# Policy as Code, Policy as Type

Access Control through Dependent Typing

Matthew Fuchs, PhD
mattdfuchs@gmail.com

## Abstract

With large sums, confidential data, or reputation at risk, correctly applied access policies can be the only barrier between business as usual and cyber disaster. The increasing complexity of the digital environment only makes this more true as time goes on. We improve the state of the art by providing provable correctness for the most complex decidable policies through the application of dependent typing. An access policy, even a complex one, is a type in a sufficiently powerful (dependent) type system, but one that can still be statically type-checked. Only code obeying the policies will pass. With that we can provide provable guarantees a policy is correctly applied well beyond the capabilities of existing systems.

There are a number of access control paradigms available, such as access control lists (ACL) - just a triple of principal, object and right, role based access control (RBAC) - principles have roles, roles have rights over certain objects, relation based access control (ReBAC) - roles are just groupings and there are relationships across these groups, and attribute based access control (ABAC) - access is a function over attributes of the accessor, the object, and the current environment, in order of increasing sophistication. Each subsumes the previous model.

We argue for the importance of ABAC for web services especially, and further that our approach represents a significant step forwards beyond existing ABAC technologies. In the extended introduction we will consider 4, Rego, Sentinel, Cedar, and XACML.

In particular, access control is implemented by technologies. These technologies require a meta-model, such as roles or relations, to describe what we want to control by modeling in our particular domain through, for example, creating roles. As a consequence, what cannot be expressed in the meta-model cannot be directly protected by the technology; i.e., you cannot control what you cannot model. Where a model is too rigid to express a policy, additional, uncontrolled code must be written to surround the control technology and map into the limited model, creating a vector for bugs and attacks. If access depends on some attribute which must be determined at the time of evaluation, such as geography, recent withdrawals or other activity, then only ABAC is sufficient to model a policy safely.

Within ABAC solutions we can further distinguish between solutions requiring all attributes be delivered to the engine at evaluation time, meaning some other system must know how to assemble attributes from some “ground truth”, and those allowing the engine to find values

itself, the latter decreasing the distance of policy evaluation from attribute values by one remove and simplifying the move to a distributed credential model, such as from the W3C. Solutions also divide on the strength of their language, between those whose policies are stated as static structures to be evaluated and those with the strength of a programming language.

But just as you can't protect what you cannot model, you cannot trust your model if you can't be sure your code obeys it. The power of code means powerful policies can be implemented, but not not necessarily that they are implemented correctly. Likewise, even if individual policies are correct, we may still want to prove their consistent application enjoys other properties. Reduced expressive power can get you the first, but none of the available systems provides the second.

Needless to say, our solution, based on dependent types, provides these guarantees with the full expressive power of existing solutions. A policy evaluates a quadruple - right, accessor, object, environment - for correctness. Alternatively stated, a policy is a *type*, permitting quadruples that are correctly typed and preventing the ill typed. Our dependently typed approach turns this into a reality; your policies of arbitrary complexity are types defined in the language and code is statically type checked to ensure only well typed (policy obeying) access is allowed at run time. Furthermore, the language comes equipped with a proof assistant; not only are individual policies provable, but one can also prove invariants hold over longer stretches of code.

The extended introduction makes this argument in the abstract. The body of the paper goes head to head with Rego, a popular, open source ABAC language/system. In it we concretely demonstrate our claims. As we include a lot of code, this may also work as a bit of an introduction to the power of programming with dependent types. Our implementation is written in Agda but everything we say holds equally in other dependently languages, such as Lean.

Within the body we also demonstrate how the approach can be applied to a world of distributed truth, where evaluating a policy may itself involve calling external services.

## Introduction

Access control systems (ACS) are the line of defense protecting your assets from the malicious and the incompetent (and the buggy). Policies are the expression of how you want that control to be applied. Policies can be expressed in many ways, from ordinary language to pseudocode to various formalisms. They can be implemented fully if your ACS supports everything expressed in your policies, or indirectly, either by dropping part of the policy, or by surrounding the ACS with additional code to fill the gap; either approach provides an attack vector for the malicious and the incompetent. Systems of increasing expressiveness have been proposed to close the gap between policy and implementation. We introduce Policy as Type (PAT), a new approach which significantly improves the state of the art.

Different assets will require different policies, and there is a spectrum of meta-models, from Access Control Lists (ACLs) to Role Based (RBAC), Relationship Based (ReBAC), and Attribute

Based (ABAC) technologies, in increasing order of complexity, to name the most common. Evolving needs have pushed users along this spectrum; the complexities of real time, on line, high frequency web services and digital contracts push us to more complex policies.

The ever increasing complexity of the digital landscape, with real time digital contracts, distributed digital identity, digital contracts, and rampant impersonation and fraud, only make this more difficult.

ACLs can be seen as a set of triples containing an object, a principal, and a right. It can get a bit more complicated, such as in network ACLs, where rules can apply to similar IP addresses without enumerating each of them. While a correct ACL will provide an answer to any access request, it provides no abstraction. Every triple is independent. Adding/removing/updating anything more complex than a single user or object is complex because one must consider and update many entries. In a sense there is neither model nor metamodel; you just store the answers.

RBAC provides some abstraction; principals are assigned to one or more roles which they use when accessing the system. Objects, such as applications, give access rights to roles, not individuals, so one need only change an individual's roles without affecting any objects. This significantly reduces overhead from ACLs - removing an individual means removing their roles, updating likewise means changing roles and no need to consider objects. However it cannot model simple relationships among sets of users (i.e., roles) or sets of objects.

ReBAC, which originated with Google's Zanzibar, was developed to handle this issue. There are derivative systems, such as OpenFGA and Ory, based on the Zanzibar model. It allows easy definition of direct rights (user Alice can edit the Calendar because there is a directly defined relationship), and role based access (Bob can edit the Calendar because he's an Editor, and Editors can edit the Calendar). But roles can propagate through indirect relationships, (Bill can edit the Calendar because he has FullAccess to the Project containing the Calendar, and there is a relationship between the Calendar and its Project). This allows much finer control, but all these relationships are extrinsic to the users, objects, and situations it operates in - they cannot peer into properties of objects or users.

The ABAC model adds further expressiveness by allowing attributes of the sender, receiver, and the environment to be included in a policy. Up to this point the first two of these have been considered opaque - the system essentially manipulates labels applied to them - and the last has been completely absent. This allows policies to consider ages, locations, recent transactions, credit lines, ownership, etc., in policies. In theory, we can represent policies of arbitrary complexity. In practice, models need to be decidable, so some restrictions of power are necessary, but a policy that cannot be evaluated in finite time is itself a problem. Each ABAC system has its own compromises, and we will examine 4 commercially available ones -  Rego, Sentinel, XACML, and Cedar - as well as PAT.

For each of these approaches there are implementing technologies. However, a technology can only control what the model can express. It may seem you can shoehorn a policy into a model, but where there is no fit you are inevitably writing uncontrolled code to map into an inadequate model, code which leaves you open to attack. You can only control what you can model.

Our goal is to argue for ABAC as safely evaluating the most general set of policies; other models fall short of providing adequate defense. Further we will show even the existing ABAC languages fall short and propose a more powerful, but equally implentable, alternative.

Because we are particularly concerned with web services, we will consider the following kind of policy in particular: may *sender* S, send *message* M to *receiver* R in current *context* C?

More practically, we would like to be able to specify policies succinctly, evaluate them quickly, and update them easily while maintaining consistency (i.e., updating policies may create race conditions where part of a policy is implemented before a change and part is implemented after).

These practical considerations conflict with expressibility - the kinds of policies we can state in our language. Consider two:

1. Users cannot view videos rated PG unless they are over 13 or have parental permission.
2. An officer can only transfer money from the corporate account during banking hours, no more than 5 times a day, and must be using a known device with the request signed by their private key and physically present in the USA (we could go on with conditions here).

Infringing the first may lead to legal or reputational damages, while the second may protect a digital contract against exploitation; millions of dollars have been lost through the failure to implement such constraints. How can we enforce these given our models? What can each model express?

What cannot be stated directly within our policy language, and enforced by the technology behind it, relies on the goodness of programmers and attackers, meaning it’s inherently unreliable. There is the dependence on the programmer, and there are additional aspects of coordination between this external code and the policy machinery. Of course, badly defined policies are always possible, but then it is clear where the blame lies.

## Arguing for ABAC

With a traditional ACL, getting an answer is easy but correctness is hard to maintain. Given lots of users and lots of movies, we would need to track every user’s right to view every movie. When a user turns 13, we’d need to go through all movies to grant access. Somewhere we’d need to keep a list of PG movies anyway. This little thought experiment shows ACLs have clear scaling and responsiveness issues. The problems of ACLs have given birth to our other challengers, and we will leave it here.

