Hydra: high-level design

bit.ly/hydra-design-doc

Note: this is a design document which was written before Hydra was implemented. The implementation follows this design in broad strokes, but has also evolved significantly (e.g. instead of STLC, Hydra's type system is based on System F, with a variant of Hindley-Milner type inference. Additional progress has been made in unifying graphs and lambda calculus). A more formal and complete specification now exists at LinkedIn, and will be published eventually.

Overview

Hydra is a new open-source framework for data integration, currently in the design and prototyping stage. Desired capabilities of Hydra include building property graphs and RDF graphs out of diverse non-graph data sources, transforming schemas and data consistently between languages, performing strongly typed graph traversal and graph stream processing, and enforcing integrity constraints in heterogeneous data environments. One particular application area of interest is boosting interoperability in Apache TinkerPop.

Hydra is a successor of the proprietary Dragon toolkit (see also How to Build a Dragon), but whereas Dragon's graph data model (APG) is based on a system of algebraic data types, Hydra's is based on an generalization of APG which also makes it a typed lambda calculus. That means Hydra will be as much a programming language as a schema language, and much more dynamic and flexible than Dragon with respect to the operations you can perform on your data. For specific differences, see below. One key feature of Hydra is that it will be capable of transforming its own source code into multiple compiled programming languages, making it highly portable. The first implementations will be Hydra-Haskell (the bootstrapping implementation) and Hydra-Scala.

This document provides an overview of the current design plan for Hydra, primarily the data model and computational model, as well as some aspects of implementation and future applications. The document will evolve as development of Hydra progresses. For more information, join the Hydra Discord server and/or watch the CategoricalData/hydra GitHub repository.

For an overview of the terminology used in this document, see below.

Initial features

Some anticipated features and components of Hydra include:

- A core data model (Hydra Core) which is both a graph data model and a typed lambda calculus
- Abstractions for bidirectional mappings (Step), errors and warnings (Qual), supported languages and features (Language), graph constraints (Constraints), etc.
- An abstraction for primitive functions (including UDFs) with language-specific implementations
- An abstraction (Module) for hierarchical organization of graph elements
- Type inference and type checking according to the Hydra Core type system
- Basic evaluation / reduction of terms, including application of primitive functions
- Bidirectional coder to/from JSON and YAML
- Bidirectional coder of Hydra Core to and from RDF with SHACL
- Bidirectional support for one or more data exchange languages, such as Avro, Protobuf, or PDL
- File I/O for dependency fetching and data output

Use cases

Initial Hydra use cases will be similar as those of Dragon, with some additional possibilities around user-specified transformations, and integration with open-source frameworks including Apache TinkerPop. For example:

- Transforming standardized data type definitions into multiple application-specific languages
- Validating, analyzing, and cataloging data type definitions in various formats
- Migrating services from one schema and data language to another
- Exporting schemas and data to RDF for visualization or querying
- Transforming data on the fly between exchange languages (e.g. Avro to Protobuf)
- Carrying out domain-specific transformations (i.e. schema migration)
- Transforming structured but non-graph data into graph data
- Serving as a well-specified schema and data language for graph applications

- Serving as a neutral language for programming logic which is needed in more than one language (e.g. Gremlin steps in language variants)

See also Applications.

Development process

Hydra development will proceed in three overlapping stages:

- **Design**: writing of this document, prototyping, settling the open design questions, and creating the initial Hydra Core model
- **Bootstrapping**: completion of an initial, minimal implementation in an appropriate language (the "bootstrapping language"), such as Scala or Haskell. Development of initial encoders and decoders, including (at a minimum) general-purpose YAML and JSON coders, as well as a coder for the bootstrapping language.
- **Closing the loop**: at the end of this stage, Hydra will be capable of reading its own source code (modulo built-in library functions) and generating implementations in one or more additional programming languages. Scala, Java, Go, Python, and Haskell are of initial interest.

Encoders and decoders

Transformations of data between languages are to be performed with the Hydra Core model as an intermediate representation. Those transformation steps which carry data from the Hydra Core model into an external model are called **encoders**, while those which carry data from external models into Hydra Core are called **decoders**. We will refer to an information-preserving encoder/decoder pair as a **coder**.

First-mile and last-mile steps

We can also speak of encoders and decoders with respect to some other reference model, such as **first-mile** and **last-mile** steps which carry data between a third-party API and a Hydra-specific model of a given language.

> **Example**: an Avro schema model will be included as part of Hydra, and will be the target of an Avro schema encoder and decoder. However, in order to map Avro schemas into the org.apache.avro Java API, you need a last-mile encoder. This step consumes expressions in the internal Avro model and produces org.apache.avro.Schema objects.

Unlike Hydra-internal steps, first-mile and last-mile steps must be written in a specific implementation language, and cannot be generalized or transformed between languages. Supporting them will typically require adding a dependency upon a third-party library.

Implementation

As mentioned above, it is a design goal to avoid having to create entirely separate Hydra implementations when they are needed in different languages, which involves a lot of duplicated effort, and concerns about feature parity across implementations. Dragon has this problem (though it alleviates the problem by generating over half of its own Haskell and Java source code from common YAML models), and so does Gremlin with its language variants. Hydra aims to close that gap by generating all of its own source code, minus primitive function implementations and language-specific DSLs, from a common source of truth.

Function libraries

There are some functions, particularly basic functions like string manipulation, arithmetic, etc. which are best implemented on a per-language basis, and which we call primitive functions (see below). For example, a string-concatenation function cannot be implemented natively in Hydra in a practical way. Instead, we will define a “concat” function and require a separate implementation of “concat” in each Hydra implementation. The work of implementing a primitive function will therefore be duplicated in each language variant, although the core primitives will generally be extremely simple -- just a matter of populating a generated PrimitiveFunction object with an appropriate function definition, like “\x y -> x ++ y”.

Source-of-truth language

The initial source-of-truth language for Hydra is Haskell, with some prototyping also being done in Scala 3. After the bootstrapping phase, more emphasIs will probably be placed on Scala as a source of truth due to its greater popularity, better IDE support, etc. While all of Hydra’s core logic will eventually be available in each language variant thanks to code generation, the original, hand-edited sources for any given module will only be available in one language, and these will tend to be the most readable for developers. It is possible that we will have a mix of source-of-truth languages, depending on what is contributed by whom.

Regardless of Scala vs. Haskell, several options were considered for bootstrapping:

1. Pick a source of truth language, and do not attempt to map function implementations to Hydra terms (i.e. no parallel implementations during the bootstrapping stage). This is the easiest option initially, but all of these functions would have to be rewritten later, in the “closing the loop” stage.
2. Specify the implementations in Hydra’s YAML format. This is easiest from a mapping point of view, with the option of “throwing away” the YAML later on without loss of

information. However, the implementation code would be very verbose -- possibly verbose enough that it would be hard to actually write programs this way.

3. **Use language-specific DSLs**. Create a convenient domain-specific language within the source-of-truth language, and use that to specify Hydra's core models. For example, instead of creating a lambda in YAML or JSON like: {"lambda": {"parameter": "x", "body": { "application": { "left": {...}, "right": {...}}}}}, we can write Scala expressions like Lambda("x", Application(..., …)), with helper functions as needed. This makes the code more compact, readable, and extensible, without making it less portable -- we can still map the code from the source-of-truth into any other language for which we have an encoder. All we need is a way of mapping the DSL-based expressions into Terms, which is easy enough using reflection or code generation.
4. Implement the needed functions idiomatically in Scala or Haskell. E.g. instead of creating a Lambda object as above, just define a function in Scala: def myfunc(x:Int) = x + 1. We could lean even more on IDEs and compilers for developing the code, but Instead of compiling it directly as in Option 1, parse it and map it into Hydra. This would give us the most compact and natural implementation code, though there is a higher up-front development cost to creating the parsing and mapping logic.

