specification Section 7.4 Condition Evaluation states:

> The ***condition*** value SHALL be "True" if the <Condition> element is absent, or if it evaluates to "True" for the ***attribute*** values supplied in the request ***context***. Its value is "False" if the <Condition> element evaluates to "False" for the ***attribute*** values supplied in the request ***context***. If any ***attribute*** value referenced in the ***condition*** cannot be obtained, then the ***condition*** SHALL evaluate to "Indeterminate".

The XACML 1.1 Specification Section 7.5 Rule Evaluation states:

> If the ***target*** value is "No-match" or “Indeterminate” then the ***rule*** value SHALL be “NotApplicable” or “Indeterminate”, respectively, regardless of the value of the ***condition***. For these cases, therefore, the ***condition*** need not be evaluated in order to determine the ***rule*** value.
>
> If the ***target*** value is “Match” and the ***condition*** value is “True”, then the ***effect*** specified in the ***rule*** SHALL determine the ***rule*** value.

In the above description, “No-Match” is synonymous with False, and “Match” is synonymous with true.

Therefore, we model these semantics of evaluating a rule function is described as follows with the target predicate as the first argument, and the condition predicate as the second argument.

```
ruleSemantics :: PredicateF -> PredicateF -> EffectT -> RuleF

ruleSemantics targetf condf effect req =
    check (targetf req) (check (condf req) effect)

    where check (BoolAtom True)  decision = decision
          check (BoolAtom False) _        = NotApplicable
          check _                _        = Indeterminate
```

Since XACML 1.1. Specification states that the Target and Condition elements are specified as optional we must model that with predicates that true true regardless.

The “EffectT” type is synonymous with the “DecisionT” type, but its values are restricted to the “Permit” and “Deny” constructors.

The above definition formally describes that a target will be checked first. If the application of the target is false, then the result is NotApplicable. If the result is True, then the condition is checked in the same manner. If the target or condition evaluate to Indeterminate (or any other unexpected value, as such with no type checking) the result is specified to be indeterminate.

Rule Combining Algorithm Function

The rule combining algorithms have somewhat of a complex nature besides just combining the resulting decisions from each applied rule, as they take into account the potential effects of other rules. We model the common functionality of a rule combining algorithm function with the following type:

```
type RuleCombiningAlgF = [(EffectT,RuleF)] -> RequestContextT -> DecisionT
```

The first argument is a list containing the rules, but each listed with its effect. Rule combining algorithms such as “deny-overrides” take into account the potential effect of the rules when determining the combination in the face of Indeterminate.

The formal semantics for the standard XACML rule combining algorithms are presented in the Appendix.

Policy Function

The XACML 1.1 Specification Section 7.6 Policy Evaluation states:

> The ***policy's target*** SHALL be evaluated to determine the applicability of the ***policy***. If the ***target*** evaluates to "Match", then the value of the ***policy*** SHALL be determined by evaluation of the ***policy's rules***, according to the specified ***rule-combining algorithm***. If the ***target*** evaluates to "No-Match", then the value of the ***policy*** SHALL be "NotApplicable". If the ***target*** evaluates to "Indeterminate", then the value of the ***policy*** SHALL be "Indeterminate".

The value “Match” is synonymous with true, and “No-Match” is synonymous with false. However, polices when evaluated return not only a decision, but may return obligations as well. A policy evaluation function is modeled with the following type:

```
type PolicyF = RequestContextT -> ObligatedDecisionT
type ObligatedDecisionT = (DecisionT, [ObligationT])
```

XACML states that the evaluation of a policy is equivalent to evaluating the policy target and if the policy target is true, the policy is evaluated according to is rule combining algorithm and rules.

The same section in the XACML 1.1 Specification states:

> A Rules value of "At-least-one-applicable" SHALL be used if the <Rule> element is absent, or if one or more of the ***rules*** contained in the ***policy*** is applicable to the ***decision request*** (i.e., returns a value of “Effect”; see Section). A value of “None-applicable” SHALL be used if no ***rule*** contained in the ***policy*** is applicable to the request and if no ***rule*** contained in the ***policy*** returns a value of “Indeterminate”. If no ***rule*** contained in the ***policy*** is applicable to the request but one or more ***rule*** returns a value of “Indeterminate”, then ***rules*** SHALL evaluate to "Indeterminate".

This paragraph seems to lay a restriction on the rule combining algorithms that they shall always operate in a consistent manner in the face of no supplied rules, or rules that evaluate to Indeterminate. Rather than lay out special cases here for policy rule evaluation and combination, we leave these semantic requirements to be followed in each of the formal semantics of the rule combining algorithm functions described in the Appendix.

The same section in the XACML 1.1 Specification states about the policy itself:

> If the ***target*** value is "No-match" or “Indeterminate” then the ***policy*** value SHALL be “NotApplicable” or “Indeterminate”, respectively, regardless of the value of the ***rules***. For these cases, therefore, the ***rules*** need not be evaluated in order to determine the ***policy*** value.
>
> If the ***target*** value is “Match” and the ***rules*** value is “At-least-one-applicable” or “Indeterminate”, then the ***rule-combining algorithm*** specified in the ***policy*** SHALL determine the ***policy*** value.

The evaluation of a Policy is formally described by the following function constructing the policy function abstractly from arbitrary target functions, combinator functions, rule functions, and obligations. We must take into account the obligations as follows:

> A ***policy*** or ***policy set*** may contain one or more ***obligations***. When such a ***policy*** or ***policy set*** is evaluated, an ***obligation*** SHALL be passed up to the next level of evaluation (the enclosing or referencing ***policy set*** or ***authorization decision***)

only if the ***effect*** of the ***policy*** or ***policy set*** being evaluated matches the value of the `xacml:FulfillOn` attribute of the ***obligation***.

We model the policy evaluation semantics with the following function:

```
policySemantics :: PredicateF -> RuleCombiningAlgF -> [(EffectT,RuleF)] ->
                   [ObligationT] -> PolicyF

policySemantics targetf combf rules obligations req =

          check (targetf req) (oblige (combf rules req))

    where check (BoolAtom False) _      = (NotApplicable,[])
          check (BoolAtom True)  result = result
          check _                _      = (Indeterminate,[])

          oblige effect = (effect,relevant effect obligations)

          relevant _             [] = []
          relevant Indeterminate _  = []
          relevant decision ((effect,obl):obls) =
                   if decision == effect
                       then (effect,obl) : relevant decision obls
                       else relevant decision obls
```

The first clause of the “check” function states that the target predicate that is applied to the request is not true then the result is “NotApplicable” with no obligations. If the target predicate applied to the request context evaluates to true, then the result is the application of the combining function on the list of results made by applying each rule to the request obliged to contain any relevant obligations. If the application of the target predicate evaluates to Indeterminate the result is Indeterminate with no obligations.

The “relevant” function formally specifies the obligations that are relevant with respect to the policy decision. The “relevant” function selects from the list of obligations the ones that apply to the decision, of either Permit or Deny.

Note: According to these formal semantics, a policy evaluation will never return obligations that are not relevant to the decision. This point is important when considering the policy combining algorithms.

Policy Combining Algorithm Function

Likewise with a rule combining algorithm function, the policy combining algorithms have a complex nature besides just combining the resulting decisions from each applied policy. At least one standard combining algorithm, “only-one-applicable”, only evaluates the targets of a policy before it decides to evaluate any policy in full. Therefore, this approach must be enabled for the type of policy combining algorithm functions. Also, a policy combining algorithm must take into account the obligations that are associated with subordinate policies. We model the common functionality of a policy combining algorithm function with the following type:

```
type PolicyCombiningAlgF =
          [(PredicateF,PolicyF)] -> [ObligationT] -> RequestContextT ->
ObligatedDecisionT
```

The first argument is a list containing the policy functions, but each listed with its respective target. This semantic is an implicit requirement from one of the standard XACML Policy combining algorithms such as “only-one-applicable”. This combining algorithm evaluates the target of each of the policies when determining which policy to evaluate.

