NORM

Abstract

We propose a neural logic programming language, **N****eural** **O****bject** **R****elational** **M****odels** (**Norm**), primarily for human experts conducting data analytics and artificial intelligence computations. Humans are taught to reason through logic while the most advanced AI today computes through tensors. Norm is trying to close the gap between human logic reasoning and machine tensor computation and to keep human experts in the loop.

Norm expresses the world in **higher-order-logic** and compiles the logic program to **neural network frameworks** and **distributed computation frameworks** to take advantage of advanced algorithms and large-scale parallel computation. Norm takes an **object-relational** semantics to ground logic expressions into a directed factor graph. Deterministic logic reasoning follows the graphical structure and assigns each object a **false** or **true** value. Probabilistic computation follows the same structure but relaxes the binary value to a range between 0 and 1. The relaxation parameterizes the logic reasoning with a neural network that can be optimized through learning algorithms.

Norm is suitable for bridging the collaboration between human experts and machine intelligence where people provide high-level cross-domain knowledge while machine provides specific in-domain optimization. Since the knowledge representation by logic is abstracted from the actual tensor computation, human experts will be able to hypothesize theories, fit and test them against data, and conduct counterfactual analysis without a deep understanding of how neural networks computing. We hope that Norm can enable more people to innovate in their own domains at a much faster pace.

Introduction

Programming languages are designed for controlling machine to compute. As formal languages, they are unambiguous and precise, but sometimes quite verbose and entangled with the implementations carried out in different computing environments. For example, in neural networks, *CNN*, *RNN*, and other layers are invented to architect the computation. People are forced to take a detour from their regular logical thinking to learn about these languages. Natural language, on the other hand, was invented for communication between people. Being concise is one of the objectives to reduce communication costs: that is being more declarative than imperative. According to the **typelogical** formalism, natural language composes meaning through a sub-structural logic. The parsed meaning is represented by a logical expression. Inspired by this formalism, Norm aims to facilitate precise and concise communication between human experts and intelligent machines through **neural logic programming**.

Traditional logic programming languages like Prolog or Datalog work as a deterministic rule engine that searches values of variables that satisfy the logic rules. Both of them are rooted in **first-order logic** (FOL) with subtle differences. Prolog has been extended to combine features like higher order predicates and object-oriented programming to deliver richer programming experiences. Datalog, on the other hand, simplifies Prolog to focus on the deductive database. It is well-known that deterministic rules become unmanageable quickly as the domain size increases. Rules that conflict with each other are quite common. Exceptions always exist. These require significant efforts to keep the system sound and complete. The probabilistic approach relaxes the rigid rules and turns the satisfiability problem into an optimization problem that finds the maximum probable assignments of variables. The model parameters can be learned from data so that conflicts can be minimized during the training process. This has significantly improved the scalability of the approach.

Research work like **Markov logic networks** (MLNs) and **ProbLog** extend the conventional logic programming with a joint probability model. In the training process, the model parameters are estimated from the data. In the inference process, marginal probabilities are computed over variables of interest. Although these have been proven to be more effective than deterministic logic programming, the globally consistent Markovian property is still very difficult to obtain. Certain assumptions like **Closed World Assumption** (CWA) have to be made. Approximation algorithms have to be adopted to reduce the computational time. Even equipped with a theory like **arithmetic circuits**, estimation of parameters from data remains quite a challenging task. In general, probabilistic logic programs are found inferior to **deep neural networks** (DNNs).

During the past decade, AI has enjoyed a huge progress due to DNNs. Instead of symbols, it represents the world in multi-dimensional tensors which carry more contextual information. The computation is undertaken through numerical operations over tensors, e.g., *matrix multiplication* or *gradient descent*. The model usually contains millions of parameters and many hyper-parameters, which relies on big datasets to avoid overfitting and powerful GPUs to train these parameters within a reasonable time limit. They relax combinatorial optimization problems into numerical optimization problems and smoothly parametrize complicated subspaces through nonlinear activation functions. When they are trained well, DNNs perform much better than other AI models during inference. However, because of the high-dimensionality and the non-linearity, DNNs have difficulties to interpret the reasoning to human experts. Approximation mostly loses the fidelity and deviates significantly from the original models on interesting cases. The reasoning provided lacks logical coherence and depth. On the other hand, human logic is also difficult to incorporate into the DNNs. It does not improve the performance of the model. Most of the time DNNs overrule the logic and render it useless. Therefore, experts from domains with in-depth human knowledge like medical, chemistry, and law have a hard time to utilize deep learning technology.

To enable an effective collaboration between logic reasoning and tensor computation, Norm unifies the symbolic view of the world and the tensor view of the world under the **object-relational semantics** (an extension of the **Kripke semantics**): the world is composed of an infinite number of objects where relations form directed edges among these objects. Each object is represented by both a symbol and a tensor. Each relation computes a probability value of the output object given the input objects. The logic computation produces 0 or 1 **probability** representing **false** or **true** value. The probabilistic computation produces a probability between 0 and 1 representing how likely the object is true. For each object, Norm assigns a label to indicate whether it is **witnessed**[1] or not. Witnessed object is labeled as 1 or 0 for being **positive** or **negative**. Norm can construct objects with no witnessed label, i.e., **na**, due to the **open-world assumption** (OWA) that unknowns do not equal negatives. The probability computation of un-witnessed objects is called **prediction**. Otherwise, it is called **interpretation**. The quality of the interpretation can be measured based on how consistent probability values and label values, e.g., **entropy** or **accuracy**. These measures also drive parameter optimization through different learning algorithms. The statistical principles provide generalization guarantees for the prediction quality from the measured interpretation quality. Meanwhile, because the inference still follows the same semantics as human experts expressed, people can understand the computation intuitively compared to a pure DNN architecture. This object-relational semantics establishes a common communication protocol between human experts and machines.

In this document, we introduce the main features of Norm. We describe the computational systems for the deterministic logic and the probabilistic model briefly.

[1] It is also called **observed**

Basic Concepts

Type, Relation and Predicate

A type declares a relation over a set of variables. For example,

```
History(patient: Patient, record: String);
```

Here, `History` is a relation between a patient and a record. The keyword "*History*" is called the predicate. The above **declaration** can be read as *A patient of type Patient has a medical history record of type String*. Semantically, these three concepts are interchangeable in Norm.

An **entity** is a special relation whose variables are considered as **attributes**. For example,

```
Patient(name: String, age: Integer);
```

Object

Constants are objects, e.g. `"Adam"` and `34`. Each object has a particular type. Each type refers to a set of such objects. The **world** is composed of objects. We assume an **open world** which contains an infinite number of objects where most of them are unknowns.

