A Programing Language for Data and Configuration!

MARK MARRON, University of Kentucky, USA

A day in the life of a developer often involves more time working with schemas, configurations, and data description systems than writing code and logic in a classical programming language. As more systems move into distributed worlds, *e.g.* cloud and microservices, and developers make increasing use of libraries and frameworks, the need to interact with a range of data formats and configuration mechanisms is only increasing. This is a treacherous world, where a misspelled property name or missing field can render an entire service inoperable, a mistake that a number in an API represents seconds instead of milli-seconds can lead to a message being set for delivery in several months instead of in an hour, a misconfigured schema can lead to public exposure of sensitive data, and corrupt or erroneous results from a misunderstood data format could result in massive financial and/or reputational damage.

To address these challenges this paper casts the problems of data and configuration descriptions, not as a problem of data representation, but as a programming language problem, that can be addressed with well understood and highly effective programming language and type system techniques! The novel challenge is that data representation and configuration are universal concerns in a system and, particularly in modern cloud or micro-service systems, these system may involve many programming languages. In the past this has led to specification systems that use a least-common-denominator set of data types, often little more than strings and numbers, and then rely on conventions or (out-of-date) documentation to ensure that the data is interpreted correctly. This paper shows that, with careful design, it is possible to create a rich universal system that can be used to express data and configuration specifications in a way that is human readable/writable and that can be produced/consumed, much like JSON, by a wide range of programming languages and systems. This paper presents BsqON, a programming language for data and configuration, that provides a unified and powerful system with all of the high-value features, including type-safety, versioning, packaging, and IDE support, expected from a modern programming language.

ACM Reference Format:
Mark Marron. 2024. A Programing Language for Data and Configuration!. 1, 1 (April 2024), 24 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Modern software systems are complex constructs, often involving many different systems, components, and programming languages. These are built and connected using a host of build & deployment configuration files, are managed via various runtime configuration systems, and interoperate using a variety of data communication formats. Despite the critical role that these configuration and data formats play, the languages and tools that they are written in and managed with are often afterthoughts in the development process. As a result a trivial mistake like a misspelled property name, spurious whitespace, or a missing field can render an entire service inoperable, silently corrupt data, produce spurious results, or result in the violation of regulatory requirements!

In practice data formats like JSON [15], YAML [39], CSV, or TOML [36] are commonly used for configuration and data exchange. These formats are simple, human readable, and ubiquitous, but they are also limited in their expressiveness and are often used in ways that are not safe or secure. For example, JSON does not provide a means to express that a value is a date, a BigNum, or a UUID and, as a result, these values are often encoded as strings and then re-parsed into the expected types.

Author's address: Mark Marron, marron@cs.uky.edu, University of Kentucky, Lexington, Kentucky, USA.

YAML is notorious for ambiguous syntax and gotchas – the literals UK and US are interpreted as strings but NO is the boolean value false! Features like the implicit treatment of missing properties can lead to month long bug hunts when a upstream service renames a property! In fact maintaining parsing data in these formats is one of the top sources of bugs in RESTful services [1].

These formats are also often used in ways that are not safe or secure. For example there is no standard/enforcable way to track and mark data that may be sensitive (PII) or should not be used in certain ways (GDPR). There is no way to express that a value is a secret, or that it should be encrypted in transit or at rest. These issues when combined for the risks of mis-interpreting data and leading to inappropriate outcomes creates serious reputational, financial, and legal risks for organizations. In 2020 one of the largest banks in North America was fined $400 million, in large part, for failing to properly manage their data systems and poor data quality practices [24].

To address these challenges this paper casts the problems of data representation and configuration description, not as a problem of a creating a data format or a limited DSL for schema definition, but instead as a problem of creating a first-class programming language specifically for building logical/semantic models for data and configuration. This language, BsqON, is designed to provide a unified and powerful system with all of the high-value features, including type-safety, versioning, annotations, packaging, and IDE support, expected from a modern programming language. Further, as required to function in a polyglot system composed of multiple programs written in many languages, this type system is designed to be easily interacted with from a wide range of programming models and languages.

The approach taken in this paper is centered around three technical contributions. The first is a type system that is simultaneously rich enough to cover the range of foundational types that real-world systems use, expressive enough to allows developers to concisely encode their particular domain, and that fits into the range of languages that modern polyglot systems are composed of. To allow for validations that are beyond the expressive power of a reasonable type system, such as uniqueness of values in a list or order relations on properties in a value, our approach provides an expression language that can be used to express rich properties while being safe, sandboxed, and suitable for formal analysis. Finally, this paper describes a canonical literal data representation for values that is designed to be easily readable/writable by humans, that supports full IDE integration, and that admits efficient encoding/parsing implementations.

Thus, this paper makes the following contributions:

- Framing the problem of data specification and configurations as a programming language problem and treating it with the same rigour as classical computationally focused languages.
- Defining a formal type system specialized for the needs of a data or configuration specification language.
- Identifying an assertion language for data invariants that is specialized for the safety concerns and expressive needs of this domains.
- Defining a canonical literal data representation format that connects to type system and enables easy, IDE supported, authoring and consumption of data values.
- Provide case studies of how this language can be used to express a wide range of data and configuration formats, how this concept is being actively used to address the challenges described in this paper at top technology and banking companies, and demonstrate the viability of further technical advances in system design and testing based on this approach.

2 DOMAIN CHALLENGES

There are numerous requirements and (conflicting) constraints that must be addressed when designing a programming language for data and configuration. The problem involves both a mix of semantic type system design as well as practical concerns around efficient data formatting and human (AI) centric experiences for literal value authoring. This is layered on top of the need to do things in a manner that is agnostic toward the programming language that may be used to implement these APIs and the desire to support administration and compliance tasks associated with data and configuration management as well!

2.1 Type and Invariant Language

2.1.1 Polyglot World. Modern applications operate in a connected world and communicate with systems written in a wide range of programming languages. This means that the data formats must be easily consumed and produced using range of programming languages and, thus, the description language cannot assume an opinionated object model or programming paradigm. This often leads to the adoption of least-common-denominator data formats, such as JSON [15, 27] or formats that closely tie the specification to a representation, Protocol Buffers [28] or Avro [2]. A compounding aspect to this polyglot challenge is that the data formats, and their representations, are also used directly by human (and soon AI) developers as well. Thus, we want a type system that provides an expressive means for describing types and data in a system while, simultaneously, being easily consumed by the wide range of programming languages and systems in our polyglot world. Additionally, we want to provide a rich set of primitive types that capture a wide range of common data types and provide means to concisely and clearly express intent instead of relying on conventions and documentation to ensure that the data is interpreted correctly.

2.1.2 Policy Issues. Data specifications often include compliance and policy issues, such as the need to ensure that sensitive data is not exposed, that data is encrypted in transit and at rest, and that data is only used in ways that are consistent with the user's consent. While these are cross-cutting concerns that must be addressed in a holistic way, the specification language plays a foundational role in expressing these source, category, and sink aspects of the data. Later tooling can then be used to check end-to-end properties of the system to ensure that the policies are being enforced correctly. Our challenge is to create a type, and declaration, system that provides a means to express that data belongs to a certain category, allow users to specify new categories relevant to their domains, and where possible provide simple solutions to common compliance issues.

2.1.3 Safe & Sandboxed Invariants. To provide more expressive power than is possible with a simple type system, the language must provide invariants over the data or configurations that it describes. These invariants may need to be executed at runtime, and as such, there are concerns with sandboxing this code, minimizing the risk of Denial-of-Service issues [8], and ensuring that it cannot leak (accidentally or maliciously) sensitive data as it runs. This requires the careful development of an expression language and library that is specialized for this task **and** that is easy for developers, who are coming from other languages, to understand and use.

2.1.4 Tooling and Discoverability. Tooling is a critical part of the this work. IDE support including syntax highlighting, type checking, auto-completion, *etc.* for authoring and consuming these data/configuration specifications is considered basic table-stakes. However, we also want to, transparently, provide the same level of tooling for any specs or configuration formats defined in this language. That is to say, if a developer authors a configuration format `FooConfig`, then we want to automatically provide IDE highlighting, completion, and type checking for working with `FooConfig` configuration files! Our challenge is to conceptualize and co-develop a native representational format for values described by the type system in our specification language. Using the combined logical and canonical physical model we can ensure that there is a well-defined workflow around values in a specification that can easily be written/read by humans (or AI agents) and that has extensive and consistent IDE support.