The formal semantics for the standard XACML policy combining algorithms are presented in the Appendix.

Policy Set Function

The policy set function is much like the policy function with two minor differences. The combining algorithms apply to policies instead of rules.

The XACML 1.1 Specification Section 7.7 Policy Set Evaluation states:

> The ***policy set's target*** SHALL be evaluated to determine the applicability of the ***policy set***. If the ***target*** evaluates to "Match" then the value of the ***policy set*** SHALL be determined by evaluation of the ***policy set's policies*** and ***policy sets***, according to the specified ***policy-combining algorithm***. If the ***target*** evaluates to "Not-Match", then the value of the ***policy set*** shall be "NotApplicable". If the ***target*** evaluates to "Indeterminate", then the value of the ***policy s***et SHALL be "Indeterminate".

Without reprinting the section on Policy Set Evaluation, the general evaluation strategy is functionally equivalent to evaluating policies like the rules. returning the relevant obligations. The relevant obligations are determined by the policy combining algorithm function.

```
type PolicySetF = RequestContextT -> ObligatedDecisionT

policySetSemantics :: PredicateF -> PolicyCombiningAlgF ->
                [(PredicateF,PolicyF)] -> [ObligationT]-> PolicySetF

policySetSemantics targetf combf policies obls req =

          check (targetf req) (combf policies obls req)

    where check (BoolAtom False) _      = (NotApplicable,[])
          check (BoolAtom True)  result = result
          check _                _      = (Indeterminate,[])
```

The astute Haskell reader will notice that the types. “PoliicyF” and “PolicySetF” are equivalent.

The PDP Function

A PDP may be generated by taking a function of type “PolicyF” or “PolicySetF” function (since they are equivalent in type) and produce a PDP function.

```
pdpf :: PolicyF -> PDP
pdpf policyf req =
          check (policyf req)
    where check (Permit,obls) =
                ResponseContext
                        [Result Permit (Status "ok" "" "") obls]
          check (Deny,obls)   =
                ResponseContext
                        [Result Deny (Status "ok" "" "") obls]
          check (NotApplicable,[]) =
                ResponseContext
                        [Result NotApplicable (Status "ok" "" "") []]
          check (Indeterminate, []) =
                ResponseContext
                        [Result Indeterminate (Status "error?" "" "") []]
```

XACML has not yet formalized the specific semantics of status, and this situation is quite obvious from looking at the formal description.

Converting the Policy Description

We have formally specified the abstract data types that model the XACML Request Context, Response Context, Policy, Policy Set, which are straight forwardly convertible to their Haskell counter parts. We have also formally specified the semantics of evaluating aspects of XACML as functions. The correlations between those functions and their purpose over the XACML context is intuitive, yet we formalize this with a function that takes an instance of the “PolicyT” data type and returns a function of type “PolicyF” according to the formally specified functions in the last section.

Function Environment

XACML has a number of functions that are named by URN identifiers of which their semantics are defined by other standards, or in the document itself. There must be some way of retrieving the corresponding Haskell function that the identifier represents. We use the following function:

```
get :: [(String,b)]-> String -> b
get ((a,v):vs) a' = if  a == a' then v else get vs a'
```

Note that a clause for the empty list is not declared for “get”. Therefore, if the identifier is not in the list of tuples. This stipulates that this particular application of this function

would fail. Therefore, we do not model policies that have undefined functions.

Our Function Environment consists of a list of identifier, FunctionF pairs.

```
type FunctionEnv = [(String, FunctionF)]
fenv :: FunctionEnv
menv :: FunctionEnv
funcEnv :: String -> FunctionF
funcEnv = get fenv
matchEnv :: String -> FunctionF
matchEnv = get menv
```

Combinator Environments

Rule and policy combinators are also contained in environments using the same identifier, function pairing mechanism..

```
type RuleCombinatorEnv = [(String,RuleCombiningAlgF)]
ruleCenv :: RuleCombinatorEnv
ruleCombinator :: String -> RuleCombiningAlgF
ruleCombinator = get ruleCenv

type PolicyCombinatorEnv = [(String, PolicyCombiningAlgF)]
policyCenv :: PolicyCombinatorEnv
policyCombinator :: String -> PolicyCombiningAlgF
policyCombinator = get policyCenv
```

The elements “ruleCenv” and “policyCenv” are described in the Appendix.

Compiling a Policy

Compiling an instance of a “PolicyT” data type to a function of type “PolicyF” is straight forward with the following functions.

```
compilePolicy :: PolicyT -> (PredicateF,PolicyF)
compilePolicy (Policy id target comb rules obls) = (targ,policyf)
    where targ = compileTarget target
          policyf = policySemantics targ
                                    (ruleCombinator comb)
                                    (map compileRule rules) obls
```

```
compilePolicy (PolicySet id target comb policies obls) = (targ,policyf)
    where targ = compileTarget target
          policyf = policySetSemantics targ
                                    (policyCombinator comb)
                                    (map compilePolicy policies) obls
```

The “compilePolicy” function compiles both “Policy” and “PolicySet” constructions. It returns a tuple containing the policy or policy sets compiled target, and the compiled policy function. The Policy function for a Policy construction is an application of the “policySemantics” function applied to the compiled target, retrieved rule combinator and a list of compiled rules, and last but not least, the obligations. The Policy function for a “PolicySet” construction is an application of the “policySetSemantics” function applied to the compiled target, retrieved policy combinator and a list of compiled policies, and last but not least, the obligations.

```
compileRule (Rule target cond effect) = (effect,rulef)
    where
      rulef = ruleSemantics (compileTarget target)
                            (compileExpression cond) effect
```

The “compileRule” function applies the “ruleSemantics” function to the compiled target, the compiled expression for the condition, and the effect.

```
compileTarget EmptyTarget            = constVal (BoolAtom True)
compileTarget (Target subs res acts) =
             targetSemantics (compileSubjects subs)
                             (compileResources res)
                             (compileActions acts)
```

The “compileTarget” function returns a function that is constantly true for the “EmptyTarget”. For an explicitly defined target, the result is defined by applying the “targetSemantics” to the compiled version of the Subjects, Resources, and Actions sections.

```
compileSubjects (Subjects subs) =
             subjects (map compileSubjectTarget subs)
compileSubjects (AnySubject) = constVal (BoolAtom True)

compileSubjectTarget (SubjectTarget matches) =
             subject (map compileSubjectMatch matches)

compileSubjectMatch (SubjectMatch id val desig) =
     match (matchEnv id) val (compileExpression desig)
```

The “compileSubjects” function returns a function that is constantly true for the “AnySubject”. For an explicitly defined subjects section, the result is defined by applying the “subjects” function to a list containing the compiled version of each target subject. Compiling a “SubjectTarget” construction applies the “subject” function to a list

containing the compiled subject matches. Finally, compiling a “SubjectMatch” construction applies the “match” function to the retrieved matching function, embedded attribute value, and the complied version of the attribute designator.

```
compileResources (Resources res) =
             resources (map compileResourceTarget res)
compileResources (AnyResource) = constVal (BoolAtom True)

compileResourceTarget (ResourceTarget matches) =
             resource (map compileResourceMatch matches)

compileResourceMatch (ResourceMatch id val desig) =
     match (matchEnv id) val (compileExpression  desig)
```