With roles we can get a bit further. Roles apply to principals not objects, but we can divide, in the first case, principals into Everyone, Over13, HasParentalPermission. Every G-rated video (assuming only G and PG) allows a principal with the Everyone role to view, while every PG-rated video only allows principals with the roles Over13 or HasParentalPermission to view. This seems to work for a simple case, but there are still issues. The policy language is “principals with role R can send messages of type X to object O”. Assigning principals to Over13 and HasParentalPermission is outside the scope of the policy language - external code needs to monitor it. If permission may be given for just a single movie, then we need a PermissionToSeeMovieM role for every movie, membership again maintained in code. If permissions can expire, we have even more code. We cannot model the (human) parent/child relationship. Each movie stands on its own; we cannot treat G movies or PG movies as groups. If something changes, we need to individually update every affected movie.

Relations add more power to the model. We can create 2 objects,  G and PG. For any individual movie we can assign G or PG to a “rating” relationship and then inherit who can view from them. We can create a userset of all principals and another of just those over 13. G has a “can view” relationship with both, while PG has with just the over 13 one. Each movie can now inherit the “can view” relationship from its parent, either the G or PG object. Permissions are modeled as additional permission objects with a “permission” relationship (one might normally view the relationship as proceeding from the permission object to the movie, but it’s not clear from the paper whether relations can be inverted, so we will add them as relations from a given movie). The permission object would have a “permits” relation to a parent which then has a “parent” relation to the child. Zanzibar follows these using “userset rewrite rules”. We can even provide the right to see any PG movie by attaching the permission to the PG object instead of the movie.

This brings us much closer; we have a reasonable construct for designating permission and we can connect that through parenthood with the permitted child. Expressing this in Zanzibar may be a bit tortured. However, membership in the over 13 group needs to be externally monitored and appropriate relations added. If the permission has a time limit, that must be externally managed. As specified, the relational policy language unions sets across relations. As such, it is difficult to describe a version of a permission that both includes parents and is restricted to a single child; i.e., a permission should have relation to the granting parent and the child, but must also require there be a transitive relationship - the grantor of the permission must be the parent of the child.

Considering our second proposed policy, the issue of change is even more significant; an external system needs to monitor transactions to move individuals in and out of relations depending on temporal circumstances. These cannot be stated within the relational policy language and might change frequently. While Zanzibar has distributed consistency protocols, continually changing rights could cause significant churn.

Finally we consider the attribute-based approach. The models we have discussed are unable to access either attributes of the entities involved (sender age, receiver rating, message signature) or of the context in which a decision is being made (time of day, sender's location). Those aspects of a policy, therefore, need to be handled external to the policy technology.

We are limited to labels the system places on them for us. This requires us to create constructs to emulate the properties and execute external code to maintain them. In an ABAC model, we address them directly. We can directly access the age of a principal, we may even be able to ask about additional objects, such as if the permission property includes a permission for the given movie by another principal who is the parent of the first.

In an ABAC system, objects and contexts have attributes we can query. Of course, the underlying system needs to model and maintain the attributes we need. But, we can just ask for the age of the sender and check if it is greater than 13. If not, we can check for, say, a permissions attribute pointing to a list of permissions, filtering to permissions pointing to the receiver (the video), and a parent attribute pointing to one or more other objects, one of which is the grantor of the permission.

In our second policy, we can consider the user's actual credit line, how much they've transferred in the last 24 hours, where they are at the time of the request, etc. This lets us precisely define the constraints of our policy in the language of our implementing technology.

In theory, then, ABAC gives us access to the full power of our information system. All the information we have about sender, receiver, environment, and message, are available to us to use in evaluating a request.

On the reasonable assumption that our underlying system can keep track of these values (the ABAC technology can only be as good as the system supplying the attributes), we then need to consider the power of the policy language under consideration.

## Comparing ABAC Languages

We can consider four commercially available ABAC systems, as well as the one we will propose. The existing languages are XACML, an XML based language created by OASIS; Cedar, a more recent language developed by Amazon; Sentinal, a language developed by Hashicorp, creators of Terraform; and Rego, an open source policy language developed by ?.

We will compare them along the following lines:
- Where do the attributes come from?
- What is the expressive power of the language?
- Are policies provably correct?

### Where do the attributes come from?

The attributes a policy engine uses can either be supplied as some data structure by an outside system, such as the caller, or retrieved directly by the engine itself. In the former case, the results achieved from the policy engine can only be as good as the external system assembling the input, while in the latter the attributes are as good as the underlying system itself. While we can try to drive out sources of doubt, attribute values need to come from somewhere, some ground truth that we need accept as correct.

Languages in the former include XACML, Cedar, and Rego. A call to any of these includes a data structure, XML or JSON, with the necessary attributes, so the policies are actually a function over this data structure. However the creation of this data structure is problematic: it is beyond the control and verification of the policy apparatus; it relies on ensuring the attribute generation process (or processes) evolves correctly with the policies; and it is vulnerable to the time of check vs time of use race condition (TOCTOU). Between the time the attributes are gathered and when the policy is evaluated, values may have changed. Collecting attributes during evaluation reduces the time delay, but the situation is really akin to atomic transactions in databases.

Sentinel has the ability to make http calls during the evaluation of a policy, meaning it can go directly to the sources of attribute values. As such it is not dependent on a data structure of dubious quality and policies are free to use any attributes supplied by an http call. However there is no control on the authenticity of these URLs or the structure of the data returned.

PAT, being implemented in a strong, dependently typed language, can access any external data to locate attributes, like Sentinel, but the safety of those endpoints is explicitly managed. In Agda, for example, an external call whose results are just accepted as true is a *postulate*, distinguishing what is true because it has been proven, and what we will assume is true (i.e., external data). With the full power of monastic functional programming we can look at deeper solutions to issues like TOCTOU. This is a problem which will only be exacerbated in a more distributed world.

### What is the expressive power of the language

XACML and Cedar express policies as static structures containing some relationship information and some logic over attribute values. As such the language expressed directly in these two systems is quite limited. Any additional computation must be pushed back to the calling system.

Rego and Sentinel, on the other hand, both derive from datalog, a descendant of Prolog. Policies are arranged in logic-based rules capable of calling each other. Each rule may have several bodies, with the system using backtracking across multiple rules with multiple bodies to find a final answer.

This comes close to the power of a general purpose programming language, but these languages are deliberately weakened to ensure they terminate with an answer for each call. Sentinel’s ability to make http calls can certainly create trouble here, as backtracking could lead to multiple calls with potentially different answers each time.

PAT fits into the same arena. The underlying language - currently Agda, but eventually Lean - is deliberately designed to terminate. Indeed, all recursive calls and loops are analyzed to ensure they terminate in a finite number of steps. It is possible to program with potentially infinite data structures, but that then becomes a clear choice.  However, dependently typed languages do not have backtracking as a resolution method.

This does not seem to cause any issues, as no one wishes to create policies which cannot be evaluated in finite time.

Being directly tied to an existing language with all the power that entails, PAT has access to the full power of the underlying language as well as developments in other fields (for example, extensive areas of mathematics have been formalized in both Agda and Lean), while Rego and Sentinel are restrained in the libraries and other code they can import.

### Are policies provably correct?

Among our four existing languages, Cedar has gone the farthest in attempting to prove the correctness of its implementation, having started with a provably correct implementation and then carefully rewriting that in Rust. While XACML predates more recent efforts in provable correctness, it could certainly be subjected to a similar methodology.

Rego and Sentinel both trade off correctness for power. They adhere to the concept of *policy as code*, directly inspired by the idea of *infrastructure as code*. While they are more expressive than Cedar and XACML, policies written in Rego and Sentinel are simply programs, like any other programs (although based on a logical framework with backtracking unfamiliar to most programmers). They are as vulnerable to bugs as any other program with only testing and the strength of their type systems to save the developer, and neither has particularly strong type system.

PAT, on the other hand, exploits powerful advances in programming languages based on dependent types. In PAT, a policy is a *type*, a very sophisticated type, but one for which there are type checkers. As such it specifies the set of legal quads (sender, receiver, message, context) and the built-in type-checker will statically check that code actually implements the policies. PAT provides the safety guarantees of a Cedar with the power of Rego or Sentinel.

Beyond this, Agda and Lean both double as proof assistants. Not only can one prove, for example, prove your policies are consistent and that your code obeys them individually, one can prove that larger chunks of code interacting with multiple policies will still maintain invariants. One could, for example, encode ReBAC constraints into your policies and then directly interact with Zanzibar or a clone by postulating they can correctly provide the relation between two objects.

