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

2 Types

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

- Base types
  - number types (all unbounded in size):
    - natural numbers (including 0): Nat
    - integer numbers: Int
    - rational numbers: Rat
    - real numbers: $Real$ [1] EdN:1
    - complex numbers: $Complex$
  - 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] EdN:2
  - labeled disjoint union types: $[a_1 : T_1, \ldots, a_n : T_n]$ for identifiers a_i [3] EdN:3
- 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 a natural number n

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

[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

  - 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 natural numbers m and n
  - 2-dimensional list (matrices of any finite dimensions): $List2(T)$
- Mathematical structures
  - finite maps (partial functions with finite support): $FiniteMap(T_1, T_2)$ for types T_i
  - polynomials: $Polynomial(R, [x_1, \dots x_n])$ for a commutative ring R and identifiers x_i
  - fields: $Field$

3 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 $\overline{t}$ for $D(t)$.

A codec C is a *string-codec* if encodes every value as a JSON string. For codecs C_i, we write $C_1|\dots|C_n$ for the codec that encodes using C_1 and decodes according to the first C_i that is applicable to the input

3.1 JSON

We use the following JSON values:
- *null*
- 32-bit integers
- booleans
- strings
- lists $[j_1, \dots, j_n]$ for JSON values j_i
- objects $\{k_1 : j_1, \dots, k_n j_n\}$ for strings k_i and JSON values j_i

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

3.2 Base Types

Natural and Integer Numbers Natural numbers and integers 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.[4]

`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 $\text{"}e'/d'\text{"}$ where e' and d' are the string encodings of e and d are also accepted.[4] EdN:4

Real Numbers A real number can be given in one of the following forms:

- a rational number
- a root $\sqrt[n]{x}$ for a natural number n and an integer x
- the strings "pi" and "e"

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

- rational numbers like `StandardRat`
- $\sqrt[n]{x}$ as the list $[\text{"}root\text{"}, \overline{n}, \overline{x}]$ for a natural number n and an integer x
- the strings "pi" or "e"

Complex Numbers A complex number can be given in one of the following forms:

- Cartesian form $x + yi$
- polar form $re^{i\varphi}$
- root of unity ζ_n

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

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] 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

3.3 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 $\{"k_i" : C_i(t_i)\}$.

3.4 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`.

If C encodes 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.

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 $m \in Nat$. 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`.

Two-dimensional List The codecs `RowList2`, `ColumnList2`, and `StandardList2` encode values of type $List2(T)$ in the same way as `RowMatrix`, `ColumnMatrix`, and `StandardMatrix`.

3.5 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.[5]

Consider polynomials of the form $p = \Sigma_{\vec{i} \in Nat^n} a_{\vec{i}} \vec{x^i} \in Polynomial(R, [x_1, \ldots, x_n])$ where $(x_1, \ldots, x^n)^{(i_1, \ldots, i_n)}$ abbreviates $x_1^{i_1} \cdot \ldots \cdot x_n^{i_n}$. 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 following forms of fields are supported:[5] EdN:5

- base fields Rat, $Real$, and $Complex$
- named fields identified by a string
- polynomial field extensions $Rat(p)$ of Rat for a polynomial $p \in Polynomial(Rat, [x])$ (for any variable name x)
- extensions of Rat with $\sqrt{n}$ written as $Qsqrt(n)$
- extensions of Rat with a root ζ_n of unity written as $Qzeta(n)$

Given a codec C for $Polynomial(Rat, [x])$, the codec `StandardField`(C) for the type $Field$ encodes fields as follows:[6]

- base fields as the string "Rat", "$Real$", and "$Complex$"
- named fields as strings

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

[5] EdNote: @John: Fields are the most difficult type so far. Please check if this in particular.

[6] Note that is different from codecs for encoding the elements of an individual fields.

– field extensions $Rat(p)$ as $C(p)$
– $Qsqrt(n)$ as the pair $[\text{"}Qsqrt\text{"}, \overline{n}]$
– $Qzeta(n)$ as the pair $[\text{"}Qzeta\text{"}, \overline{n}]$

4 Conclusion

References