2.2 Data Representation

2.2.1 Efficient & Human Readable/Writable Format. The ability for human users to comfortably read and write the data format is a key concern. This is particularly important in the configuration domain where a primary use involves manual human authoring. This is also important in the data domain where the ability to easily read and write data is a key part of the development and debugging process. In these scenarios developers are manually authoring data values to test their code, and hopefully, building up a suite of test cases for future use. Thus, we want a data representation format that is human readable and writable, and that can be easily string encoded, while also being efficient to parse and consume. This is a difficult balance as human friendly formats are often less efficient to parse while machine friendly formats often make extensive us of magic constants or injected meta-data.

2.2.2 Universality and Expressiveness. In addition to being human readable and writable, the data format should provide concise and explicit support for common data types such as dates, BigNums, UUIDs, byte-buffers *etc.* In most existing systems these values are, by convention, encoded as strings and then re-parsed into the expected value type in the consuming system. This is a common source of confusion and errors as well as leading to performance issues by introducing redundant string manipulation operations. Our challenge is to provide bespoke support for a rich set of common data types in the data format to give explicit information to human and AI agents about the types of the values and to ensure these formats can be identified and handled efficiently in parsing. Additionally, we want to provide a simple means for developers to extend and wrap these types to support specifics of their domain, such as tagging an number as being a temperature in Fahrenheit vs a windspeed in MPH.

2.2.3 Polyglot Parsing and Multiple Representations. The data representation should support parsing into many languages, not exclusively or in a way that is heavily biased towards a single language. This is critical as the formats are, by definition, intended for use in a polyglot system with a wide range of programming languages co-operating. Additionally, the language should map reasonably into a range of well known formats, such as JSON, XML, SQL, protobuf, *etc.* and not preclude the option of using a high-efficiency binary format if desired. Our challenge is to balance the data representation format to ensure a linear left-lookahead (LL) parser can be built for the data without overburdening the format with verbose declarations. Further, as much as possible, we want to infer tag and type values to avoid the need to repeatedly parse and lookup these values (as strings).

2.2.4 Deduplication and References. Finally, there are pragmatic concerns for common uses of data that should be supported. Common issues include preserving aliasing and cycles in the data, de-duplicating common values, embedding environment values or constants in the data value, and defining a value as a delta from another existing value. Sometimes these values may be explicitly given, implicit relative in a workflow, or defined as part of the specification itself. Our challenge for these tasks is to provide a means of naming and resolving these references in the data format in a controlled, safe, and understandable manner.

3 TYPE LANGUAGE

By framing the description of a data value or a configuration via types, instead of a data schema, we automatically inherit the compositionality, modularity, reuse, and safety benefits of a full programming type system. From this perspective the critical issue is how to structure this type system so that it is simultaneously expressive, easy to use, and can be effectively mapped into SDKs for a wide range of popular programming languages.

The type system for the BsqON specification language is designed as a mix of structural and nominal types that are familiar from languages like C++, C#, or TypeScript that a modern developer is likely to be familiar with. Although these type systems are incompatible in various ways, some have implicit `null` values while others provide `nullable` types or union types along with variations in inheritance and subtyping rules, the BsqON type system is designed to be reasonably mappable into any these languages.

3.1 Primitive Types

Providing a small set of builtin primitive types is simple and easy to understand but can force developers to continually and manually reinvent definitions of common types. Conversely, providing a large set of primitive types can be overwhelming and can lead to a proliferation of types that are difficult to remember and that are rarely useful (or used).

For the set of builtin primitive types, BsqON looks broadly at the most common foundation types seen in APIs, and that we want to support directly such as `BigInt` and `BigNat`, `UUIDs`, or dates and times (Figure 1). Instead of forcing developers to manually define these types, or encode them as strings with a special structure, we provide them as primitives that have well known syntax and can be extended for domain specific concepts. To handle the long tail of semi-common types that are not provided as primitives, but that foundationally are really just a wrapper around a primitive, we provide a `typedecl` (Section 3.2) mechanism that allows developers to define their own types that are based on the provided primitives.

Many of the primitive types in Figure 1 are self-explanatory. However, there are a number of subtle design choices such as the use of 63 bits for the max magnitude of the `Int` and `Nat` types. This ensures that unsigned to signed casts and signed negation are always safe. Further, this magnitude allows for safe encoding of all JSON `SAFE_INTEGER` values. The `Rational` type is a novel but useful precise fractional number type with a `BigInt` numerator and a `Nat` denominator. This allows for a arbitrarily large value with a precision of up to $1/(2^{63} - 1)$. This is controlled roundoff and fixed precision is a useful alternative to floating point numbers (binary or decimal which BsqON also provides) for many applications. The inclusion of `DecimalDegree` and `LatLongCoordinate` types reflections their use in many applications and the utility of having a foundational way to talk about geographic location. Further, these particular values also have a very distinct literal value representation that allows for easy parsing and validation.

The inclusion of various Date/Time types is a reflection of the fact that these are common types and that they have subtle semantics which make the different representational choices critical for different scenarios. The `DateTime` *vs.* `UTCDateTime` *vs.* `PlainDate` distinction is one such example. In some cases the timezone is important while in other cases a UTC time, or even just a date in a consistent locale, is sufficient and simpler than worrying about timezone conversions. In addition the `DateTime` timezone is a free form string, `{[a-zA-Z0-9/, _-]+}` that allows for a wide range of custom locale representations of a time to be used – *e.g.* `{New-York Trading Day}`.

The `TimeStamp` provides millisecond precision ISO timestamps (in UTC time) as these are widely used. The `TickTime` and `LogicalTime` types are designed to be used for times taken in terms of processor tick time or a logical time measured in steps. The former is useful for performance monitoring and the latter is critical for distributed systems or monotonic values for an `ETag` [21]. Similarly the `Time Deltas` (one delta type for each time type) provide support for the common need to represent time differences. This is particularly important for combined date and time representations as the result of a delta in *seconds* applied to a Data/Time value is not always the

None	:=	Singleton none value type
Nothing	:=	Singleton `nothing` value for `Optional<T>` types
Bool	:=	`true | false`
Int/Nat	:=	Signed/Unsigned 63 bit numbers
BigInt/BigNat	:=	Arbitrary precision signed/unsigned numbers
Float/Decimal/Complex	:=	Floating point numbers
Rational	:=	Finite denominator resolution Rational
DecmialDegree	:=	Decimal Degree to 8 places
LatLongCoordinate	:=	Latitude/Longitude pair
String	:=	UTF-8 encoded string
ASCIIString	:=	ASCII encoded string
ByteBuffer	:=	Byte Buffer
UUID	:=	UUID of various formats
SHAHashCode	:=	SHA 256 Content Hash
DateTime	:=	Date and Time with Timezone
UTCDateTime	:=	UTC Date and Time
PlainDate/PlainTime	:=	Data or Time only
TimeStamp	:=	Timestamp (UTC) with milliseconds
TickTime/LogicalTime	:=	Time count in processor or logical ticks
Time Deltas ...	:=	Deltas for time values
UnicodeRegex	:=	A regex over unicode strings
ASCIIRegex	:=	A regex over ascii strings
PathRegex	:=	A regex that can be used on Paths

Fig. 1. Primitive Types in BsqON.

same as the result of applying a delta in *months* (or *hours*) due to issues with days in a month, daylight saving, or historical leap seconds!

The BsqON regex types derive from the designs in BREX [19]. This reworking of regular expression and path globbing languages addresses many error prone features in common PCRE regex languages and introduces a number of novel features designed to support using regexes for validation. This new regex design also makes a strong distinction between regexes that are designed to be used on ASCII strings, Unicode strings, and paths. This matches well with the split between `String`, `ASCIIString`, and the `Path` types in BsqON.

3.1.1 StringOf and ASCIIStringOf. The `StringOf` type is a special type that is used to represent a (unicode or ASCII) string that conforms to a specific structure. This type is parameterized by a regular expression [20]. This is a superpower for data specification and, a variation of this feature, is heavily used in two popular existing web-programming API specification formats OpenAPI [27] and Microsoft's TypeSpec [37].

```
typedecl Zipcode = /[0-9]{5}("-"[0-9]{4})?/;

const NYCode: StringOf<Zipcode> = "abc"Zipcode;   %%error not in Zipcode language
const NYCode: StringOf<Zipcode> = "10001";        %%error a string -- not StringOf
const NYCode: StringOf<Zipcode> = "10001"Zipcode; %%ok
```

In this example the specification declares a named Unicode regex called `Zipcode` and then uses it to define a `StringOf<Zipcode>` typed constant. The first two assignments are errors as the literal values do not match the regex and are not validated respectively. The third assignment is correct as the literal value matches the `Zipcode` regex. In a spec the developer can safely assume that `NYCode` is a valid zipcode and use it as such.