## From here

From here we will explain our approach in a head to head comparison with Rego, currently the open source champion of the ABAC approach. This will involve a lot of Agda code and can also be seen as a bit of an introduction to the value of dependently typed programming. We start with a translation of the introductory example from the Rego documentation and move on from there. We implement the movie policy in detail and describe how this can work in a decentralized trust environment, such as envisioned by the W3C Verifiable Claims WG.

## Dependent types and propositions as types - turning a policy into a type

There is a new class of programming languages called dependently typed. This family includes Agda, Lean, and Idris, among others. Our examples will use Agda, but our reasoning applies to all members of the family. Because dependent types are unfamiliar to most programmers, we will spend some time describing a particular class of dependent types called propositional types which extend the familiar structural types, and then use those to rewrite the Rego introductory example in Agda.

Dependent typing breaks down the barrier between types and terms deeply rooted in other programming languages. The canonical example is vectors of length *n*, defined as:

```
data Vec (A : Set a) : ℕ → Set a where
  []  : Vec A zero
  _∷_ : ∀ (x : A) (xs : Vec A n) → Vec A (suc n)
```

Here `A` gives the type of entries in the `Vector`. Bypassing much subtlety, `Set` is Agda-speak for a type, so given a type `A`, `Vec A` is a type constructor which, given a natural number, *n*, creates the type `Vec A n`. (ℕ is the set recursively consisting of `zero` and `(suc n)`, where *n* is a member of ℕ.) There are two constructors, `[]` creates the unique object of type `Vec A 0`, and `_::_` adds an object of type `A` to the front of a vector of type `Vec A n`, creating a vector of size *n* + 1, of type `Vec A (suc n)`. This shows a dependent type, a type containing a term, in this case an ℕ, so `'a' :: 'b' :: []` is of type `Vec Char 2`. In most languages programmers are used to, you cannot have a value, such as 2, as part of a type definition. Note that the constructors have complete control over *n* - you can only declare an object of `Vec A n` by constructing it one object at a time.

More interesting are types such as

```
data _≤_ : Rel ℕ 0ℓ where
  z≤n : ∀ {n}                 → zero  ≤ n
  s≤s : ∀ {m n} (m≤n : m ≤ n) → suc m ≤ suc n
```

`_≤_` is a relation between two natural numbers. This describes types such as `5 ≤ 6`. For someone just exposed to this approach that takes a while to sink in. `5 ≤ 6` is *type*, not a function from a pair of numbers to a boolean value. How do we get such a thing? We have two constructors for the type, `z≤n` and `s≤s`. The first takes a single number, *n*, and constructs the type `zero ≤ n` for whatever that *n* is. The curly braces indicate that Agda will attempt to find the appropriate n, so you don't need to do so. Clearly that won't give us an example of `5 ≤ 6`, so we look to the other constructor. It says, if we have an object of type `m ≤ n` then we can get back an object of type `suc m ≤ suc n`. In this case `m` is 4 and `n` is 5. We need (s≤s …*object of type 4* ≤ *5*). How do we get an object of type `4 ≤ 5`? We repeat, until we can get down to using `z≤n`, at which point n is just 1 (i.e., 6 - 5). The (only) object of type `5 ≤ 6` (given these definitions) is `(s≤s (s≤s (s≤s (s≤s (s≤s z≤n {1})))))`. In general Agda infers the argument to `z≤n`, it is an *implicit* argument. We supply it here for clarity; the curly braces indicate to the compiler we are replacing an implicit parameter. The object is a proof that `5 ≤ 6`.

Another, almost ubiquitous, example is the equivalence type constructor (_≡_), defined as

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

Equivalence takes an object, *x*, of (implied) type `A` and constructs the type `x ≡ x`. At first glance this might appear trivial, but it is quite subtle. We can write the type `x ≡ y`, and try to construct an object of that type with `refl`, but it will only type check if we can convince Agda that *x* and *y* are always the same. It turns out needing to show two objects with different definitions are actually the same happens a lot, and we will see examples shortly. (Because dependently typed languages allow terms into type definitions, they also allow function calls and other constructs. These may need to be executed while type checking, sometimes leading to long compile times. However Agda, and similar languages, forces iteration and recursion to terminate so compile times are finite. Agda has other constructs to support controlled non-termination.)

We will dive deeper into this as we look at defining policies through dependent types. The introductory example in the Rego documentation, which we rewrite in Agda, is a filter on a putative server configuration. The configuration is expressed in YAML and the Rego code walks the tree. As given, the typing of structures and `Server`, `Protocol`, `Port` and `Network` is entirely implicit in the Rego code, although that can also be validated using JSON-Schema.

We now duplicate the Rego example with the following Agda code. Agda, like Haskell and other languages (both functional and non-functional), is strongly typed, so type declarations missing from Rego are present.

```
data Protocol : Set where
   https : Protocol
   ssh : Protocol
   mysql : Protocol
   memcache : Protocol
   http : Protocol
   telnet : Protocol

data Network : Set where
   network : String → Bool → Network

data Port : Set where
   port : String → Network → Port
```

```
data Server : Set where
   server : String → List Protocol → List Port → Server

protocols : Server → List Protocol
protocols (server _ a _) = a

net1 = network "net1" Bool.false
net2 = network "net2" Bool.false
net3 = network "net3" Bool.true
net4 = network "net4" Bool.true

publicNetwork : Network → Bool
publicNetwork (network _ a) = a

getNetwork : Port → Network
getNetwork (port _ n) = n

p1 = port "p1" net1
p2 = port "p2" net3
p3 = port "p3" net2

app = server "app" (https ∷ ssh ∷ []) (p1 ∷ p2 ∷ p3 ∷ [])
db = server "db" (mysql ∷ []) (p3 ∷ [])
cache = server "cache" (memcache ∷ []) (p3 ∷ [])
ci = server "ci" (http ∷ []) (p1 ∷ p2 ∷ [])
busybox = server "busybox" (telnet ∷ []) (p1 ∷ [])

servers = app ∷ db ∷ cache ∷ ci ∷ busybox ∷ []
networks = net1 ∷ net2 ∷ net3 ∷ net4 ∷ []
ports = p1 ∷ p2 ∷ p3 ∷ []
```

We first define the data types and then define the values from the example.

The Rego example then introduces two policies:

1. Servers reachable from the Internet must not expose the insecure 'http' protocol.
2. Servers are not allowed to expose the 'telnet' protocol.

We will slightly generalize this and consider 3 lists of protocols:

1. badProtocols, which just includes `telnet`
   ```
   badProtocols : List Protocol
   badProtocols = telnet ∷ []
   ```
2. weakProtocols, which just includes `http`
   ```
   weakProtocols : List Protocol
   weakProtocols = http ∷ []
   ```
3. strongProtocols, which contains everything else
   ```
   strongProtocols : List Protocol
   ```

```
strongProtocols = https ∷ ssh ∷ mysql ∷ memcache ∷ http ∷ []
```

So far Agda looks like any conventional functional language, like Haskell. We now wade into the weeds.

We have a bunch of servers and we can define three properties of servers:

1. A `Server` is good if it exposes no bad protocols
2. A `Server` is private if it is not exposed to the Internet
3. A `Server` is compliant if it is good and either it is private or it exposes no weak protocols

They appear in Agda code as:

```
data GoodProtos : Server → Set where
    *good* : (s : Server) → ((protocols s) ∩ badProtocols) ≡ [] →
GoodProtos s

data PrivateServer : Server → Set where
    *private* : (s : Server) →
(Data.List.Base.length (anyExposed (portList s))) ≡ 0 → PrivateServer s

data GoodServer : (s : Server) → Set where
   *goodserver* : ∀ (s : Server) → (GoodProtos s) →
       (PrivateServer s) → (GoodServer s)
   *safeserver* : ∀ (s : Server) → (GoodProtos s) →
                   ¬ (PrivateServer s) →
                  ((protocols s) ∩ weakProtocols ≡ []) →
                  (GoodServer s)
```

We now start using dependent types in earnest, and they show their power. Here, the types serve as propositions about `Servers`. `GoodProtos s` represents that we have a `Server` none of whose protocols are prohibited (protocols is a helper function). It is dependent because the type *depends* on the particular `Server`, `s`, as well as the value of `badProtocols`.

(The helper function `protocols` is defined as

```
protocols : Server → List Protocol
protocols (server _ a _) = a
```

and just returns the list of `Protocol` defined for the server.)