At this time, Option 3 is being pursued, though Option 2 will be supported during the bootstrapping phase in any case. Option 4 is still being considered for the "closing the loop" stage, though it is speculative until we try it. In general, Option 1 is best avoided, as it could be easy to become dependent on this non-portable code if it works well initially, but there probably will be a few hard-coded functions as we transition from prototyping to bootstrapping.

Data model

Hydra is based on a system of algebraic data types together with function abstractions. While Hydra's data model deviates from the original formulation of Algebraic Property Graphs (APG), it is closely related, and can be considered as a generalization of APG. See below for specific differences. Currently, Hydra's formal type system is a variation on the simply typed lambda calculus.

Graphs

A Hydra **graph** G is a set of valid **data** terms G(D) together with a set G(E) of distinguished (name, schema term, data term) triplets called **elements**, where the data term **δ**(e) of an element e ∈ G(E) is a term in G(D), and the schema term **σ**(e) is a term in $G_S(D)$, where G_S is another graph, called the **schema graph** of G. Elements and terms will be described in greater detail in sections that follow. The set of data terms G(D) is usually infinite, and simply tells you what sort of expressions are allowed in a particular graph, while the set of elements G(E) is usually finite, and represents the meaningful information in the graph.

Example: you can think of a graph element such as a “knows” edge between Jack and Jill as an ordered triplet having a name (a unique identifier; usually called “id” in property graphs), a type (represented by a term, or expression in Hydra’s Term grammar), and a value (represented by another term): (e1, {“out”: ->Jack, “in”: ->Jill}, ->knows), where ->e can be read as “reference to element e”.

An element can have any name, but it must be the case that the data term of the element (indicated with the **δ** arrows in the diagram below) conforms to the schema term of the element (indicated with the **σ** arrows), where the type conformance relation (indicated with the **τ** arrows) is graph-specific.

$$\begin{tikzcd}
G(E) \arrow[d, "\delta"] \arrow[dr, "\sigma"] & G_S(E) \arrow[d, "\delta"] \arrow[dr, "\sigma"] & (\ldots) \\
G(D) \arrow[r, "\tau"] & G_S(D) \arrow[r, "\tau"] & G_{S'}(D)
\end{tikzcd}$$

Notice that the schema terms of one graph -- G -- are the data terms of another graph -- G_S -- the schema graph of G. The Hydra Core model (see below) is its own schema graph; it defines the Term type as well as the Type type. The latter is an instance of itself according to the Hydra Core type conformance relation. You might call Hydra Core a single-level graph.

Example: a practical example of a multi-level graph is a collection of messages in Protocol Buffers, or their representation in Hydra. G(D) is the set of Protobuf messages and subexpressions which are valid in a given domain, while G(E) is empty (there are no distinguished expressions). G_S(D) is the set of types in a particular Protobuf schema, while G_S(E) is the set of message and enum types (distinguished types identified by a type name) defined in that schema. Going further, $G_{S'}$ is Hydra’s Protobuf model, with $G_{S'}$(D) the set of valid Protobuf type expressions, and $G_{S'}$(E) the set of named types like ProtobufType which generate all possible Protobuf schemas. Finally, $G_{S''}$ is the Hydra Core model, with $G_{S''}$(D) the set of all valid Hydra type expressions, and $G_{S''}$(E) the set of named types in the Hydra Core model, such as Term, FunctionType, and AtomicType (see below).

Models

Like Dragon, Hydra will rely heavily on abstract **models**, i.e. small, human-friendly graphs that usually consist only of at most a few dozen elements intended as types for other graphs, each with descriptive names, good documentation, and a clear purpose.

Example: a model of Neo4j graphs would have an element named Node with a description like "A node in the graph with properties and relationships to other entities", an element named "Relationship" with a description like "A relationship between two nodes in the graph", etc. Let's call this model Neo4jGraph. An actual Neo4j graph G would be a Hydra graph in which the elements are Nodes and Relationships, i.e. σ(e) for each e is either ->Node, ->Relationship, etc. and δ(e) is a term representing the data of a Node (including its id, label(s), and properties) or of a Relationship (including its id, label, properties, and a reference to the out- and in-nodes), etc. The conformance relation τ between G and its schema graph Neo4jGraph is specific to that graph, but the same rules could be used for many different Neo4j graphs.

A model is a lightweight specification of a **data model** -- enough for Hydra to map valid expressions into or out of them. In the case of TinkerPop graphs, for example, A model is usually provided with a specification of **constraints**, as well. These fall into two broad categories:

- Type constraints: limit the types which can be defined, e.g.
    - Is a given Type allowed as an element's schema term?
    - Is a given **type relation** allowed between two given type variants, e.g.
        - Can a record have union-typed fields?
        - Can a map be keyed on lists?
        - Can lists contain other lists?
- Structural constraints:
    - Are cyclical dependencies among elements (types) allowed?
    - Are cyclical dependencies within namespaces allowed?

Essentially, the practical graphs we use in applications are instances of models, and the usual goal of a Hydra transformation is to carry any given graph whose schema is a particular model (e.g. the Thrift Schema model) to a graph whose schema graph is another model (e.g. the Protobuf Schema model) in a reversible way, transforming unsupported expressions into supported expressions plus annotations (**stashing**).

Here are some likely models to be included in Hydra at some point, in addition to Hydra Core:

- Arrow,
- Atom,
- Avro Data, Avro Schema, Avsc,
- FlatBuffers,
- Go,
- GraphML,
- GraphQL,
- Haskell,
- Ion,
- Parquet,
- PDL (Pegasus Data Language),
- Protobuf Data, Protobuf Schema,
- Python,
- RDF,
- RDF-Star,
- RDFS (RDF Schema),
- Scala,
- SHACL,

- Java,
- JavaScript
- JSON,
- OWL,
- Thrift Data, Thrift IDL,
- XML Schema,
- YAML

Where programming languages like Go or Java are mentioned in the above, it is to be understood that only a subset of the language will be represented in the model -- namely, the subset we are interested in expressing data and programs in. For example, the Java model needs to allow us to define classes with the sort of functions (methods) and types we use in Hydra graphs, etc. but we may not care about main methods, I/O, for loops, or much of the rest of the language. In order to create a fully functioning Hydra implementation in a language like Java or Go, we need to be able to map elements from Hydra's built-in models into the language in a semantics-preserving way, which includes function types and function implementations. Note that not all terms in HydraCore(D) need be mappable; for example, arbitrary-precision integers are provided by the type system, but will not be used anywhere in Hydra's core programming logic, precisely because they are not very portable.

Also note that existence of a model for a given language does not mean that Hydra supports transformations into or out of that language; the model just defines the data structures, while the logic for transformations has to be provided separately. It is typically simpler and easier to create a model than it is to create a transformation into or out of one.

Names

A name is just a unique identifier. In Dragon, names are lists of strings which correspond to a hierarchy of modules, e.g. [“org/example/mytypes”, “MyOuterType”, “MyInnerType”] is a name identifying a type called “MyInnerType” which is nested under a type called “MyOuterType” in a module called “org/example/mytypes”. This nesting helps to organize values and types into reasonable chunks, and enables a close correspondence with other schema and programming languages which break things down into modules. In Hydra, names might be lists of strings as in Dragon, or simple strings (which are often easier to work with, e.g. when data is mapped into languages which can't use a list as a map key), or they may be parameterized (allowing the developer to choose), TBD.

