Specifying JSON Encodings of Mathematical Objects

John Cremona[3] Dennis Müller[1] Michael Kohlhase[1] David Lowry-Duda[3]
Florian Rabe[1,2] Tom Wiesing[1]

[1] FAU Erlangen-Nürnberg
[2] LRI Paris
[3] University of Warwick

Abstract. We describe a set of types and codecs for specifying the mathematical semantics and concrete encoding when storing mathematical objects in databases.

1 Introduction

Choice of Encoding Format We use JSON as the target format of our encodings. The most obvious alternative would have been XML.

Both JSON and XML are untyped and provide basic structuring features like lists and records. Moreover, both are easy to parse and serialize, and libraries exist for most programming languages.

JSON has the advantage that it is simpler (e.g., avoiding namespace issues) and has a more human-readable concrete syntax[4] It is also helpful that it explicitly distinguishes booleans, integers, and strings.

Overview

2 Types

2.1 Types

A **type** T is one of the following:

- Base types
  - number types (all unbounded in size):
    - natural numbers: N (including 0), Pos (excluding 0), $Prime$ (prime numbers, not excluding 0 and 1)
    - integer numbers: Z
    - integer numbers modulo m: $Z(m)$ (for $m \in Pos$)
    - rational numbers: Q
    - real numbers: R [1] EdN:1

[4] This is important for practitioners, which often do not have the time to learn the abstract syntax and depend on an immediately-understandable concrete syntax.

[1] EdNote: FR: it's unclear which irrational numbers we need to support and how to represent them

    - ∗ complex numbers: C
    - ∗ p-adic numbers: $Qp(p)$ (for prime p)
  - strings: $String$
  - booleans: $Boolean$
- Aggregating type constructors for types $T_1, \ldots, T_n$
  - product types: $T_1 * \ldots * T_n$
  - record types: $\{a_1 : T_1, \ldots, a_n : T_n\}$ for identifiers a_i
  - disjoint union types: $T_1 + \ldots + T_n$ [2]
  - labeled disjoint union types: $[a_1 : T_1, \ldots, a_n : T_n]$ for identifiers a_i [3]

  In record and labeled disjoint union types, the order of fields is not relevant.
- Collecting type constructors for any type T
  - option types (list of length up to 1): $Opt(T)$
  - vectors (fixed-length lists): $Vec(T, n)$ for $n \in N$
  - lists (arbitrary finite length): $List(T)$
  - sets (finite subsets): $FiniteSet(T)$
  - multisets (finite multiset subsets): $FiniteMultiset(T)$
  - finite hybrid sets (like multisets but also allowing negative mulitiplicities): $FiniteHybridset(T)$
  - matrices: $Mat(T, m, n)$ for $m, n \in N$
- Erasure of dependent parameters: for a type constructor $T(k)$ (not necessarily unary) that takes a parameter k of number type, we write $T(\_)$ for the type obtained by taking the unions over all possible parameters. In particular, we have
  - $Mat(T, m, \_)$, $Mat(T, \_, n)$ and $Mat(T, \_, \_)$ for matrices with arbitrary dimensions
  - $List(T) = Vec(T, \_)$
- Mathematical structures
  - finite maps (partial functions with finite support): $FiniteMap(T_1, T_2)$ for types T_i
  - polynomials: $Polynomial(r, [x_1, \ldots x_n])$ for ring $r \in Ring$ and identifiers x_i
  - rings: $Ring$ (multiplication is a commutative monoid)
  - fields: $Field$
  - elements of structures: every structure S (e.g., $S \in Ring$ or $S \in Field$) is a type itself; the elements of S are represented as elements of the underlying type of S, which must be defined separately for every structure S.

2.2 Subtyping

The subtyping relation $S \subseteq T$ is the order generated by

[2] EdNote: FR: not sure if we need these as they rarely come up yet as they are dual to product types

[3] EdNote: FR: not sure if we need these as they rarely come up yet; they are dual to record types

- $Prime \subseteq Pos \subseteq N \subseteq Z \subseteq Q \subseteq R \subseteq C$
- $T(k) \subseteq T(\_)$ for any type constructor T (not necessarily unary) that allows a wildcard parameter
- All collecting and aggregating type constructors are covariant in all type arguments. For example, if $S \subseteq T$, then $List(S) \subseteq List(T)$.
- Horizontal subtyping: Record types become smaller, labeled disjoint union types bigger when adding fields.
- $Ring \subseteq Field$

3 Elements

The elements of most types are straightforward. In any case, for the purposes of this document, we do not need to fix concrete syntax for the elements of most types.