A relation combines a tuple of input objects into an output object, e.g., `@Patient("Adam", 34)`. This is called **grounding**. In Norm, type itself can be grounded to a **type-object**, e.g., `@Patient`. Type-object has a generic type `Type`. See section for more details about Norm's type system.

Every object has a unique id, ***_oid***. We can refer to the object with the *_oid* or with the fully specified symbol, e.g., `@Patient("Adam", 34)`.

However, a partially grounded type is not an object, e.g., `@Patient("Adam")` does not have a *_oid* associated with that symbol.

Unlike in common databases where objects are allowed to be duplicated, Norm specifies that every object has a **unique** *_oid*. In this way, **deduplication** is automatically applied during the construction process.

Variable

Variables are placeholders for objects. A Norm expression is first grounded to fill variables with objects, then infers the probability of these objects. Norm is a 3-valued logic where **na** denotes the unknown objects. In many cases, variable can take na objects.

Type of variables

Every variable has a type which restricts the set of objects to select from. For example, `name` of `Patient` is a `String`. This implies that any string can be a person’s name. We use the symbol “:” to denote the **isa** relationship between a variable and a type.

Scope of variables

A variable can belong to a type or another variable. For example,

```
Patient.name;
History.patient.name;
```

Accessing `name` in both cases are valid but their domains are different. `Patient.name` has the domain of names of every `Patient`, while `History.patient.name` only has names of all patients who have a medical record.

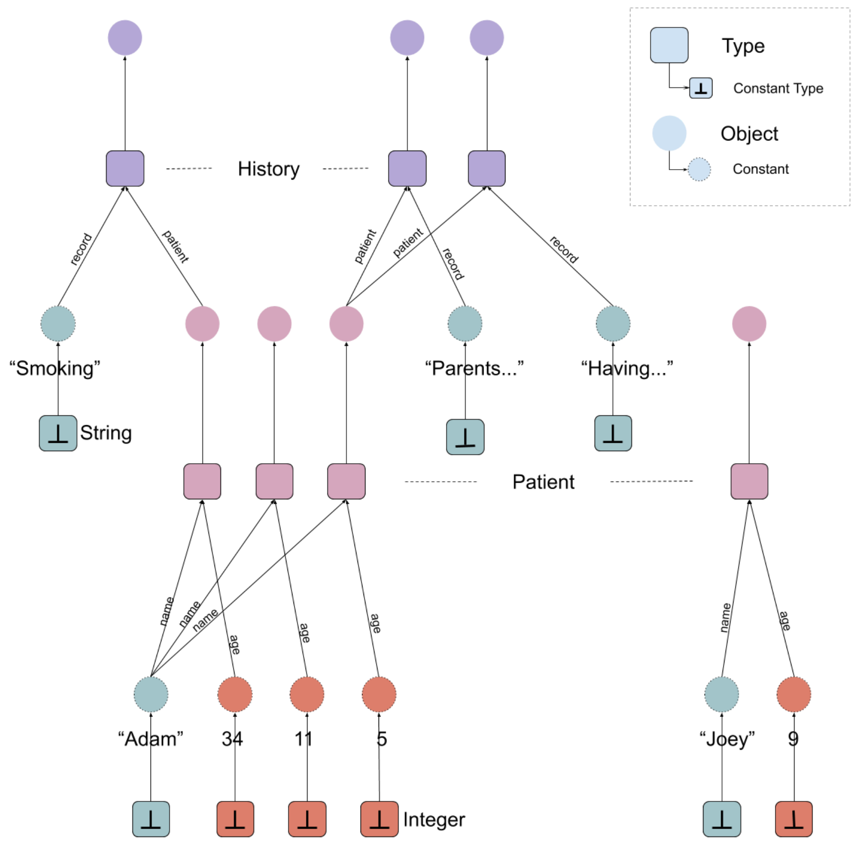
Definition

Declaration of types informs the machine about the structure of types. For example, we know that `History` has a subset of objects from `Patient`. Definition informs the machine about the structure of objects. For example,

```
Patient := "Adam", 34
         | "Adam", 5
         | "Adam", 11
         | "Joey", 9;
History := @Patient("Adam", 34), "smoking"
         | @Patient("Adam", 5), "parents are heavy smokers"
         | @Patient("Adam", 5), "had a fever for 3 days";
```

Here, symbol “:=” denotes **means** relationship, and ‘|’ denotes **or** relationship. Examples above show the **extensional** definition that constructs objects and label them as positives. If a negative label is required, we can specify it at the end of each row, e.g., `Patient |= "Adam", 34, 0`. An **intensional** definition is a logical expression that can be evaluated to the extensional objects.

At any moment, the full extensional definition of the world results in a factor graph. For example,



Every type is represented by a box with different color. Every object is represented by a circle with the same color as its type. Input objects to the type have variable names on the label. The output object simply reflects the composition of input objects. Types like Integer or String are constant types without any inputs. Objects witnessed as positives filled with color. Objects witnessed as negatives filled with gray but edged with color. Unknown objects are dark matters in the world that might not be depicted in the graph.

Expression

With the basic concepts of type, variable, and object, we can form logical expressions to validate a statement or infer variables. For example,

```
// Any child less than 10 year old does not have a tobacco history
forany p: Patient, p.age < 10 =>
     not exist r in p.History.record, r ~ 'smoke';
```

This is a first order logic expression, where symbol "=>" denotes **imply** relation, a.k.a., if … then. Symbol "~" denotes the **like** relationship. Any record evaluating to true is returned as a **positive**.

Any record evaluating to false is returned as a **negative**. Any failed record is returned as **exceptions** or **unknowns**.

For the given example, we have the following results where each object is listed as a row in the table. The variable _label represents the **witnessed**[2] value while _prob represents the **inferred** value. Clearly, the negative object is a counter-example (false negative) that indicates the conflicts of given logic expression and the witnessed world. Domain experts can further investigate and modify hypothesis accordingly.