3.1.2 Path & Glob. A similar situation exists for the `Path` type. This type is used to represent a URI style path (excluding query params and segments) [4]. Although the language of URI Path values is theoretically regular, and thus could be described by regexes, in practice these regexes are complex and error prone for a developer to write out in full. Instead, the BsqON specification language provides a `Path` type that is parameterized by a *Path Glob* expression from the BREX package. These globs are a pseudo-regex language specifically for describing URI-like paths.

```
const UserOutputLog: PathGlob = g\file:///home/user/[env['USER_NAME']]/**/["output_"[0-9]+].log\;
const MeNotTmp: PathGlob = g\file:///home/user/me/Splash/MyPaper/*.[!("log" | "aux" | "gz")]\;

const MyPaper: Path<NotTmp> = \file:///home/user/you/Splash/MyPaper/paper1.pdf\MeNotTmp; %%error not my paper
const MyPaper: Path<NotTmp> = \file:///home/user/me/Splash/MyPaper/paper1.pdf\MeNotTmp;  %%ok this is mine
```

In the example above, the `UserOutputLog` and `MeNotTmp` constants are defined using Path Glob expressions. These expressions are then used to define a `Path<MeNotTmp>` typed constant. The first assignment is an error as the literal value does not match the `MeNotTmp` glob. The second assignment is correct as the literal value matches the `MeNotTmp` glob. As also seen in this example the BREX Path Glob language supports the use of environmental variables in the regular expressions, full regex matching on each component, or sequence of components in the path, and even allows limited negation on segments of the path pattern!

3.2 TypeDecls

Type declarations are the feature that allow us to lift from this the set of primitive types built into the language to a set of types that are specific to the domain of the application. A `typedecl` is a new-type definition that takes a primitive type, StringOf, or Path type and wraps it in a *distinct* new name. This definition can also include a set of constraints that are used to validate the values of the new type (Section 4). The accessor syntax `.value()` is a way to access the underlying primitive value from a `typedecl` value.

The ability to create distinct semantic/conceptual types using primitive data is a powerful tool for developers and can be easily used to prevent eliminate entire classes of subtle bugs [30]. Consider the motivating example from [30], converted to use BsqON types, where there is a function with 2 string parameters.

```
function getUser (companyId: String, userId: String): User {...}

function doSomethingWithUser(companyId: String, userId: String): User {
    let user = getUser(userId, companyId); %%oops -- swapped the parameters
    ...
}
```

In this example the `companyId` and `userId` parameters are both of type `String` so the type-checker will miss the accidental swap of the parameters leading to an error or accidental disclosure of data! However, if we define a `typedecl` for each of these parameters, even if they are still using strings as the underlying type, we can catch this error at compile time.

```
typedecl CompanyId = String;
typedecl UserId = String;

function getUser (companyId: CompanyId, userId: UserId): User {...}
function doSomethingWithUser(companyId: String, userId: String): User {
    let user = getUser(userId, companyId); %%now detected at compile time!
    ...
    let user2 = getUser("Robert'); DROP TABLE Students;--"CompanyId, ""UserId); %%maybe still some problems!
    ...
}
```

As can be seen in this example, a creative programmer can still introduce bugs since the underlying data-types are still just arbitrary strings. However, when combined with validated strings (`StringOf` or `ASCIIStringOf`) and validations (Section 4) the `typedecl` feature can be used to create powerful static and automatic dynamic checks to catch these correctness and security issues.

```
typedecl ValidCompanyId = /[a-zA-Z0-9_]+/;
typedecl CompanyId = StringOf<ValidCompanyId>;

typedecl UserId = String & { validate !value.empty(); };

function doSomethingWithUser(companyId: String, userId: String): User {
    ...
    let user2 = getUser("Robert'); DROP TABLE Students;--"CompanyId, ""UserId); %%Now a type error for both!
    ...
}
```

The `typedecl` feature has been used to create a `CompanyId` type that is a `String` that is restricted to a specific (non-empty) set of SQL safe ASCII characters, which can be checked at compile time and/or automatically at runtime, and catch issues like the possible SQL injection attack in the `companyId` type. Similarly, the `UserId` type has been defined to be a `String` with an automatic runtime invariant check. These examples are in the context of a function call (as that was the original example) but are equally applicable when considering general API specification and use.

3.3 Structural Types

Structural types, specifically tuples and records, are a natural part of polyglot service based programming. The BsqON language provides both Tuples and Record types with the following syntax:

$$\begin{array}{rcl} \text{Tuple} & := & [\text{Type}_1 \ldots, \text{Type}_k] \\ \text{Record} & := & \{\text{PropertyName}_1 : \ \text{Type}_1 \ldots, \text{PropertyName}_j : \ \text{Type}_j\} \end{array}$$

The `Tuple` type is a simple ordered collection of types. The `Record` type is a collection of named properties with associated types. In both cases we disallow subtyping and require exact structural matches on both number and type of elements.

3.4 Nominal Types

User defined nominal types are the workhorse of the BsqON type system. These types are used to define the majority of the composite types in a specification. The BsqON language provides for multiple inheritance of nominal types and distinguishes between abstract `Concept` types and concrete `Entity` types as shown in Figure 2.

The `FieldDecl` syntax supports the declaration of a new (non-shadowing field name) of a given type. A field may have an optional default value, the expression language is covered in detail in Section 4, but we note that based on the properties of the language this initialization is not limited to constants. In fact it can any expression, including references to other fields in the object with the only restriction being circular dependency in the initializations, which are checked at compile time. The `ValidateDecl` syntax supports the declaration of a new check for the type as a boolean expression that is used to validate the values of the fields in the type.

The `provides` list enumerates the set of other Concept types that this type provides. The concept of provides is a form of multiple inheritance that:

$$
\begin{array}{rcl}
\text{FieldDecl} & := & \text{Annotations } \texttt{field FieldName:} \text{ Type } (= \text{ Expression})? \; ; \\
\text{ValidateDecl} & := & \texttt{validate Expression} \; ; \\
\text{TemplateDecl} & := & \text{<Type}_1 \text{ Type}_{\text{restrict1}}? \ldots, \text{Type}_k \text{ Type}_{\text{restrictk}}?\text{>} \\
\text{TypeNameDecl} & := & \text{Typename TemplateDecl?} \\
\text{Provides} & := & \texttt{provides } \text{Type}_1 \ldots, \text{Type}_j \\
\text{ConceptDecl} & := & \text{Annotations } \texttt{concept TypeNameDecl} \text{ Provides? } \{(\text{ValidateDecl} \mid \text{FieldDecl})^*\} \\
\text{EntityDecl} & := & \text{Annotations } \texttt{entity TypeNameDecl} \text{ Provides? } \{(\text{ValidateDecl} \mid \text{FieldDecl})^*\}
\end{array}
$$

Fig. 2. User defined Nominal Types in BsqON.

```
typedecl RequestID = String & { invariant !value.empty(); };
typedecl ProductID = String & { invariant !value.empty(); };
typedecl OrderPrice = Decimal;
typedecl Quantity = Nat;

concept Request {
    field id: RequestID;
    field requestPrice: OrderPrice
}

entity BuyRequest provides Request {
    field quantity: Quantity
    field product: ProductID;

    invariant $requestPrice = $quantity * $requestPrice;
}

entity SellRequest provides Request {
    field buyerId: RequestID;
}
```

Fig. 3. Example Financial Processing Application types.

(1) Defines a subtype relation where A `provides` $B \Rightarrow A <: B$.
(2) Defines the inheritance of all fields and validates from the provided types – name conflicts are a compile time errors.

The `ConceptDecl` syntax supports the declaration of a new fully abstract type. These types cannot be instantiated and attempting to do so is a compile time error. The `EntityDecl` syntax supports the declaration of a new concrete type. These types can be instantiated and used in the specification but cannot be further extended. Thus, there is a strong guarantee around type checking and structure on where subtyping is possible. Further, a BsqON specification is considered closed from a type perspective. A consumer cannot create additional type instantiations of a template type or create a new subtype of a concept type. Only the instantiations (or concepts/entities) created explicitly in the specification are valid. These nominal types are primary mechanism for building an ontology of data. A simple example is a (simplified) sample demo application published by Morgan Stanley [23] as shown in Figure 3 and refined in Figure 5.