However, some types have multiple representations that are different in meaningful ways (e.g., because converting between representations is expensive or imprecise). Moreover, some types have multiple constructors similar to an inductive type. In the sequel, we describe which representations are supported in those cases.

Integers Modulo The elements of $Z(m)$ are represented by $0, \ldots, m-1$.

Real Numbers A real number can be one of the following:
- a rational number
- an IEEE double precision float
- a root $\sqrt[n]{x}$ for $n \in N$ and $x \in Z$
- the strings "pi" and "e"

Complex Numbers A complex number can be one of the following:
- Cartesian form $x + yi$
- polar form $re^{i\varphi}$
- root of unity ζ_n

p-Adic Numbers A p-adic number x consists of unit $u \in N$ (u, p coprime), valuation $v \in Z$, and precision $r \in N$ (for $u < p^r$).

Polynomials For $r \in Ring$ and distinct strings x_i, we consider polynomials $p \in Polynomial(r, [x_1, \ldots, x_n])$ to be of the form $p = \Sigma_{\vec{i} \in N^n} a_{\vec{i}} \vec{x^i}$ where and $(x_1, \ldots, x^n)^{(i_1, \ldots, i_n)}$ abbreviates $x_1^{i_1} \cdot \ldots \cdot x_n^{i_n}$.

Rings A ring can be one of the following:
- a field
- $Polynomial(r, [x_1, \ldots, x_n])$ for $r \in Ring$

Fields A field can be one of the following:
- base fields Q, R, and C
- finite fields $Z(p)$ for $p \in Prime$ (same type as integers modulo p)
- polynomial field extensions $FieldExtension(F, p, a)$ of $F \in Field$ for a polynomial $p \in Polynomial(F, [x])$ (for any variable name x)
- named fields identified by a string

We define some abbreviations for common fields:
- $Q(p, a) = FieldExtension(Q, p, a)$
- $Qsqrt(n, a) = Q(x^2 - n, a)$
- $Qzeta(n, a) = Q(y_n, a)$ where y_n is the n-th cyclotomic polynomial

We do not define $GF(q)$ for $q = p^n$ as an abbreviation for $FieldExtension(Z(p), g)$ for some irreducible polynomial $g \in Polynomial(Z(p))$ of degree n because there is no way to choose g canonically and it is necessary to know g to represent the elements of $GF(q)$.

Structure Elements Every structure has an underlying type, which is used to represent its elements.

The underlying types of fields are defined as follows: For Q, R, C, and $Z(p)$, the underlying type is the field itself. The underlying type of $FieldExtension(F, p, a)$ is $Polynomial(F, [a])$ ($a = x$ is allowed).

4 Codecs

For each type, we define codecs. These are pairs of bijective maps between the elements of that type and subsets of JSON.

To encode/decode a mathematical object into/from JSON, we need to provide its type and a codec for that type. (Different codecs might encode different mathematical objects as the same JSON. Thus, the meaning of an arbitrary JSON is only determined if the codec is known.)

For codec c for type T and a mathematical object t of T, we write $c(t)$ for the encoding of t. For some types, we define a *default codec* (whose name will start with `Standard`. If a default codec d exists, we write $\bar{t}$ for $d(t)$.

A codec c is a *string-codec* if encodes every value as a JSON string. For codecs c_i, we write $c_1 | \ldots | c_n$ for the codec that encodes using c_1 and decodes according to the first c_i that is applicable to the input

4.1 JSON

We use the following JSON values:
- *null*
- 32-bit integers
- IEEE double precision floating point numbers; technically, JSON does not distinguish between integers and floats, and we assume that, e.g., 1 is an integer and 1.0 is a float
- booleans

– strings
– lists $[j_1, \ldots, j_n]$ for JSON values j_i
– objects $\{k_1 : j_1, \ldots, k_n j_n\}$ for strings k_i and JSON values j_i

We also say *pair* for lists $[j, j']$ of length 2.

4.2 Base Types

Natural and Integer Numbers All subtypes of Z as well as integers modulo m are encoded in the same way.