The “compileResources” function returns a function that is constantly true for the “AnyResource”. For an explicitly defined subjects section, the result is defined by applying the “subjects” function to a list containing the compiled version of each target subject. Compiling a “ResourceTarget” construction applies the “subject” function to a list containing the compiled resource matches. Finally, compiling a “ResourceMatch” construction applies the “match” function to the retrieved matching function, embedded attribute value, and the complied version of the attribute designator.

```
compileActions (Actions acts) =
             actions (map compileActionTarget acts)
compileActions (AnyAction) = constVal (BoolAtom True)

compileActionTarget (ActionTarget matches) =
             action (map compileActionMatch matches)

compileActionMatch (ActionMatch id val desig) =
     match (matchEnv id) val (compileExpression  desig)
```

The “compileActionss” function returns a function that is constantly true for the “AnyAction”. For an explicitly defined subjects section, the result is defined by applying the “subjects” function to a list containing the compiled version of each target subject. Compiling a “ActionTarget” construction applies the “subject” function to a list containing the compiled action matches. Finally, compiling a “ResourceMatch” construction applies the “match” function to the retrieved matching function, embedded attribute value, and the complied version of the attribute designator.

```
compileExpression (Value x) = constVal x
compileExpression (Apply id exps) =
                           apply (funcEnv id) (map compileExpression exps)
compileExpression (SubjectAttributeDesignator cat id dt issuer mbp) =
                          subjectAttributeDesignator cat id dt issuer mbp
compileExpression (ResourceAttributeDesignator id dt issuer mbp) =
                           resourceAttributeDesignator id dt issuer mbp
compileExpression (ActionAttributeDesignator id dt issuer mbp) =
                           actionAttributeDesignator id dt issuer mbp
compileExpression (EnvironmentAttributeDesignator id dt issuer mbp) =
                           environmentAttributeDesignator id dt issuer mbp
```

Compiling an Expression is straight forward.

The PAP function

A PAP complies the policy and we apply the function that specifies status to make the PDP.

```
pap ::  PAP
pap policyt = pdpf policyf
                where (targetf,policyf) = (compilePolicy policyt)
```

Conclusions

We present the formal semantics of XACML. Instead of standard denotational semantics, we use a functional, declarative programming language, Haskell. This approach gives us the benefit of a standard syntax, and advantage of all the formal theory already behind the Haskell language.

Some important points, where the ambiguities lie, are called out as a result of this formal analysis.

Future Work

This document will serve as a way in which the community may discuss XACML with respect to its semantics. When new versions of XACML gain new features, using these semantics, perhaps properties can be proved about policies in general, or even a specific policy.

References

[DS] D. Eastlake et al., *XML-Signature Syntax and Processing*, **http://www.w3.org/TR/xmldsig-core/**, World Wide Web Consortium.

[Hancock] Hancock, "Polymorphic Type Checking", in Simon L. Peyton Jones,
"Implementation of Functional Programming Languages", Section 8,
Prentice-Hall International, 1987

[Haskell] Haskell, a purely functional language. Available at **http://www.haskell.org/**

[Hinton94] Hinton, H, M, Lee,, E, S, The Compatibility of Policies, Proceedings 2nd ACM Conference on Computer and Communications Security, Nov 1994, Fairfax, Virginia, USA.

[SAML] Security Assertion Markup Language available from

http://www.oasis-open.org/committees/security/#documents

[Sloman94] Sloman, M. Policy Driven Management for Distributed Systems. Journal of Network and Systems Management, Volume 2, part 4. Plenum Press. 1994.

Appendix

Functions

We will explain via Haskell the semantics that are specified in the XACML document. Functions that refer to other standards and are explicitly clear, we will assume that they are primitive functions, such as string equality. These functions include all matching functions, arithmetic functions, string conversion functions, primitive type conversion functions, arithmetic comparison functions, date and time functions, non numeric comparison functions, set functions, and the special match functions.

In this section we do define the logical functions, the bag functions, and the higher order bag functions that are explicitly defined in the XACML specification. In the XACML specification the higher order bag functions are already specified in Haskell, however, we will respecify them here in the context the scheme we used for our semantics.

Logical Functions

We will formally describe the logical functions as specified in the XACML document.

Logical Or

The logical OR function takes a list of boolean values and returns a boolean value according to standard propositional logic use of OR. The XACML 1.1 Specification Section A.14.5 Logical Functions states the semantics of “or” as follows:

> This function SHALL return "False" if it has no arguments and SHALL return "True" if one of its arguments evaluates to "True". The order of evaluation SHALL be from first argument to last. The evaluation SHALL stop with a result of "True" if any argument evaluates to "True", leaving the rest of the arguments unevaluated. In an expression that contains any of these functions, if any argument is "Indeterminate", then the expression SHALL evaluate to "Indeterminate".

The description states that the order of evaluation shall be from first to last and if any element evaluates to True the evaluation shall stop. However, it does state that if any element is deemed to evaluate to indeterminate, the result shall be indeterminate.

Note: This situation is ambiguous because “leaving the rest of the arguments unevaluated” and the last clause states that all arguments really must already be evaluated before applying the function. The last sentence implies strict evaluation for all arguments regardless.

We will model this function with the “fold left” standard Haskell function, which evaluates every argument since it must traverse the entire list of its argument.

```
logical_or :: FunctionF
logical_or = foldl f (BoolAtom False)
     where
         f IndeterminateVal _                 = IndeterminateVal
         f _                IndeterminateVal  = IndeterminateVal
         f (BoolAtom a)     (BoolAtom b)      = BoolAtom (a || b)
         f _                 _                = IndeterminateVal
```

Note: This semantics specifies that Indeterminate takes precedence over an evaluation of True.

Logical And

The logical and function takes a list of boolean values and returns a boolean value according to standard propositional logic. The XACML 1.1 Specification Section A.14.5 Logical Functions states the semantics of “and” as follows:

> This function SHALL return "True" if it has no arguments and SHALL return "False" if one of its arguments evaluates to "False". The order of evaluation SHALL be from first argument to last. The evaluation SHALL stop with a result of "False" if any argument evaluates to "False", leaving the rest of the arguments unevaluated. In an expression that contains any of these functions, if any argument is "Indeterminate", then the expression SHALL evaluate to "Indeterminate".

The description states that the order of evaluation shall be from first to last and if any element evaluates to False the evaluation shall stop. However, it does state that if any element is deemed to evaluate to Indeterminate, the result shall be indeterminate.

Note: This situation is ambiguous because “leaving the rest of the arguments unevaluated” and the last clause states that all arguments really must already be evaluated before applying the function. The last sentence implies strict evaluation for all arguments regardless.

We model this function with the “fold left” standard Haskell function, which evaluates every argument since it must traverse the entire list of its argument.

```
logical_and :: FunctionF
```

```
logical_and = foldl f (BoolAtom True)
     where
        f IndeterminateVal _                 = IndeterminateVal
        f _                IndeterminateVal = IndeterminateVal
        f (BoolAtom a)     (BoolAtom b)      = BoolAtom (a && b)
        f _                _                 = IndeterminateVal
```

Note: This semantics specifies that Indeterminate takes precedence over an evaluation of False.

Logical N-Of

The logical n-of function requires that there be a certain number of Trues in a list of arguments. The XACML 1.1 Specification Section A.14.5 Logical Functions states the semantics of “n-of” as follows:

> The first argument to this function SHALL be of data-type “http://www.w3.org/2001/XMLSchema#integer”, specifying the number of the remaining arguments that MUST evaluate to "True" for the expression to be considered "True". If the first argument is 0, the result SHALL be "True". If the number of arguments after the first one is less than the value of the first argument, then the expression SHALL result in "Indeterminate". The order of evaluation SHALL be: first evaluate the integer value, then evaluate each subsequent argument. The evaluation SHALL stop and return "True" if the specified number of arguments evaluate to "True". The evaluation of arguments SHALL stop if it is determined that evaluating the remaining arguments will not satisfy the requirement. In an expression that contains any of these functions, if any argument is "Indeterminate", then the expression SHALL evaluate to "Indeterminate".