3.4.1 Extended Algebraic DataTypes. The example trading app code shows a common pattern in data specification, and programming languages generally. In the example there is a simple root concept type that is extended in minor ways by multiple entities. Functional languages often provide *Algebraic Data Types* (ADTs) to support this pattern, but in their classic form these are not

$$\begin{array}{rcl}
\text{PFieldDecl} & := & \text{Annotations } \texttt{FieldName:} \text{ Type } (= \text{ Expression})? \text{ ;} \\
\text{RootDecl} & := & \text{Annotations } \texttt{datatype} \texttt{ TypeNameDecl} \text{ Provides? } \texttt{using} \texttt{\{}(\text{ValidateDecl} \mid \text{PFieldDecl})^{*}\texttt{\}} \\
\text{SubtypeDecl} & := & \text{Annotations } \texttt{TypeNameDecl} \text{ Provides? } \texttt{\{}(\text{ValidateDecl} \mid \text{PFieldDecl})^{*}\texttt{\}} \\
\text{EADTDecl} & := & \text{RootDecl } \texttt{of} \text{ SubtypeDecl } (\mid \text{SubtypeDecl})^{*} \text{ ;}
\end{array}$$

Fig. 4. User defined Nominal Types in BsqON.

```
datatype Request using {
    id: RequestID;
    requestPrice: OrderPrice
} of
BuyRequest {
    quantity: Quantity
    product: ProductID;

    validate $requestPrice = $quantity * $requestPrice;
}
| SellRequest {
    buyerId: RequestID;
}
;
```

Fig. 5. Financial Processing Application types using Extended Algebraic DataTypes (E-ADTs).

$$\begin{array}{rcl}
\text{List} & := & \texttt{List<}\text{Type}\texttt{>} \\
\text{Set} & := & \texttt{Set<}\text{Type}_s\texttt{>} \\
\text{Map} & := & \texttt{Map<}\text{Type}_k\texttt{,}\text{Type}_v\texttt{>} \\
\text{Option} & := & \texttt{Optional<}\text{Type}\texttt{>} \mid \texttt{Something<}\text{Type}\texttt{>} \mid \texttt{nothing} \\
\text{Result} & := & \texttt{Result<}\text{Type}_t\texttt{,}\text{Type}_e\texttt{>} \mid \texttt{Result<}\text{Type}_t\texttt{,}\text{Type}_e\texttt{>::Ok} \mid \texttt{Result<}\text{Type}_t\texttt{,}\text{Type}_e\texttt{>::err}
\end{array}$$

Fig. 6. Utility Types in BsqON.

well suited to many of the scenarios we encounter in practice. In particular there are often shared fields that are common to all subtypes and we want to declare them (and related invariants) in the root concept for all the subtypes to inherit.

The desire to mixing OOP and ADT patterns leads us to introduce a specialized datatype declaration for *Extended ADTs* (EADTs) that is a combination of the concept and entity declarations in a pseudo-ADT style shown in Figure 4. This notation allows us to concisely express the same structure as the previous example and in practice this simplified syntax can reduce the amount of boilerplate and increase the clarity of the code substantially. Using the notation from Figure 4 we can rewrite the previous example as shown in Figure 5.

3.4.2 Utility Types & Containers. The BsqON language provides a suite of utility types and containers that are commonly used in data specifications and configuration files. As with the Primitive types in Section 3.1 the goal is to provide a small set of types that are commonly used and that have a distinct literal representation. These types are described in Figure 6.

The List, Set, and Map types are the standard collection types. We require that Type_s and Type_k in the Set and Map types are either simple equality comparable primitive types, None, Bool, `Int/Nat`, `BigInt/BigNat`, `String/ASCIIString`, `UUID`, `SHAHashCode`, *etc.*, or are `typedecl` types with one of these as the underlying type.

The `Option` and `Somthing` types along with the `Result` and `Ok/Err` types are standard error handling and optional value types.

3.5 Union and Intersection Types

To support ad-hoc composition and organization BsqON provides a restricted version of intersection types and a general union type.

Intersection types are denoted with the syntax CType_1 `&` CType_2, and are only supported on `concept` (or other intersection) types. This type indicates that all subtypes must be declared as providing *both* CType_1 and CType_2. More formally:

$$\begin{aligned}
T <: C_1 \ \texttt{\&} \ \ldots \ \texttt{\&} \ C_i \quad & \text{if} \quad \forall C_m,\ T \ \texttt{provides}\ C_m \lor T = C_m \\
C_1 \ \texttt{\&} \ \ldots \ \texttt{\&} \ C_i <: T \quad & \text{if} \quad \exists C_m,\ C_m \ \texttt{provides}\ T \lor C_m = T \\
C_1 \ \texttt{\&} \ \ldots \ \texttt{\&} \ C_i <: C'_1 \ \texttt{\&} \ \ldots \ \texttt{\&} \ C'_j \quad & \text{if} \quad \forall C'_n \exists C_m,\ C_m \ \texttt{provides}\ C'_n \lor C_m = C'_n
\end{aligned}$$

Union types are denoted with the syntax Type_1 `|` Type_2, and are supported on all types. This type indicates that a subtype is allowed to be either Type_1 or Type_2. More formally:

$$\begin{aligned}
T <: T_1 \ | \ \ldots \ | \ T_i \quad & \text{if} \quad \exists T_m, T <: T_m \\
T_1 \ | \ \ldots \ | \ T_i <: T \quad & \text{if} \quad \forall T_m, T_m <: T \\
T_1 \ | \ \ldots \ | \ T_i <: T'_1 \ | \ \ldots \ | \ T'_j \quad & \text{if} \quad \forall T_m \exists T'_n, T_m <: T'_n
\end{aligned}$$

Example of the uses of these types is shown below:

```
%% ad-hoc type constraints
concept Respose {
    field status: Int;
    ...
}

concept Cacheable {
    field cacheKey: SHAHashCode;
}

type CacheableRespose = Response & Cacheable;

%% nullable types
type MaybeInt = Int | none;

%% union type of possible ETag values
type ETag = UUID | SHAHashCode | LogicalTime;
```

The first set of declarations shows how intersection types allow the ad-hoc creation of refined types, in this case that a value should be both a `Response` *and* `Cacheable`. The second set of declarations shows how union types can be used in various ways. The `MaybeInt` type shows how a nullable type can be built using a union of `Int` and none. The `ETag` type shows how a union of multiple types can be used to represent a set options for an *ETag* that various parts of the system may use – *e.g.* some might generate UUID while other might take a hash of the state content.

3.6 Type Annotations

To support common information control tasks and to provide the basis for more specialized end-to-end data flow tooling the BsqON language provides a explicit `sensitive` type and element annotation mechanism. This mechanism allows for the explicit annotation of a type or a field within a nominal type (Section 3.4). Simple examples of this are shown below:

```
enum LevelTag {
    pii,
    nomarket
}

sensitive typedecl Secret = String;

entity User {
    sensitive[LevelTag::pii] field name: String;
    password: Secret;
}
```

In this example the typedecl type `Secret` is annotated as `sensitive` so all uses of this type will carry that trait. The entity `User` declares the specific field `name` as `sensitive` with the specific tag of `pii`. The `sensitive` annotation allows specialization with a set of enum values to provide a more detailed ontology of sensitivity. In our example we may want to mark the `name` as `pii` but useable for marketing purposes but that must be treated as sensitive Personally Identifiable Information (PII) data.

This simple annotation allows for direct implementation of several universally useful patterns – scrubbing sensitive values from general logs or debug dumps and monitoring the set of API endpoints + storage systems that handle sensitive data. In these cases the exact level/form of sensitivity does not matter, instead we know that storing any sensitive data in general logs is a risk and likely a compliance violation. These annotations also provide the basis for more advanced information flow control and end-to-end dataflow analysis tools as well.

4 EXPRESSION LANGUAGE

The expression language that BsqON provides for validating data values is based on the concept of a regularized programming language introduced by Bosque [18]. This approach to programming language design focuses on simplifying various features to create a predictable, visually intuitive, and analyzable language. In our domain these features are particularly important in a number of ways. Firstly, this design ensures that the language has only immutable values and thus no accidental mutation of data that is being parsed/validated. The language is also focused on "lifting semantic information into syntactically explicit forms" which increases the readability, and thus value of these expressions as human and AI consumable, documentation. The fact that the language, and core libraries, are fully specified and have no indeterminate or implementation defined behaviors ensures that, in our polyglot world, all clients have the same behavior when accepting/rejecting data. Finally, the language is designed to map into efficiently checkable SMT formulas to support automated reasoning tasks (such as generating parameterized testing values Section 6.2).

An example of these expressions and their use for validation is shown in below. This example is derived from a sample trading application [23] that manages various trades and produces various summary results. The specifications for the data types and other associated information is shown in Figure 8.

```
validate $orders.unique(pred(a, b) => a.id !== b.id);

validate $startAvailable - $orders.sumOf<Quantity>(fn(o) => o.quantity) == $available;
```

The first validate expression is used to ensure that all member `orders` in a List have a unique identifier value. The second validation expression is used to ensure that the available quantity at the start of trading, less the sum of the new new orders, is equal to the currently available quantity. As Bosque does not have loops, these expressions are written using named higher order transformations, which provide clear names describing their semantics, and in this case, can be converted to efficiently checkable SMT formulas. As the underlying BsqON type-system, including the regular expression language, is encodable as decidable logics it is possible to automatically generate counter-examples, generate finite-model proofs of correctness, and in many cases even full-proofs for arbitrary assertions about BsqON specifications [10, 18, 20]!