Modularity

One practical use of hierarchical names is for organizing elements into modules. For example, Dragon usually takes the top-level portion of a name to be a file name, and creates the corresponding files when writing files to disk, whether the target format is Thrift, Protobuf, Avro, OWL, etc. In Protobuf, which supports inner types, type names double as module names. In all cases, the modules serve an organizational purpose only; there is no data hiding in the sense of Java classes or Haskell modules. However, there are some structural constraints (see above) which are based on modules, and there are other possible constraints we could consider which have more of a data-hiding flavor. For example, one could prohibit references to elements in a module based on certain metadata on the element (e.g. no references to a “protected” element from outside its module, or no references to an element in a “development” module from a “production” module; something similar is already done in Dragon). It is TBD whether the use cases for such constraints are strong enough to warrant further consideration, but if other programming languages are any guide, they might become so as Hydra programs become more sophisticated.

Elements

An element is a named value annotated with a type. You can think of the set G(E) as a dictionary or index which tells you exactly which elements are in the graph G, what their names (ids) and data types are, and what data they contain. For those familiar with Gremlin, G(E) is like g.V() combined with g.E() -- it gives you all of the vertices together with all of the edges. The schema term **σ**(e) of each element e ∈ G(E) allows you to classify the element as a vertex, an edge, or something else, as the case may be, and the data **δ**(e) gives you its actual information content. In a property graph context, elements are “vertices and edges”, while in a programming language context, they are declarations like “foo = 42 :: Int” which bind a name to a typed value.

Data of an element

To every element e ∈ G(E), there is a term **δ**(e) ∈ G(D), called the “data” or “data term” of the element. Intuitively, the elements G(E) are the “important things” in the graph G, and are usually finite in number. The other set G(D) is the set of all potential data terms which can be attached to elements, and is usually infinite. Unlike G(E), Hydra APIs will provide no way of accessing G(D) except to check a term for membership; it is just a logical construct. However, you cannot create an element out of a data term unless that term is in G(D) -- a data term cannot reach outside of its own graph, and different graphs may have different constraints on legal data terms.

> **Example**: some examples of ordinary values are the number 42, the record {“foo”: 42, “bar”: true}, and also abstractions like lambdas (\x. 42 + x). See below for more information on terms. Again, you can make an element out of any term in G(D), so an element add42 ∈ G(E) with **δ**(add42) = (\x. 42 + x) is perfectly legal as long as lambdas are in G(D). You probably wouldn’t refer to this element as either a vertex or an edge.

Element references

For every element e ∈ G(E), there is a unique term (->e) ∈ G(D). This is the term which represents a reference to the element, and it can be freely used in other terms. For example, suppose the record {“out”: ->Jack, “in”: ->Jill} is the data of a “Jack knows Jill” edge. The terms

(->Jack) and (->Jill) are the only ways to reference Jack or Jill in graph G, and there is no way at all to reference Jack or Jill from a graph other than G. Further, if we give our “Jack knows Jill” edge the name jackKnowsJill in G, then we can refer to it from other elements’ data terms, e.g. we can an edge property with the term {“out”: ->jackKnowsJill, “in”: “Probably her brother”}.

The type of an element reference is the same as the schema term (see below) of the referenced element: **τ**(->e) = **σ**(e), so the type of (->Jack) is Person, if Jack is a Person, and **τ**({“out”: ->Jack, “in”: ->Jill}) = **σ**(jackKnowsJill) = knows (for example). This type is inferred based on the context of the term in an element’s data.

Schema term of an element

The schema term **σ**(e) of an element e ∈ G(E) provides a data type to which the data **δ**(e) of the element is expected to conform exactly: **τ**(**δ**(e)) = **σ**(e). Whereas the types of terms are inferred (based on their context within an element or computation), the schema terms of elements are explicitly stated. Think of them as the type signature you use when declaring a variable in program code.

An element’s schema term is, by definition, a data term in the schema graph G_S: e ∈ G(E) ⇒ **σ**(e) ∈ G_S(D). In typical property graphs, the schema term is what is usually called a “label”. The Hydra equivalent of a label is a reference (->e’) to an element e’ ∈ G_S(E) in the schema graph G_S. For example, the schema term of the jackKnowsJill edge illustrated above might be (->knows), and the type of the edge property might be (->comments). Note that element schema terms are terms in G_S(D), not G(D), so they are allowed to reference elements in G_S(E). Also note that in general, schema terms do not need to be element references; sometimes they represent atomic types, record types, etc. For example, we can define an element fortytwo ∈ G(E) with the simple integer value 42: **δ**(fortytwo) = 42 and **σ**(fortytwo) = int32.

Terms

Terms are expressions in a controlled vocabulary (see below) which is used to represent all data in Hydra. In this document, “value” (an instance of a type) is used interchangeably with “term” (an expression representing a value), though in some contexts they may be distinguished (e.g. we might only call a term a value if it is fully reduced, i.e. contains no function applications, or we might only want to call it a value if it has no applications, no lambdas, and no UDF references -- if it’s “just data, no functions”).

Ignoring syntactic sugar like enums, optionals, lists, etc. the table below illustrates all of the term constructors (“variants”) currently being considered. The Hydra Core type Term is a union of these variants. A term in a Hydra graph is always accompanied by a type (a term in the schema graph) to which it conforms. If that type is Type from Hydra Core, then type inference (see below) can be used to find the type of any subterms, including element references.

Term constructors

The following are all constructors in the Term grammar of Hydra Core. First, some notation:

- ->v1 is a reference to an element with the name “v1”
- !add is a reference to a primitive function with the name “add”
- int32(100) is an instance of the atomic type int32, with the 32-bit value 100. int64(100) is an instance of the atomic type int64, with the 64-bit value 100. For brevity, we may just write 100 when the specific atomic type is not important. Similarly, we will usually write “foo” instead of string(“foo”).
- [1, 2, 3] is a list with list elements 1, then 2, then 3
- Two terms separated by a space, e.g. (!add 1) represent an application term with a function on the left, and an argument on the right. The term ((a b) c) may also be written (a b c).
- (\x.f x) is lambda notation, with x as a bound variable
- {“shape”: ->Circle, “color”: “blue”} is record syntax, which is also used for variant terms, e.g. {“left”: 99} could be read either as a record with one field, or as inl(99).
- .color indicates the projection of a “color” field. For example, (.color {“shape”: ->Circle, “color”: “blue”}) would evaluate to “blue”.

Terms	Types	Examples	Description
Application	(any)	!neg 42	The application of a function to an argument
Atomic	Atomic	int32(42), string(“foo”)	An instance of a atomic type
Cases	Function	If … then … else ...	Pattern match which eliminates a union
CompareTo	Function	compareTo(42)	Orders two terms in the presence of a common type
Data[1]	Function	data[2]	The **δ** function which maps an element to its data
Element	(any)	->v1, ->e1	A reference to an element by name
Function	Function	!concat	A primitive function whose implementation is opaque
Lambda	Function	\x.\y.!plus x y	A function abstraction (lambda)

[1] Notice that while **δ** is provided as a primitive in the language of terms, **σ** is not (for now), because the latter produces terms which are not necessarily in G(D).

[2] An instance of the data constructor contains no other information, so it may be more helpful to illustrate the constructor with another kind of term, such as the application term (data ->point1337). This would evaluate to **δ**(point1337), e.g. a record with “lat” and “lon” fields.