To construct a value of type `GoodProtos`, we need to call a constructor. There is only one, `*good*`. We need to pass it first a `Server`, `s`, and then something of *type* `(protocols s) ∩ badProtocols) ≡ []`, at which point we receive back a value of type `GoodProtos`. Look at the definition of the `Server` name `db`. We can get a proof of `GoodProtos db` with the statement `*good* db refl`. Since the definition of `db` is known at compile time, Agda can automatically reduce `((protocols s) ∩ badProtocols)` to `((mysql ∷ []) ∩ telnet ∷ [])` and then to `[]`, at which point `refl`, implicitly given the result of this operation (the Agda compiler can infer many arguments automatically), generates a value of type `[] ≡ []`. This type checks, and we have proof that `db` does not expose any bad protocols. You can enter `*good* db refl` into the interpreter or compile it and it type checks; it is a value of type `GoodProtos db`. However, you cannot do the same with `busybox`, which exposes `telnet`. Because the intersection is no longer empty, you would need to create a value of type `(telnet ∷ []) ≡ []`. But that type is empty - the only constructor for equivalence is `refl`, which only takes one value. You cannot create a value of type `GoodProtos busybox`, so `*good* busybox refl` does not type check.

Getting a value of `PrivateServer s` proceeds similarly, although here we ensure the number of exposed ports is 0. Once again, a value of the appropriate type can only be established if, indeed, there are no ports on the server exposed to the Internet.

Finally, a `Server`, *s,* is acceptable if we have a proof of `GoodServer s`. There are two possibilities here: only acceptable protocols are used and either the server is `Private` or it is not `Private` but no weak protocols are exposed.

While it is good to prove such properties, an object of type `GoodServer s` is fairly opaque and doesn't give access to *s* at the term level. We really want both the object, *s*, and the proof together.

A dependent pair is the combination of an object and a statement about that object. The type `Σ[ s ∈ (Server) ](GoodServer s)` is an existentially quantified type. It says here exists *s*, a `Server` which is a `GoodServer`. The constructor for a dependent pair is _,_. An example is:

```
(db , *goodserver* db (*good* db refl) (*private* db refl)
```

where `db` is defined as above. This type checks, meaning I have a pair of `db` and a proof that `GoodServer db` holds.

From this, one can prove servers are good at construction time. For example, in

```
randomServer : Σ[ s ∈ (Server) ] (GoodServer s)
randomServer = let foo = server "foo" (mysql ∷ []) (p3 ∷ []) in
     (foo , *goodserver* foo (*good* foo refl) (*private* foo refl))
```

`randomServer` is a dependent pair. One can get the `Server` object by projection:

```
goodServer = proj₁ randomServer
```

and be guaranteed to have a usable `Server`.

This immediately contrasts with Rego, where the decision that a server is good lies far from its definition. With Rego, the definition of a `Server` must be serialized into JSON or other understood format, transmitted to a Rego server with a query, the answer received and interpreted, and finally some decision must be made. You cannot bring the policy into the construction of the `Server`, only through a convoluted check.

As argued above, if you create your `Server` where the compiler can “see” it, it can move policy checking into a compile time action. However, it may be the `Server` configuration is loaded or in a separate module. Then the compiler cannot see the configuration and we need to deal with a list of `Server`s which may be either good or not. The function `goodServerCheck` examines a `Server` and returns a `Maybe (Σ[ s ∈ (Server) ] (GoodServer s))`- either a dependent pair or `nothing`.

```
goodServerCheck : (s : Server) → Maybe (Σ[ s ∈ (Server) ] (GoodServer s))
goodServerCheck s@(server _ protList portList) with
     (emptyIntersection protList badProtocols)
...       | no _ = nothing
...       | yes noBad with isPrivate s
...              | yes yesPrivate = just (s , *goodserver* s
                                        (*good* s noBad) yesPrivate)
...              | no notPrivate with (emptyIntersection protList
                                                        weakProtocols)
...                      | no _ = nothing
...                      | yes noWeak = just (s , *safeserver* s
                                      (*good* s noBad) notPrivate noWeak)
```

Here, `emptyIntersection` is `Dec([] ≡ [])`. If it fails on the 3rd line, then return `nothing`. Otherwise, check if it’s private. If so, return a pair, using `*goodserver*` for proof.  The values `noBad` and `yesPrivate` are evidence for the proof.  If it’s not private, then test for `weakProtocols`. If there are none, return a pair, using `*safeserver*` as proof - or `nothing`

With this function we can get either the list of good `Server`s or of violations by simply filtering the list. For the good servers one might return the list with a proof that all servers in the list are good.

```
goodServerList : List Server → List Server
goodServerList [] = []
goodServerList (h ∷ t) with goodServerCheck h
...                      | nothing = goodServerList t
...                      | just _  = h ∷ (goodServerList t)
```

There are a few additional desiderata to consider.

As mentioned, `emptyIntersection` is `Dec([] ≡ [])`. Decidable, `Dec`, is defined as

```
data Dec {a} (P : Set a) : Set a where
  yes : ( p :   P) → Dec P
  no  : (¬p : ¬ P) → Dec P
```

`Dec` takes a propositional type and has two constructors. `Yes` returns with a proof of type *P* and `no` returns with a proof of its negation. For a propositional type to be decidable then one must be able to prove it holds or prove it doesn't.

There are two ways to prove a `Server` is a `GoodServer`. One, `*safeserver*`, requires proving a `Server` is `¬(PrivateServer s).` This requires a brief detour into the treatment of negation in Agda.

Proof in Agda is based on the theories of [Per Martin-Löf.](#) It is a [constructive intuitionistic logic](#) lacking the law of the excluded middle, i.e., `(P ∨ ¬P)`. Instead, `¬(PrivateServer s)` is shorthand for `(PrivateServer s) → ⊥.` ⊥ (bottom) is the empty type, and deriving it means there's been a contradiction, i.e., from bottom you can prove anything. Essentially, `¬(PrivateServer s)` means you should not be able to prove `(PrivateServer s)`, because if you could, that would be a contradiction.

While Agda does not employ the law of the excluded middle for proof, there still are types that are decidable, as just mentioned. For example, a natural number is either 0 or it isn't, so there is a decidable version of equivalence for ℕ: `_≟_`. A value of this type is either a `yes` or a `no`, and we use that in the definition of `isPrivate`, where we check how many exposed ports the server has:

```
isPrivate : (s : Server) → Dec (PrivateServer s)
isPrivate s with (Data.List.Base.length (anyExposed (portList s))) ≟ 0
...          | yes eq = yes (*private* s eq)
...          | no neq = no (negatePrivateServer s neg)
```

Being decidable, `(Data.List.Base.length (anyExposed (portList s))) ≟ 0` actually returns either `yes  0 ≡ 0` or `no (0 ≡ 0 → ⊥)`. We will use both.

The first case uses `(*private* s eq)` to return a value of `PrivateServer s.` For `¬(PrivateServer s)` we consider the helper function `negatePrivateServer`, defined as

```
negatePrivateServer : ∀ (s : Server) →
    {Data.List.Base.length (anyExposed (portList s)) ≢ 0 } →
    ¬ (PrivateServer s)
negatePrivateServer s { neq } (*private* _ p) = ⊥-elim (neq p)
```

As explained, `¬ (PrivateServer s)` is shorthand for `(PrivateServer s) → ⊥` and `{Data.List.Base.length (anyExposed (portList s)) ≢ 0` is really shorthand for `0 ≡ 0 → ⊥`. In the body of `negatePrivateServer` we first pass in a function that derives a contradiction when handed `0 ≡ 0` (we have this from `neg` in isPrivate, as described above). That is then curried to a function of type `Private p → ⊥.` `(*private* _ p)` is the parameter to that function (because, being `*private*`, *p* cannot contain any exposed ports) and `⊥-elim (neq p)` derives the contradiction we need.

## Regression Tests vs. Regression Proofs

Programmers are all familiar with the concept of regression tests - a set of tests run after changes to try to ensure that new code doesn't break existing behavior. Regression tests basically run sets of sample data on various parts of the application to try to ensure behavior is correct. Likewise we can prove properties of our current system which may fail if it is changed.

Just as we can prove properties of a transaction request, we can prove properties of the underlying model. For example, we have established a `Protocol` type with a number of different protocols. We have divided them into three categories, `badProtocols`, `weakProtocols`, and `strongProtocols`. We might want to ensure every `Protocol` is in one of these three categories, and that they are exclusive. We can prove these properties for the current code, and if a `Protocol` is added but not categorized, or if a `Protocol` is entered in two categories, the code proving these properties will not compile. This is a regression *proof*, not just a test.

To prove the three categories are exclusive do as follows:

```
separate : (badProtocols ∩ weakProtocols ≡ []) × (badProtocols ∩ 
strongProtocols ≡ [])
          × (weakProtocols ∩ strongProtocols ≡ [])
separate = refl , refl , refl
```

The type of `separate` is an **and** over the three conditions. This is represented by `×` at the type level. We generate an object of the type by proving the type, which we do with three repetitions of `refl`. For each of these, the compilation has evaluated the condition (e.g., `badProtocols ∩ weakProtocols` and passed the results to the only constructor for equivalence, `refl`. Since the intersections are all empty, this proves the proposition.