5 LITERAL DATA REPRESENTATION

Human and AI driven authoring of BsqON values is a fundamental part of many workflows. Clearly, configuration files are a case where human authoring is the norm. However, the ability to author BsqON values in a human-friendly way is also important for debugging, testing, and troubleshooting as the primacy of JSON style structured logging[1] and values in RESTful systems attests. With this in mind, this section covers the design of a data format for BsqON values that is optimized with syntactic hints, common case ergonomics, and structure to ease human authoring and reading while, as much as possible, maximizing support for efficient parsing.

5.1 Explicit Syntax

5.1.1 Base Values. The first area where we apply the principle of explicitly encoding semantic information in syntax is the representation of the primitive values. Figure 7 shows the literal representation for each of the primitive types. We note that these representations (as *regular languages*) are mutually disjoint. Thus, the type of any value in this format is explicitly encoded in the syntax of the value itself.

The syntax for each primitive type is explicit, unambiguous, and builds on natural intuition that a developer from a Java/C#/JS/TypeScript background would have. This makes it easy for humans, AI agents, a parser, or developer tools to identify values and their types without any additional context. One subtle deviation from the ISO 8601 representation of timezones [14] is that we allow arbitrary timezone strings to be used in the BsqON format. This allows users to tag times with domain specific timezone strings, such as "NY Trading Region" or "London Trading Region", which allows for customized decisions around location based time processing.

The representation for the structured strings (`StringOf` and `ASCIIStringOf`) along with the `Path` types simply take the literal forms and tag with a suffix of the declared validator type. Examples of these are shown in subsubsection 3.1.1 and subsubsection 3.1.2. Similarly, for the user defined new-types (`typedecls`) the syntax is simply a primitive literal value followed by an _ and the type name – *e.g.* `23.0f_Fahrenheit` or `3/8R_Inch`. In unambiguous cases, when the type is known from the context, the type name can be omitted – *e.g.* `23.0f` or `3/8R`.

5.1.2 Composite Values. The structural types `Tuple` and `Record` borrow syntax existing in many programming languages. The `Tuple` type is represented as a comma separated list of values in square brackets – *e.g.* `[1n, true, "ok"]`. The `Record` type is represented as a comma separated list of key-value (equality) pairs in curly brackets – *e.g.* `{a=1n, b=true, c="ok"}`.

Nominal types are represented with the (optional) type reference followed by the constructor arguments for the type. These arguments are permitted to be positional or named, and arguments for fields with default values can be omitted.

$$
\begin{aligned}
\text{PosArg} \quad &:= \quad \text{Value} \\
\text{NamedArg} \quad &:= \quad \texttt{FieldName}\text{=Value} \\
\text{ArgList} \quad &:= \quad (\text{PosArg} \ \mid \ \text{NamedArg})^* \\
\text{NominalCons} \quad &:= \quad \text{TypeName?}\{\text{ArgList}\}
\end{aligned}
$$

[1]Monitoring with JSON formatted data can consume between 10-30% of the total execution time for Node.js applications! [16]

None	:=	`none`
Nothing	:=	`nothing`
Bool	:=	`true | false`
Int/Nat	:=	sign and digits with (`i`|`n`) suffix
BigInt/BigNat	:=	sign and digits with (`I`|`N`) suffix
Float/Decimal/Complex	:=	sign and digits with (`f`|`d`) suffix or complex $r \pm c$`i` format
Rational	:=	sign and n/d`R` format
DecmialDegree	:=	sign and digits with `dd` suffix
LatLongCoordinate	:=	`...lat` ± `...long` Decimal Latitude/Longitude pair
String	:=	`"..."` double quoted string + `%...;` style escapes
ASCIIString	:=	`'...'` single quoted string + `%...;` style escapes
ByteBuffer	:=	`0x[...]` array of hex bytes
UUID	:=	`uuidX"{...}` UUID of version X in byte-hypen form
SHAHashCode	:=	`shaX{...}` 256bit Content Hash using SHA version X
DateTime	:=	Date and Time with Timezone – ISO 8601 format + arbitrary timezone strings
UTCDateTime	:=	UTC Date and Time – ISO 8601 format
PlainDate/PlainTime	:=	Data or Time only – ISO 8601 format
TimeStamp	:=	Timestamp (UTC) – ISO 8601 format with milliseconds
TickTime/LogicalTime	:=	digits with (`t`|`l`) suffix
Time Deltas ...	:=	explicit ± followed by corresponding time format
Regexs	:=	`/.../` with (`a`|`p`) for ASCII/Path flavors

Fig. 7. Primitive Types in BsqON.

The builtin constructors have a similar representation, with `lists/sets` having only positional arguments and `maps` using a notation of Value=>Value for the key-value pairs. The special utility types can be constructed explicitly from their type names or, if the context is explicit, using shorthand `something(v)`, `ok(v)`, `err(v)` notation.

Using the types described in Figure 3 a `BuyRequest` value (and some invalid variations) in the BsqON literal format is:

```
%%fully named and tagged construction
BuyRequest{
  id="b1"_RequestID,
  requestPrice=10.5d_OrderPrice,
  quantity=100n_Quantity,
  product="Apples"_ProductID
}

%%shorthand literal with positional arguments and one named argument
BuyRequest{product="Apples"_ProductID, "b1"_RequestID, 10.5d_OrderPrice, 100n_Quantity}

%%shorthand literal with positional arguments and omitted typedecl decorations
BuyRequest{id="b1", 10.5d, 100n, "Apples"}

%%shorthand literal in known type context with all omitted decorations
{"b1", 10.5d, 100n, "Apples"}

%%error on missing quantity property
BuyRequest{id="b1"_RequestID, requestPrice=10.5d_OrderPrice, product="Apples"_ProductID}
```

```
%%type error on parameter swap (quantity and requestPrice)
BuyRequest{id="b1", 100n_Quantity, 10.5d_OrderPrice, "Apples"}
```

5.1.3 Unions, Concepts, and Tagging. A frequent challenge in literal data representations is in representing union types or other situations where there may be multiple valid types in a given position. In practice systems rely on some form of tagging to disambiguate the intended type. In JSON popular approaches of this tagging include “envelopes” or adding a known tag field to the data. These approaches can be cumbersome and/or error prone. The envelope approach wraps the data in a new tuple or record with a tag field and the underlying value as the only elements – requiring an extra object per disambiguated value. The tag field approach requires a special tag field, which is at risk of collision with a data field name.

The BsqON literal format is designed to both minimize the need for additional explicit tags and to simplify the syntax/cost when they are required. By construction, for all of the primitive types, they are already implicitly diambiguatable in the syntax so no additional tagging is required. For entity types and typedecls, their literal syntax already contains a natural constructor or tagged form that includes the type name, and thus they are already resolvable as well. The only remaining types are the structural tuples and records – using the syntax `<Type> Value` as a cast-style tag. Using this syntax allows us to disambiguate the intended type of the value in a single token.

```
typedecl Foo = Nat;
entity Bar { field f: Nat; }

type MyUnion = None | Nat | Foo | Bar | [Nat, String] | [Int, String]

%%Parsing a MyUnion type as
none    %%None
2n      %%Nat
2n_Foo  %%Foo typedecl

Bar{ 2n } %%Bar entity

<[Nat, String]> [2, "ok"] %%[Nat, String]
<[Int, String]> [2, "ok"] %%[Int, String]
```

5.1.4 Value References. To support explicit control of aliasing (sharing of values) and as an additional mechanism to reduce redundancy in the literal format, we allow for the use of value references and accesses. Further, we provide support for the common scenarios of using environment variables and a various special names.

The first form of naming, explicit value references, is a simple mechanism for binding a name to a value and then permitting that name to be used in place of the value inside a literal sub-value. The syntax use for this is a classic parenthesized let-in form – `(let` Name`:` Type = Value `in` Value`)`. This construct allows us to provide explicit aliasing (or cyclic structure) semantics for values and also provides a uniform means to eliminate duplicative/redundant value descriptions. This can be used in an automated fashion to deduplicate values in a storage or transmission context or by a human developer to improve the readability of a data value they are authoring.