The description states that each argument will be evaluated according to the order they are listed. However, if the number of expressions supplied is less than the number specified as the first argument, then it must return indeterminate. The evaluation should not proceed once the requirement is satisfied. However, it does state that if any element is deemed to evaluate to Indeterminate, the result shall be indeterminate.

Note: This situation is ambiguous because “leaving the rest of the arguments unevaluated” and the last clause states that all arguments really must already be evaluated before applying the function. The last sentence implies strict evaluation for all arguments regardless.

We model this function, in which the length is checked, and if that is okay, then all the arguments are technically evaluated until an indeterminate is found, regardless of evaluating to true..

```
logical_n_of :: FunctionF
```

```
logical_n_of ((IntAtom n):vs) =
        if (n > length vs) then IndeterminateVal
                           else n_of n vs
    where
          n_of 0     (IndeterminateVal:_)  = IndeterminateVal
          n_of 0     (BoolAtom True:vs)    = n_of 0 vs
          n_of 0     (BoolAtom False:vs)   = n_of 0 vs
          n_of n     []                    = BoolAtom (n /= 0)
          n_of (n+1) (IndeterminateVal:_)  = IndeterminateVal
          n_of (n+1) (BoolAtom True:vs)    = n_of n vs
          n_of (n+1) (BoolAtom False:vs)   = n_of (n+1) vs
          n_of _     _                     = IndeterminateVal
```

The first clause of “n_of” states that 0 with an indeterminate value for the next argument is indeterminate, regardless of the other arguments. However, if 0 with a true for the next argument is dependent on the rest of the arguments regardless. The same is true with 0 and false. However, if we have no arguments, the result is only true if we are looking 0 to be true. For the inductive cases, If the next argument is true, the result is the same as applying the “n_of” function to one less and the rest of the arguments.. However, if the next argument is false, then the result is the same as applying the “n_of” function to the same number and the rest of the arguments. If we encounter an indeterminate along the way, it immediately results in indeterminate.

Note: This semantics specifies that Indeterminate takes precedence over an evaluation of True or False.

Logical Not

The logical not function evaluates to the logical opposite, except if the argument is Indeterminate. The XACML 1.1 Specification A. 14.5 Logical Functions states the semantics for “not” as the following:

> This function SHALL take one logical argument. If the argument evaluates to "True", then the result of the expression SHALL be "False". If the argument evaluates to "False", then the result of the expression SHALL be "True". In an expression that contains any of these functions, if any argument is "Indeterminate", then the expression SHALL evaluate to "Indeterminate".

```
logical_not :: FunctionF
logical_not [BoolAtom t] = BoolAtom (not t)
logical_not _            = IndeterminateVal
```

Bag Functions

The Bag Functions specified in the XACML document are specific for each type. However, their descriptions are polymorphic. Our formal specification of their semantics can be polymorphic as well.

One and Only

The XACML 1.1 Specification Section 14.9 Bag Functions states the semantics for the "one-and-only" function as:

> This function SHALL take an argument of a ***bag*** of *type* values and SHALL return a value of *data-type*. It SHALL return the only value in the ***bag***. If the ***bag*** does not have one and only one value, then the expression SHALL evaluate to "Indeterminate".

```
bag_one_and_only :: FunctionF

bag_one_and_only [Bag [v]] = v
bag_one_and_only _         = IndeterminateVal
```

The above specification matches if there is only a Bag with one element in it, otherwise indeterminate results.

Bag Size

The XACML 1.1 Specification Section 14.9 Bag Functions states the semantics for the "bag-size" function as:

> This function SHALL take a ***bag*** of *type* values as an argument and SHALL return an "http://www.w3.org/2001/XMLSchema#integer" indicating the number of values in the ***bag***.

```
bag_size :: FunctionF

bag_size [Bag []]     = IntAtom 0
bag_size [Bag (v:vs)] = IntAtom (length (v:vs))
```

Is In

The XACML 1.1 Specification Section 14.9 Bag Functions states the semantics for the "is_in" function as:

> This function SHALL take an argument of data-type *type* as the first argument and a ***bag*** of *type* values as the second argument. The expression SHALL evaluate to "True" if the first argument matches by the "urn:oasis:names:tc:xacml:1.0:function:type-equal" to any value in the ***bag***.

We model this semantic with the "bag_is_in" function that takes a list containing only two elements.

```
bag_is_in :: FunctionF
```

```
bag_is_in [a, Bag as] = if (not (elem a as)) then checkDT a as
                                             else BoolAtom True
bag_is_in _           = IndeterminateVal

checkDT _                              [] = BoolAtom True
checkDT a@(IntAtom _)                  (IntAtom _:as)          = checkDT a as
checkDT a@(StringAtom _)               (StringAtom _:as)       = checkDT a as
checkDT a@(BoolAtom _)                 (BoolAtom _:as)         = checkDT a as
checkDT a@(DoubleAtom _)               (DoubleAtom _:as)       = checkDT a as
checkDT a@(DateTimeAtom _)             (DateTimeAtom _:as)     = checkDT a as
checkDT a@(DateAtom _)                 (DateAtom _:as)         = checkDT a as
checkDT a@(TimeAtom _)                 (TimeAtom _:as)         = checkDT a as
checkDT a@(HexBinaryAtom _)            (HexBinaryAtom _:as)    = checkDT a as
checkDT  a@(Base64BinaryAtom _)        (Base64BinaryAtom _:as) = checkDT a as
checkDT a@(AnyURIAtom _)               (AnyURIAtom _:as)       = checkDT a as
checkDT a@(YearMonthDurationAtom _ _) (YearMonthDurationAtom _ _:as) =
                                                    checkDT a as
checkDT a@(MonthDayDurationAtom _ _)  (MonthDayDurationAtom _ _:as) =
                                                    checkDT a as
checkDT _                              _ = IndeterminateVal
```

The function “elem” is a standard Haskell function that returns true if the first argument is equal to an element in the second argument, which is a list. The “elem” function will apply the correct equality predicate. The function “checkDT” is needed to specify the semantics for checking to see if the data type of the compared value is consistent with the type of the Bag. This check can be factored away for compilations that conform to XACML's type safety. The “checkDT” function returns true or indeterminate.

Bag

The XACML 1.1 Specification Section A.14.9 Bag Functions states the semantics for the “bag” function as:

> This function SHALL take any number of arguments of a single data-type and return a ***bag*** of *type* values containing the values of the arguments. An application of this function to zero arguments SHALL produce an empty ***bag*** of the specified data-type.

We model this with the “bag” function. It takes a list of arguments of the same type and turn them into a bag.

Note: It is not specified in the document, but if any of those arguments evaluate to Indeterminate, the entire bag expression should evaluate to Indeterminate to be consistent.