Proving each Protocol is in one category or another requires a proof for each Protocol:

```
allProtosAssigned : (p : Protocol) → ((p ProtDecSetoid.∈ badProtocols) ⊎
                                      (p ProtDecSetoid.∈ weakProtocols) ⊎
                                      (p ProtDecSetoid.∈ strongProtocols))
allProtosAssigned https = inj₂ (inj₂ (here refl))
allProtosAssigned ssh = inj₂ (inj₂ (there (here refl)))
allProtosAssigned mysql = inj₂ (inj₂ (there (there (here refl))))
allProtosAssigned memcache = inj₂ (inj₂ (there (there (there (here refl)))))
allProtosAssigned http = inj₂ (inj₁ (here refl))
allProtosAssigned telnet = inj₁ (here refl)
```

Here, the symbol `⊎` represents an `or`, so a disjoint union type. To follow the first case, for `https`, we first need to prove it is in `strongProtocols`. After Agda is done applying implicits, the calls to `inj₂ (inj₂)` indicate we are looking at the second type of the second union, i.e., `strongProtocols`. The call `here refl` grabs the first item in the list and proves it's equivalent to `https`. For the next entry, `there` grabs the tail of the list, and so on.

Were another protocol to by added to the type, say `smtp`, and not included in one of our lists, or included in more than one, these proofs would fail, just as a regression test may fail.

These proofs are small but ensure an important property. More complex proofs can be applied to other parts of the system. When the script is monitoring actions in a financial contract, proving certain conditions hold (or do not) can be worth millions of dollars.

## Approving a transaction

Generically, a transaction, such as a call from an entity to a service or a credit card purchase, involves multiple parties.  Each party has some set of attributes (we may also call these properties or, to foreshadow, claims). A policy needs to determine if the transaction is valid or not. To do so it evaluates some proposition for each of the parties, and if all are proven, then the transaction can proceed. The value of dependent types is we can logically specify the policies in code and enforce them in code.

Consider a service providing videos with a simple policy: a user must be older than a video's minimum age to see it, or have permission from their parent to do so. Just to add a little networking, we require that the request pass over https before noon and the answering server is approved. Finally, we want to establish that the returning video is the one requested.

We define Agda record types for the various parties to the transaction. Each record type has a field for a unique name and additional fields to provide attributes. For our purposes, we assume there is some system to provide the appropriate attributes given a unique name.

```
record ItemRequest : Set where
 field
   name : String
   videoName : String
   ageLimit : ℕ

record Item : Set where
 field
   name : String
   ageLimit : ℕ

{-# NO_POSITIVITY_CHECK #-}
record User : Set where
 inductive
 field
   name : String
   age : ℕ
   parent : Maybe User
   grantsPermission : User → ItemRequest → Bool
   isParent : User → Bool

record Transport : Set where
 field
   name : String
   protocol : Protocol

record Context : Set where
 field
   name : String
   timeOfDay : ℕ

record Service : Set where
 field
   name : String
   isApproved : Bool
```

We also provide an equivalence operation over Users to be used when looking for parents.

```
data _≡U_ : Rel User (# 0) where
 *≡U* : ∀ ( p q : User ) → (User.name p) ≡ (User.name q) → p ≡U q

_≟U_ : (a : User) → (b : User) → Dec ( a ≡U b)
a ≟U b with (User.name a) Data.String.Properties.≟ (User.name b)
...        | yes y = yes (*≡U* a b y)
...        | no  n = no (nent n)
 where
   nent : ∀ {s t : User} → (User.name s) ≢ (User.name t) → ¬ (s ≡U t)
   nent neq (*≡U* s t ff) = ⊥-elim (neq ff)
```

This assumes names are unique.

Next we provide policies as propositional dependent types. These are the equivalent of Rego's rules.

```
data SafeContext : Context → Set where
 *contextProp* : (c : Context) → (Context.timeOfDay c) Data.Nat.Base.≤ 12
                 → SafeContext c

data HasPermission : (s : User) → (p : ItemRequest) → Set where
 *hp* : (s : User) → (p : ItemRequest) → (r : User)
       → (User.isParent s r) ≡ Bool.true
       → ((User.grantsPermission r) s p) ≡ Bool.true
       → HasPermission s p

data SafeSender : User → Set where
 *senderPropOldEnough* : (s : User) → (p : ItemRequest)
        → (ItemRequest.ageLimit p) Data.Nat.Base.≤ (User.age s) → SafeSender s
 *senderPropYoung* : (s : User) → (p : ItemRequest) → HasPermission s p
        → SafeSender s

data SafeChannel : Transport → Set where
 *isHttps* : (c : Transport) → (Transport.protocol c) ≡ https → SafeChannel c

data SafePayload : ItemRequest → Set where
 *safePayload* : (i : ItemRequest) → SafePayload i

data SafeService : Service → Set where
```

```
 *safeService* : (s : Service) → (Service.isApproved s) ≡ Bool.true →
SafeService s

data SafeResponse : Item → Set where
 *safeResponse* : (r : ItemRequest) → (i : Item)
     → (ItemRequest.videoName r) ≡ (Item.name i)
     → SafeResponse i
```

The most interesting of these is `SafeSender`. There are two rules, following the two allowed conditions. First is *`senderPropOldEnough`* which holds if a `User`'s age is greater or equal to the `ageLimit` of the requested `Item`. Otherwise we have *`senderPropYoung`* where `SafeSender` holds if the sender `HasPermission` to request the `Item`. `HasPermission` holds for a sender and a requested `Item` if there is another `User`, that `User` is the parent of the sender, and that parent grants permission for the sender to see the `Item`.

The core of this is `safeCall`, a function that only takes dependent pairs and performs the actual server call underneath.

```
safeCall : Σ[ c ∈ Context ] (SafeContext c) →
           Σ[ u ∈ User ] (SafeSender u) →
           Σ[ ch ∈ Transport ] (SafeChannel ch) →
           Σ[ s ∈ ItemRequest ] (SafePayload s) →
           Σ[ s ∈ Service ] (SafeService s) →
           Maybe Item
safeCall context sender channel payload service = answer
where
   doCall : Transport → ItemRequest → Service → Maybe Item
   doCall a b c = — The server call happens here
   answer : Maybe Item
   answer with doCall (proj₁ channel) (proj₁ payload) (proj₁ service)
   ...     | nothing = nothing
   ...     | just result with constrainResponse response (proj₁ payload)
   ...                    | nothing = nothing
   ...                    | just isGood = just (proj₁ isGood)
```

Each parameter to `safeCall` is a dependent pair proving the first object has passed the policies applied to it. Since the actual server call is embedded in `safeCall`, the call doesn't occur unless the policy is met.

Accessing `safeCall` is `preCall` which just tries to gather those proofs.

```
preCall : Context → User → Transport → ItemRequest → Service → Maybe Item
preCall   context  sender   channel  payload service with (checkContext
context)
...    | nothing = nothing
...    | just safeContext with (checkSender sender payload)
...          | nothing = nothing
...          | just safeSender with (checkChannel channel)
...                 | nothing = nothing
...                 | just safeChannel with (constrainPayload payload)
...                        | nothing = nothing
...                       | just safePayload with (checkService service)
...                             | nothing = nothing
...                             | just safeService = safeCall
                                     safeContext safeSender
                                    safeChannel safePayload safeService
```

It is a cascade of calls to check each proof, which (following intuitionistic logic) either returns `just` a proof or returns `nothing`. If all the proofs are available it can call `safeCall` with the appropriate dependent pairs. One could also replace `Maybe` with a type containing `just` a value or `error` and a message

In between are a number of boring routines to generate the necessary proofs, such as

```
checkChannel : (s : Transport) → Maybe (Σ[ s ∈ Transport ] (SafeChannel s))
checkChannel s with (Transport.protocol s) ≟P https
...             | yes p = just (s , *isHttps* s p)
...             | no _ = nothing
```

Eventually these should be auto generated.

