Nano JSON

Working with JSON formatted data in Isabelle/HOL and Isabelle/ML

Achim D. Brucker

October 27, 2022

Department of Computer Science, University of Exeter, Exeter, UK
`a.brucker@exeter.ac.uk`

Abstract

JSON (JavaScript Object Notation) is a common format for exchanging data, based on a collection of key/value-pairs (the JSON objects) and lists. Its syntax is inspired by JavaScript with the aim of being easy to read and write for humans and easy to parse and generate for machines.

Despite its origin in the JavaScript world, JSON is language-independent and many programming languages support working with JSON-encoded data. This makes JSON an interesting format for exchanging data with Isabelle/HOL.

This AFP entry provides a JSON-like import-expert format for both Isabelle/ML and Isabelle/HOL. On the one hand, this AFP entry provides means for Isabelle/HOL users to work with JSON encoded data without the need using Isabelle/ML. On the other and, the provided Isabelle/ML interfaces allow additional extensions or integration into Isabelle extensions written in Isabelle/ML. While format is not fully JSON compliant (e.g., due to limitations in the range of supported Unicode characters), it works in most situations: the provided implementation in Isabelle/ML and its representation in Isabelle/HOL have been used successfully in several projects for exchanging data sets of several hundredths of megabyte between Isabelle and external tools.

Keywords: JSON, JavaScript Object Notation, data exchange, data import, data export

Contents

1 Introduction **7**

2 Implementation **9**

2.1 An Import/Export of JSON-like Formats for Isabelle/HOL (Nano_JSON) . . . 9
2.1.1 Defining a JSON-like Data Structure . . . 9
A HOL Data Type for JSON . . . 10
ML Implementation . . . 10
2.1.2 Parsing Nano JSON . . . 11
ML Implementation . . . 11
Isar Setup: Cartouche and Isar-Top-level Binding . . . 13
Examples . . . 15
2.1.3 Serializing Nano JSON . . . 15
ML Implementation . . . 16
Isar Setup . . . 16
Examples . . . 17
2.1.4 Putting Everything Together . . . 17
2.2 Query Infrastructure (Nano_JSON_Query) . . . 17
Isabelle/ML . . . 17
Isabelle/HOL . . . 19
2.3 The Main Theory of Nano JSON (Nano_JSON_Main) . . . 20

3 User's Guide and Examples **21**

3.1 The Nano JSON Manual (Nano_JSON_Manual) . . . 21
3.1.1 Isabelle/HOL . . . 21
3.1.2 Isabelle/ML . . . 22
3.2 Examples (Example) . . . 22
3.3 Examples Real (Example_Real) . . . 25

1 Introduction

JSON (JavaScript Object Notation) [5, 1] is a widely-used format for exchanging data, having replaced XML [2] in many areas. The structure of JSON is based on:

- A collection of key-value (name-value) pairs, also called JSON object. In programming languages, this corresponds, e.g., to data structures such as objects, records, structs, or hash tables.
- A finite list of values. In programming languages, this corresponds, e.g., to lists, sequences, or arrays.

The syntax of is inspired by JavaScript with the aim of being easy to read and write for humans (that are comfortable with curly-bracket programming languages) and easy to parse and generate for machines.

Despite its origin in the JavaScript world, JSON is language-independent and many programming languages support working with JSON-encoded data. This makes JSON an interesting format for exchanging data with Isabelle/HOL.

This AFP entry provides a JSON-like import-expert format for both Isabelle/ML and Isabelle/HOL. While this entry provides a simple formal model of JSON, formalizing an object-based data structure is not the aim of this AFP entry (for this, see, e.g., [4, 3]). The aim of this AFP entry is to provide means to work with JSON-encoded data in Isabelle, similar to the AFP entry "XML" [6] provides means to work with XML documents in Isabelle.

In more detail, this AFP entry provides two different ways for importing and exporting JSON-encoded data in Isabelle:

Isabelle/HOL: This AFP entry provides means for Isabelle/HOL users to work with JSON encoded data: it provides a HOL data type modelling the JSON format and new top-level Isar [7] commands to import and export JSON encoded files. Moreover, a new top-level Isar command is provided to parse JSON data that is stored "inline" within the Isabelle/HOL theory file.

Isabelle/ML: This AFP entry also provides interfaces to the underlying implementation in Isabelle/ML, allowing to write users to write their on extensions that require features not (yet) exposed to the Isabelle/HOL level. For example, data type packages for specific variants of JSON that, e.g., should directly map to specific HOL data types or that need further encoding or conversions during export or import.