List	List	[1, 2, 10]	A list of terms, all with the same type
Map	Map	{“Sunday”:1, “Monday”:2}	A map of terms to terms
Projection	Function	.lat	Projects a field from a record.
Record	Record	{“lat”: 37.8, “lon”: 122.4}	An instance of a (string-indexed) product type
Set	Set	{“Sunday”, “Monday”}	A set of terms
Union	Union	{“right”: 42}	An instance of a (string-indexed) sum type
Variable	(any)	x	A bound or free variable

Equality and comparison

In Hydra Core, two terms of the same type are equal if they have exactly the same structure, i.e. they are literally “the same” term. Furthermore, terms of a particular type are totally ordered, such that we can define the term constructor compareTo which produces a lessThan, equalTo, or greaterThan comparison value for any two terms conforming to the type. Hydra’s term grammar contains a special compareTo constructor, whose signature in Hydra Core is (a -> a -> boolean), to perform comparison operations. Without a type, or in the presence of two different types, terms are not comparable, i.e. compareTo cannot be used.

Example: the following are **not** legal in the Hydra Core type system:

- 42 :: int32 == 42 :: int64
- “foo” :: string == foo :: Element<string>, where **δ**(foo) = “foo”

However, the following would be legal, and true, equality and inequality statements:

- (intToLong 42) :: int64 = 42 :: int64
- “foo” :: string = **δ**(foo) :: string
- “aba” :: string < “abb” :: string

In other words, there is no notion of equality or comparison which cross-cuts types, and no notion of a subtype, or automatic casting, in Hydra Core, though other graphs are free to define their own rules, and use the compareTo primitive in graph-specific ways. Moreover, an element is distinct from its data term, so you cannot simply substitute an element reference with its value when comparing or evaluating terms. Two elements with the same data term are distinct because they have different names / different identities in the graph. However, you are free to use functions which map unequal terms to equal ones, as in the second set of example above:

Example: if elements Bob and OtherBob both have the data “Bob”, then it is true that **δ**(Bob) = **δ**(OtherBob) even though Bob ≠ OtherBob (because they are two different Person instances in the graph, possibly representing two different real-world people).

Note that built-in comparability of terms in Hydra Core will make it straightforward to use terms -- even function terms like lambdas and primitive functions -- in hash tables and ordered data structures like lists, as in Haskell. However, some caution must be used when mapping such terms into less expressive languages. For example, you can’t have a list as a map key in Protobuf, so Hydra will be forced to make a pragmatic choice like substituting a serialized version of the list (a string) with the list itself.

Equality not invariant under reduction

Once again, terms are equal only if they are identical, so a function application which becomes identical to another term after beta reduction is not the same even though it is guaranteed to have the same type, e.g. (capitalize “foo”) ≠ “FOO”.

Collections

You will notice three natively supported collection types in the above: lists, sets, and maps. Whereas lists can be thought of as a special case of union terms, sets and maps are different, because in principle they require uniqueness for their elements or keys. Formally, sets and maps in Hydra are simply additional list constructors, but they are given distinct constructors for the sake of more natural integration with other data and programming languages, very many of which have native support for sets and maps which is distinct from their support for lists. Additionally, we will provide distinct primitive functions for working with lists, sets, and maps in Hydra.

Types

In Hydra, a “type” is anything to which we can draw a **τ** arrow from a data term (an element of G(D) in some graph G). This admits type systems which are foreign to Hydra. In this document, we will use Type (uppercase) to refer to a specific element of the Hydra Core model, i.e. a “Hydra Core type”. Similar to the Term grammar described above, Type is a union of the following constructors (again, ignoring syntactic sugar like enumerations, vertex and edge types, etc.).

Type constructors

Types	Examples	Description
Atomic	See below.	A primitive / atomic type.
Element	->int32	The type of an element reference[3]
Function	Person -> boolean	The type of a function, with a domain and codomain
List	list[int32] -- a list of integers	The type of a list of terms, all of the same type
Map	map[->PhoneNumber, ->Person]	The type of a map from keys to values
Nominal	Person	A reference to a type by name (i.e. as an element); a nominal type
Record	{“x”: float32, “y”: float32}	A labeled product type
Set	set[int32]	The type of a set of terms
Union	{“left”: boolean, “right”: int32}	A labeled sum type

Notice that there are fewer Type constructors than there are Term constructors.

Atomic types

The following atomic types are proposed for Hydra:

Atomic types	Description
binary	Byte arrays/lists
boolean	true/false
float	Floating point / real numbers
bigfloat	Unlimited precision floating-point number
float32	32-bit floating point number (float)
float64	64-bit floating point number (double)
Integer	Integer values

[3] As mentioned above, an element reference is not interchangeable with the data term of the element, e.g. ->foo is not the same as 42, even if δ(foo) = 42. We say that 42 :: int32, but ->foo :: element<int32>.

bigint	Signed, unlimited-precision integer
int8	Signed 8-bit integer
int16	Signed 16-bit integer (short)
int32	Signed 32-bit integer (int)
int64	Signed 64-bit integer (long)
uint8	Unsigned 8-bit integer (byte)
uint16	Unsigned 16-bit integer
uint32	Unsigned 32-bit integer
uint64	Unsigned 64-bit integer
string	Character string

Identical constructors will be defined for these variants under the AtomicValue and AtomicType types in Hydra Core.

Syntactic sugar

Dragon's built-in types are somewhat different from those described above. Apart from parameterizing atomic types ("primitive types"), Dragon includes a number of additional type constructors which can be considered as syntactic sugar:

- Optional types
- Enumeration types
- Vertex types
- Edge types
- Property types

It is still to be determined whether Hydra will include Type and/or Term constructors for these "convenience types", as well as others such as integer-indexed products (tuples) and sums. In addition, Dragon has limited support for parameterized types (see below), which are still being explored in a Hydra context.

Advanced types

All of the above can be captured, formally, using a variant of the Simply Typed Lambda Calculus. Other useful constructs, however, cannot, and require additional formalisms such as the Hindley-Milner type system.

Specifically, will we attempt to support:

- Parameterized types?
- Type classes?

The Hydra Core vocabulary

The Hydra Core model is a specific graph which is built into Hydra, defining the core data model. The following elements will be defined in Hydra Core (where indentation indicates the structure of a union):

- **Comparison**: the result of a comparison judgement: lessThan, equalTo, or greaterThan.
- **Field**: a Term paired with a FieldName. Used in record and union terms, as well as case analysis over unions.
- **FieldName**: a string or other object which indexes (labels) a field within a record, variant record, or case statement. Eventually, FieldName may be parameterized rather than defined as a type. I.e. instead of defining the Field type in Hydra Core as a product of FieldName×Term, we might define it as a×Term, leaving the choice of a (string, integer, list, etc.) up to the developer.
- **FieldType**: a Type paired with a FieldName, used in the definition of record and union types. May be combined with Field if/when parameters are supported, i.e. Field<Type> vs. Field<Term>.
- **Term**: common type for all data in Hydra (see Terms)
    - **Application**: the application of a function to an argument (see Function application)
    - **AtomicValue**: a union of legal atomic values. Corresponds 1:1 with AtomicType.
        - **BooleanValue**: an enumeration of the values “false” and “true”
        - **FloatValue**: a union of legal floating-point values, by type
        - **IntegerValue**: a union of legal integer values, by type
    - **CaseStatement**: a pattern which applies to a union term (see Pattern matching)
    - **Lambda**: a function abstraction (see Function abstractions)
    - **Variable**: a variable which may be bound by a lambda
- **Type**: data type for native Hydra data types (see Types)
    - **AtomicType**: a union of Hydra’s atomic types (see Atomic types)
        - **FloatType**: an enumeration of the three floating-point atomic types
        - **IntegerType**: an enumeration of the nine integer atomic types
    - **FunctionType**: a function type, having a domain and codomain
    - **MapType**: the type of a map, having a key type and a value type

Note: “*Variant” types including TypeVariant, TermVariant, AtomicVariant, FloatVariant, and IntegerVariant are omitted from the above for the sake of brevity. These are merely a convenience; for example, TypeVariant is just an enumeration of the Type constructors: {atomic, element, function, …}.