Positives:

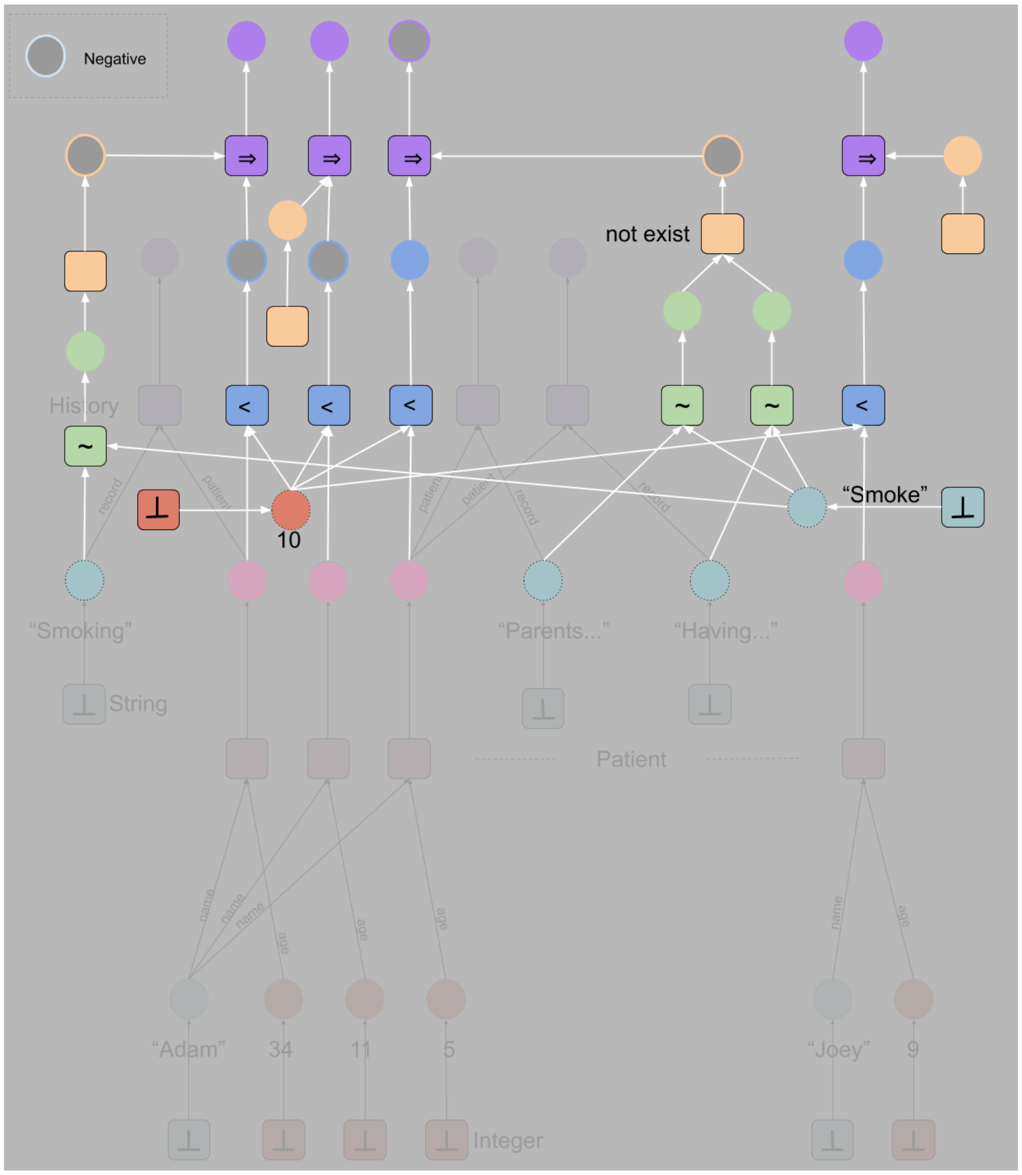
_oid	**p**	**r**	**_prob**	**_label**
$12faefe	@Patient("Adam", 34)	"smoking"	1.0	1
$21feab	@Patient("Adam", 11)	NA	1.0	1
$fa32b3	@Patient("Joey", 9)	NA	1.0	1

Negatives:

_oid	**p**	**r**	**_prob**	**_label**
$a14aee	@Patient("Adam", 5)	"parents are heavy smokers"	0.0	1

The graph representation of the logical path can be depicted in the following figure. The old world is grayed out to emphasize the newly added parts. The actual implementation of the computation depends on the engine we use. For example, when data fit in memory, a python framework **Pandas** can be used to execute the computation. When data are distributed across multiple nodes, a python framework **Dask** can be used to push down the predicates. If GPUs are available, a python framework **cudf** can be used to run the expression. Of course, for the probabilistic computation, Norm compiles to tensorflow or pytorch. We will discuss the system in later section.

[2] Sometimes, it is also called **observed**.

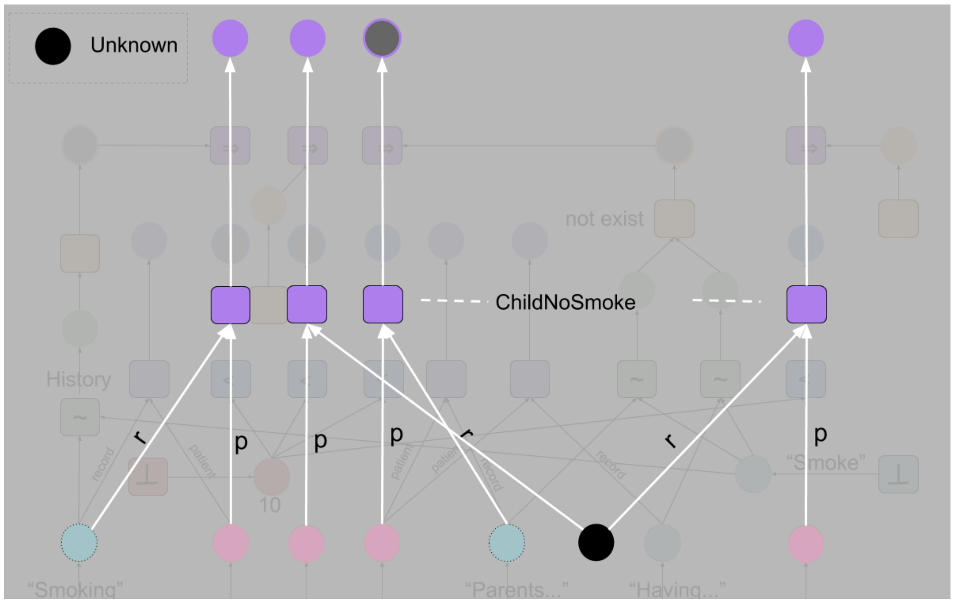


Logic relations like '<', '~', *not exist* and '=>' take the input objects and compute the probability for the output object. For logic computation, the probability to be either a 0 or 1. For probabilistic computation, logic relation produces a probability between 0 and 1. For example, if we query the record by `'tobacco'` instead of `'smoke'`, the "~" relation can produce a similarity measure to decide whether the record implies a "tobacco" history or not. This relation can be as simple as a cosine similarity or as complicated as a Transformer model. Norm utilizes a versioning mechanism to override relations for different definitions either deterministic or probabilistic. People can specify which version to use in the logic expression or rely on an auto machine learning algorithm to decide which version is the best to fit the witnessed world.