While the supported JSON-variant is not fully JSON compliant (e.g., due to limitations in the range of supported Unicode characters), it works in most situations: the provided implementation in Isabelle/ML and its representation in Isabelle/HOL have been used successfully in several projects for exchanging data sets of several hundredths of megabyte between Isabelle and external tools.

In summary, Nano JSON should enable you to work with JSON encoded data in Isabelle/HOL without the need of implementing parsers or serialization in Isabelle/ML. You should be able to implement mapping from the Nano JSON HOL data types to your own data types on the level of Isabelle/HOL (i.e., as executable HOL functions). Nevertheless, the provided ML routine that converts between the ML representation and the HOL representation of Nano JSON can also serve as a starting point for converting the ML representation to your own, domain-specific, HOL encoding.

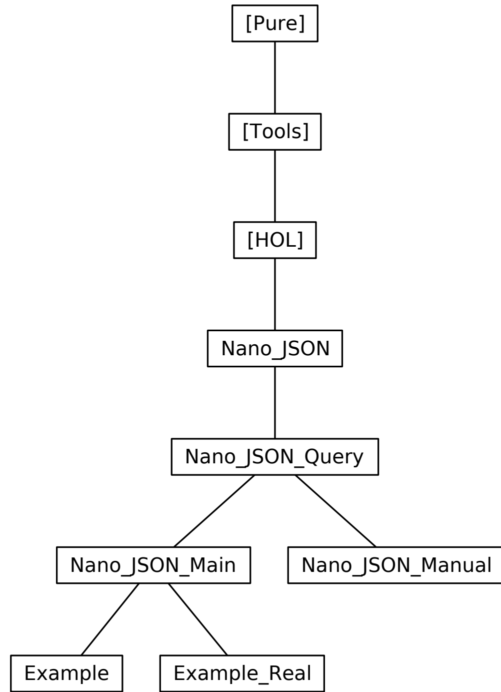


Figure 1.1: The Dependency Graph of the Isabelle Theories.

The rest of this document is automatically generated from the formalization in Isabelle/HOL, i.e., all content is checked by Isabelle. Overall, the structure of this document follows the theory dependencies (see Figure 1.1). If you want to understand the implementation of Nano JSON or want to use its API on the level of Isabelle/ML, you should consult chapter 2. If you are interested in using Nano JSON on the level of Isabelle/HOL or want to explore examples, please consult chapter 3.

2 Implementation

2.1 An Import/Export of JSON-like Formats for Isabelle/HOL (Nano_JSON)

```
theory
  "Nano_JSON"
imports
  Main
keywords
      "JSON_file" :: thy_load
  and "JSON" :: thy_decl
  and "JSON_export" :: thy_decl
  and "defining"::quasi_command

begin
```

This theory implements an import/export format for Isabelle/HOL that is inspired by JSON (JavaScript Object Notation). While the format defined in this theory is inspired by the JSON standard (`https://www.json.org`), it is not fully compliant. Most notably,

- only basic support for Unicode characters. In particular, JSON strings are mapped to a polymorphic type usually is either instantiated with the `string` or `String.literal`. Hence, the strings that can be represented in JSON are limited by the characters that Isabelle/HOL can handle (support on the Isabelle/ML level is less constrained).
- numbers are mapped to a polymorphic type, which can, e.g., be instantiated with
  - `real`. Note that this is not a faithful representation of IEEE754 floating point numbers that are assumed in the JSON standard. Moreover, this required that parent theories include Complex_Main.
  - `int`. This is recommended configuration, if the JSON files only contain integers as numerical data.
  - `string`. If not numerical operations need to be done, numberical values can also be encoded as HOL strings (or `String.literal`).

  While the provided JSON import and export mechanism is not fully compliant to the JSON standard (hence, its name "Nano JSON"), it should work with most real-world JSON files. Actually, it has already served well in various projects, allowing us to simply exchange data between Isabelle/HOL and external tools.

2.1.1 Defining a JSON-like Data Structure

We start by modelling a HOL data type for representing the abstract syntax of JSON, which consists out of objects, lists (called arrays), numbers, strings, and Boolean values.