The literal language also allows for dereferencing sub-values of tuples, records, and collections. Elements in records and tuples are accessed via a '`.`' dereference followed by the field or tuple index. For collections, the syntax is Value`[`Index`]` where index is an integer value (with negative values indicating offsets from the end). Finally, the syntax Value`[`Key`]` is used to access the value associated with a key in a map.

```
(let x: String = "A really long string that is used multiple times ......" in
  [x, x, x] %% deduplicate
)
(let x: [Nat, Nat] = [1n, 2n] in
  {a=x, alias=x} %% aliasing
)
(let x: ListNode = ListNode{next=x} in
  x %% cyclic structure
)
(let x: [Nat, Nat] = [1n, 2n] in
  x.0 %% 1n
)
(let x: Map<Nat, Int> = {1n => -1i} in
  x[1n] %% -1i
)
```

Beyond these manually bound names/values, the BsqON literal format also supports the use of environment variables and special names. The environment variables are accessed using the syntax `env['Name']` where name is an ascii string. These names are resolved from an environment that is passed into the parser as described by the literal meta-data (Section 5.3).

BsqON provides three special names that are programmatically bound to values before or during parsing. The first is a special variable `$src` that can be used to define a value based on a diff from an existing value provided at parse time! The second are useful shorthands, `$index` and `$key`, that are bound to the current index or key being processed in a collection or map.

```
List<Nat>{$index, 1n, $index}                              %% same as List<Nat>{0n, 1n, 2n}
Map<Int, {k: Int, v: String}>{2i => {k: $key, v: "two"}} %% same as {2i => {k: 2i, v: "two"}}
```

Finally, the BsqON literal format provides a special extension point naming mechanism for integration with generative AI, fuzzing, or testing tools. Specifically, there is a special name syntax $?Name that indicates a placeholder value (or ($?Name:Type) typed placeholder) that is to be filled in by a later stage in the processing pipeline. In the parsing process these placeholders are resolved and checked for both type correctness and any related validation constraints.

5.2 Efficient Parsing

Given the type language described in Section 3 and literal syntax described above we can now address the question of parsing. Specifically, given a BsqON assembly A, a specific type T, and a literal value l we want to parse l into a value of type T – that is lex/parse the structure, resolve names, check the types, and extract the meaningful values from the string representations.

Our parser implementation is a simple single-pass recursive descent parser that is driven by the type information. The parse function signature is –

```
Result<Value, Error> parse(char8_t* str, Type* type, Assembly* assembly)
```

– where the parser performs type checking inference based on the literal values seen at each step. By design the BsqON literal format is a $LL(1)$ language and, in conjunction with, propagating the type information down as the parsing occurs we can parse and type check in a single pass. Thus, the format can be stream parsed for efficiency or we can produce a full AST of the value for tasks like code completion or error analysis in an IDE.

This structure also allows us to relax some of the syntax requirements as well. For example we can omit constructor types when there is only a single option based on the type inference from the context or we can allow dropping the extra types on typedecl or StringOf types to simplify the authoring experience or reduce the size of the value data representation. Since the type information is an explicit part of the process (Section 5.3) we can also compile the parser into an optimized form specific to a given assembly and type.

5.3 Metadata Integration

The previous sections describe the literal value syntax and parser in terms that take a type and assembly as well as information on an environment and possibly a special relational context value. In some systems these may be implicit in the use of the parser – like in a command line utility or embedded in a programming language. However, for other scenarios, like a configuration file, or a RESTful API, these values need to be communicated in some manner.

Instead of requiring an external (and ad-hoc) mechanism to communicate this information the BsqON literal syntax provides explicit support for optional meta-data in the format. For information on the assembly we provide a *shebang* notation at the top-level of the literal value that points to a reference to the assembly via a URI path – *e.g.* a local file, a URL, a package database, or even a github repo. The type of the value in a literal can also be explicitly included in the meta-data and we allow for the inclusion of partial (or full) environmental information in the meta-data as well. This allows for a single literal and the required meta-data to be represented as a single file, string, or buffer and easily passed between various parts of a system.

Consider a simple example of a data value for a weather forecast request as specified from a package database:

```
#!api://weatherapi/1.0.0/forecast?ForecastRequest
env{'UNITS'=>UnitOptions, 'CURRENT_LOCATION'=>LatLongCoordinate}
{
    location=env['CURRENT_LOCATION'],
    date=2024-04-05,
    forecastKind=Forecast::daily,
    units=env['UNITS']
}
```

In this example we have a complete literal value that is resolved using the assembly `forecast` from the given data-type specification repository of the type `ForecastRequest`. The meta-data includes the required environment variables, `UNITS` and `CURRENT_LOCATION`, along with the expected type of the values, that are expected to be defined in the client and instantiated when it loads/parses this value to make a request to the remote server. The parser will also check that all the required fields are initialized and set to appropriate values, the date format is correct and does not have an hour/min/second component, the enumerations for `forecastKind` and `units` are valid, and that any optional fields are set if/as needed.

6 CASE STUDIES

This section uses several case studies to explore different aspects of the BsqON language and its use. The first case study is on data specifications (Section 6.1) and corresponds to investments being done in a top-5 investment bank on data modeling and quality based on this work. The next two case studies look at using BsqON for configuration files (Section 6.3) and for parameterized testing and fuzzing (Section 6.2). The final case study is on using BsqON for API SDK generation (Section 6.4) which is a problem that is being addressed at a top-3 cloud provider using ideas from this work.

6.1 Data Specification

To illustrate how BsqON can be used for data specifications we look at a sample trading application published by Morgan Stanley [23]. This application manages various trades and produces various summary results. A representative example for these data types is the `DaySalesSummary` type, and associated declarations, as shown in Figure 8.

The first thing to note about this definition, as opposed to an equivalent Swagger or OpenAPI definition, is that it is fundamentally a type definition. As such it can be composed, versioned, and reused in the same way and with the same tools as any other programming language! This is a major advantage as it allows for us to build complex systems from simpler components and manage complexity using well-known and effective engineering practices.

```
%%%%
%% Data Specification
typedecl Identifier = /[A-Za-z][A-Za-z0-9_]{0-9}/a;
typedecl CUSIP = /[0-9]{6}[0-9A-Z]{3}/a;

typedecl Quantity = BigNat;

entity SaleOrder {
    field id: ASCIIStringOf<Identifier>;
    field instrument: ASCIIStringOf<CUSIP>;
    field quantity: Quantity;
}

entity DaySalesSummary {
    field available: Quantity;
    field startAvailable: Quantity;
    field orders: List<SaleOrder>;

    validate $orders.unique(pred(a, b) => a.id !== b.id);
    validate $startAvailable - $orders.sumOf<Quantity>(fn(o) => o.quantity) == $available;
}

%%%%
%% Literal
DaySalesSummary{
    available: 800N_Quantity,
    startAvailable: 1000N_Quantity,
    orders: List<SaleOrder>{
        SaleOrder{ 'A1'Identifier, '123456789'CUSIP, 100N_Quantity },
        SaleOrder{ 'A2'Identifier, '123456789'CUSIP, 100N_Quantity }
    }
}
```

Fig. 8. Declarations from Sample Trading App

Beyond the baseline value of moving into a full type-system for this specification, this example also shows how the extensive support for defining types in terms of their business requirements increases the clarity of the intent behind each component and the correctness of the system – as generated parsers are able to automatically run the specified data validation. An example of this is the `SaleOrder` field `instrument` which is declared to be of type `StringOf<CUSIP>`. In a JSON based encoding this would be simply declared as a string, leaving developers unclear on what instrument is expected to be, instead with BsqON it is expressly declared to be a 9 digit ASCII encoded value and, even better, the validator name tells us that it represents a CUSIP value! For a user in this domain this is a common term referring to the standard CUSIP[2] identifier for financial instruments. Further, the fact that the `id` and `instrument` fields have distinct declared types, even thought they are both physically ASCII strings, provides easy/efficient interop with other systems while eliminating chances of parameter confusion bugs [30] bugs in the use of this type.

The data validation operations provide another level of clarity and correctness to the system. They allow us to automatically check that the data conforms not only to the basic structural requirements of the specification but also to the business rules that are required for the system to function correctly. For example, the `DaySalesSummary` type has a validation that ensures the `available` field is always equal to the difference between the `startAvailable` and the sum of the `quantity` fields of the `orders` list.