The following need to be defined in Hydra, but may or may not be in Hydra Core itself:

- **Context**: a lexical and typing context for data/type operations
- **Element**: a (name, data term, schema term) triplet (see Elements).
- **Graph**: an iterable set of elements G(E) together with a logical set G(D). One cannot iterate over G(D), but one can test a Term for membership. Some definition of **τ**, and some definition of graph constraints -- generating G(D) -- may be included in Graph, as well (see Graphs).
- **Name**: a string or a list of strings representing an element name. Like FieldName, this type might eventually be replaced with a type parameter, for greater flexibility.

The following elements are also being considered for inclusion in Hydra Core:

- **EnumerationType**: shorthand for a (finite) union of units
- **Optional**, **OptionalType**: shorthand for union with unit
- **Tuple**, **TupleType**: possibly a special case of Record and RecordType
- **VariantType**: or other name for an unlabeled union type. Possibly a special case of UnionType

Computation and inference

The current proposal for Hydra’s type system (see above) can be seen as an extension of the simply typed lambda calculus (STLC). It differs from the most basic formulations of STLC in that it includes element references and reference terms, record and union types, conditionals based on record and union types, and built-in functions (UDFs). The type system is likely to be extended further in order to support parametric polymorphism.

In any case, terms in Hydra can represent computations as well as “ordinary” data like numbers, strings, and records containing “ordinary” fields. The main purpose of Hydra is not to evaluate such expressions -- although since STLC is not complicated, it will definitely have that capability -- but to apply transformations to them, and transpose them meaningfully into and out of other languages which also support function abstractions. We want to lean on the Scala compiler, the Haskell compiler, etc. to do most of the hard work of optimizing program code for efficient execution.

Function abstractions (lambdas)

A lambda in STLC is an abstraction which consumes an argument of a given (sometimes explicitly provided) type and produces a result of another type. In other words, it is an instance of a function type. Some lambdas can be reduced to concrete values by providing concrete values as arguments. In Hydra, the domain and codomain types of a lambda are provided if the lambda is the data of an element, otherwise inferred.

> **Example**: consider a term/type pair like (\p.(.firstName p + .lastName p)) :: ->Person -> string. This is a function whose type is given. It could even be used to define a graph element in G(E), call it fullName:
>
> > {“name”: “fullName”, “data”: \p.(.firstName p + .lastName p), “schema”: ->Person -> string}
>
> You could treat this element like any other vertex in the graph -- use it in edges, apply properties, etc. -- but it also “does” something: you can pass Person vertices to it, and get back strings like “Arthur Dent” or “John Doe”.
>
> Suppose we have a more complicated version of the same function, like (\p.((\f.f p) (\p'.(.firstName p' + .lastName p')))), also typed with (->Person -> string). This is also a lambda, but it has a function application inside. In this case it is straightforward to infer that the type of the inner lambda is the same as the type of the outer lambda. See the section on type inference below for more details.

Function application

A function application is a term (f x), where f is a term with function type (A -> B) and x is a term with type A. The application itself has type B. An application is a “computation waiting to happen”. Unlike a lambda, which needs an argument in order to be applied, an application already has its argument, and can be reduced to a (usually) simpler term. However, Hydra does not require that applications be reduced; it would be legal to define an element like:

> arthurDentName : string = ->fullName ->arthurDent

...which represents the result of applying the “fullName” function to the Person arthurDent. More practically, applications generally contain unbound variables and are used as subterms in function abstractions.

Pattern matching

Two kinds of conditionals are currently being considered for Hydra:

1. Elimination of union terms (see term constructor above)
    - E.g. (\x -> case x of: left a -> …, right b -> …) produces a different result depending on whether x is a “left” or a “right”.
2. Generic conditionals based on equality judgements (see above)

    ○ E.g. if x = 42 then … else if x = 35 then … else …

Equality in Hydra is determined using the compareTo constructor, which produces a union term, so (2) can be considered as a special case of (1).

These conditionals are sufficient for pattern matching as seen in many functional programming languages such as Haskell or Scala, though unless a surface syntax is provided (see Hydrascript), it will be up to developers to build more complex pattern matching constructs (e.g. not only switching on the alternatives of a union type, but also providing a convenient way of matching an empty list, binding variables to the head and tail of a list, etc.) on top of the basic term grammar.

Primitive functions

Primitive functions are black-box functions whose implementation is built into Hydra (or a particular configuration of Hydra), rather than specified as a function abstraction. At the graph level, all we know is the function's name and type. They have the same status in a graph as atomic values like "foo" or 42, but they are identified by names, like elements are. For example, the name of a string concatenation primitive might be "hydra/primitives/strings.concat", and there will be an element with that name in a standard Hydra model which can be loaded so as to access metadata for the function.

Primitive functions behave like function abstractions, with some important differences. Like function abstractions, their types are function types: applying either a primitive function or a lambda of type A->B to an argument of type A will give you a term of type B. Unlike a lambda, a primitive function has a minimal number of arguments (its **arity**) to which it needs to be applied before it will reduce to a term other than a lambda; until then, arguments are curried. Primitive functions are not allowed to curry arguments internally.

> **Example**: a primitive !concat function, which has the type (string -> string -> string), requires two arguments before it can be applied. The expression (!concat "foo") expands to (\x.!concat "foo" x), and evaluation (see below) stops there. The primitive function is now allowed to consume a single argument and give back "something else" of type (string -> string), unless it is another primitive function.

It is still to be determined whether primitive functions will be allowed to consume and produce arbitrary terms (including lambdas, applications, conditionals, and/or other function references), or only passive values. It is likely that built-in Hydra primitive functions will be completely generic, while user-defined functions (see below) will be more constrained.

When a given primitive function is included in an evaluation environment, it is bound to an implementation which can be called on actual arguments to produce an actual result. The implementation will always be dependent on the host programming language, so Hydra-Scala, Hydra-Haskell, Hydra-Python etc. will all need separate implementations of the standard primitives included with Hydra. For that reason, it is important to keep these mandatory primitives to a minimum, so as to reduce the burden of creating new Hydra language variants.

Likely standard libraries of primitive functions include:

- Integer and floating-point arithmetic
- String functions
- List, map, and set functions
- Hydra primitive operations (compiled for performance), e.g.
    - Type inference and type checking
    - Evaluation/reduction
    - Qual (exceptions)
    - Serialization

These libraries may be organized as models, just as Hydra Core and other built-in models are.

User-defined functions

Whereas the libraries above will be built into Hydra, application developers will have the option of defining their own primitive functions -- called user-defined functions (UDFs) -- and include them in Hydra's runtime environment. These will behave exactly like the built-in functions except that additional constraints may be applied for the sake of type safety and other runtime guarantees. Unlike built-in primitives, UDFs will not need to be implemented in many languages in parallel; they can simply be implemented in the language in which they are needed. If the "same" UDF is needed in different language environments, developers can use the same name; it will then be up to the developer to guarantee that the implementations are also equivalent.