A HOL Data Type for JSON

```
datatype ('string, 'number) json =
     OBJECT "('string * ('string, 'number) json) list"
     | ARRAY "('string, 'number) json list"
  | NUMBER "'number"
  | STRING "'string"
  | BOOL "bool"
  | NULL
```

Using the data type `('string, 'number) json`, we can now represent JSON encoded data easily in HOL, e.g., using the concrete instance `(string, int) json`:

```
definition example01::‹(string, int) json› where
"example01 =
  OBJECT [(''menu'', OBJECT [(''id'', STRING ''file''), (''value'', STRING ''File''),
          (''popup'', OBJECT [(''menuitem'', ARRAY
                       [OBJECT [(''value'', STRING ''New''),
                                (''onclick'', STRING ''CreateNewDoc()'')],
                        OBJECT [(''value'', STRING ''Open''),
                                (''onclick'', STRING ''OpenDoc()'')],
                        OBJECT [(''value'', STRING ''Close''),
                                (''onclick'', STRING ''CloseDoc()'')]
                       ])]
           )]),(''flag'', BOOL True), (''number'', NUMBER 42)
         ]"
```

ML Implementation

The translation of the data type `('string, 'number) json` to Isabelle/ML is straight forward with the exception that we do not need to distinguish different String representations. In addition, we also provide methods for converting JSON instances between the representation as Isabelle terms and the representation as Isabelle/ML data structure.

```
ML‹
signature NANO_JSON_TYPE = sig
    datatype NUMBER = INTEGER of int | REAL of IEEEReal.decimal_approx
    datatype json = OBJECT of (string * json) list
                  | ARRAY of json list
                  | NUMBER of NUMBER
                  | STRING of string
                  | BOOL of bool
                  | NULL

    val term_of_real: bool -> IEEEReal.decimal_approx -> term
    val term_of_json: bool -> typ -> typ -> json -> term
    val json_of_term: term -> json
end
›
```

```
ML_file Nano_JSON_Type.ML
```

The file `Nano_JSON_Type.ML` provides the Isabelle/ML structure `Nano_Json_Type:NANO_JSON_TYPE`. When first argument of the functions `Nano_Json_Type.term_of_real` and `Nano_Json_Type.term_of_json` is `true`,

then warnings are reported to the the output window of Isabelle. Otherwise, warning will be ignored. Furthermore, the two arguments of type `typ` of the function `Nano_Json_Type.term_of_json` represent the HOL target type for JSON strings and numerical values. An example invocation of this function looks as follows:

```
ML‹Nano_Json_Type.term_of_json false @{typ "string"} @{typ ‹int›} Nano_Json_Type.NULL›
```

2.1.2 Parsing Nano JSON

In this section, we define the infrastructure for parsing JSON-like data structures as well as for importing them into Isabelle/HOL. This implementation was inspired by the "Simple Standard ML JSON parser" from Chris Cannam.

ML Implementation

Configuration Attributes. We start by preparing the infrastructure for three configuration attributes, using the Isabelle/Isar attribute mechanism:

```
ML‹
  val json_num_type = let
    val (json_num_type_config, json_num_type_setup) =
      Attrib.config_string (Binding.name "JSON_num_type") (K "int")
  in
    Context.>>(Context.map_theory json_num_type_setup);
    json_num_type_config
  end
›
```

The attribute "JSON_num_type" (default `int`) allows for configuring the HOL-type used representing JSON numerals.

```
ML‹
  val json_string_type = let
    val (json_string_type_config, json_string_type_setup) =
      Attrib.config_string (Binding.name "JSON_string_type") (K "string")
  in
    Context.>>(Context.map_theory json_string_type_setup);
    json_string_type_config
  end
›
```

The attribute "JSON_string_type" (default `string`) allows for configuring the HOL-type used representing JSON string.

```
ML‹
  val json_verbose = let
    val (json_string_type_config, json_string_type_setup) =
      Attrib.config_bool (Binding.name "JSON_verbose") (K false)
  in
    Context.>>(Context.map_theory json_string_type_setup);
    json_string_type_config
  end
›
```

The Boolean attribute "JSON_verbose" (default: false) allows for enabling warnings during the JSON processing.

Lexer. The following Isabelle/ML signatures captures the lexer:

ML ‹
```
signature NANO_JSON_LEXER = sig
    structure T : sig
        datatype token = NUMBER of char list
                       | STRING of string
                       | BOOL of bool
                       | NULL
                       | CURLY_L
                       | CURLY_R
                       | SQUARE_L
                       | SQUARE_R
                       | COLON
                       | COMMA

        val string_of_T : token -> string
    end
    val tokenize_string: string -> T.token list
end
```
›

ML_file Nano_JSON_Lexer.ML

The file Nano_JSON_Lexer.ML provides the Isabelle/ML structure Nano_Json_Lexer:NANO_JSON_LEXER. It provides the function Nano_Json_Lexer.tokenize_string which, as the name suggests, tokenizes a string (containing a JSON object).