We have the following encodings for an integer n, we

`IntAsNumber`: the JSON integer n if $|n|$ is small enough and like `IntAsString` otherwise

`IntAsString`: a string containing the decimal expansion of n

`IntAsList`: a list $[\text{"}base\text{"}, b, l, d_1, \ldots, d_{|l|}]$ where b, l, d_i are JSON integers such that $\mathrm{sgn} l = \mathrm{sgn} n$ and $(d_1, \ldots, d_|l|)$ is the list of digits of n relative to base 2^b.[5]

`StandardInt`: the codec `IntAsNumber`|`IntAsString`|`IntAsList`

Rational Numbers `StandardRat` encodes the rational number e/d for integers e, d as

– $\overline{e}$ if $d = 1$
– the pair $[e, d]$

For encodings, the e and d are fully canceled and $d > 0$. For decoding, all pairs with $d \neq 0$ are accepted. For decoding the string "e'/d'" where e' and d' are the string encodings of e and d are also accepted.[4] EdN:4

Real Numbers The codec `StandardReal` encodes a real number as follows:

– rational numbers like `StandardRat`
– floats as JSON floats
– $n\sqrt{x}$ as the list $[\text{"}root\text{"}, \overline{n}, \overline{x}]$ for $n \in N$ and $x \in Z$
– the strings "pi" or "e"

Complex Numbers The codec `StandardComplex` encodes a complex number z (in any form) as

– if z is in Cartesian form:
  • $\overline{x}$ if $y = 0$
  • the object $\{\text{"}re\text{"} : \overline{x}, \text{"}im\text{"} : \overline{y}\}$ otherwise
– if z is in polar form
  • the object $\{\text{"}abs\text{"} : \overline{r}, \text{"}unitarg\text{"} : \overline{\alpha}\}$ if $\varphi = 2\pi\alpha$ for $\alpha \in [0, 1[$
  • the object $\{\text{"}abs\text{"} : \overline{r}, \text{"}arg\text{"} : \overline{\varphi}\}$ otherwise
– the object $[\text{"}root - of - unity\text{"}, \overline{n}]$ if z is a root of unity

[5] The value l may seem redundant in the encoding (except for carrying the sign). But it has the advantage that the canonical order on integers corresponds to the lexicographic order of JSON lists.

[4] EdNote: FR: is this needed

p-Adic Numbers The codec `StandardQp`(p) for $Qp(p)$ encodes the p-adic number with unit u, valuation v, and precision r as an object $\{"unit" : \overline{u}, "valuation" : \overline{v}, "precision" : \overline{r}\}$. [5]

EdN:5

Strings `StandardString` encodes strings as JSON strings.

Booleans `BooleanAsBoolean` encodes booleans as JSON boolean.
`BooleanAsInt` encodes booleans as 0 or 1.
`BooleanAsString` encodes booleans as the strings "true" or "false".
`StandardBoolean` is `BooleanAsBoolean`|`BooleanAsInt`|`BooleanAsString`

4.3 Aggregating Type Constructors

Corresponding to type constructors T that form types $T(T_1, \ldots, T_n)$ from existing types, we use codec constructors C that form codecs $C(C_1, \ldots, C_n)$ for $T(T_1, \ldots, T_n)$ from codecs c_i for T_i.

In the following, we assume codes $c_1, \ldots, c_n$ for the types $T_1, \ldots, T_n$. We write $\vec{c}$ for $c_1, \ldots, c_n$.

Products The codec `StandardProduct`($\vec{c}$) for $T_1 * \ldots * T_n$ encodes tuples $(t_1, \ldots, t_n)$ as JSON lists $[c_1(t_1), \ldots, c_n(t_n)]$.

Records The codec `StandardRecord`($\vec{c}$) for $\{k_1 : T_1, \ldots, k_n : T_n\}$ encodes records $\{k_1 = t_1, \ldots, k_n = t_n\}$ as JSON objects $\{"k_1" : c_1(t_1), \ldots, "k_n" : c_n(t_n)\}$.

Unions The codec `StandardUnion`($\vec{c}$) for $\{T_1 + \ldots T_n\}$ encodes values t of T_i as the JSON pair $[i, c_i(t)]$.

Labeled Unions The codec `StandardLabeledUnion`($\vec{c}$) for $[k_1 : T_1, \ldots, k_n : T_n]$ encodes values $k_i(t)$ for $t \in T_i$ as the JSON object $\{"k_i" : c_i(t_i)\}$.

4.4 Collecting Type Constructors

We assume a codec c for a type T.