Type inference

Typing judgments are to be well-defined in Hydra; there is no such thing as an untyped term, and no operation will proceed if a term/type mismatch is discovered. Such a mismatch could occur, for example, if an element is defined with a data term which does not conform to its schema term, a condition which automated tools should help prevent. Given a term and a typing context (see below) including a single type t to which the term is expected to conform, the type of every subterm can be inferred, according to the variant of the term and the variant of the type. Say we have term v and expected type t. A type checking function check(t, v) works as follows:

- If v is an element reference (->e): confirm that t = element<$\boldsymbol{\sigma}$(e)>. Do not descend into the element.
- If v is a variable, confirm that v::t is in the typing context
- If t is a atomic type and v a base term, ensure they match (e.g. int32 vs. 42 :: int32)
- If t is a record type with of $t_1..t_n$ and v a record of terms $v_1..v_n$: $\bigwedge$check(t_i, v_i)
- If t is a variant record type of fields $t_1..t_n$, expect a variant term v_i: check(t_i, v_i)

- If t is a function type (t_1->t_2) with t_1 a record type of fields t'_1..t'_n, and v is a projection on field i: confirm that t2 = t'_i
- If t is a function type (t_1->t_2) with t_1 a variant record type of fields t'_1..t'_n, and v is a case statement with cases v'_1..v'_n: $\wedge$check(t_2, v'_i)
- If t is a function type and v a primitive function reference, confirm that t = (declared type of the function)
- If t is a function type (t_1->t_2) and v a lambda (\x.v'): TBD; need to update the typing context before descending into the function body
- If v is a function application: TBD; need to update the typing context before descending into the left and right side of the application

If the above falls through, the term does not match the type. Note that the above assumes that the checked term is closed, i.e. has no free variables.

Evaluation

As mentioned above, Hydra is not envisioned to be a full-fledged programming language with an optimizing compiler. However, it will have the ability to evaluate terms in the manner of a strongly typed interpreter. This evaluation will probably be fairly tunable (e.g. allowing the user to select lazy or strict evaluation, and different conditions for reduction), as we want to be able to serve applications with different needs and expectations. In all cases, evaluation will take the form of beta reduction in the basic lambda calculus sense, with a slight difference: the results of function evaluation will always be wrapped in the Qual monad, allowing Hydra to capture error and warning information in the process of performing the evaluation. Any term of type T will reduce not to T, but to Qual<T>. A Qual<T> object contains a T if the evaluation was successful, and nothing but error information otherwise. Likewise, if we are applying a function of type A->B, the result will be a Qual<B>, likely starting from a Qual<A>.

The reduction rules for Hydra's version of STLC may not be listed in full here, but they are as you would expect. Where a choice is possible (e.g. whether to descend into the body of a function abstraction while reducing), we will tend to make the choice configurable.

Evaluation context

In order to evaluate a term within a graph, we need at least the following pieces of context:

1. The lexical context which allows us to look up elements in G(E) by name
2. The lexical context which allows us to look up primitive functions by name
3. The typing context (typing environment) of the term. This depends on (1).
4. Global options, such as the option which chooses between lazy and strict evaluation

A Hydra evaluation context will contain all of this information, and will be passed along with every term to be reduced or transformed.

Concrete syntax

Hydra will be intentionally minimal with respect to surface syntax, at least initially. All of the important constructs of Hydra are to be captured in the Hydra Core model, and related models. To that, we will add a general-purpose coder for YAML, and another for JSON, capable of serializing and deserializing terms to/from either format. Since all data in Hydra -- regardless of model -- is expressed as terms, this encompasses everything. I.e. there will be no special “schema YAML” format as in Dragon, with a separate format for “values” conforming to a schema.

However, since the generic YAML format will probably be fairly verbose, at some point we may introduce additional formats as a convenience. One possibility for such a format is a “compact” YAML syntax, and another is a programming language syntax which we will cann “Hydrascript” for the purpose of this document.

Generic YAML and JSON coders

As mentioned Hydra, will support a YAML-based syntax, as a superset of a generic JSON-based syntax, similar to Dragon's YAML schema formats. Unlike Dragon (in which there is a special reader and writer for Dragon YAML at the schema level and the data level), Hydra's YAML syntax will correspond one-to-one with a model specification (the Hydra YAML model). The generic YAML coder can then be used for arbitrary graphs G or G_S -- there is no distinction between “data” and “schemas” in terms of the YAML format.

Hydrascript

“Hydrascript” may or may not be necessary; it is to be determined based on the Hydra developer experience whether a separate programming language syntax is worth the effort. See the discussion of bootstrapping languages above.

If undertaken, Hydrascript would be a compact syntax for Hydra models which is much less verbose than YAML for complex transformations, such as function implementations, but also more tailored to Hydra than general-purpose programming languages like Scala or Haskell. Developers would use Hydrascript much as they would use a typical functional programming language. Hydrascript code would typically be written by hand rather than generated, though a full (bidirectional) Hydrascript coder would be provided. Note that as a programming language, Hydra can be thought of as statically typed (regardless of the surface syntax: Hydra YAML, Hydrascript, or another format).

Apart from development cost, other factors to consider are the initially lacking IDE and third-party API support for Hydrascript, vis-a-vis Scala or other options. Not to be undertaken lightly, unless working with Scala turns out to be harder.

Embedding Hydrascript in YAML, and vice versa

In some cases, it would be useful to include short snippets of Hydrascript, as strings, in YAML-based model files. The Hydra YAML model will include special "script" constructors to enable the Hydrascript snippets to be treated as if they were equivalent YAML expressions. For example:

```
value:
 name: "AddOne"
 function:
   script: "\x -> x + 1"
```

Likewise, it would sometimes be useful to include Hydra YAML snippets in Hydrascript files. Here, too, there will be special constructors to allow the YAML to be treated like part of the script. The same could be said of other potential implementation languages like Scala.

Comparison with Dragon and APG

In terms of functionality, Hydra can be considered as a successor to Dragon, but its design is fundamentally pretty different. Some specific differences:

- **Unification of types and values**: in Algebraic Property Graphs and in Dragon, values and types are different things, despite having a similar structure. For example, a record can be thought of as a tuple of values, whereas a record type (product type) is a tuple of types. In Hydra, there are only terms. The type of a term is again a term, albeit in a different graph. This will make Hydra's code base somewhat simpler than Dragon's because Dragon has to grapple with values and types separately -- separate data structures, separate transformation steps, etc.
- **Unification of labels and ids**: in APG, a "label" is an identifier bound to a type. Every element has a label, as well as its own id. In Hydra, types are just terms, and elements are just (id, term, term) triplets, so there is no such thing as a label. In a classic property graph, the type of each element will always be a reference which can be considered as a label (e.g. the label of a given element might be "Person", which in Hydra would be a reference to a "Person" element in the type graph), but in other kinds of graphs, it need not be. This again will simplify Hydra's code, while shifting the burden of working with property graphs out of the core of the language and into Hydra domain models.
- **Unification of schema-level and data-level mappings**: since there is no difference between terms and types in Hydra (it's just a matter of level), there will not be a hard distinction between schema-level mappings (“transformers”) and data-level mappings as there is in Dragon. Both are referred to as “coders” here. Separate mappings will still be required, e.g. we will have separate steps for Protobuf schemas and Protobuf data because they are expressed in different languages (which map to different Hydra models), but the same API/interface will be used for both.