ML ‹
```

signature NANO_JSON_PARSER = sig
    val json_of_string : string -> Nano_Json_Type.json
    val term_of_json_string : bool -> typ -> typ -> string -> term
    val numT: theory -> typ
    val stringT: theory -> typ
end
```
›

ML_file "Nano_JSON_Parser.ML"

The file Nano_JSON_Parser.ML provides the Isabelle/ML structure Nano_Json_Parser:NANO_JSON_PARSER. The two main functions:

- First, Nano_Json_Parser.json_of_string parsing a string (containing a JSON object) and returning its abstract syntax (i.e., an instance of the type Nano_Json_Type.json.
- Second, Nano_Json_Parser.term_of_json_string, which parses a string into a HOL term. As for Nano_Json_Type.term_of_json, the first argument decides if warnings are printed and the next wo arguments determine the HOL type representations for JSON strings and numerical values.

  The parser MLNano_Json_Parser.term_of_json_string can now be used, on the Isabelle/ML-level as follows:

ML ‹
```
  Nano_Json_Parser.term_of_json_string true (@{typ string}) (@{typ int})
```

```
                                        "{\"name\": [true,false,\"test\"]}"
›
```

Isar Setup: Cartouche and Isar-Top-level Binding

The JSON Cartouche. First, we define a cartouche that allows using JSON syntax within HOL expressions:

```
syntax "_cartouche_nano_json" :: "cartouche_position ⇒ 'a"  ("JSON _")
parse_translation‹
let
  fun translation u args = let
      val thy = Context.the_global_context u
      val verbose = Config.get_global thy json_verbose
      val strT = Nano_Json_Parser.stringT thy
      val numT = Nano_Json_Parser.numT thy
      fun err () = raise TERM ("Common._cartouche_nano_json", args)
      fun input s pos = Symbol_Pos.implode (Symbol_Pos.cartouche_content
                                         (Symbol_Pos.explode (s, pos)))
    in
      case args of
        [(c as Const (@{syntax_const "_constrain"}, _)) $ Free (s, _) $ p] =>
          (case Term_Position.decode_position p of
            SOME (pos, _) => c
                          $ Nano_Json_Parser.term_of_json_string verbose strT numT (input s
pos)
                          $ p
          | NONE => err ())
      | _ => err ()
  end
in
  [(@{syntax_const "_cartouche_nano_json"}, K (translation ()))]
end
›
```

In the following, we briefly illustrate the use of the JSON cartouche and the attribute for mapping JSON types to HOL types:

```
declare [[JSON_string_type = string]]
lemma ‹y == JSON ‹{"name": true}› ›
  oops

declare [[JSON_string_type = String.literal]]
lemma ‹y == JSON ‹{"name": true}› ›
  oops
declare [[JSON_string_type = string]]

lemma ‹y == JSON‹{"name": [true,false,"test"]}› ›
  oops
lemma ‹y == JSON‹{"name": [true,false,"test"],
                  "bool": true,
                  "number" : 1 }› ›
  oops
```

Isar Top-Level Commands. Furthermore, we define two Isar top-level commands: one that allows for importing JSON data from the file system, and one for defining JSON "inline" within Isabelle theory files.

```
ML‹
structure Nano_Json_Parser_Isar = struct
    fun make_const_def (binding, trm) lthy = let
            val lthy' =  snd ( Local_Theory.begin_nested lthy )
            val arg = ((binding, NoSyn),
                       ((Thm.def_binding binding,@{attributes [code]}), trm))
            val ((_, (_ , thm)), lthy'') = Local_Theory.define arg lthy'
        in
            (thm, Local_Theory.end_nested lthy'')
        end


    fun def_json name json lthy = let
            val thy = Proof_Context.theory_of lthy
            val strT = Nano_Json_Parser.stringT thy
            val numT = Nano_Json_Parser.numT thy
            val verbose = Config.get_global thy json_verbose
    in
       (snd o (make_const_def (Binding.name name,
                               Nano_Json_Parser.term_of_json_string verbose strT numT json)))

       lthy
    end

    val typeCfgParse  = Scan.optional
                             (Args.parens (Parse.name -- Args.$$$ "," -- Parse.name))
                             (("",""),"");
    val jsonP = (Parse.name -- Parse.cartouche)

end
›

ML‹
val _ = Outer_Syntax.local_theory @{command_keyword "JSON"}
        "Define JSON."
        ((Parse.cartouche -- keyword ‹defining› -- Parse.name) >> (fn ((json, _), name)
        => Nano_Json_Parser_Isar.def_json name json))

val _ = Outer_Syntax.command command_keyword ‹JSON_file›
        "Import JSON and bind it to a definition."
        ((Resources.parse_file -- keyword ‹defining› -- Parse.name) >>
         (fn ((get_file,_),name) =>
           Toplevel.theory (fn thy =>
           let
              val ({lines, ...}:Token.file) = get_file thy;
              val thy'' = Named_Target.theory_map
                             (Nano_Json_Parser_Isar.def_json name (String.concat lines))
                             thy
           in thy'' end)))
›
```