Encapsulation

At last, a new type can be defined by removing the quantifiers.

```
ChildNoSmoke(p: Patient, r: String)
       := p.age < 10 => r !~ 'smoke';
```



This practice is called **encapsulation** which is a similar concept in Object Oriented Programming (OOP). Encapsulation not only modularizes the logic expression to reduce the overall code complexity, but it also **factors** out a **lemma** from the world. Other logic expressions can sign up the lemma without re-proving it. The more logic expressions depend on this lemma, the higher impact it is. These high impact in-domain or cross-domain lemmas will be the value of proposition. Inventing new lemmas from data is the task known as *inductive logic programming*. In Norm, this process becomes a complementary structural learning for DNNs. We will discuss these in the later section.

Advanced Features

Data and constants

In Norm, objects are data persisted in disk or memory. Constants are special objects with no further decomposition. The following table lists constant types and their examples:

Type		Example	Default
Boolean		true, false	true
Integer		322, -23	0
Float		2.3, -3e-5	0.0
String		“test”	“”
UUID		$012abe, $latest, $best	$
Datetime		t”12/31/1999 23:59:59”	t”1/1/0001 00:00:00”
Measurement	Distance	3m, 2.3mm	0m
	Time	3M, 41s	0s
	Coordinate	37.7N, 122.4W	0N
	Weight	100lb, 19kg	0g
	Currency	42USD	0USD
NA		none, na, null	na
Tuple		(“test”, 3)	()
List		[1, 2, 3, (“test”, 2.0)]	[]

Norm does not distinguish data from code. An extensional definition can ingest constant data to a type and construct objects automatically. It works just like codes. For example,

```
History := ("Adam", 34), "smoking"
         | ("Adam", 5), "parents are heavy smokers"
         | ("Adam", 5), "had a fever for 3 days";
```

Any intensional definition is first grounded to objects. These ground objects are merged to the previous data together. One convenience provided by Norm is that without explicitly specification of the type `Patient`, Norm aligns the constants with the declared variables and convert the generic tuple `("Adam", 34)` to a `Patient` object. This also helps validate data when a complete and explicit variable declaration is provided in advance.

List type

Similar to normalization in a relational database, it is recommended to have all variables single valued. However, it is convenient to allow multi-valued variables. List type is created to support multi-valued variables. For example,

```
// Explicit declaration of variable is unnecessary most of the time
sentence: [String] = History.record.split();
```

The variable `sentence` is a list of words derived from the medical *history record*. Slicing operator allows referring to individual object, e.g., `sentence[2]`. Given beginning and end indices produce a subset of objects, e.g., `sentence[2..7]`. As a convention, we refer to the last element by -1, and the first element by 0. In general, objects stored for any type are in a list. Slicing any variable refers to a particular object. For example, `History.record[1]` refers to `"parents are heavy smokers"`. Meanwhile, `sentence[3][1]` refers to `"heavy"`.
Note that List is of variable length. If the index is beyond the bounds, out of bound exception is raised, e.g., `sentence[3][0]`.

Continuous Definition

As we have seen in the previous section, the relationship ':=' defines a given type with either objects or logic expressions. Norm supports continuous definitions, '|=' and '&='. They represent "**or means**" and "**and means**" relationships respectively.

```
History |= ("Joey", 46), "never smoked"
        |  ("Jane", 7), "heart problem";
History &= ChildNoSmoke(patient, record)?no_smoke;
```

In the above example, we first append two additional records and then add an additional variable `no_smoke`. The values of `no_smoke` are produced by evaluating `ChildNoSmoke`. Symbol '?' represents a query intention that queries the relation and assign the values to a variable. '|=' acts like a union while '&=' acts like an intersection. After we enforce the medical history to obey the rule of `ChildNoSmoke`, those data who disagree with it turned negative.

Unlike traditional logic program where the same predicate can be defined with multiple rules, Norm enforces that every predicate can have only one definition, all continuous definitions are logically combined to the original one. The modifications can be versioned and isolated to reduce conflicts.

Recursive Definition

The logic relation definition can be recursive. For example,

```
Influence(p1: Patient, p2: Patient);
Smoke(p: Patient) |= exist o = Smoke.p, o.Influence.p2?p & o != p;
```

For the definition, we choose witnessed smokers so far and search for persons that are influenced by them. These persons are unioned to the original smokers. This continuous definition operates one step. To support full recursive definition, Norm use '||=' and '&&=' to denote the recursive operation. For example,

```
Smoke(p: Patient) ||= exist o = Smoke.p, o.Influence.p2?p & o != p;
```

The previous operation will continue until no more smokers added to the relation `Smoke`. We will talk about this fixed point algorithm in later section.

Type of types

Norm conforms to the higher order logic where quantification over types is feasible. For example,

```
Patient(name: String, age: Integer, profession: Type);
```

Here, `profession` is a variable with a type of types. For example, *Adam, age of 34, teaches*:

```
@Patient("Adam", 34, @Teach);
```

`Teach` can be a relation between a person and a course.

Type hierarchy

Inheritance is an **isa** relation between two types. We use the symbol '::' to denote this relation. For example,

```
Teach:: Work;
Build:: Work;
Patient(name: String, age: Integer, profession: Type<Work>);
```

Here, we re-declare the `profession` of `Patient` to be subtypes of `Work`, i.e., `Teach` and `Build`.

Inheritance relationship implies a **possessive** relation that subtypes can be accessed from their parents, e.g., `Work.Teach` and `Work.Build`.

Multiple inheritance can be expressed through a combination, for example:

```
Teach:: Honorable Work;
```

This reads as *Teach is an Honorable Work*. It inherits the intensional definitions of both `Honorable` and `Work`, but not the extensional data because they might not be objects of `Teach`. One can try to convert them by explicitly appending them with '|=', i.e.,

```
Teach |= Honorable? & Work?;
```

To avoid the diamond problem, we follow the **MRO** algorithm to resolve inheritance by the order. Therefore, `Honorable Work` is different than `Work Honorable`.

An inductive type definition is an alternative way to declare the inheritance, for example,

```
Work := Teach | Build;
```

Essentially, it creates a *named union type*. All extensional objects of subtypes will be included. Note that Norm does not support anonymous union types.

Namespace

Namespace allows types to be organized in a hierarchy for the purpose of packaging and scope isolation. When interacting with a Norm interpreter, every session creates a temporary namespace. All types created in the session live in this namespace. Types from other namespaces can be imported to the current scope, for example,

`with www.example.com.Work$112 as Work`.

Types from current namespace can be exported to a specific namespace, for example,

`export Work to www.example.com as WK`.