In practice this allows the BsqON parser to catch invalid and ill formed data like the following without any additional manual checks or code:

```
DaySalesSummary{
    available: 900N_Quantity,                                     %% fails startAvailable - orders check
    startAvailable: 1000.5,                                       %% fails type check
```

[2]https://www.cusip.com/identifiers.html

```
    orders: List<SaleOrder>{
        SaleOrder{ 'A1'Identifier, '123$56789'CUSIP, 100N_Quantity }, %% fails CUSIP format check
        SaleOrder{ 'A1'Identifier, '123456789'CUSIP, 100N_Quantity }  %% fails invariant on unique id
    }
}
```

This example illustrates how the BsqON language can can be used to define rich data specifications that are both clear and correct. In combination with the BsqON literal syntax this allows for efficient parsing and the ability to automatically preform data validation which is known to be a major source of defects in systems citerestler and leads to incompatibilities [7] between how different parts of the system accept/reject values! Interestingly, as a result of lifting semantic information into the explicit syntax of the types LLM systems, like GPT-4, are quite effective at generating valid literals for a given type. In fact the literal in Figure 8 was generated by GPT-4, with one fix on the available/startAvailable quantities.

6.2 Testing & Parameterized Fuzzing

In addition to supporting easy human authoring for configuration files, the BsqON literal syntax is also quite effective for authoring test cases, and with the addition of the *placeholder* syntax in subsubsection 5.1.4 it can be used to generate parameterized test cases and fuzzing inputs as well [9, 20, 35]. Given our sales example declaration we could create a single concrete value to use as a test case or we could create a parameterized input to specifically test code that involves subsets of the properties:

```
%% parameterize id values to cover how they are handled
SaleOrder{ $?id_fuzz1, '123456789'CUSIP, 100N_Quantity }

%% parameterize the cusips to cover how they are handled
DaySalesSummary{
    available: 800N_Quantity,
    startAvailable: 1000N_Quantity,
    orders: List<SaleOrder>{
        SaleOrder{ 'A1'Identifier, $?cusip_fuzz1, 100N_Quantity },
        SaleOrder{ 'A2'Identifier, $?cusip_fuzz1, 100N_Quantity }
    }
}

%% parameterize each element in the list
DaySalesSummary{
    available: 800N_Quantity,
    startAvailable: 1000N_Quantity,
    orders: List<SaleOrder>{
        $?order_fuzz1,
        $?order_fuzz2
    }
}
```

These parameterized values show how we can use the syntax to generate test cases with a range of possible (valid) values. In the first case just the values of the `id` fields but keeping all of the other values fixed. The second case shows a unique ability that comes from the explicit naming were we can ensure that the test uses the same (aliased if needed) value for the CUSIP value in both fields simply by giving them the same parametric name. The third case shows how this feature can be used to generate full (compound) subvalues. In this case for each of the entries in the list. As noted in Section 4, both the type/regex checks *and* the Bosque language can be mapped into decidable fragments of logic for Z3 [10, 18]. This feature enables us to effectively solve for the values parameters or results must satisfy in order to meet the constraints of the specification [20].

6.3 Configuration File

The next case is the specification of a configuration language based on the `launch.json` configuration used for setting up debugging tasks in Visual Studio Code. The `launch.json` file is a JSON file that is used to configure the debugger in the IDE. Our version of the configuration [38] is shown below. The description needed for the JSON specs is 100+ lines of normative text such as "program - executable or file to run when launching the debugger". Further, this format requires a hand rolled string substitution DSL, validation logic, and IDE tooling for the parsing/processing of this file. In contrast the BsqON specification for this configuration file can be concisely expressed as:

```
...
enum DebuggerRequestEnum { launch, attach }

concept DebugLaunchConfig {
    field name: String;
    field dbgType: DebuggerTypeEnum;
    field request: DebuggerRequestEnum;

    field program: PathOf<FSPath>;
    field args: List<String> = List<String>{};
    field cwd: PathOf<FSPath> = env['workspaceRoot'];
    field dbgEnv: Map<ASCIIString, String?> = Map<ASCIIString, String?>{};
    field envFile: PathOf<FSPath>? = none;
    field stopOnEntry: boolean = false;

    validate $name != "";
}
```

In this example the `DebugLaunchConfig` concept represents the base for all launch configurations. Instead of declaring almost every field as a string the BsqON declaration can use enumerations, structured path types, and explicitly optional fields (and their default values) to clearly and concisely define this configuration. Further, the `validate` operation is used to ensure the additional constraint that the `name` field is never empty.

In the current VSCode model defining the configurations for specific languages relies on ad-hoc extensions of this type. Instead in BsqON we can use the subtyping and inheritance features to cleanly create this hierarchy of configurations, manage their definitions via symbolic lookups, and version them just as we would if they were types in a full-fledged programming language.

The design of the literal representation provides a clean and simple to read/write configuration file that has IDE support and validation provided by construction. An examples of a BsqON configuration file for the given specification is shown below:

```
{
    name: "Launch Program",
    dbgType: DebuggerTypeEnum::node,
    request: DebuggerRequestEnum::launch,
    program: \file:///<env['workspaceRoot']>/bin/app.js\FSPath,
    args: ["--arg1", "value1"],
    %% cwd: not provided use default value
    dbgEnv: { 'NODE_ENV' => "development" },
    %% envFile: not provided use default value
    stopOnEntry: true
}
```

In an IDE this file can be automatically validated, we can provide code completion, and hover-over documentation. Given the explicitness of the syntax, the ability to include (or drop) type names and property names, along with supporting comments, the BsqON configuration file is substantially more readable and maintainable than the equivalent JSON, TOML or YAML configuration.

6.4 API SDK Generation

The final case study we present is using BsqON for API SDK generation. While microservice, serverless, and cloud based systems have adopted either JSON or protobufs as defacto standards for for shared interchange, most programs handle these external API calls using a language specific SDK that wraps the underlying RPC or REST operations in a more idiomatic (and often type-safe) language specific library. These are frequently hand-written, or at least hand-refined, based on the RESTful API documentation – often a mixture of Swagger, some design notes, and the implementation code. Unsurprisingly, the result is code that involves substantial engineering effort to write and maintain, there are inconsistencies between how the API functions in the various SDKs, and input validation is often buggy or incorrect [1].

Instead, we can use BsqON to define the API and then compile this canonical specification into the SDKs for the various languages. By design the BsqON type system is intended to be broadly interoperable, with particular attention to major languages/formats in the cloud and web space such as Java, C#, TypeScript, Python, C/C++, SQL, JSON, and OpenAPI. Consider the `DaySalesSummary` type from the first case study, we can define SDK bindings for C++ and TypeScript as follows:

```
////
//C++ SDK
type boost::multiprecision::mpz_int Quantity;

class DaySalesSummary {
public:
    Quantity available;
    Quantity startAvailable;
    std::vector<SaleOrder*>;

    //constructors/destructors
    ...
};

////
//TypeScript SDK
type Quantity = bigint;

class DaySalesSummary {
    available: Quantity;
    startAvailable: Quantity;
    orders: SaleOrder[];
}
```

The BsqON parser provides a C binding for traversing the ASTs generated from parsing BsqON literal values (or JSON values) and this can be linked into the SDK runtime to provide the necessary parsing and validation support. Thus, once we have, manually, written a translator & runtime for a target language we can then automatically generate the needed SDK bindings[3] and have confidence that they are correct and consistent with the ground truth API specification. This is a major advantage over the current state of the art and this technique already being used in production at a top-3 cloud provider to generate their SDKs for their cloud services.

7 RELATED WORK

Data intercommunication formats (specifications) and configuration languages are a critical part of modern computing. They are the dark matter that defines the meaning of data in a business content, that glues together microservices, that describes what and how a system should do to access resources, *etc.* Given this ad-hoc set of uses, there are just as many ad-hoc DSLs and formats that have grown organically to support them! In recent years there has been a shift de-facto standards

[3]Currently the service endpoints (functions) are specified in a custom DSL. Future work is to extend BsqON with explicit support for these definitions.

of JSON [15], XML, YAML [39], TOML [36], and the classic INI format. While these formats are widely used, they have serious limitations that manifest in the form of security vulnerabilities, poor readability, and error prone features.

Recent industrial led work has started to address some of these challenges, particularly in the context of defining cloud (RESTful) APIs. The OpenAPI Specification [27], RAML [29], CUE [6], smithy [32], and typespec [37] are all examples of these efforts. Approaches like OpenAPI, RAML, and CUE focus on providing a superset of JSON and various DSL mechanisms for describing JSON structures/encodings. Systems like the smithy and typespec languages, like this work, define actual type systems for describing the logical semantics of data, independent of the encoding, and then providing generators for JSON, OpenAPI, or other language SDKs based on these types. However, these systems stop at this point and cannot be used to define the arbitrarily rich semantic checks that are possible with the BsqON `validates` construct and do not provide the type of expressive literal object language support described in Section 5. Instead users are left authoring literal data and configurations in JSON (or similar) and the limitations that these formats impose. Protobuf [28] and gRPC [12] are two other well known tools in this space. These systems provide a way to define a schema for data that can be used to generate code for serializing and deserializing data in a variety of languages. However, these systems are focused on the serialization and deserialization of data not the creation of a logical model of the data or the issue of validation.