Examples

Now we can use the JSON Cartouche for defining JSON-like data “on-the-fly”. Note that you need to escape quotes within the JSON Cartouche, if you are using quotes as lemma delimiters, e.g.,:

```
lemma "y == JSON‹{"name": [true,false,"test"]}›"
  oops
```

Thus, we recommend to use the Cartouche delimiters when using the JSON Cartouche with non-trivial data structures:

```
lemma ‹ example01 == JSON ‹{"menu": {
                            "id": "file",
                            "value": "File",
                            "popup": {
                              "menuitem": [
                               {"value": "New", "onclick": "CreateNewDoc()"},
                               {"value": "Open", "onclick": "OpenDoc()"},
                               {"value": "Close", "onclick": "CloseDoc()"}
                              ]
                            }},
                           "flag": true,
                           "number": 42
                           }››
  by(simp add: example01_def)
```

We can define new JSON data “inline”, using the Isar keyword **JSON**:

```
JSON ‹
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}, "flag":true, "number":42}
› defining example04
```

Moreover, we can define new JSON data by reading it from a file, using the Isar keyword **JSON_file**:

```
JSON_file "example.json" defining example03

thm example03_def example04_def

lemma "example03 = example04"
  by (simp add:example03_def example04_def)
```

2.1.3 Serializing Nano JSON

In this section, we define the necessary infrastructure to serialize (export) data from HOL using a JSON-like data structure that other JSON tools should be able to import.

ML Implementation

ML ‹
```
signature NANO_JSON_SERIALIZER = sig
    val serialize_json: Nano_Json_Type.json -> string
    val serialize_json_pretty: Nano_Json_Type.json -> string
    val serialize_term: term -> string
    val serialize_term_pretty: term -> string
end
```
›

ML_file "Nano_JSON_Serializer.ML"

The file `Nano_JSON_Serializer.ML` provides the Isabelle/ML structure `Nano_Json_Serializer:NANO_JSON_SERIALIZER`. It provides functions for serializing JSON data from it abstract syntax as well as from its HOL term representations. Moreover, variants are provided that try to pretty print the output with the goal of making it easier to read for humans.

Isar Setup

ML ‹
```
structure Nano_Json_Serialize_Isar = struct
  fun export_json thy json_const filename =
    let
        val term = Thm.concl_of (Global_Theory.get_thm thy (json_const^"_def"))
         fun export binding content thy =
             let
               val thy' = thy |> Generated_Files.add_files (binding, content);
               val _ = Export.export thy' binding (Bytes.contents_blob content)
             in thy' end;
        val json_term = case term of
                        Const (@{const_name "Pure.eq"}, _) $ _ $ json_term => json_term
                      | _ $ (_ $ json_term) => json_term
                      | _ => error ("Term structure not supported.")
        val json_string  = Nano_Json_Serializer.serialize_term_pretty json_term
        val json_bytes = (XML.add_content (YXML.parse json_string) Buffer.empty)
                           |> Bytes.buffer
    in
        case filename of
             SOME filename => let
                                val filename = Path.explode (filename^".json")
                                val thy' = export (Path.binding
                                                    (Path.append (Path.explode "json")
                                                       filename,Position.none))
                                                    json_bytes thy
                                val _ =  writeln (Export.message thy (Path.basic "json"))
                              in
                                 thy'
                              end
           | NONE =>  (tracing json_string; thy)
    end
end
```
›

```
ML‹
  Outer_Syntax.command ("JSON_export", Position.none)
  "export JSON data to an external file"
  ((Parse.name -- Scan.option (keyword ‹file›-- Parse.name))
   >> (fn (const_name,filename) =>
        (Toplevel.theory (fn state => Nano_Json_Serialize_Isar.export_json state
                                          const_name (Option.map snd filename)))));
›
```