Options The codec `StandardOption`(c) for $Opt(T)$ encodes values $t \in T$ as $c(t)$ and omitted values as the *null* value.

Vectors The codec `StandardVector`(n, c) for $Vec(n, T)$ encodes vectors $(t_1, \ldots, t_n)$ as the JSON list $[c(t_1), \ldots, c(t_n)]$.

Lists The codec `StandardList`(c) encodes lists in the same way as `StandardVector`.

Sets The codec `StandardSet`(c) encodes sets in the same way as `StandardVector`, where the elements are listed in any order but without repetitions.

[5] EdNote: FR: treatment of 0?

Multisets The codec `StandardMultiset`(c) encodes multisets as lists $[[c(t_1), \overline{m_1}], \ldots, [c(t_n), \overline{m_n}]]$ of pairs (t, m) where $t \in T$ is an element of the multiset and with multiplicity $m \in N$. The order of pairs is not specified. For encoding, the same t_i will not occur twice, and $m_i \neq 0$. For decoding, these cases are accepted.

Hybrid Sets The codec `StandardHypbridSet`(c) encodes hybrid sets in the same way as `StandardMultiset` except that the multiplicities may be negative.

Matrices The codec `RowMatrix`(m, n, c) for $Mat(m, n, T)$ encodes matrices (t_{ij}) as the list of lists $[[c(t_{11}), \ldots, c(t_{1n})], \ldots, [c(t_{m1}), \ldots, c(t_{mn})]$.

The codec `ColumnMatrix`(m, n, c) for $Mat(m, n, T)$ is the corresponding column-wise encoding.

We put `StandardMatrix` = `RowMatrix`.

4.5 Mathematical Structures

Finite Maps Assume codecs c, d for types S, T.

The codec $MapAsList(c, d)$ encodes the map $f \in FiniteMap(S, T)$ as the list of pairs $[[c(s_1), d(f(s_1))], \ldots, [c(s_n), d(f(s_n))]]$ where the $s_i \in S$ are the pairwise distinct arguments for which f is defined (in any order).

If c is a string-codec, the codec $MapAsObject$ encodes f as the object $\{c(s_1) : d(f(s_1)), \ldots, c(s_n) : d(f(s_n))\}$.

We put `StandardMap` = `MapAsList`.

Polynomials Assume a codec c for the underlying type T of a ring r.[6]

The codec $PolynomialAsSparseList(r, [x_1, \ldots x_n], c)$ encodes p as the list $[\ldots, [c(a_{\vec{i}}), [\overline{i_1}, \ldots, \overline{i_n}]], \ldots,]$. The entries of that list occur in any order, and no $\vec{i}$ occurs twice. In the special case $n = 1$, encoding uses $\overline{i_1}$ instead of $[\overline{i_1}]$; decoding accepts both forms.

For $n = 1$, the codec $UnaryPolynomialAsDenseList(r, [x], c)$ encodes $p = \Sigma_{i=0}^{d}$ as the list $[c(a_0), \ldots, c(a_n)]$.

Fields The codec `StandardField`(c) for the type $Field$ encodes fields as follows:

- base fields as the string "Q", "R", and "C"
- named fields as strings
- field extensions $FieldExtension(F, p, a)$ with $p \in Polynomial(F, [x])$ as $[\text{"}extension\text{"}, StandardField(F), \overline{p}, a]$ [6] EdN:6
- field extensions $Q(p, a)$ as $[\overline{p}, a]$ [7] EdN:7
- $Qsqrt(n)$ as the pair $[\text{"}Qsqrt\text{"}, \overline{n}]$
- $Qzeta(n)$ as the pair $[\text{"}Qzeta\text{"}, \overline{n}]$

[6] For example, we can use $T = C$ and $c =$ `StandardComplex` for all polynomials whose coefficients are chosen from a subset of the complex numbers.

[6] EdNote: FR: the choice of encoding for p is not easy; here I'm assuming we always have a default encoding

[7] EdNote: FR: Do we need a special case for this? What encoding should it be?

Structure Elements A codec of the type of elements of a structure S is the same as a codec for the underlying type of S.

4.6 Subtyping Property

Our standard codecs mirror the subtype relationship between the types. If $S <: T$, then the elements of S are encoded in the same way by the standard codecs for S and T.

EdN:8 [8]

5 Conclusion

References

[8] EdNote: FR: this remark could be promoted to a theorem