Given a small set of objects

```
payload : ItemRequest
payload = record { name = "Payload" ; videoName = "I'm PG 13"; ageLimit =
13 }

response = record { name = "I'm PG 13" ; ageLimit = 13 }

daddyPerm : User → ItemRequest → Bool
daddyPerm child item with (User.name child) ≟S "Sender"
...           | no _ = Bool.false
...           | y with (ItemRequest.videoName item) ≟S "I'm PG 13"
...                 | no _ = Bool.false
...                 | yes p = Bool.true
daddy : User
daddy = record { name = "Daddy" ; age = 40 ; parent = nothing ;
                 grantsPermission = daddyPerm ; isParent =
                                                λ { _ → Bool.false }
 }

isParently : (a : User) → (b : User) → Bool
isParently a b with a ≟U b
...             | no _ = Bool.false
...             | yes _ = Bool.true

sender = record { name = "Sender" ; age = 10 ; parent = just daddy ;
                 grantsPermission = λ { u i → Bool.false }
                 ; isParent = λ { pq → isParently pq daddy }
 }

youngSender = record { name = "YoungSender" ; age = 10 ; parent = just
daddy ;
                 grantsPermission = λ { u i → Bool.false }
                 ; isParent = λ { pq → isParently pq daddy }
 }

channel : Transport
channel = record {name = "Channel" ; protocol = https }

context : Context
context = record { name = "Context" ; timeOfDay = 10 }

service : Service
service = record { name = "Foo" ; isApproved = Bool.true }

service2 : Service
service2 = record { name = "Foo" ; isApproved = Bool.false }
```

We can evaluate a few calls. For example, `preCall context sender channel payload service` will evaluate to `just response` (while `sender` is underage, they have permission from `daddy`), but `preCall context youngSender channel payload service` fails, as `youngSender` does not have permission. Likewise, using `service2` will fail as it is not approved.

This simple example provides a framework for more sophisticated examples. Like Rego, this entire system can be run as a sidecar in conjunction with the basic application. A client communicates values for sender, receiver, etc., to a script. The script finds the appropriate attributes and evaluates them. Unlike Rego, however, the script performs the actual service call. This allows the script to operate on both the payload and response. For example, the payload may contain information requiring a higher security level to view than that of the service; the script can filter out, mask, or tokenize values in the payload to lower its security level. Likewise, the return value may need to be manipulated before being returned to the caller.

Agda being a functional language, we have a main routine in the IO Monad pulling invocations from a Stream. An invocation consists of identifiers for sender, receiver, etc., as well as some representation of the payload. Within the IO Monad we can retrieve or generate attributes for all of these, evaluate the policy functionally, and then perform the call within the IO Monad.

## Building a decentralized authorization ecosystem

In a distributed system, if a call crosses a trust boundary, the recipient must ensure any policies are applied, even if the sender has already proven the receiver’s policies were upheld. However, if a policy is viewed, as we’ve shown, as the validation of a proposition over some set of properties, and the sender can send some trustworthy values for those properties, which we will now call claims, along with a sketch of the proof, the receiver can just validate the correctness of the proof.

Some claims, such as claims about the payload itself, may be easy to validate, but others may be claims supplied by third parties. For example, a claim containing a purchase limit, credit score, or current age attribute is a claim by some third party, such as a bank. For the sender to pass this to the receiver, the claim needs to be authentic. In the current web architecture, this implies either a record on a blockchain or a token signed with the private key of the third party.

Consider the work of the W3C Verifiable Claims WG on their data model. In the model, a claim is a statement about a subject, expressed as a **subject-property-value** relationship. A credential is a set of claims about the same subject, and a *verified* credential is a credential packaged with some metadata and cryptographic proof of the issuer. Finally, a presentation takes this one step further, packaging a set of verified credentials from a single party (the "holder") with, again, cryptographic proof they come from the holder.

For our current purposes, we will just consider a credential packaging a single claim about an entity or the context. We will posit the existence of a set of `ClaimServer`s at particular URLs. A qualified entity can query a `ClaimServer` to get information about an entity. Every party to a transaction will have a list of `ClaimsServer`s they trust, and so would trust a verified claim signed by that `ClaimsServer`.

Clearly there is a separate set of interactions around becoming qualified to query a `ClaimServer` and further complexity around which claims for which entities a specific `ClaimServer` can make.

For two parties to do business, there must be some overlap in the `ClaimServer`s they trust and further overlap in which claims for which entities they trust a particular `ClaimServer`. For our purposes we will elide that code and assume there's a single `ClaimServer` all parties trust.

We add new types supporting trust of the form

```
data ITrustYou : (s : Service) → (a : ClaimServer) → Set where
  iTrustYou : (s : Service) → (a : ClaimServer) → ITrustYou s a
```

showing an entity, in this case a Service, trusts some `ClaimServer`

A verified claim is a subject-property-value triple signed by a `ClaimServer`. Within Agda it is a dependent pair. The first part of the pair is the signed claim, a chunk of text. The second part is a type with three fields. The name of the type reflects the property the claim is about, and the fields are the `ClaimServer`, the entity, and the value.

For example, when a ClaimServer is queried about a `User`'s age, this propositional type

```
data AgeClaim : ClaimServer → User → Set where
  ageClaim : (a : ClaimServer) → (e : User) → (n : ℕ) → AgeClaim a e
```

represents an `AgeClaim` and says the `ClaimServer` says the `User` is *n* years old. All our properties of interest can be specified as claims. The signed claim is kept for further communication, but the instantiated object can be used in proof.

Proofs are now longer as they need to specify the trust relationships. For example, the proof that a sender was old enough to view an item was

```
 *senderPropOldEnough* : (s : User) → (p : ItemRequest)
```

```
               → (ItemRequest.ageLimit p) Data.Nat.Base.≤ (User.age s)
               → SafeSender s
```
But now it becomes
```
 *senderPropOldEnough* : (u : User) → (a : ClaimServer) → (s : Service)
                     → ITrustYou s a → (ac : AgeClaim a u) → (p : ItemRequest)
                     → (ItemRequest.ageLimit p) Data.Nat.Base.≤ (getAge ac)
                     → SafeSender s
```
i.e., we have a `User`, a `ClaimServer`, and a `Service`, the `Service` trusts the `ClaimServer`, the `ClaimServer` makes a claim about the `User`'s age (*ac*), and the age specified by the claim is greater or equal to the age specified by the `ItemRequest`.

Any party can represent its view of a transaction by an Agda record type. The record has fields for all the appropriate parties and fields for each of the proofs that party requires. For example,
```
record ServerSide : Set where
 field
   callContext : Context
   callSender : User
   callTransport : Transport
   requestedItem : ItemRequest
   recipient : Service
   ------------------------
   safeContext : SafeContext callContext
   safeSender : SafeSender callSender
   safeChannel : SafeChannel callTransport
   safePayload : SafePayload requestedItem
   safeService : SafeService recipient
```
The `ServerSide` record type has fields for the entities of interest in the first part, as we've discussed so far. The bottom half has fields with propositional types referring to fields defined in the record. To instantiate a `ServerSide` record, one not only needs entities of the appropriate types, but one needs to also provide the necessary proofs about those entities. A `ClientSide` record would include a field for the returning Item and at least one accompanying proof.

Other definitions, such as data type definitions, can be included in shared Agda modules.

Now clients can generate appropriate records and send them to the server. The work of validating a policy is shifted from server to client. However, we have no simple serialization mechanism to implement this. Instead we need to convert this for transport ourselves.

For the entities in the first part of the record, each should have a unique URI. The pair (field name, URI) should be sufficient for the recipient to create a local object of the appropriate type (as the record type gives the type of each field). For the proofs we hope to rely on Agda's internal reflection mechanism, which provides more than enough information to glean the proof tree. Of course, not all information is necessary. We need the tree of proof constructors and the claims used. In the process of serializing, some information local to where the proof is being executed may need to be signed to verify to the recipient that it was generated by the sender. If reflection turns out to be inadequate, we can alter the code that finds the proof to add in that serialization, so in the end we either have a nothing or a proof with its serialization and the list of verified claims used in generating it.

With this information the recipient will be able to reconstruct a proof from the proof tree and the given claims without needing to either find the proof or perform the process of gathering the claims itself, as the claims are all signed by trusted `ClaimServer`s.

This provides a framework for a decentralized trust architecture, but also a means for collaborating parties to share policy definitions and allow each party to specify their requirements.

## Beyond Synchrony

We have described a simple synchronous call and response architecture and a decentralized authorization system. The latter, as well as many financial systems, require a more complex architecture, capable of juggling multiple overlapping communications. We will sketch such an architecture.

Consider a credit card purchase request. Nominally, this has five parties involved - the purchaser, the merchant, the merchant's bank, the card issuer, and the card network. The dance starts when a merchant sends purchase details to its bank. The merchant's bank sends these details to the card network, which sends them to the card issuer. If all is above board, the purchase details were precipitated by an exchange between the purchaser and the merchant, but perhaps there is fraud. What should the card issuer do?

Clearly the issuer's policy will require answering many questions, such as:

- What is the issuer's history with the merchant? Have they ever been seen? Any past examples of fraud?
- What is the size of the purchase?
- Was it point-of-sale or ordered over the internet?
- Where is the card holder? Can we contact them?
- … and so on.

Answering these questions, vital to the business, may cause some delay, during which other requests may come in from the merchant, the purchaser, or even the merchant's bank. Should these be allowed to proceed, or be paused? If paused, how do we communicate that across requests?