Versioning

Norm models the world in a continuous fashion, i.e., *interacting is developing*. To maintain the system sound, a versioning mechanism is adopted to track changes, lock dependencies and isolate concurrent operations. Following the **Multi-Version Concurrency Control** (MVCC) strategy, new versions are created when operations like continuous definitions occur. The delta changes are logged to save the storage while a merkle tree is employed to create the delta between two versions. In general, one can refer to a specific version by `Patient$1121abc`. By default, `Patient` is equivalent to `Patient$latest` which always links to the latest version of the type at the compile time. Commands like **merge** and **rollback** are supported to maintain the versions.

Overriding and overloading

The same predicate might work on different types of variables. For example, the **possessive** relation '.' can access variable from a type, and also access a type from a parent type. Both of them active at the same time as long as the context disambiguates the relation semantics by itself. Norm simply uses versions to overload and override types. For example,

```
History(patient: Patient, record: String);
History(person: Person, record: String);
p.History.record;
```

Depending on the actual type of `p`, it links to a different `History` during the compilation. Moreover, if `Patient` is a subtype of `Person`, the compiler will link the lowest type in the type hierarchy that has a version of `History` associated with that type. We will discuss the linking and compilation at later sections.

Anonymous type object

In many functional programming languages, anonymous function is the key building block. In Norm, it can be declared by symbol '@'. For example, `@(x,x2)(x2=sin(x * x))`. This anonymous declaration does not create an explicit type or persist any data. It is treated as an unnamed type object.

Dependent type

A complete static type checking is a *formal verification* which requires a system to implement *dependent type theory*. For example, according to **λ-abstraction**, `@Patient` is an object of type `String & Integer => Boolean`[3]. According to **π-abstraction**, we can declare a parameterized type: `Append<T>(collection: [T], element: T)`. When the parameter is not clear at the time of programming, a generic type `Any` can be used to simplify the type declarations, i.e., `Append(collection: [Any], element: Any)`.

Although Norm encourages a specific type declaration for relations, variables used in expressions are not necessarily specified. *Hindley-Milner* style algorithm can help infer the types and check the validity of the expression. In fact, type checking is a theorem prover that ideally is able to prove whether the expression is true or false before it is semantically executed. In reality, static type checking is limited when the software solution is not obvious because it omits the induction process. Norm, on the other hand, grounds types to objects and seeks a **statistical inductive proof** of the world through machine learning. Whether a formal prover can help a semantic prover or vice versa is an interesting open question.

Evaluation and intention

Construction intention

Given assignments to the variables, a type can be evaluated accordingly. For example,

```
// Construct a new entry in the data if it does not exist
History(patient=("Adam", 34), record="coughing all night");
```

This creates a new entry for the `History` of the patient `@Patient("Adam", 34)`.

[3] According to Curry-Howard correspondence, conjunction is equivalent to product and implication is equivalent to functional, i.e., String x Integer -> Boolean

By default, the constructed object is labeled as positive unless the explicit label is provided: `History(patient=("Adam", 34), record="coughing all night", _label=false);`

`History` has a variable `no_smoke` that is computed from variables `patient` and `record`. During the construction, the intensional definition applies on the input objects and the probability is computed immediately. Therefore, some objects might yield negative probability. For example,

```
// no_smoke is evaluated to false and hence this is a negative as well
History(patient=("Adam", 6), record="heavy smoker");
```

The query returns the object:

_oid	history	_prob	_label
$12faefe	@History(@Patient("Adam", 6), "heavy smoker")	0.0	1

The constructed object is a false positive. To override certain part of logic, we can simply provide the value of that corresponding variable:
`History(patient=("Adam", 6), record="heavy smoker", no_smoke=true);`
The computation shortcuts the graph if the value of the objects is provided.

Sometimes, the intensional definition does not provide logic to compute a certain variable. These variables are expected to be given. In case of missing variables, Norm fills it with a default value: `History(patient=("Adam", 13))` has `record` default to "". See section for all default values of constant types. The default value of type `History` is an object with all variables take default values, i.e., `History();`

Query intention

Besides the construction intention as described above, sometimes we evaluate to **query**. Norm uses a symbol '?' to specify the query intention. For example,

```
history = History(patient=("Adam", 34), record="smoking")?;
```

The query returns the object:

_oid	history	_prob	_label
$12faefe	@History(@Patient("Adam", 34), "smoking")	1.0	1

We can also query any variable instead of the type. For example,

```
History(patient.name="Adam", record?);
```

The above query returns all records for patients named Adam,

_oid	record	_prob	_label
$12faefe	"smoking"	1.0	1
$21fsfsf	"parents are heavy smokers"	1.0	1
$fa1231	"had fever for several days"	1.0	1

For any object who have already been witnessed, query intention evaluates to the same results as the construction intention. However, when un-witnessed objects provided to the type, query intention does not label it with positive or negative, but unknown instead. The result is different than the construction intention. For example,

```
Patient("Jack", 41)?
```

The above query returns the following object:

_oid	Patient	_prob	_label
$12fase	@Patient("Jack", 41)	1.0	na

Query projection

Norm provides a syntax to project the query results into different variables. For example,

```
History(patient.name="Adam"?patient, record?history);
```

It returns two variables:

_oid	patient	history	_prob	_label
$12faefe	@Patient("Adam", 34)	"smoking"	1.0	1
$21fsfsf	@Patient("Adam", 5)	"parents are heavy smokers"	1.0	1
$fa1231	@Patient("Adam", 5)	"had fever for several days"	1.0	1

Projection is a convenient syntax to assign variables in a *left-to-right* composing convention. Of course, Norm still supports traditional assignment relation that assigns objects to a variable:

```
patient, history = History(patient.name="Adam"?, record?);
```

Limited query

When the extensional definition contains too many data, query result might exceed memory limit. For example, `History?` returns all records. To limit the number of results, Norm allows query to limit. For example, `History?100` returns results up to 100. If the number of records is less than 100, the results return all records.

Limit query often combines with sorting. By default, Norm sorts according to probability from high to low then by primary variables in the declaration order. However, one can always declare a different ordering for any variable, see section.

Possessive Evaluation

Norm allows the type to be evaluated as a possessive access, for example,

```
Patient("Adam", 34).History(record?history);
```

The above treats the entire expression on the left side of the '.' symbol as the first variable argument. For a binary relation, it is further simplified:

```
Patient("Adam", 34).History.record?history;
```

If there are no more arguments passed to the variables, we can further save the brackets:

```
Patient("Adam", 34).History?history
```

Currying

Currying is very similar to evaluation except that it produces a type object instead of an object of the type. For example, `Patient<"Adam">` produces `@Patient<"Adam">` which is a type-object represents a subset of patients whose name is *Adam*. On the other hand, `Patient("Adam")` produces `@Patient("Adam", 0)` which is an object of `Patient`.
Currying facilitates more complicated higher order logic computation.