The way this function is defined, ensures that the data types will be consistent amongst the arguments, otherwise the expression results in Indeterminate.

```
bag :: FunctionF
```

```
bag []   = Bag []

bag (IntAtom a:vs) = foldl f (Bag [(IntAtom a)]) vs
    where f (Bag a@((IntAtom _):bs)) b@(IntAtom _) = Bag (a ++ [b])
          f _ _                  = IndeterminateVal

bag (DoubleAtom a:vs) = foldl f (Bag [(DoubleAtom a)]) vs
    where f (Bag a@((DoubleAtom _):bs)) b@(DoubleAtom _) =
                                                 Bag (a ++ [b])
          f _ _                  = IndeterminateVal

bag (StringAtom a:vs) = foldl f (Bag [(StringAtom a)]) vs
    where f (Bag a@((StringAtom _):bs)) b@(StringAtom _) =
                                                 Bag (a ++ [b])
          f _ _                  = IndeterminateVal

bag (BoolAtom a:vs) = foldl f (Bag [(BoolAtom a)]) vs
    where f (Bag a@((BoolAtom _):bs)) b@(BoolAtom _) =
                                                 Bag (a ++ [b])
          f _ _                  = IndeterminateVal

bag (DateAtom a:vs) = foldl f (Bag [(DateAtom a)]) vs
    where f (Bag a@((DateAtom _):bs)) b@(DateAtom _) =
                                                 Bag (a ++ [b])
          f _ _                  = IndeterminateVal
bag (TimeAtom a:vs) = foldl f (Bag [(TimeAtom a)]) vs
    where f (Bag a@((TimeAtom _:bs))) b@(TimeAtom _) =
                                                 Bag (a ++ [b])
          f _ _                  = IndeterminateVal
bag (DateTimeAtom a:vs) = foldl f (Bag [(DateTimeAtom a)]) vs
    where f (Bag a@((DateTimeAtom _):bs)) b@(DateTimeAtom _) =
                                                 Bag (a ++ [b])
          f _ _                  = IndeterminateVal
bag (HexBinaryAtom a:vs) = foldl f (Bag [(HexBinaryAtom a)]) vs
    where f (Bag a@((HexBinaryAtom _):bs)) b@(HexBinaryAtom _) =
                                                 Bag (a ++ [b])
          f _ _                  = IndeterminateVal

bag (Base64BinaryAtom a:vs) = foldl f (Bag [(Base64BinaryAtom a)]) vs
    where f (Bag a@((Base64BinaryAtom _):bs)) b@(Base64BinaryAtom _) =
                                                 Bag (a ++ [b])
          f _ _                  = IndeterminateVal

bag (DoubleAtom a:vs) = foldl f (Bag [(DoubleAtom a)]) vs
    where f (Bag a@((DoubleAtom _):bs)) b@(DoubleAtom _) =
                                                 Bag (a ++ [b])
          f _ _                  = IndeterminateVal

bag (AnyURIAtom a:vs) = foldl f (Bag [(AnyURIAtom a)]) vs
    where f (Bag a@((AnyURIAtom _):bs)) b@(AnyURIAtom _) =
                                                 Bag (a ++ [b])
          f _ _                  = IndeterminateVal

bag (YearMonthDurationAtom y d:vs) =
                        foldl f (Bag [(YearMonthDurationAtom y d)]) vs
    where f (Bag a@( YearMonthDurationAtom _ _ :bs))
                 b@( YearMonthDurationAtom _ _) = Bag (a ++ [b])
          f _ _                  = IndeterminateVal

bag (MonthDayDurationAtom m d:vs) =
                        foldl f (Bag [(MonthDayDurationAtom m d)]) vs
    where f (Bag a@( MonthDayDurationAtom _ _ :bs))
                 b@( MonthDayDurationAtom _ _) = Bag (a ++ [b])
          f _ _                  = IndeterminateVal
```

Higher Order Bag Functions

In this section we redefine the higher order bag functions to work with our value scheme. They follow the same logical semantics, except our arguments are lists of "ValueT" and functions return "ValueT".

Any of

The XACML 1.1 Specification Section A.14.11 Higher-order bag functions states the semantics for the "any-of" function as:

> This function SHALL take three arguments. The first argument SHALL be a `<Function>` element that names a boolean function that takes two arguments of primitive types. The second argument SHALL be a value of a primitive data-type. The third argument SHALL be a ***bag*** of a primitive data-type. The expression SHALL be evaluated as if the function named in the `<Function>` element is applied to the second argument and each element of the third argument (the ***bag***) and the results are combined with "urn:oasis:names:tc:xacml:1.0:function:or".

```
any_of :: FunctionF
any_of [f,a,Bag []]                        = BoolAtom False
any_of [f@(StringAtom fid),a,Bag (x:xs)] =
             logical_or [(funcEnv fid) [a,x],(any_of [f,a,Bag xs])]
any_of _                =  IndeterminateVal
```

All of

The XACML 1.1 Specification Section A.14.11 Higher-order bag functions states the semantics for the "all-of" function as:

> This function SHALL take three arguments. The first argument SHALL be a `<Function>` element that names a boolean function that takes two arguments of primitive types. The second argument SHALL be a value of a primitive data-type. The third argument SHALL be a ***bag*** of a primitive data-type. The expression SHALL be evaluated as if the function named in the `<Function>` element were applied to the second argument and each element of the third argument (the ***bag***) and the results were combined using "urn:oasis:names:tc:xacml:1.0:function:and".

```
all_of :: FunctionF
all_of [f,a,Bag []]                        = BoolAtom True
all_of [f@(StringAtom fid),a,Bag (x:xs)] =
            logical_and [(funcEnv fid) [a,x],(all_of [f,a,Bag xs])]
all_of _                = IndeterminateVal
```

Any Of Any

The XACML 1.1 Specification Section A.14.11 Higher-order bag functions states the semantics for the “any-of-any” function as:

> This function SHALL take three arguments. The first argument SHALL be a `<Function>` element that names a boolean function that takes two arguments of primitive types. The second argument SHALL be a ***bag*** of a primitive data-type. The third argument SHALL be a ***bag*** of a primitive data-type. The expression SHALL be evaluated as if the function named in the `<Function>` element were applied between *every* element in the second argument and *every* element of the third argument (the ***bag***) and the results were combined using “urn:oasis:names:tc:xacml:1.0:function:or”. The semantics are that the result of the expression SHALL be "True" if and only if the applied predicate is "True" for *any* comparison of elements from the two ***bags***.

```
any_of_any :: FunctionF
any_of_any [f,Bag [],Bag ys]     = BoolAtom False
any_of_any [f,Bag (x:xs),Bag ys] =
           logical_or [any_of [f,x,Bag ys,any_of_any [f,Bag xs,Bag ys]]]
any_of_any _                     = IndeterminateVal
```

All of Any

The XACML 1.1 Specification Section A.14.11 Higher-order bag functions states the semantics for the “all-of-any” function as:

> This function SHALL take three arguments. The first argument SHALL be a `<Function>` element that names a boolean function that takes two arguments of primitive types. The second argument SHALL be a ***bag*** of a primitive data-type. The third argument SHALL be a ***bag*** of a primitive data-type. The expression SHALL be evaluated as if function named in the `<Function>` element were applied between every element in the second argument and every element of the third argument (the ***bag***) using “urn:oasis:names:tc:xacml:1.0:function:and”. The semantics are that the result of the expression SHALL be "True" if and only if the applied predicate is "True" for each element of the first ***bag*** and any element of the second ***bag***.

```
all_of_any :: FunctionF
all_of_any [f,Bag [],Bag ys]     = BoolAtom False
all_of_any [f,Bag (x:xs),Bag ys] = logical_and
                      [any_of [f,x,Bag ys,all_of_any [f,Bag xs,Bag ys]]]
all_of_any _                     = IndeterminateVal
```

Any of All

The XACML 1.1 Specification Section A.14.11 Higher-order bag functions states the semantics for the “any-of-all” function as:

> This function SHALL take three arguments. The first argument SHALL be a `<Function>` element that names a boolean function that takes two arguments of primitive types. The second argument SHALL be a ***bag*** of a primitive data-type. The third argument SHALL be a ***bag*** of a primitive data-type. The expression SHALL be evaluated as if the function named in the `<Function>` element were applied between *every* element in the second argument and *every* element of the third argument (the ***bag***) and the results were combined using “urn:oasis:names:tc:xacml:1.0:function:or”. The semantics are that the result of the expression SHALL be "True" if and only if the applied predicate is "True" for *any* element of the first ***bag*** compared to *all* the elements of the second ***bag***.

```
any_of_all :: FunctionF
any_of_all [f,Bag [],Bag ys]     = BoolAtom False
any_of_all [f,Bag (x:xs),Bag ys] =
         logical_or [all_of [f,x,Bag ys,any_of_all [f,Bag xs,Bag ys]]]
any_of_all _                     = IndeterminateVal
```