### A dependently typed response

In the dependently typed functional world we abandon our synchronous architecture. Instead, as has really been the case, we adopt message passing. Messages come in from an unbounded Stream and are handled quickly. Perhaps more messages are sent to be later collected before sending a final approval or denial message back to the original sender, after using the collected evidence to evaluate approval or denial against the issuer's policies.

In the functional world these messages and their processing are handled by a State Monad which is continuously updated before the next message is seen. Some messages, such as a new purchase request, start a new process, but others, such as a response from another system (such as credit scoring), may just update the continuation of an existing process. Sending a message to retrieve a value updates the State with a new continuation to handle the response. Eventually all information is gathered, a response generated, and that process disappears from the State.

When a request comes in requiring additional information, the State can be updated to prevent other transactions from moving forward before that information is gathered and resolved. A policy may place a sliding window transaction limit on an account; the sum of transactions for that window becomes an important attribute. A transaction may pass through a number of states as information is gathered and evaluated; these states can be embedded directly into the types to prevent erroneous messages.

This kind of event loop programming is very common in user interfaces where a user may update any visible item. Techniques such as those in [my dissertation] can be used to lower the burden.

Underlying this approach needs to be a persistent store, whether database, blockchain, or persistent queue. The State Monad retains information between messages; if there is a failure that information would be lost. Providing a persistent store can also allow different executing policy scripts to share information about parties in different transactions.

The format of this store can be derived from the data of the attributes as understood by the scripting language, e.g. Agda. The policies determine which attributes are needed to prove or disprove a policy, and which attributes need to persist across messages.

In a complex system such as credit card processing, the evidence we'll need to prove policies will come from a variety of systems, such as bank records, internal credit scoring systems, balances, etc.  Nevertheless, each logical statement in a policy specifies the kind of evidence it needs. The code in the script must have access to routines supplying that evidence or a script will not type check. Any pass through a script leading to a claim a policy is applied needs to find all the necessary evidence.

The discussion above leads directly to a particular approach to policy. Policies are statements of conditions; under certain conditions, certain operations are permitted. As such they are expressible as logical statements. In Agda (and other dependently typed languages serving as theorem provers and programming languages) we can express these policies in a logical language. We can prove properties of a policy directly using Agda as a proof assistant. Policies require kinds of evidence; the policies in Agda specify exactly what kind of evidence is needed.

From there one can implement functions to retrieve that evidence. Finally code can be written or likely generated to gather evidence and prove a given operation is valid or invalid under the appropriate conditions. All this together provides very strong guarantees of correctness.

We can go beyond this. As multiple parties are involved in long running financial interactions, should we not consider implementing digital contracts this way?

## A Rego response

Rego performs as an oracle with respect to its clients. In theory, using Rego splits policy from processing; code doesn't need to know about policy, just needs to get an answer. However, when a client queries Rego it needs to ensure Rego has all the information it needs. As described in the Rego documentation, as that becomes more complex, the processes surrounding Rego to ensure its knowledge base is up to date becomes more baroque. As we've seen, proving is policy is being correctly applied requires various pieces of evidence. In the dependently typed setting, it is obvious what that evidence is and the script can retrieve evidence as necessary. For Rego, all that information needs to be supplied before at query time, meaning that knowledge needs to be exported from the script and into the supporting infrastructure so all possible evidence is available.

We have shown how gathering evidence and applying policies can become quite complex. By functioning only as an oracle, Rego can only function in such a scenario by deep cooperation with supporting code. For example, when approving a transaction is based on the status of other outstanding transactions, the code supporting Rego needs to understand how to provide and update that kind of information, leading to further integration. As Rego does not control the actual communication, it cannot operate on the response before the client. The client needs to understand the various stages of a complex communication and access Rego multiple times as the state of the process evolves.

Inevitably, to keep policy matters away from application code, there will need to be another layer of code that sits between the application and Rego, accessing all the necessary code to add data to the final Rego call and perhaps performing some updates based on the results before returning the final decision to the application. All of this coordination is based on documents flowing between developers of these three components, little of which can be type checked and none of which can be proven correct.

# Other programming languages

What we have demonstrated is based on the particular characteristics of dependently typed languages; it would not generally hold for other languages as it would require a change to the type system. A notable exception is Java's allowance for type annotations and its integration with the Checker Framework.

Java has included annotations since version 1.5 and extended those to types in SE 8. Annotations on types allow developers to add additional constraints on types. At the same time, Java added a plugin framework for the compiler. The Checker Framework (CF) is a compiler plugin enabling users to extend compiler functionality by processing typed annotations at compile time.

CF ships with many annotations, such as a nullness checker, a lock checker, and, closest to this work, a tainting checker. Tainted and untainted values can also be seen private and public, where no private value should be exposed publicly.

With this assist, annotations on Java code can be used to force various operations to be performed with compile time checks, including with relationships among annotations. It can also be used to ensure callouts to policy code, such as Agda or other, and that the results are properly handled. Otherwise a policy framework is entirely dependent on the client to actually implement the decisions of the policy server.

Python has both optional type hints to provide some static checking and decorators to perform a similar service to annotations. While decorators only execute at run time, the pydantic system has managed to combine them both with great power. It would be interesting to combine that with our proposals.

## Conclusion

Current approaches to the access control problem rely on ad hoc policies written in untyped languages. As the policy is implemented in the body of rules, there is no way to test correctness except through observation and testing use cases. There is also no way to separate the specification of the policy and the implementation code checking individual requests.

We demonstrate the applicability of dependent types, especially propositional types, to the access control problem. By specifying policies directly as types in a strongly typed programming language we get the usual compile time guarantees that our policies are well formed as types. However, a policy is also a set of requirements to be fulfilled before an action is taken. Our approach ensures these steps are taken through compile time guarantees; i.e., specifying the policies as types forces the code validating an action to necessarily perform all the necessary steps or it will not compile; the code needs to always prove the policy has been enforced or it does not type check.

Having specified a policy as a set of propositions separate from its implementation, we can directly analyze the policy.  Agda, like other dependently typed languages, such as Lean, doubles as a proof assistant. We can use this to prove desirable properties of our policies, such as preventing simultaneous withdrawals from an account. By including these proofs in our code, we can test for regressions when policies evolve. A regression proof provides much stronger guarantees than any number of regressions tests, the only way to test existing systems. We can also translate the policy into other formats so we can apply both SAT solvers and model checking to provide guarantees.

As there is no separate specification of a policy in Rego, there is no separate specification of the attributes required to validate that policy. Therefore there will be a need for an intermediate layer between application and script to provide them. In Agda, attributes are well defined and with an API the script can retrieve the values it needs.

As policies become more complex, there are interactions among transactions due to race conditions, account limits, time delays, etc. These require maintaining state across invocations. We show how to apply functional programming techniques to minimize the overhead of checking these dependencies, as well as interacting with external transactional stores to share state. We believe the intermediate layer mentioned will become more complex with existing policy languages, breaking down the application/policy barrier.

Currently, any service receiving a message needs to apply policies from a no trust perspective. However we show how to integrate our approach with a verifiable claims infrastructure, such as that specified by the W3C Verifiable Claims WG. In such a system, the client can undertake the overhead of retrieving attributes and proving claims. The proof sketch and signed, verified attribute values can be passed to the server, which only needs to verify the proof and not start from the beginning.

This implies the ability to share definitions of attributes and policies among the entities involved in some transaction, such as a digital contract. We show how the Agda language constructs, such as modules and record types, can serve as a means to specify these, as well as share proofs and back and forth.

, as well as embedding existing models, such as Google's Zanzibar or Amazon's Cedar, so they can be melded with much more sophisticated policies. Uniquely, this includes policies extending over multiple exchanges. Provability extends from just policy definition to ensuring correctness within the code itself.

Provable correctness without loss of power provides several advantages: ensuring your policies are decidable, statically proving your code *implements* your policies under all circumstances, and allow

Provable correctness is a guarantee your code will properly implement your policies. Furthermore, you can prove additional properties, demonstrating your policies are not only obeyed but accomplish your goals.

Policies are a way of discovering the relationship between two objects, a sender and a receiver, and whether that relationship permits the sender to send a message to the receiver.

provably correct
inline, rather than callout
complex
shareable
stateful

I want to convince the reader

1. Web services, even with fairly straightforward requirements, require the flexibility of attribute based access control
2. ABAC languages are not all the same; some are more powerful and/or safer than others. The more your policy can be expressed in the language of the chosen system,

Value of this effort:

-

There are a variety of access control models and languages of varying power to specify them. Of these, Attribute Based Access Control is the most general and OPA's Rego the most powerful language implementing it. Rego (and Hashicorp's Sentinel) implements ABAC through rules in the logic programming tradition: validating a policy is proving some set of attributes resolve to true given the rules. Dependently typed languages (in this case Agda) can not only support policies as logical statement over attributes, the policies are actually dependent types in the language, providing much stronger guarantees, including compile time rule checking for correctness, regression proofs rather than regression tests, the ability to prove properties of the policy independent of examples to exclude harmful behavior by construction, all important properties when large sums are on the line. One can also abstract out policies and look to SAT solvers and model checkers to prove properties.