Quotation and Unquote

In natural language, people use quotations to mention the literal text instead of the meaning of it. But sometimes, we do want to use the meaning of the quotation. This operation is called **unquote** in Norm. "{<expression>}" denotes the unquote relationship. For example,

```
// We assign values to the variable, it could be results from an expression
worker = ["Teach.teacher", "Build.engineer"];
{worker}.name;
```

The expression returns the union of names from `"Teach.teacher"` and `"Build.engineer"`.
Of course, the example is a bit trivial. We can achieve the same by simply using disjunction.

```
Teach.teacher.name | Build.engineer.name;
```

Unquoting becomes useful when dealing with a large number of parsing work, e.g., pivoting.

Time series

Norm is a continuous programming language, every object it computes is stamped with a UTC time. Internally assigned to a variable **_time**. By default, Norm is a single time series where _time refers to the created time and the recorded time. If two-time series is needed, _time is considered as the recorded time, the created time needs to be specified explicitly.

Norm provides two windowing mechanisms, interval and rolling. For example,

```
Device.interval(5s).temperature.mean;
Device.rolling(5s).temperature.mean;
```

This computes the mean of the temperature every 5 second with interval windows or rolling windows.

Variable property

The declared variables in the function can take properties.

```
History(patient: Patient primary, record: String primary);
```

Primary/optional

Primary variable is used to generate oid which determines the uniqueness. The default declaration is primary, e.g., `History(patient: Patient, record: String);`
Any variable created without explicit declaration is considered optional, e.g., `no_smoke`.
If specified, the declaration should be:

```
History(patient: Patient primary, record: String primary,
        no_smoke: ChildNoSmoke optional);
```

A constant can passed to the primary to define the default value, for example,

```
History(patient: Patient primary, record: String primary(na),
        no_smoke: ChildNoSmoke optional);
```

Instead of empty string, `record` defaults to `na` now.

Time

This indicates that the variable is also aliased to _time. For example,

```
Device(name: String, measured_at: Datetime time, temperature: Temperature
optional);
```

This declares that a device records the temperature at `measured_at` time which is also the _time internally. This `measured_at` is also a primary variable used to distinguish the records.

Oid

By default, Norm computes the oid by hashing the combination of primary variables with a cryptographical function. The result is difficult to remember. If the user knows a variable which contains unique values, oid can be used to specify the alias to _oid.

Desc/asc

Ordering of the variable accordingly. By default, desc order is taken. The ordering of the variables also matters during sorting. For example,
`Device(name: String, measured_at: Datetime asc, temperature: Temperature);`
sorts by name, measured_at, temperature in order by ascending on measured_at.

Parameter

Some variables might not be changing for different values in other variables. These can be specified as parameters. For example,

```
Device(name: String, measured_at: time, temperature: Temperature optional,
unit: parameter(‘F’));
```

In probabilistic computation, a parameter can be used for optimization with respect to an objective function.

Output

This property is used to declare a functional type instead of relational type. We will discuss it in the following section.

Functional type

Norm is not a functional programming language, but supports functions by declaring variables to be output. For example,

```
ChildNoSmoke(p: Patient, r: String, no_smoke: Boolean output)
     := (p.age < 10 => r !~ 'smoke')?no_smoke;
ChildNoSmoke(Patient("Adam", 8), 'drink');
```

The above can be considered as a convenient shortcut for:

`ChildNoSmoke(Patient("Adam", 8), 'drink', no_smoke?);`

Functional type supports functional chaining instead of relational chaining, for example:

```
// functional chaining
Device.temperature.mean().log();
// relational chaining
Device.temperature.mean()?average_temperature & log(average_temperature);
```

It might also affect the neural network layer to be used. An explicit output variable might be associated with a **Softmax** layer while a relational type might use a **Sigmoid** layer. We will discuss this in a later section.

Atomic type

Atomic type does not keep the extensional definition a.k.a. data. Many math functions, e.g., sum and std, are atomic since no actual input and output are memorized for those functions.
A type can be declared to be atomic if no variable is primary, for example,

```
// explicit optional variable declarations
ChildNoSmoke(p: Patient optional, r: String optional);

// implicit optional variable declarations within definition body
ChildNoSmoke := p: Patient & r: String &
                p.age < 10 => r !~ 'smoke';
```

Without primary variables, a type can not produce _oid, which disables the extensional definition.

Quantifier

Quantifiers in higher-order logic, i.e., ∀ ∃ are represented by **forany** and **exist** respectively. In Norm, **forany**, **forevery**, **foreach** and **forall** are considered equivalent to each other. They define variable **domains** for the logic. For example,

```
// variable p is a Patient
foreach p: Patient, p.age < 10;

// variable age and name are in the scope of Patient
forevery age, name in Patient, age < 10 & name?;

// variable age is in the range of 1 to 10
forall age in [1..10], sum(age < 8);

// variable age in the scope of Patient
// count the number of children for every age less than 10
forany age in Patient, age < 10 & num_of_children = count();

// variable r is in domain of History.record
exist r = History.record, r ~ 'smoke';
```

There are three ways to quantify the variables: **isa (':')**, **is ('=')**, **in**. The last one has two versions, variables **in** the scope of a type and variables **in** an expression.

In terms of the graph representation, quantifiers specify the relations among particular sets of objects. This structural definition allows human experts to experiment or explore different compositions over the domain. We will discuss the structural specification for deep learning in later section.

Scoping

In Norm, each type has a scope of its namespace or its parent type. We can specify the scope by the phrase **with**. For example,

```
with Patient from www.example.com,
    forany p: Patient, not exist r = p.History.record,
        p.age < 10 => r ~ 'smoke';
```

Computation

Table and schema

Semantics of a logic expression in Norm are a set of objects in the world. They are considered as data. Norm takes a relational database approach to organize data in tables. Each non-atomic relation can be considered as a schema in the database. Primary variables are the primary columns. Variable type defines the foreign keys that allow tables to be joined. Norm takes columnar store as the persistent layer to store these objects. To support distributed computation, Norm enforces the data to be immutable. The continuous definition computes a delta over the data and stores them separately. In this way, Norm tracks the full lineage of the data to ensure the integrity and reproducibility.

Relational algebra

The semantics of logic connectives can be mapped to the relational algebra, so deterministic propositional logic computation can be implemented through relational operators similar to SQL.