Despite the critical role data and configurations play in modern computing, the topic of data specification and static configuration languages has been relatively unstudied in a theoretical context. The majority of work has focused on build systems [5, 22], end-to-end policy languages (Cedar) [17], or validating properties of configuration systems [3, 13, 17, 31]. These are critical problems but their focus is on the dynamic aspects of configuration such as identifying dependencies for build ordering and incremental rebuilding or transitive policy interactions leading to globally undesirable behaviors. They assume the existence of some underlying static resource specification system – which could in practice even be BsqON. In this respect the work presented here is complementary to these efforts as it provides a richer base for these systems to build on.

A recent and notable project in this space is the S3 project [11, 33, 34]. This work looks at building many of the same workflows that are supported by BsqON, such as testing and mocking APIs. However, it approaches this topic from the viewpoint of a grammar for the physical encoding of the inputs/outputs of the API using a novel DSL (ISLa) instead of from the perspective of a programming language and type system. Interestingly our experiences, along with interviews and feedback from developers, showed limited value in modeling a service at this level as opposed to the higher level of the logical types it operated on. Further, we observed that many developers were not comfortable or interested in learning (and remembering) another specialized DSL. This led to an aggressive focus to "meet developers where they are" and the realization that combination of a type system for data (configuration) + a rich literal language for authoring values was a powerful model that could be readily understood and easily adopted by practicing developers. Further, the ability to do SDK generation and data validation from the specifications provided us with a strong motivating scenario to drive initial adoption.

Work on *typed literal macros* [25] (or *type specific languages* [26]) explore how to add custom syntax or DSLs to provide specialized literal notation for specific types, *e.g.* SQL or HTML, as well as custom syntax extensions for user defined types. In some sense the literal format in this paper can be seen as a specialized case of this prior work, specifically it provides custom syntax for a larger set of types that would be found in most programming languages. However, the literal format in Section 5 explicitly and intensionally excludes a mechanism (DSL) for building extensible custom syntax for user defined types. This choice was motivated by the DSL-itis issues that arise around onboarding and comprehension when each configuration has custom syntax that must be understood.

8 ONWARD!

This work is intended as a first step in addressing the critical but often overlooked role that data communication formats and configuration files play in modern computing. These are the dark matter that defines the meaning of data and actual running of systems in a business but that are generally treated as second-class concerns by developers and programming language academics alike! Our central contribution is that by treating these issues as a first-class concerns and viewing them as programming language problems we can create a type & validation language, along with an expressive value literal notation, that addresses many of the major pain points in this domain.

This work is ongoing and opens up multiple avenues for future research with substantial potential for practical impact. We have alluded to the incredible need for tooling to support end-to-end checks on sensitive data for compliance concerns, the need to build additional tooling for data quality and provenance tracking, and the potential for using the type system to drive automated testing and fuzzing of systems. Our initial experiments in these areas have been promising and have seen active interest from industrial partners. This is also a unique area at the intersection of programming languages and the recent explosion of work on AI powered coding assistants and end-user agents. We believe that, fundamentally, APIs are the way AI systems interact with the real world and that the description of these APIs has a foundational role in the ability of these agents to use them effectively. With these opportunities we believe the work in this paper presents the first step in a much larger area of work.

ACKNOWLEDGEMENTS

I would like to thank Brian Terlson, Stephen Goldbaum, Richard Perris, and Claudia Babescu for their sharing their insights and experiences. These have been invaluable in grounding the ideas in this paper and helping shape them into a cohesive vision.

REFERENCES

[1] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: Stateful REST API Fuzzing *(ICSE)*.
[2] Avro 2023. Apache Avro. https://avro.apache.org/.
[3] Salman Baset, Sahil Suneja, Nilton Bila, Ozan Tuncer, and Canturk Isci. 2017. Usable declarative configuration specification and validation for applications, systems, and cloud *(Middleware)*.
[4] Tim Berners-Lee, Larry M Masinter, and Roy T. Fielding. 1998. Uniform Resource Identifiers (URI): Generic Syntax. RFC 2396. https://doi.org/10.17487/RFC2396
[5] Maria Christakis, K. Rustan M. Leino, and Wolfram Schulte. 2014. Formalizing and Verifying a Modern Build Language *(FM)*.
[6] CUE 2023. CUE Documentation. https://cuelang.org/.
[7] Data Validation & Parsing 2019. Inconsistent Data Validation. Private Communication.
[8] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2018. The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale *(ESEC/FSE 2018)*.
[9] fuzz 2023. American Fuzzy Lop. https://github.com/google/AFL.
[10] Stephen Goldbaum, Attila Mihaly, Tosha Ellison, Earl T. Barr, and Mark Marron. 2022. High Assurance Software for Financial Regulation and Business Platforms *(VMCAI)*.
[11] Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2020. Mining input grammars from dynamic control flow *(ESEC/FSE)*.
[12] gRPC 2023. Project Page. https://grpc.io/.
[13] Peng Huang, William J. Bolosky, Abhishek Singh, and Yuanyuan Zhou. 2015. ConfValley: a systematic configuration validation framework for cloud services *(EuroSys)*.
[14] ISO 8601 2019. Standard for date and time representations. https://www.isotc154.org/posts/2019-08-27-introduction-to-the-new-8601/.
[15] JSON 2023. JSON Format. https://www.json.org/json-en.html.

[16] JSON Parsing 2019. JSON Overhead. Private Communication.

[17] John Kastner, Aaron Eline, Joseph W. Cutler, Shaobo He, Emina Torlak, Anwar Mamat, Lef Ioannidis, Darin McAdams, Matt McCutchen, Andrew Wells, MIchael Hicks, Neha Rungta, Kyle Headley, Kesha Hietala, and Craig Disselkoen. 2024. Cedar: A New Language for Expressive, Fast, Safe, and Analyzable Authorization *(OOPSLA)*.

[18] Mark Marron. 2023. Toward Programming Languages for Reasoning: Humans, Symbolic Systems, and AI Agents *(Onward!)*.

[19] Mark Marron. 2024. *Better Regexing with BREX – A Modern Regex DSL.* Technical Report.

[20] Marron, Mark and Goldbaum, Stephen 2021. Agile and Dependable Service Development with Bosque and Morphir. Linux Foundation Open Source Summit (https://www.youtube.com/watch?v=olAWcFOK-IU).

[21] MDN ETag 2023. ETag Description. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/ETag.

[22] Andret Mokhov, Neil Mitchell, and Simon Peyton Jones. 2020. Build systems à la carte: Theory and practice. *Journal of Functional Programming* 30 (2020).

[23] Morphir 2021. Morphir. https://github.com/finos/morphir.

[24] Office of the Comptroller of the Currency 2020. Citibank $400M Fine. https://www.occ.gov/static/enforcement-actions/ea2020-056.pdf.

[25] Cyrus Omar and Jonathan Aldrich. 2018. Reasonably programmable literal notation *(ICFP)*.

[26] Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. 2014. Safely Composable Type-Specific Languages *(ECOOP)*.

[27] OpenAPI 2023. OpenAPI 3.0 Format. https://swagger.io/specification/.

[28] protobuf 2023. Protocol Buffers. https://protobuf.dev/.

[29] RAML 2023. RAML Documentation. https://raml.org/.

[30] Andrew Rice, Edward Aftandilian, Ciera Jaspan, Emily Johnston, Michael Pradel, and Yulissa Arroyo-Paredes. 2017. Detecting argument selection defects. *Proceedings ACM Programming Languages* 1 (2017).

[31] Rian Shambaugh, Aaron Weiss, and Arjun Guha. 2016. Rehearsal: a configuration verification tool for puppet *(PLDI)*.

[32] Smithy 2023. AWS Smithy 2.0 Documentation. https://smithy.io/2.0/index.html.

[33] Dominic Steinhöfel and Andreas Zeller. 2022. Input invariants *(ESEC/FSE)*.

[34] Dominic Steinhöfel and Andreas Zeller. 2024. Language-Based Software Testing. *Commun. ACM* 67 (2024).

[35] Nikolai Tillmann and Wolfram Schulte. 2005. Parameterized unit tests *(ESEC/FSE-13)*.

[36] TOML 2023. TOML Format. https://www.toml.io.

[37] TypeSpec 2023. Microsoft TypeSpec. https://microsoft.github.io/typespec/standard-library/built-in-decorators.

[38] VSCode 2024. launch.json Attributes. https://code.visualstudio.com/.

[39] YAML 2023. YAML Format. https://www.yaml.org.