All of All

The XACML 1.1 Specification Section A.14.11 Higher-order bag functions states the semantics for the “any-of-all” function as:

> This function SHALL take three arguments. The first argument SHALL be a `<Function>` element that names a boolean function that takes two arguments of primitive types. The second argument SHALL be a ***bag*** of a primitive data-type. The third argument SHALL be a ***bag*** of a primitive data-type. The expression is evaluated as if the function named in the `<Function>` element were applied between *every* element in the second argument and *every* element of the third argument (the ***bag***) and the results were combined using “urn:oasis:names:tc:xacml:1.0:function:and”. The semantics are that the result of the expression is "True" if and only if the applied predicate is "True" for *all* elements of the first ***bag*** compared to *all* the elements of the second ***bag***.

```
all_of_all :: FunctionF
all_of_all [f,Bag [],Bag ys]     = BoolAtom False
all_of_all [f,Bag (x:xs),Bag ys] =
         logical_and [all_of [f,x,Bag ys,all_of_all [f,Bag xs,Bag ys]]]
all_of_all _                     = IndeterminateVal
```

Lookup Environments

The functions that supply the functions, predicates, and combinators are listed as identifier, function name pairs. We refer you to the XACML document for their specific names, and other than the functions described above, their semantics are described elsewhere.

Function Environment

The Function Environment holds all the functions that are nameable in an XACML expression.

```
value_eq :: [ValueT] -> ValueT
value_eq [a,b] = BoolAtom (a == b)

integer_greaterthan :: [ValueT] -> ValueT
integer_greaterthan [IntAtom a, IntAtom b] = BoolAtom (a > b)

fenv = [
    ("or",                   logical_or),
    ("and",                  logical_and),
    ("any-of",               any_of),
    ("all-of",               all_of),
    ("any-of-any",           any_of_any),
    ("all-of-any",           all_of_any),
    ("any-of-all",           any_of_all),
    ("all-of-all",           all_of_all),
    ("integer-one-and-only", bag_one_and_only),
    ("integer-greaterthan",  integer_greaterthan),
    ("string-equal",         value_eq),
    ("boolean-equal",        value_eq),
    ("integer-equal",        value_eq)

     --- etc
 ]
```

Match Environment

The Match Environment holds all the functions that are allowed in the matching elements. These names should also be in the Function Environment.

```
menv = [
    ("string-equal",  value_eq),
    ("boolean-equal", value_eq),
    ("integer-equal", value_eq)

     --- etc
 ]
```

Rule Combinator Environment

The Rule Combinator Environment contains the standard combinators.

```
ruleCenv = [
    (“deny-overrides”,   rule_deny_overrides),
    ("permit-overrides", rule_permit_overrides),
    ("first-applicable", rule_first_appl)
          ]
```

Rule Deny Overrides

The XACML 1.1 Specification Section C.1 Deny Overrides states the semantics of the rule combining algorithm “deny-overrides” as follows:

> In the entire set of ***rules*** in the ***policy***, if any ***rule*** evaluates to "Deny", then the result of the ***rule*** combination SHALL be "Deny". If any ***rule*** evaluates to "Permit" and all other ***rules*** evaluate to "NotApplicable", then the result of the ***rule*** combination SHALL be "Permit". In other words, "Deny" takes precedence, regardless of the result of evaluating any of the other ***rules*** in the combination. If all ***rules*** are found to be "NotApplicable" to the ***decision request***, then the ***rule*** combination SHALL evaluate to "NotApplicable".
>
> If an error occurs while evaluating the ***target*** or ***condition*** of a ***rule*** that contains an ***effect*** value of "Deny" then the evaluation SHALL continue to evaluate subsequent ***rules***, looking for a result of "Deny". If no other ***rule*** evaluates to "Deny", then the combination SHALL evaluate to "Indeterminate", with the appropriate error status.
>
> If at least one ***rule*** evaluates to "Permit", all other ***rules*** that do not have evaluation errors evaluate to "Permit" or "NotApplicable" and all ***rules*** that do have evaluation errors contain ***effects*** of "Permit", then the result of the combination SHALL be "Permit".

The semantics of the Rule Deny Overrides algorithm takes into account the effect of the rule. It is formally described as follows:

```
rule_deny_overrides :: RuleCombiningAlgF

rule_deny_overrides []             req = NotApplicable

rule_deny_overrides ((eff,rf): rs) req =
               check eff (rf req) (rule_deny_overrides rs req)

    where check _      Permit        NotApplicable = Permit
          check _      Permit        next          = next
          check _      Deny          _             = Deny
          check Deny   Indeterminate Indeterminate = Indeterminate
          check Deny   Indeterminate Deny          = Deny
          check Deny   Indeterminate Permit        = Indeterminate
          check Deny   Indeterminate NotApplicable = Indeterminate
          check Permit Indeterminate Indeterminate = Indeterminate
          check Permit Indeterminate Deny          = Deny
          check Permit Indeterminate Permit        = Permit
```

```
          check _      NotApplicable next          = next
```

The rule_deny_overrides function specifies the complex semantics for the this combinator. The first clause holds the property that if there are no rules, NotApplicable is always the result. The next clause evaluates each rule and potential effect with the combination of the rest of the rules.

The check function checks a rule with respect to its potential effect with the rest of the rules. For example, the first clause states that if a rule results in “Permit” and the rest of the combinator is “NotApplicable” the result is “Permit”. Noted is the set of clauses that say “Deny Indeterminate” which states that if the potential effect is the rule is “Deny” and the rule results in “Indeterminate” and the rest of the combination is anything but “Deny”, the result must be “Indeterminate”.

Rule Permit Overrides

The XACML 1.1 Specification Section C.1 Deny Overrides states the semantics of the rule combining algorithm “permit-overrides” as follows:

> In the entire set of ***rules*** in the ***policy***, if any ***rule*** evaluates to "Permit", then the result of the ***rule*** combination SHALL be "Permit". If any ***rule*** evaluates to "Deny" and all other ***rules*** evaluate to "NotApplicable", then the ***policy*** SHALL evaluate to "Deny". In other words, "Permit" takes precedence, regardless of the result of evaluating any of the other ***rules*** in the ***policy***. If all ***rules*** are found to be "NotApplicable" to the ***decision request***, then the ***policy*** SHALL evaluate to "NotApplicable".
>
> If an error occurs while evaluating the ***target*** or ***condition*** of a ***rule*** that contains an ***effect*** of "Permit" then the evaluation SHALL continue looking for a result of "Permit". If no other ***rule*** evaluates to "Permit", then the ***policy*** SHALL evaluate to "Indeterminate", with the appropriate error status.
>
> If at least one ***rule*** evaluates to "Deny", all other ***rules*** that do not have evaluation errors evaluate to "Deny" or "NotApplicable" and all ***rules*** that do have evaluation errors contain an ***effect*** value of "Deny", then the ***policy*** SHALL evaluate to "Deny".

The semantics of the Permit Deny Overrides algorithm takes into account the effect of the rule. It is formally described as follows:

```
rule_permit_overrides :: RuleCombiningAlgF
rule_permit_overrides []             req = NotApplicable
rule_permit_overrides ((eff,rf): rs) req =
          check eff (rf req) (rule_permit_overrides rs req)
```

```
    where check _      Deny          NotApplicable = Deny
          check _      Deny          next          = next
          check _      Permit        _             = Permit
          check Permit Indeterminate Indeterminate = Indeterminate
          check Permit Indeterminate Permit        = Permit
          check Permit Indeterminate Deny        = Indeterminate
          check Permit Indeterminate NotApplicable = Indeterminate
          check Deny   Indeterminate Indeterminate = Indeterminate
          check Deny   Indeterminate Permit        = Permit
          check Deny   Indeterminate Deny          = Deny

          check _      NotApplicable next          = next
```

The rule_permit_overrides function specifies the complex semantics for the this combinator. The first clause holds the property that if there are no rules, NotApplicable is always the result. The next clause evaluates each rule and potential effect with the combination of the rest of the rules.