	Relational algebra	Logic	
Intersection	R(x,y) ∩ T(x,y)	R(x,y) & T(x,y)	
Union	R(x,y) ∪ T(x,y)	R(x,y) | T(x,y)	
Difference	R(x,y) - T(x,y)	R(x,y) & !T(x,y)	
Join	R(x,y) ⋈ T(y,z)	R(x,y) & T(y,z)	R(x,y) & T(z=y,w)
Filtering	σ(x>10)R	R(x,y) & x > 10	
Product	R(x,y) × T(z, w)	R(x,y) & T(z)	

In Norm, positives and negatives are represented by _label and _prob for witnessed value and predicted value respectively. Unknowns are not treated as negatives. In this sense, logic connectives are not exactly the same as the relational operators. Norm first apply relational operators to construct objects, and then apply the logic operators to compute the probability. According to the intention of the evaluation, the constructed object may or may not be witnessed.

Stratified Recursion

When recursive definition contains negated relations, it might cause an infinite loop. For example,

```
Influence(p1: Patient, p2: Patient);
Smoke(p: Patient) ||= exist o in Smoke.p, o.Influence.p2?p & o != p
                       & !Smoke(p)?;
```

The probability of a patient smoking could oscillate between 0 and 1 from iteration to iteration. Norm raises an exception after a certain number of iterations. To avoid the infinite loop, we can restrict the number of negated relations involved and enforce that no self negation.

Control flow

There are two ways to branch logic. The first one is the ‘=>’, a.k.a., ‘if .. then’ logic. For example,

```
ChildNoSmoke := p.age < 10 => r !~ 'smoke';
```

The second approach to branch logic is through dynamic dispatching. For example,

```
Child:: Patient := age < 10;
Adult:: Patient := age >= 10;

ChildNoSmoke(p: Child, r: String) := r !~ 'smoke';
ChildNoSmoke(p: Adult, r: String) := true;

ChildNoSmoke(('John', 4), 'smoke'); // evaluate to a negative object
```

Dynamic dispatching unwraps complex logic, and allows cleaner code. One advantage of the first implementation is that the branching can be parameterized, for example,

```
ChildNoSmoke(p: Patient, r: String, th: Integer parameter(10))
      := p.age < th => r !~ 'smoke';
History.fit();
```

After fitting, we might find out that 6 year old is the best threshold instead of 10.

Aggregation

Data analytics involves many different kinds of aggregations. Norm express these aggregations logically. For example,

```
// Find those publishers who has a publication sold at an average price
//     1000 and a publication sold lower than 500;
With publication,
   foreach publisher,
      exist publication_id, price.mean > 1000
     & exist publication_id, price.min > 500;
```

As a comparison to SQL, here is another example:

How many states have some college students playing in the mid position but not in the goalie position?

A regular SQL is written as:

```
SELECT COUNT(*) FROM
  (SELECT T1.state FROM college AS T1
                   JOIN tryout AS T2 ON T1.cName = T2.cName
                   WHERE T2.pPos = 'mid'
   EXCEPT
   SELECT T1.state FROM college AS T1
                   JOIN tryout AS T2 ON T1.cName = T2.cName
                   WHERE T2.pPos = 'goalie')
```

People have to focus on the actual implementation like join and except instead of the logic itself.

Norm is much more straightforward:

```
forall state in college,
        exist pPos in tryout, pPos == 'mid'
     & !exist pPos in tryout, pPos == 'goalie';
// ‘that’ is a special variable refers to the previous expression result
count(that);
```

Pivot

Pivot is a common analytics tool to summarize aggregation results for each category in one or multiple dimensions. For example,

```
// Alarm tracks events from an ip address
Alarm(event: String, ip: String, time: Datetime, tally: Integer);

// Computes on each ip what are greater than 1000 total tally of each event
with Alarm, foreach ip,
       tally.sum > 1000 ?total_tally_of_{event};
```

Pivoting in Norm is expressed by the unquoting syntax. Manually constructing variables for event categories is possible but tedious and prone to errors. These variables provide new insights and are possibly used as additional features to improve the predication performance of the model.

Relaxation

Logic expressions are often strict, i.e., ‘hard’. Sometimes they are precise, but miss a lot of cases. The relaxation softens the hard rule so that it can increase the coverage significantly without losing much precision. One way to do it is through parameterization like the example shown in the previous section:

```
ChildNoSmoke(p: Patient, r: String, th: Integer parameter(10))
      := p.age < th => r !~ 'smoke';
History.fit();
```

Whenever a relation contains a parameter, the process of fitting searches for the best value with respect to an objective, e.g., accuracy.
Moreover, any relation in Norm can be relaxed with a different version. For example,

```
ChildNoSmoke.relax(~, ~$transformer);
```

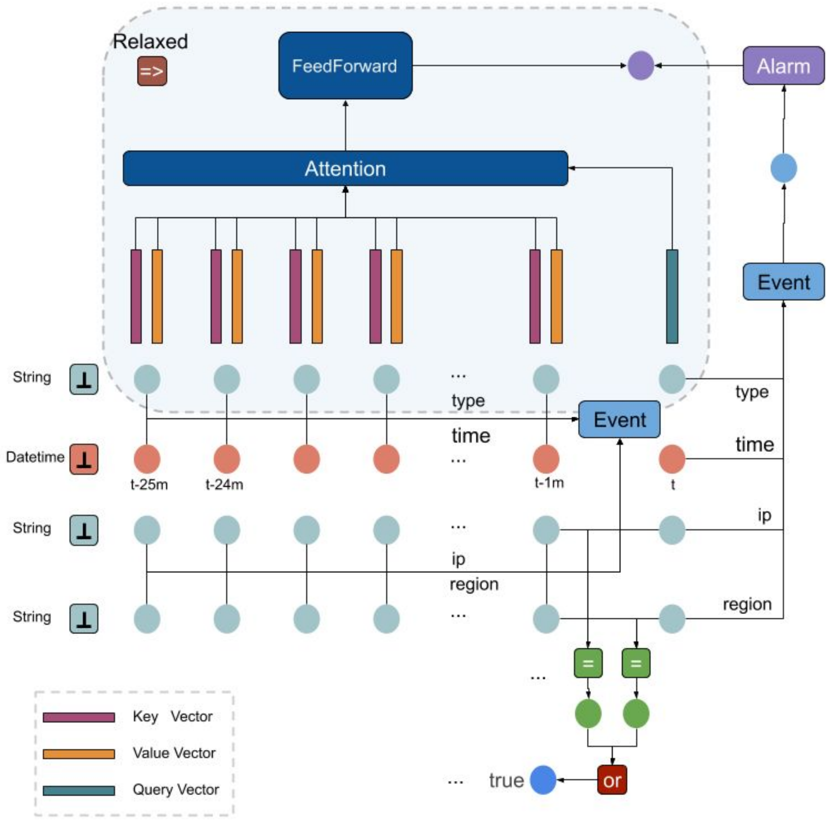
The regular expression based ‘like’ operator is replaced with a transformer based similarity operator. With this new definition, a fit process will optimize over the transformer parameter and the threshold parameter simultaneously. However, transformer contains much more parameters and is prone to overfitting. It is likely that tuning the threshold yields better accuracy (cross-validation for example) than tuning the DNNs. Automatic machine learning algorithm can choose the best relaxation from different versions if no specific relaxation is provided:

```
// Relax the 'like' operator and choose from all versions
ChildNoSmoke.relax(~);
// Relax every relation used in ChildNoSmoke if available
ChildNoSmoke.relax();
```