- **First-class support for function types and function values**: Dragon “kind of” has function types. They are not well integrated with values, and they are used only for code generation. In Hydra, function types are important because transformations (functions) will commonly operate over other transformations.
- **No parameterized primitive types**: Hydra does not currently parameterize primitive types like Dragon does. For example, there is no "precision" parameter for integer or floating-point types, no "signedness" parameter for integer types, and no "maximumLength" parameter for string types. In Hydra, the complete type of an atomic value can be inferred from the value itself, whereas Dragon's primitive type cannot. In Hydra we relieve some of the need for parameters by providing additional value constructors like int16, int32, uint64, float32, float64, etc. as well as bigint and bigfloat types. Dependent types like integers with bounds, strings with bounded length regex, etc. are possible using metadata, but are not part of the Hydra Core type system.
- **Clearly encapsulated user-defined functions (UDFs)**: while Dragon’s Haskell implementation and Java implementation have some functions in common, there is nothing in the code which links the corresponding functions. In Hydra, there is to be a clear notion of primitive functions (including user-defined-functions), which is defined in terms of a function type and which is paired with a context-specific, black-box implementation. When a function is common between two Hydra language variants, we will know it because they will have the same type definition in a common model; the UDF is then implemented as appropriate in the programming environment of each language variant.
- **Native transformations**: transformations in Dragon are written in opaque, compiled code (Java or Haskell). As such, they are not accessible for analysis, optimization, or translation into other languages. Hydra would consider them all to be primitive functions. In Hydra, most transformations will be implemented not as primitives, but as native Hydra expressions (terms) which can be mapped into different implementation languages in parallel, and which are open to analysis and optimization. This will not only lower the barrier to creating Hydra implementations in additional programming languages, but also open up Hydra's core logic for optimization and correctness testing.
- **Clear distinction between native transformations and first-mile/last-mile steps**: in Dragon, there are general-purpose steps which operate directly on Dragon's internal representations (graphs, types, values), and there are steps which map internal representations to or from string representations or third-party APIs. However, there is no clear distinction between the two. In Hydra, first- and last-mile steps are very different, in that they will generally be opaque, as primitive functions (because they are implemented entirely in a particular language variant, using native code constructs), rather than transparent, as native transformations.
- **Deep support for polymorphic types**: like function types, Dragon “kind of” has support for polymorphic types -- just enough to generate expressions like “Lossy<A>” or “Step a b” into Java or Haskell. In Hydra, we will need to perform actual type inference over polymorphic types, so they need to be built in at a more basic level.
- **No built-in REPL**: unlike data structures and transformation logic, an interactive program like a REPL would be difficult and time-consuming to carry over from a core

language into multiple implementation languages, due to the very different ways in which different languages approach I/O. Dragon's REPL is implemented in Haskell, with no attempt at a REPL in Java. For Hydra, it will be up to the developers of individual Hydra language variants to create their own language-specific REPL, although some common data structures and functions can be provided for parity.
- **Qualified instead of Lossy**: Dragon's Lossy monad will be replaced in Hydra by a similar monad called "Qualified" or "Qual". The purpose of Qualified is similar (it captures transformation results along with alerts/disclaimers, usually relating to information loss), but whereas Lossy is tied to a particular data type (called Alert) for information attached to transformation results, Qualified will allow arbitrary, domain-specific typed values. The contents of Qualified can be thought of as an extension of the graph, whereas Alert had very limited expressivity (a string plus a logging level).

Terminology

The following are terms you will see used in connection with Hydra, particularly in connection with Hydra's type system. Some, like "term" and "type" will have formal definitions as in APG, whereas others like "data" or "metadata" are only descriptive.

- **Algebraic Property Graphs (APG)**: Dragon's basic data model. Hydra's data model has yet to be completely defined, but it can be thought of as a generalization of APG as originally specified.
- **Atomic value**: a member of one of several sets of values which are built into Hydra. Dragon's atomic ("primitive") values consist of booleans, byte arrays, integers, floating-point values, and strings, whose types are parameterized along a few dimensions such as bit precision, signedness, etc. The type of an atomic value in Hydra is an atomic type, which is any of a few distinguished elements in the Hydra Core model.
- **Attribute**: a metadata property, i.e. a property in a schema graph. For example, a "description" property on a "worksFor" edge definition would be considered as an attribute, particularly if "description" is one of the built-in properties supported by Hydra. In Dragon, there are built-in fields like "description", "status", "seeAlso", etc. which are called attributes and may be attached to any symbol, but they are not formalized as properties.
- **Coder** (or "codec"): short for "encoder/decoder". If a coder is bidirectional (has both an encoder and a decoder), we require round-trips from either end to be information-preserving.
- **Data of a graph**: the set G(D) of all legal terms
- **Data term**: the component of an element which provides the "data" of an element. Compare with the schema term of the element, which provides the data *type*. Collectively, the "data terms" of a graph G are the set G(D) of all *possible* data terms **δ**(e) of elements e ∈ G(E).
- **Decoder**: a component which maps the data terms of a graph to data terms in Hydra Core

- **Dereferencing**: the action of **δ**(e), or the data() term constructor. It replaces an element reference with the element's data term. To **recursively** dereference an element or its data term is to replace all references in the term, starting with the shallowest reference. Two unequal terms (like references to the elements "forty-two" and "zweiundvierzig", both of which are aliases for the number 42) may become equal when dereferenced.
- **Encoder**: a component which maps Hydra Core data terms to the data terms of another graph. Opposite of a decoder.
- **Element**: a named, typed value. In typical property graph applications, element names are called "ids". In the original formulation of APG, the type of an element is given by its "label". In Hydra, the "schema term" of an element may be given by any data term in the schema graph G_S. If the type is an element reference, then it is a "label" in the original APG sense. However, in Hydra you can also have an element with no label (type name), whose type is simply "integer", "integer x string", or any other type expression.
- **Element reference**: a data term (->e) which references an element e in the graph. The name of the element is given in the reference, and the type is known/inferred.
- **Graph**: a set G(D) of data terms and a set G(E) of elements (named, typed terms) together with a second graph G_S (the "schema graph"), having the following properties:
    a. The name of each element e in G(E) is unique in G
    b. The data of each element e in G(E) is a term in G(D)
    c. The schema term of each element e in G(E) is a term in G_S(D), i.e. the schema term of an element in the base graph is a data term in the schema graph.
    d. The data term of each element conforms to its schema term
- **Metadata**: any data in a graph's schema graph, particularly that data which modifies but is not part of the type elements of the graph. For example, if "JohnDoe" is an element in a graph, and the type of "JohnDoe" is "Person" (which is an element in the schema graph G_S as opposed to the base graph G), then any annotations on "Person" (like a description "a person is a human being" or a "seeAlso: foaf:Person" expression") are part of the metadata. The actual "Person" element is part of the schema graph, but may or may not be considered part of the metadata.
- **Model**: a Hydra graph representing a data model in some domain of interest. Models will mainly consist of collections of Type elements, which are the Hydra Core notion of a data type. For example, Hydra will include a built-in SHACL model which provides the schema for RDF graphs, an Avro model which provides the schema for Avro datasets, etc. as well as the Hydra Core model which provides the schema of all of the built-in models, including Hydra Core itself.
- **Module**: all elements of a graph whose names belong to a particular namespace. Modules are simply a convenience for organizing elements in complex or heterogeneous graphs. They will sometimes correspond to directories, packages, etc. depending on target languages and APIs.
- **Name**: a unique identifier for an element in a graph. Although names are likely to be opaque in the abstract specification of Hydra's data model, in the implementation each name will be a list of string-valued parts, similar to a domain name, file name, or other hierarchical identifier. Dragon also defines names in this way.