Examples

We can now serialize JSON and print the result in the output window of Isabelle/HOL:

```
JSON_export example01
thm example01_def
```

Alternatively, we can export the serialized JSON into a file:

```
JSON_export example01 file example01
```

2.1.4 Putting Everything Together

For convenience, we provide an ML structure that provides access to both the parser and the serializer:

```
ML‹
structure Nano_Json = struct
    open Nano_Json_Type
    open Nano_Json_Parser
    open Nano_Json_Serializer
end
›

end
```

2.2 Query Infrastructure (Nano_JSON_Query)

```
theory
  Nano_JSON_Query
imports
  Nano_JSON
begin
```

In this theory, we define various functions for working with JSON data, i.e., the data types defined in the theory Nano_JSON.Nano_JSON. These query functions are useful for building more complex functionality of JSON encoded data. One could think of them as something like jq (https://stedolan.github.io/jq/) for Isabelle.

Isabelle/ML

```
ML‹
signature NANO_JSON_QUERY = sig
    val nj_filter:
```

```
        string -> Nano_Json_Type.json
                   -> (string list * Nano_Json_Type.json) list
    val nj_filterp:
        string list -> Nano_Json_Type.json
                     -> (string list * Nano_Json_Type.json) list
    val nj_filter_obj:
        (string * Nano_Json_Type.json option) -> Nano_Json_Type.json
                     -> (string list * Nano_Json_Type.json) list
    val nj_filterp_obj:
        (string list * Nano_Json_Type.json option) -> Nano_Json_Type.json
                     -> (string list * Nano_Json_Type.json) list
    val nj_first_value_of:
        string -> Nano_Json_Type.json
                     -> Nano_Json_Type.json option
    val nj_first_value_ofp:
        string list -> Nano_Json_Type.json
                     -> Nano_Json_Type.json option
    val nj_update:
        (Nano_Json_Type.json -> Nano_Json_Type.json) -> string -> Nano_Json_Type.json
                     -> Nano_Json_Type.json
    val nj_updatep:
        (Nano_Json_Type.json -> Nano_Json_Type.json) -> string list -> Nano_Json_Type.json

                     -> Nano_Json_Type.json
    val nj_convert:
        (Nano_Json_Type.json -> 'a) -> string -> Nano_Json_Type.json
                     -> 'a list
    val nj_string_of:
        Nano_Json_Type.json
                     -> string option
    val nj_string_of':
        string -> Nano_Json_Type.json -> string

    val nj_integer_of:
        Nano_Json_Type.json -> int option
    val nj_integer_of':
        int -> Nano_Json_Type.json -> int
    val nj_real_of:
        Nano_Json_Type.json -> IEEEReal.decimal_approx option
    val nj_real_of':
        IEEEReal.decimal_approx -> Nano_Json_Type.json -> IEEEReal.decimal_approx
    val nj_bool_of:
        Nano_Json_Type.json -> bool option
    val nj_bool_of':
        bool -> Nano_Json_Type.json -> bool
  end
```

›

ML_file Nano_JSON_Query.ML

The file Nano_JSON_Query.ML provides the Isabelle/ML structure Nano_Json_Query:NANO_JSON_QUERY.

Isabelle/HOL

```
fun nj_filter':: ‹'a ⇒ 'a list × ('a, 'b) json ⇒ ('a list × ('a, 'b) json) list›
  where
    ‹nj_filter' key (prfx, OBJECT json) = ((map (λ (k,v). (prfx@[k],v)) (filter (λ (k,_). key
= k) json) )
                        @ (List.concat (map (nj_filter' key) (map (λ (k,v). (prfx,v)) json)))
                         )›
  | ‹nj_filter' key (prfx, ARRAY json) = (List.concat (map (nj_filter' key) (map (λ v. (prfx,v))
json)))›
  | ‹nj_filter' _ (_, NUMBER _) = []›
  | ‹nj_filter' _ (_, STRING _) = []›
  | ‹nj_filter' _ (_, BOOL _) = []›
  | ‹nj_filter' _ (_, NULL) = []›

definition nj_filter::‹'a ⇒ ('a, 'b) json ⇒ ('a list × ('a, 'b) json) list› where
          ‹nj_filter key json = nj_filter' key ([],json)›

fun nj_update :: ‹(('a, 'b) json ⇒ ('a, 'b) json) ⇒ 'a ⇒ ('a, 'b) json ⇒ ('a, 'b) json›
  where
    ‹nj_update f key (OBJECT kjson) =  OBJECT (map (λ (k,json). if k = key
                                                             then (k, f json)
                                                             else (k, nj_update f key json))
kjson)›
  | ‹nj_update f key (ARRAY json) = ARRAY (map (nj_update f key) json)›
  | ‹nj_update _ _ (NUMBER n) = NUMBER n›
  | ‹nj_update _ _ (STRING s) = STRING s›
  | ‹nj_update _ _ (BOOL b) = BOOL b›
  | ‹nj_update _ _ NULL = NULL›
```