Factor graphs

The object-relational semantics grounds objects to a directed factor graph. For an example,

```
// Given an IoT device records an event stream data
//  and a triggered alarms
Event(type: String, ip: String, region: String, time: Datetime time);
Alarm(event: Event);
// Data are loaded from a real time event queue like Kafka.
// Can we discover the root cause events for each alarmed event type?
Alarm := exist e: Event, e.before(event, 25m)
             & (e.ip == event.ip | e.region == event.region)
             & e.type => event.type;
Alarm.relax(exist).relax(=>);
```



The above graph is one slice of the entire world for each alarmed event. The existential quantifier builds connections among each event and its previous events within 25 minutes who occur on the same ip address or the same region. When we relax the quantifier and the implication relation, we build an attention based model to predict whether the event should be alarmed or not. The relaxed attention cell is very similar to *Graph Attention Networks*.

Note that to enable attention mechanism, each object is associated with a **query** vector, a **key** vector, and a **value** vector. For other types of layers like CNNs or RNNs, value vector will be used. If a recursive definition is involved, the GAT layer can be stacked up to mimic the message passing algorithm.

Logical explanation

One of the major advantages of Norm is that inference follows the same semantics as human logic. Therefore, the explanation of the inference can be traced back along the attention weights. This also helps to infer the causal relationship among the objects. To further concentrate the attention weights for a focused explanation, Norm favors a sparse attention network so that most of the attention weights would be zeros.

Multi-variable inference

Norm forms a directed graph, the probability model is considered as a *Bayesian Networks*. Inference over multiple variables can be expensive. A commonly used algorithm, i.e., *beam search,* is implemented for Norm. With *branch and bound* optimization, beam-search can be shown to be ε-optimal with a high confidence under the low-noise condition.

Learning with unknowns

In Norm, unknowns do not equal negatives. There is an infinite number of unknown objects not recorded inside Norm. Meanwhile, negatives are likely mixed with unknowns. Several learning algorithms can help train the model with noisy negatives and a large number of unknowns. **Semi-supervised learning** can help to propagate positive labels to unknowns. **Active learning** can help domain experts to efficiently verify predicted examples or label critical unknown examples. **Weakly-supervised learning** can create a lot of labeled (but noisy) data from unknowns to improve baseline performance. **Adversarial learning** like GAN can help further reduce the noise in the data and improve the model robustness. **Self-supervised learning** can utilize a large amount of unknowns and obtain high quality tensor representations that can be transformed to domain with small number of labeled data. These advanced learning algorithms can be

Inductive logic programming

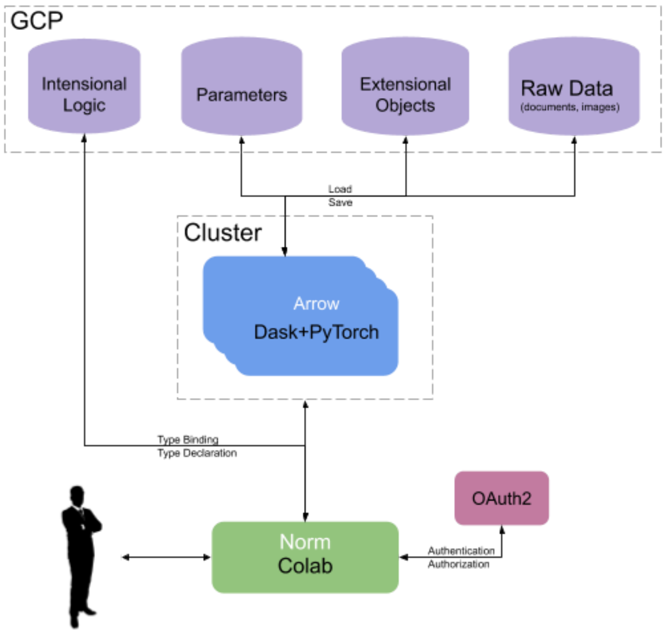
Not only domain experts can explore new logic expressions, machine intelligence can do too. **Structural learning** Algorithm like differential inductive logic programming can induce new logic program from data. Learning dependencies can help discover causal relations. Automatic factorization can optimize logic structure and improve generalizability.

Sequential decision making

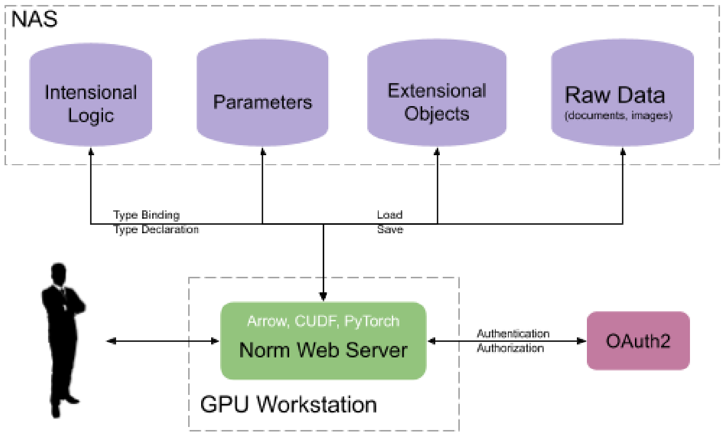
At last, when problems involve a sequence of hidden variables, e.g., the navigation path towards a goal, **reinforcement learning** can help significantly reduce the search space. Overall, advanced learning algorithms are key value of the intelligent machine to help experts optimize their logic programs. The details of these algorithms are beyond this document.

System

Norm is a protocol between human experts and the machine intelligence. It facilitates a direct conversational interactions. The logic program compiles down to the APIs provided by frameworks like *PyTorch* and *Dask* for high performance large scale computational power. Meanwhile, Norm binds to a data storage layer. File-based columnar store like Parquet to save relational data. Apache Arrow helps efficiently manage data in-memory. Logic programs are stored in a file-based relational database like sqlite for version control. Raw data like images and documents are stored in files too. A Cloud storage system like S3, GCP or Azure datalake, or an on-prem local storage can be abstracted from the storage layer in Norm. Since computation in Norm is immutable, the storage layer is ACID compliant.



Deployed on Google Cloud



Deployed on-prem