- **Namespace**: for hierarchical names, a set of names with a common prefix. For example, "examples.types.GeoPoint" and "examples.types.time.Duration" are both members of the "examples" and "examples.types" namespaces.
- **Path**: a sequence which uniquely identifies a position in a term. E.g. a path can be used to point to the number 42 in the expression ("Arthur", inl(42)). In Dragon, paths have human-friendly representations which help locate errors and warnings related to validation and transformations.
- **Property**: an element whose data is an ordered pair (->e, v'), where ->e is an element reference, and v' is some term which contains no references. A property "connects an element to a simple value".
- **Record**: a tuple, or instance of a product type. A record can also be seen as a map of keys ("field names") to terms. In Hydra, the type of a record is again a record. E.g. the type of record("foo", "bar", 42) is record(string, string, integer), where the first expression is a data term in the graph G, and the second expression is a data term in the schema G_S.
- **Schema**: a collection of data type definitions, in a loose sense. For example, an Avro avsc file or a Thrift IDL file can be referred to as “schemas”. “Schema term” and “schema graph” have more precise meanings in Hydra.
- **Schema element**: any element in the schema graph $G_S(E)$ of a graph G. In typical property graph applications, vertices and edges have “labels”, which in Hydra are references to schema elements.
- **Schema graph**: a graph G_S which provides the data model of another graph G. For every element $e \in G(E)$, $\boldsymbol{\sigma}(e) \in G_S(D)$, which is to say that the element’s data type annotation (schema term) is an expression in a language of terms and elements which is defined as a separate graph. Note: in the Property Graph Schema Working Group, the term "type graph" has been used for a similar concept.
- **Schema term**: the component of an element which provides its data type. $\boldsymbol{\sigma}(e)$ is the schema term of element e. Collectively, the “schema terms” of graph G are the set $G_S(D)$ of all *possible* schema terms $\boldsymbol{\sigma}(e)$ of elements $e \in G(E)$.
- **Step**: a bidirectional transformation, i.e. a Step<t1, t2> is a pair of transformations, one of which maps from t1 to t2, and the other of which maps from t2 to t1. A Step<t1, t2> and a Step<t2, t3> can be composed together to form a Step<t1, t3>. Many of the building blocks of practical transformations in Hydra will be encapsulated as steps.
- **Term**: an expression produced by the Hydra Core term grammar (Term). All data in Hydra is expressed as terms.
- **Transformation**: a logical instance of a function type, allowing terms of one given type to be mapped to terms of another given type. Practically, a transformation can either be a black-box primitive function, or it can be a Hydra term which explicitly specifies a function using constants, case expressions, lambdas, etc. At the implementation level, a transformation is not simply a function from t1 to t2, but is a function from t1 to Lossy<t2>, where Lossy is a monad which captures errors, warnings, and other information related to imperfections in the mapping.
- **Type**: a term in a schema graph G_S to which a term in a base graph G conforms. The type of a term is itself a term, but in a different graph; if t is in G(D), then $\boldsymbol{\tau}(t)$ is in $G_S(D)$.

We say that t *is an instance of* **τ**(t) in some context. Often when we talk about "types" in this document, we are referring more specifically to instances of Type, which is the Hydra Core notion of a data type.

- **Type conformance**: a relationship between a term and a type (which is also a term) which is captured by **τ**, and which is preserved by all operations in Hydra. Type conformance in Hydra Core follows the simply typed lambda calculus, but in general it is graph-specific.
- **Value**: a full-reduced term. Used loosely.
- **Variant record**: an instance of a labeled coproduct type. Alternatively, a variant record can be seen as a (key, value) pair, where the key is an index into a map of keys to types.

Applications

This section is for community suggestions about how a tool like Dragon would be used, if open source. Hydra will aim to serve many of these use cases.

Survey results

The following are developer responses to a survey (Release Dragon!) which was conducted as part of the How to Build a Dragon series of presentations.

- We define our data warehouse table schemas as protobufs and use or manually write mappings between: parquet/jsonschema/arrow/spark/aws glue/etc... This tends to be duplicated among our scala and python code. We would be interested in using dragon to a) begin defining more "well known types" and b) be DRYer.
- I would love to see support for other prog. languages like Idris. This tool seem related: https://typedefs.com/targets/. Possibly one could use this tool for data migrations.
- Personal knowledge bases, esp. traffic (and ideally sharing, e.g. bidirectional traffic) between Roam, org-roam, Semantic Synchrony and Hode.
- ETL, automated schema evolution, enabling new data use cases, creating consistency among different organizations' data marts
- Schema transformation for different consumption patterns, enforcing schema evolution (backcompat), code generation.
- Data integration & virtualization to support domain specific data analytics
- Verifiable Digital Credentials for Privacy Preserving and Faster Car Rental
- any customer data integration requirements across a variety of verticals.
- Data catalog, data mapping, graph data analytics, FAIR data sharing
- My use case is machine learning in both healthcare and education.

- Projection of core data into multiple read models
- Architecture prototyping for software development
- Software knowledge base construction.
- RDF / Property Graph integration
- Collaborative dataset curation
- We use Neo4j for modelling of industry standard data model/ontology for Insurance called Acord. Acord is provided in XMI format and contains a lot of information. Only a subset of it relevant for us and we load into Neo4j and create a logical model of it and then even physical. This global model is then exposed as a GraphQL endpoint and populated with data from Kafka using Kafka Streams. Dragon would be interesting to us in case of mappings between different source schemas and our target GraphQL schema but also creating data pipelines/mappings from source to target. I played myself with idea of defining these mappings in yaml files and define a bunch of functions(Java) able to execute on it. But I think Dragon has got much further on that path.

Others

- ...

(scratch pad)

APG

lambda(e) = Lat
sigma(Lat) = double
upsilon(e) = 50
tau(50) = double

Hydra

sigma(e) = ->Lat
delta'(Lat) = double
delta(e) = (50:Lat)
tau((50:Lat))=->Lat

tau(delta(e))=sigma(e)

If terms don't carry types:

delta(e) = 50
tau(50) = Lat, Lon, double (tau is not a function)

```
function APGToHydra (APG apg) {
  let G(E) = apg.G(E);
  let G(D) = apg.G(V);
  let delta (e) = (apg.G(upsilon)(e):apg.G(lambda)(e));
  let G_S(E) = apg.G(L);
  let G_S(D) = apg.G(T).replace(“Label “, “->”); // recall the type constructor “Label l” for l in G(L)
  let sigma (e) = (->(apg.lambda)(e));
  let delta’ = apg.G(sigma);
  let tau (e:v) = v;
}

function APGToHydra (APG apg) { // triple way of thinking
  let G(E), G_S(E);
  for each e in apg.G(E), add (e, apg.G(upsilon)(e), apg.G(lambda)(e)) to G(E);
  for each l in apg.G(L), add (l, apg.G(sigma)(l), t(apg.G(sigma)(l))) to G_S(E) // t is
type-inference
}

function t (x) {
  return match x with
  | a*b => hydra.TypeRecord t(a) t(b)
  | a+b => hydra.TypeUnion t(a) t(b)
  | Prim p => Hydracore_atomictype p
  | etc.
}
```

event,position,time:$G_S(E)$
delta’(event)=position*time, delta’(position)=double*double, delta’(time)=double

e,t,p:G(E)
sigma(e)=event, sigma(t)=time, sigma(p)=position

delta(e) = (->p, ->t)
delta(p) = (50,100)
delta(t) = 400

Q: Does the json for delta(e) show { position: ->p, time: ->t} OR { position: { 50, 100}, time: 400}
A: The former.