Examples. The following illustrates a simple example of the `nj_filter` function.

```
declare [[JSON_string_type=String.literal]]

JSON   ‹
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}, "flag":true, "number":42}
› defining example_literal_literal

value ‹nj_filter (STR ''onclick'') example_literal_literal›

end
```

2.3 The Main Theory of Nano JSON (Nano_JSON_Main)

theory
  Nano_JSON_Main
imports
  Nano_JSON_Query
begin

This is the main entry point into the Nano_JSON session for users of this AFP entry.

end

3 User's Guide and Examples

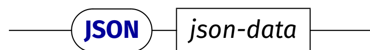
3.1 The Nano JSON Manual (Nano_JSON_Manual)

```
theory
  Nano_JSON_Manual
imports
  Nano_JSON_Query
begin
```

3.1.1 Isabelle/HOL

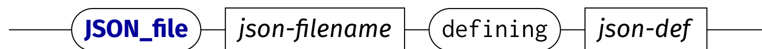
On the Isabelle/HOL level, we provide three new commands for working with JSON formatted data.

JSON. For defining JSON data within HOL terms, we provide the following command:



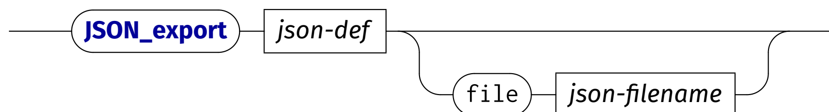
where *json-data* is a cartouche containing the JSON data.

JSON_file. For reading JSON data from an external file, we provide the command



where *json-filename* is the path (file name) of the JSON file that should be read and *json-def* is the name the HOL definition representing the JSON data is bound to.

JSON_export. For serializing (exporting) JSON encoded data, we provide the command



where *json-def* is the definition of the JSON data that should be exported. Optionally a file name (*json-filename*) can be provided. If a file name is provided, it is used for storing the serialized data in Isabelle's virtual file system. If no file name is provided, the serialized data is printed in Isabelle's output window.

Configuration. We provide three configuration attributes:

- The attribute `JSON_num_type` (default `int` allows for configuring the HOL-type used representing JSON numerals.
- The attribute `JSON_string_type` (default `string`) allows for configuring the HOL-type used representing JSON string.
- The attribute `JSON_verbose` (default: false) allows for enabling warnings during the JSON processing.

3.1.2 Isabelle/ML

We refer users who want to use or extend Nano JSON on the Isabelle/ML level to the ML signatures shown in the implementation chapter, i.e., chapter 2.

end

3.2 Examples (Example)

```
theory
  Example
imports
  Nano_JSON_Main
begin
```

In this theory, we illustrate various small examples of importing or exporting of JSON data from Isabelle/HOL. The examples in this theory do not use the type `real` to avoid the need to depend on the theory "Complex_Main".

```
JSON ‹
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}, "flag":true, "number": -42}
› defining example_default
thm example_default_def
JSON_export example_default
JSON_export example_default file example_default

declare [[JSON_string_type=string, JSON_num_type = nat]]
JSON ‹
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
```

```
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}, "flag":true, "number": 42}
› defining example_string_nat
thm example_string_nat_def
JSON_export example_string_nat
JSON_export example_string_nat file example_string_nat

declare [[JSON_string_type=string, JSON_num_type = int]]
JSON ‹
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}, "flag":true, "number":42}
› defining example_string_int

thm example_string_int_def
JSON_export example_string_int
JSON_export example_string_int file example_string_int

declare [[JSON_string_type=String.literal, JSON_num_type = int]]
JSON ‹
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}, "flag":true, "number":42}
› defining example_literal_int
thm example_literal_int_def
JSON_export example_literal_int
JSON_export example_literal_int file example_literal_int


declare [[JSON_string_type=string, JSON_num_type = string]]
JSON ‹
{"menu": {
```

```
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}, "flag":true, "number":-42}
› defining example_string_string
thm example_string_string_def
JSON_export example_string_string
JSON_export example_string_string file example_string_string

declare [[JSON_string_type=String.literal, JSON_num_type = String.literal]]
JSON ‹
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}, "flag":true, "number":42}
› defining example_literal_literal
thm example_literal_literal_def
JSON_export example_literal_literal
JSON_export example_literal_literal file example_literal_literal
```

xxxx

Using the top level Isar commands defined in the last section, we can now easily define JSON-like data:

```
JSON ‹
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}, "flag":true, "number":42}
› defining example02


thm example02_def
```

```
declare [[JSON_string_type = String.literal]]
JSON ‹
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}, "flag":true, "number":42}
› defining example02'

thm example02'_def


ML‹
@{term ‹JSON‹{"number":31}››}
›
```