The check function checks a rule with respect to its potential effect with the rest of the rules. For example, the first clause states that if a rule results in “Deny” and the rest of the combinator is “NotApplicable” the result is “Deny”. Noted is the set of clauses that say “Permit Indeterminate” which states that if the potential effect is the rule is “Permit” and the rule results in “Indeterminate” and the rest of the combination is anything but “Permit”, the result must be “Indeterminate”.

Rule First Applicable

The XACML 1.1 Specification Section C.1 Deny Overrides states the semantics of the rule combining algorithm “first-applicable” as follows:

> Each ***rule*** SHALL be evaluated in the order in which it is listed in the ***policy***. For a particular ***rule***, if the ***target*** matches and the ***condition*** evaluates to "True", then the evaluation of the ***policy*** SHALL halt and the corresponding ***effect*** of the ***rule*** SHALL be the result of the evaluation of the ***policy*** (i.e. "Permit" or "Deny"). For a particular ***rule*** selected in the evaluation, if the ***target*** evaluates to "False" or the ***condition*** evaluates to "False", then the next ***rule*** in the order SHALL be evaluated. If no further ***rule*** in the order exists, then the ***policy*** SHALL evaluate to "NotApplicable".
>
> If an error occurs while evaluating the ***target*** or ***condition*** of a ***rule,*** then the evaluation SHALL halt, and the ***policy*** shall evaluate to "Indeterminate", with the appropriate error status.

The first applicable rule selects the first rule that evaluates to anything but NotApplicable. This combining algorithm does not take into account the potential effect of the rule.

```
rule_first_appl :: RuleCombiningAlgF
```

```
rule_first_appl []             req = NotApplicable

rule_first_appl ((eff,rf): rs) req =
          check (rf req) (rule_first_appl rs req)

    where check Permit        _    = Permit
          check Deny          _    = Deny
          check Indeterminate _    = Indeterminate
          check NotApplicable next = next
```

Policy Combinator Environment

The Policy Combinator Environment contains the standard combinators.

```
policyCenv = [
    (“deny-overrides”,      policy_deny_overrides),
    ("permit-overrides",    policy_permit_overrides),
    ("first-applicable",    policy_first_appl),
    ("only-one-applicable", policy_only_one_appl)
          ]
```

Policy Deny Overrides

The semantics of the policy deny overrides combinator is vastly simpler than the rule combinator, because there is no potential effect to consider. However, we do have to formally specify the handling of obligations.

The XACML 1.1 Specification Section C.1 Deny Overrides states the semantics of the policy combining algorithm “deny-overrides” as follows:

> In the entire set of ***policies*** in the ***policy set***, if any ***policy*** evaluates to "Deny", then the result of the ***policy*** combination SHALL be "Deny". In other words, "Deny" takes precedence, regardless of the result of evaluating any of the other ***policies*** in the ***policy set***. If all ***policies*** are found to be "NotApplicable" to the ***decision request***, then the ***policy set*** SHALL evaluate to "NotApplicable".
>
> If an error occurs while evaluating the ***target*** of a ***policy***, or a reference to a ***policy*** is considered invalid or the ***policy*** evaluation results in "Indeterminate", then the ***policy set*** SHALL evaluate to "Deny".

The semantics of the Policy Deny Overrides algorithm is as follows:

```
policy_deny_overrides :: PolicyCombiningAlgF
policy_deny_overrides []             obls req = (NotApplicable,[])
policy_deny_overrides ((targ,pf): ps) obls req =
      combine obls (check (pf req) (policy_deny_overrides ps obls req))
```

```
    where check (Permit,obs) (NotApplicable,obs') = (Permit,obs ++ obs')
          check (Permit,obs)       (d,obs')       = (d,obs')
          check (Deny,obs)         (_,obs')       = (Deny,obs ++ obs')
          check (Indeterminate,[]) (_,obs')       = (Deny,obs')
          check (NotApplicable,[]) (d,obs')       = (d,obs')
          check _                  _              = (Indeterminate,[])

          combine obls (effect,obs) =
                 (effect,relevant effect (obs ++ obls))

          relevant _             [] = []
          relevant Indeterminate _  = []
          relevant decision ((effect,obl):obls) =
                 if decision == effect
                     then (effect,obl) : relevant decision obls
                     else relevant decision obls
```

The first clause of the “policy_deny_overrides” function states that if there are no subordinate policies, then the result is “NotApplicable” without obligations. The second clause checks the result of each policy with the combined result of the following policies.

The check policy combines the results and collects all the obligations from the evaluated policies. The combination of decisions is straight forward according to the specification, explicitly noting that if a policy evaluates to indeterminate, the combined result is Deny.

Note: The semantics for the policy deny overrides combinator are not specified in XACML. The formal specification given here, which is the only one that will return a consistent result regardless of evaluation strategy, is strict in its policy evaluation. Every policy in the combination must be evaluated. This conclusion may be undesirable.

The obligations are collected from every policy. The check clauses force this by evaluating at least the obligations of the rest of the combination. Since policies, according to our formal semantics, only issue obligations pertaining to their specific decision, extracting the relevant obligations from the combination of all obligations and only extracting the the obligations relevant to the combined decision will not select any undesired obligations.

A policy deny overrides combinator that does not implement obligations should not be restricted to evaluate every policy. The following combinator has those semantics. We will keep the same type signature, but ignore the obligations.

```
policy_deny_overrides_noobl :: PolicyCombiningAlgF
policy_deny_overrides_noobl []              obls req = (NotApplicable,[])
policy_deny_overrides_noobl ((targ,pf): ps) obls req =
      check (pf req) (policy_deny_overrides_noobl ps obls req)
```

```
    where check (Permit,_) (NotApplicable,_) = (Permit,[])
          check (Permit,_)          rest      = rest
          check (Deny,_)            _         = (Deny,[])
          check (Indeterminate,[]) _          = (Deny,[])
          check (NotApplicable,[]) rest       = rest
          check _                   _         = (Indeterminate,[])
```

One will notice that the check clauses only cause full evaluation of all subordinate policies by virtue of “Permit” and “NotApplicable”.

Note: Depending upon the semantics chosen for the “deny-overrides” policy combinator, mixing policies and policy sets with both is notoriously dangerous, as this combinator ditches all obligations that may have been emitted from subordinate policies. This can be a problem when policies are distributed and referenced or even executed by different PDPs.

Policy Permit Overrides

The semantics of the policy permit overrides combinator is vastly simpler than the rule combinator, because there is no potential effect to consider. However, we do have to formally specify the handling of obligations.

The XACML 1.1 Specification Section C.1 Permit Overrides states the semantics of the policy combining algorithm “deny-overrides” as follows:

> In the entire set of ***policies*** in the ***policy set***, if any ***policy*** evaluates to "Permit", then the result of the ***policy*** combination SHALL be "Permit". In other words, "Permit" takes precedence, regardless of the result of evaluating any of the other ***policies*** in the ***policy set***. If all ***policies*** are found to be "NotApplicable" to the ***decision request***, then the ***policy set*** SHALL evaluate to "NotApplicable".
>
> If an error occurs while evaluating the ***target*** of a ***policy***, a reference to a ***policy*** is considered invalid or the ***policy*** evaluation results in "Indeterminate", then the ***policy set*** SHALL evaluate to "Indeterminate", with the appropriate error status, provided no other ***policies*** evaluate to "Permit" or "Deny".