Rego, like others, tries to function as an oracle, but trying to cleanly separate policy and code requires an intermediate layer to translate, especially as transactions become more complex and interrelated. Using established functional programming techniques an Agda script can integrate that layer and coordinate more sophisticated policies across multiple transactions and multiple scripts. Indeed, we can consider how to implement digital contracts with these techniques.

In the current world, entities interacting on the Internet are on their own to check policies, pushing that overhead onto servers. However, there are attempts to build a distributed authentication environment. We show how to integrate a distributed claims architecture with authorization, pushing some of the overhead from server to client. Formalizing policies in types also makes it easier to share them among parties.

We close by showing how to share policies among parties in an Agda framework, allowing the considerable overhead of validating a policy from the server back to its clients.

OpenFGA also allows conditions so it supports some degree of attribute based processing.

This ReBAC model is a bit different. The model is, essentially, a graph database of relationships. Rights are either direct or propagate through relationships of objects. These relationships can be expressed in and processed through the techniques we've described, but the full model is probably best implemented with a graph database of some sort with traversals performed in the database and the script evaluating the subgraph connecting the principal and the resource.

Given finite principals, objects and message types, the entire answer space is a 3 dimensional matrix with boolean entries. Very quick if accurate (a few lookups), but hard to maintain. If we change a user's rights, or remove the user, we need to track down each entry for each object. But we have no way to address the general constraints of the policies, just a huge list of *true*s and *false*s.  There isn't really a policy *language*; hence the need for all our other proposals.

We will consider the core question of a policy to determine if a *principal* or object *Sender* has the right to send a message of type *M* to another object *Receiver*. There are various formulations of this in different contexts, but they are basically equivalent for our purposes, and we are particularly interested in the Web Services context.

We can create Parent and Child relationships among principals.

(A subsidiary problem of soundness is consistency in the face of updates - while a static system may always give the same answers, updating that system as requests arrive can be difficult. ACLs basically encode the answers to all policy questions, but can be very expensive to update.)

Succinctly specify policies
Prove the policies are consistent
Ensure your policies, as specified, are actually applied
Prove additional properties, i.e., not only are your policies followed but they accomplish your goals

Modern policy languages implement policy *as* code; they are small programming languages in themselves, along the lines of *infrastructure as code*. These have advantages over standard approaches, but policies and code remain separate. We propose instead to exploit dependent typing, found in languages such as Agda or Lean, to implement policy *as type*, where policies become part of the type system itself. This would be difficult in the policy as code languages, as most are not strongly typed themselves.

The primary advantage of this is a move to compile time evaluation of compliance. When either code or policy changes, the impact is immediate. This is true not just for individual calls, but for larger modules as well. Additionally, as dependent typing systems double as proof assistants, where the compiler's own static type checking fails, we can add proofs of correctness. Policy as type also moves responsibility (and overhead) from server to client and allows the client to evaluate a policy and choose among compliant paths earlier in execution. Finally, this exposes a deep relationship between policies and automata.

We will briefly discuss current policy languages for access control, then we will introduce Agda in the context of an existing example from Rego, followed by how to implement general access control in our model. All along we will demonstrate how Agda provides improved guarantees of correctness and completeness, including replacing some regression tests with regression proofs. We then go beyond current practice to specify how to create a decentralized multiparty authorization architecture using Agda's record types and modules to share common definitions and policies, as well as allow individual participants to communicate their policies, pushing much of the policy burden from server to client. Finally we consider complex policies over multiple interacting long-lived transactions, with a brief overview of some other common policy languages.

## Policies as propositions, permissions as proofs

Modern policy languages, such as Rego (OPA) or Sentinel (Hashicorp), are small, domain specific programming languages designed to evaluate a set of rules. An execution is a query against this set of rules. Rego, the more complex of the two, is based on Datalog and essentially implements a restricted Prolog-style backward chaining: a query tries to satisfy some rule, that rule tries to satisfy rules called from its body, and so on, recursively.  There are serious limitations compared with Prolog to guarantee termination and ensure speedy execution.  In both, though, rules are evaluated and either find a solution or fail.

There are a number of other policy languages around with less expressive power. We will discuss some of those before the conclusion, at which point we can address their comparative expressiveness.

We recommend readers familiarize themselves with at least the introduction to the Rego documentation. It includes a reasonably complex introductory example which we will shortly reimplement. We will confine our discussions to Rego, although they apply to Sentinel and similar languages as well.

These languages are examples of Attribute Based Access Control (ABAC). In a run, the system is presented with some (structured) data and asked a query about it.  During evaluation of the query, the system can traverse the data for various attributes and perform calculations.  In the end, the system will either find sets of attributes in the data providing solutions to the query or fail. Generally failure is seen as negation; the query is false. However, in the flawed world of programming there can be many ways to fail to prove something when inputs change subtly.

Rego assumes a world of JSON structured data, possibly validated by a JSON-Schema. This data may be defined in the script, part of the query input, or loaded periodically. The query uses path expressions to find attributes and evaluate rules. Both Rego and Sentinel allow the developer to define additional functions to manipulate attributes.  As programming languages, policy languages can operate over any structured data; policy is just the intended domain.

A program can access these engines at run time either to validate an action and the solution is irrelevant, or to provide some information for further processing (such as a set of permitted values) where the solutions are relevant.

In a general setup, the policy language is interpreted by a policy server. A client sends a query along with some information; the policy server executes the query using that information and its internal state and returns an answer, often a yes/no about an attempted transaction. However, once received, the client is free to obey or not.

## Other policy languages

Several other policy languages are in common use today, although none with the expressive power of Rego. These include Cedar (used by Amazon), OpenFGA (based on Google's Zanzibar), and XACML (Oasis). Cedar and XACML have properties of ABAC, while OpenFGA uses a more explicit graph model, called Relationship-based Access Control. Ory is another PaC language based on Zanzibar.

In Cedar, policies are a set of permit/forbid triples with possible condition clauses. A triple contains a description of a principal, an action (or set of actions), and a resource. A condition is a when clause specifying a boolean relation over attributes of the context or involved entities. Policies are compiled into a PolicySet. A Query is compiled from a tuple of entities for principal, action, resource, and an optional context. A description of the entities is compiled into an Entities object. A Query is submitted to an Authorizer, which takes the PolicySet, the Query, and the Entities. The Authorizer uses these various components to match attributes of the entities against the rules specified in the PolicySet and gives back an answer.

XACML, defined in XML, has a slightly more complex version of the same story. Several systems, such as the Policy Decision Point (which evaluates a request) and Policy Enforcement Point (which intercepts requests and sends them to the PDP), are defined. Once again, there are four components - subject, action, resource, and optional environment. These are evaluated by rules querying their various attributes. XACML has a much richer set of functions for defining rules than Cedar, but the general mechanism is similar.

While both of these are examples of ABAC, they are static descriptions; any processing of data needs to happen before a request is evaluated. They cannot have the flexibility of Agda or Rego.

OpenFGA, based on Google's Zanzibar, uses types and relations between types. It allows easy definition of direct rights (user Alice can edit the Calendar because there is a directly defined relationship), and role based access (Bob can edit the Calendar because he's an Editor, and Editors can edit the Calendar). But roles can propagate through indirect relationships, (Bill can edit the Calendar because he has FullAccess to the Project containing the Calendar, and anyone with FullAccess can edit the Calendar). OpenFGA also allows conditions so it supports some degree of attribute based processing.

This ReBAC model is a bit different. The model is, essentially, a graph database of relationships. Rights are either direct or propagate through relationships of objects. These relationships can be expressed in and processed through the techniques we've described, but the full model is probably best implemented with a graph database of some sort with traversals performed in the database and the script evaluating the subgraph connecting the principal and the resource.

None of these models challenge the power of our model. However, they have, and Zanzibar in particular, supported very large volumes. The ability of our approach, or even Rego, to scale similarly is an open question. Nevertheless, these all lack some of the power provided by an actual programming language.

In considering how to evaluate an access control paradigm, we can imagine a number of important properties we want for our technology. For example, we would like it to be sound, decidable, and expressive. Decidable means one always gets an answer, either yes or no, within a finite time. Soundness, in this context, means you will always get the same answer to the question. Of course, one must distinguish between the model and implementation; a designer may have specified a set of policies that are sound and decidable, but the implementation or the underlying technology fails to implement them appropriately. Finally, expressiveness applies to what you can say in your policy language. The underlying technology is driven by what you can express. What you cannot express you cannot secure; those additional elements require layers of insecure code around your authorization technology, code which is a big security hole.