Moreover, we can import JSON from external files:

```
lemma "example01 = example03"
  by(simp add: example01_def example03_def)


end
```

3.3 Examples Real (Example_Real)

```
theory
  Example_Real
imports
  Nano_JSON_Main
  Complex_Main
begin
```

In this theory, we illustrate various small examples of importing or exporting of JSON data from Isabelle/HOL. The examples in this theory make use `real`. This is possible, as this theory imports the theory `Complex_Main`.

```
declare [[JSON_num_type = real]]

JSON ‹
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
```

```
  }
}, "flag":true, "number":-42.8}
› defining example_default_real

thm example_default_real_def

JSON_export example_default_real
JSON_export example_default_real file example_default_real

declare [[JSON_string_type=string, JSON_num_type = int]]
JSON ‹
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}, "flag":true, "number":42}
› defining example_string_int
thm example_string_int_def
JSON_export example_string_int
JSON_export example_string_int file example_string_int

declare [[JSON_string_type=String.literal, JSON_num_type = int]]
JSON ‹
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}, "flag":true, "number":42}
› defining example_literal_int

thm example_literal_int_def
JSON_export example_literal_int
JSON_export example_literal_int file example_literal_int

declare [[JSON_string_type=string, JSON_num_type = real]]
JSON ‹
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
```

```
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}, "flag":true, "number": 42.8}
› defining example_string_real
thm example_string_real_def
JSON_export example_string_real
JSON_export example_string_real file example_string_real

declare [[JSON_string_type=String.literal, JSON_num_type = real]]
JSON ‹
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}, "flag":true, "number":42.8}
› defining example_literal_real
thm example_literal_real_def
JSON_export example_literal_real
JSON_export example_literal_real file example_literal_real


declare [[JSON_string_type=string, JSON_num_type = string]]
JSON ‹
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}, "flag":true, "number":-42.5}
› defining example_string_string
thm example_string_string_def
JSON_export example_string_string
JSON_export example_string_string file example_literal_string

declare [[JSON_string_type=String.literal, JSON_num_type = String.literal]]
JSON‹
{"menu": {
  "id": "file",
  "value": "File",
```

```
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}, "flag":true, "number":-42.5}
› defining example_literal_literal
thm example_literal_literal_def
JSON_export example_literal_literal
JSON_export example_literal_literal file example_literal_literal

end
```

Bibliography

[1] T. Bray, editor. The JavaScript object notation (JSON) data interchange format. Online: https://datatracker.ietf.org/doc/html/rfc8259. Dec. 2017.

[2] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (xml) 1.0 (fifth edition). W3C Recommendation, 2008. Available at http://www.w3.org/TR/REC-xml/.

[3] A. D. Brucker and M. Herzberg. The Core DOM. *Archive of Formal Proofs*, Dec. 26, 2018. ISSN: 2150-914x. URL: http://www.brucker.ch/bibliography/abstract/brucker.ea-afp-core-dom-2018. http://www.isa-afp.org/entries/Core_DOM.html, Formal proof development.

[4] A. D. Brucker and B. Wolff. An extensible encoding of object-oriented data models in HOL. USenglish. *Journal of Automated Reasoning*, 41:219–249, 3, 2008. ISSN: 0168-7433. DOI: 10.1007/s10817-008-9108-3. URL: http://www.brucker.ch/bibliography/abstract/brucker.ea-extensible-2008-b.

[5] ECMA-404:The JSON data interchange syntax. Online: https://www.ecma-international.org/publications-and-standards/standards/ecma-404/. Dec. 2017.

[6] C. Sternagel and R. Thiemann. XML. *Archive of Formal Proofs*, Oct. 2014. ISSN: 2150-914x. http://isa-afp.org/entries/XML.shtml, Formal proof development.

[7] M. M. Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, TU München, München, Feb. 2002. URL: http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/wenzel.html.