The semantics of the Policy Permit Overrides algorithm is as follows:

```
policy_permit_overrides :: PolicyCombiningAlgF
policy_permit_overrides []              obls req = (NotApplicable,[])
policy_permit_overrides ((targ,pf): ps) obls req =
      combine obls (check (pf req) (policy_deny_overrides ps obls req))
```

```
    where check (Deny,obs) (NotApplicable,obs')   = (Deny,obs ++ obs')
          check (Deny,obs)         (d,obs')       = (d,obs')
          check (Permit,obs)       (_,obs')       = (Permit,obs ++ obs')
          check (Indeterminate,[]) (_,obs')       = (Indeterminate,[])
          check (NotApplicable,[]) (d,obs')       = (d,obs')
          check _                  _              = (Indeterminate,[])

          combine obls (effect,obs) =
                 (effect,relevant effect (obs ++ obls))

          relevant _              [] = []
          relevant Indeterminate _  = []
          relevant decision ((effect,obl):obls) =
                 if decision == effect
                     then (effect,obl) : relevant decision obls
                     else relevant decision obls
```

The first clause of the “policy_permit_overrides” function states that if there are no subordinate policies, then the result is “NotApplicable” without obligations. The second clause checks the result of each policy with the combined result of the following policies.

The check policy combines the results and collects all the obligations from the evaluated policies. The combination of decisions is straight forward according to the specification, explicitly noting that if a policy evaluates to indeterminate, the combined result is Indeterminate.

Note: The semantics for the policy permit overrides combinator are not specified in XACML. The formal specification given here, which is the only one that will return a consistent result regardless of evaluation strategy, is strict in its policy evaluation. Every policy in the combination must be evaluated. This conclusion may be undesirable.

The obligations are collected from every policy. The check clauses force this by evaluating at least the obligations of the rest of the combination. Since policies, according to our formal semantics, only issue obligations pertaining to their specific decision, extracting the relevant obligations from the combination of all obligations and only extracting the the obligations relevant to the combined decision will not select any undesired obligations.

A policy permit overrides combinator that does not implement obligations should not be restricted to evaluate every policy. The following combinator has those semantics. We will keep the same type signature, but ignore the obligations.

```
policy_permit_overrides_noobl :: PolicyCombiningAlgF
policy_permit_overrides_noobl []             obls req = (NotApplicable,[])
policy_permit_overrides_noobl ((targ,pf):ps) obls req =
      check (pf req) (policy_permit_overrides_noobl ps obls req)
```

```
    where check (Deny,_)   (NotApplicable,_) = (Deny,[])
          check (Deny,_)           rest      = rest
          check (Permit,_)         _         = (Permit,[])
          check (Indeterminate,[]) _         = (Indeterminate,[])
          check (NotApplicable,[]) rest      = rest
          check _                  _         = (Indeterminate,[])
```

One will notice that the check clauses only cause full evaluation of all subordinate policies by virtue of “Deny” and “NotApplicable”.

Note: Depending upon the semantics chosen for the “deny-overrides” policy combinator, mixing policies and policy sets with both is notoriously dangerous, as this combinator ditches all obligations that may have been emitted from subordinate policies. This can be a problem when policies are distributed and referenced or even executed by different PDPs.

Policy First Applicable

The XACML 1.1 Specification Section C.3 First Applicable states the semantics of the policy combining algorithm “first-applicable” as follows:

> Each ***policy*** is evaluated in the order that it appears in the ***policy set***. For a particular ***policy***, if the ***target*** evaluates to "True" and the ***policy*** evaluates to a determinate value of "Permit" or "Deny", then the evaluation SHALL halt and the ***policy set*** SHALL evaluate to the ***effect*** value of that ***policy***. For a particular ***policy***, if the ***target*** evaluate to "False", or the ***policy*** evaluates to "NotApplicable", then the next ***policy*** in the order SHALL be evaluated. If no further ***policy*** exists in the order, then the ***policy set*** SHALL evaluate to "NotApplicable".
>
> If an error were to occur when evaluating the ***target***, or when evaluating a specific ***policy***, the reference to the ***policy*** is considered invalid, or the ***policy*** itself evaluates to "Indeterminate", then the evaluation of the ***policy-combining algorithm*** shall halt, and the ***policy set*** shall evaluate to "Indeterminate" with an appropriate error status.

The semantics of the Policy First Applicable algorithm is as follows:

```
policy_first_appl :: PolicyCombiningAlgF
policy_first_appl []              obls req = (NotApplicable,[])
policy_first_appl ((targ,pf): ps) obls req =
       combine obls (check (pf req) (policy_first_appl ps obls req))
```

```
    where check (Permit,obs)       _    = (Permit,obs)
          check (Deny,obs)         _    = (Deny,obs)
          check (Indeterminate,[]) _    = (Indeterminate,[])
          check (NotApplicable,[]) next = next
          check _                  _    = (Indeterminate,[])

          combine obls (effect,obs) =
                 (effect,relevant effect (obs ++ obls))

          relevant _             [] = []
          relevant Indeterminate _  = []
          relevant decision ((effect,obl):obls) =
                  if decision == effect
                        then (effect,obl) : relevant decision obls
                        else relevant decision obls
```

Policy Only One Applicable

This policy combing algorithm only selects policies based on the evaluation of their targets. The semantics is that of first applicable after ensuring that there is only one applicable policy.

The XACML 1.1 Specification Section C.4 Only One Applicable states the semantics of the policy combining algorithm “first-applicable” as follows:

> In the entire set of policies in the ***policy set***, if no ***policy*** is considered applicable by virtue of their ***targets***, then the result of the policy combination algorithm SHALL be "NotApplicable". If more than one policy is considered applicable by virtue of their ***targets***, then the result of the policy combination algorithm SHALL be "Indeterminate".
>
> If only one ***policy*** is considered applicable by evaluation of the ***policy targets***, then the result of the ***policy-combining algorithm*** SHALL be the result of evaluating the ***policy***.
>
> If an error occurs while evaluating the ***target*** of a ***policy***, or a reference to a ***policy*** is considered invalid or the ***policy*** evaluation results in "Indeterminate, then the ***policy set*** SHALL evaluate to "Indeterminate", with the appropriate error status.

```
policy_only_one_appl :: PolicyCombiningAlgF
policy_only_one_appl [] obls req = (NotApplicable,[])
policy_only_one_appl ps obls req =
       combine obls
               (check (foldl traverse notappl
                                    (map (\(t,pf) -> (t req,pf)) ps)))
```

```
where
  notappl  = ( BoolAtom False, (\r -> (NotApplicable,[])))
  indeterm = ( IndeterminateVal, (\r -> (Indeterminate,[])))

  traverse    ((BoolAtom True),_)  ((BoolAtom True),_) = indeterm
  traverse  a@((BoolAtom True),p)  _                    = a
  traverse    ((BoolAtom False),_) rest                 = rest
  traverse    (IndeterminateVal,_) _                    = indeterm

  check ((BoolAtom True),policyf)  = policyf req
  check ((BoolAtom False),policyf) = policyf req
  check _                          = (Indeterminate, [])

  combine obls (effect,obs) =
               (effect,relevant effect (obs ++ obls))

  relevant _             [] = []
  relevant Indeterminate _  = []
  relevant decision ((effect,obl):obls) =
                if decision == effect
                      then (effect,obl) : relevant decision obls
                      else relevant decision obls
```

The above description defines the semantics almost literally. The expression containing the “foldl” function traverses all the targets making sure there are not two targets that evaluate to True. The check function evaluates the selected policy and returns the decision with the selected policy's relevant obligations. Those obligations are combined with the given obligations from the enclosing policy set and returned.

Note: Whether dealing with obligations or not with this combinator, it does not matter as only one policy is selected. The result does not change with the evaluation order.