Serdica J. Computing **7** (2013), No 1, 35–72

Serdica
Journal of Computing

Bulgarian Academy of Sciences
Institute of Mathematics and Informatics

JSONYA/FN: FUNCTIONAL COMPUTATION IN JSON

Miloslav Sredkov

Abstract. Functional programming has a lot to offer to the developers of global Internet-centric applications, but is often applicable only to a small part of the system or requires major architectural changes. The data model used for functional computation is often simply considered a consequence of the chosen programming style, although inappropriate choice of such model can make integration with imperative parts much harder. In this paper we do the opposite: we start from a data model based on JSON and then derive the functional approach from it. We outline the identified principles and present Jsonya/fn — a low-level functional language that is defined in and operates with the selected data model. We use several Jsonya/fn implementations and the architecture of a recently developed application to show that our approach can improve interoperability and can achieve additional reuse of representations and operations at relatively low cost.

1. Introduction. The growing need for global Internet-centric applications pushes software engineering tools and techniques to their limits and drives

ACM Computing Classification System (1998): D.3.2, D.3.4.
Key words: JSON, data models, functional programming, intermediate languages, interoperabilityy.

the adoption of radically different approaches and technologies. While the utilisation of functional programming in commercial software development used to be small, these recent trends opened gaps which are more and more often filled by functional languages or by techniques originating from the functional world. The application of functional programming in the context of such large systems is without any doubt interesting, but we focus on a narrower, but still fundamental question: what data representation model should be used for functional computation in the context of global Internet applications?

Even though essential, this question is often neglected, as its answer is considered a mere implication of the data structures or the type system of the chosen programming language. Doing so, however, can hinder integration with imperative technologies, as the data models used by functional languages are usually fundamentally different than those of the imperative ones. As a result, unless the whole system is built with functional tools, the borders between the imperative and functional parts will most likely be problematic.

In contrast, we directly attack the data model problem as we consider it crucial for the complexity of the resulting system. We analyse whether a data model based on JSON [8], a data-interchange language with fast growing popularity, can provide smoother integration compared to the alternatives. We show that this solution provides various valuable properties and can reduce the discrepancy in complex applications at a reasonable implementation cost.

These global apps often need to spawn multiple platforms, to use a variety of communication channels and synchronisation methods to support distributed computation or to work in disconnected (offline) mode. At the same time, most programmers are not skilled enough in functional programming, so any attempt to answer these challenges in a functional way is threatened by the possibility to become obsolete before gaining sufficient community. Because of this, successful initiatives to bring technologies from the functional world to the industry are usually limited to a particular aspect of the system.

While the focus of such solutions makes them efficient for the specific problem, they still need to be combined with imperative technologies. The latter are usually general-purpose programming languages, which are capable of, although less efficiently, solving the problem in question on their own. This can cause the benefits of the functional approach to diminish compared to the overhead of the need to learn, integrate, maintain or distribute yet another tool.

Some tools, however, emerge to solve a wider problem such as scalability. Many even go further, aiming to make all parts of the Internet applications programmable in a single functional language. Unfortunately, such ambitious

technologies suffer from other problems. First, these instruments usually require massive implementation effort to be built and maintained, but more importantly, unless they provide special interoperability features, which can often pollute their otherwise clean languages, they become all-or-nothing solutions.

In heterogeneous systems, a particular source of discrepancy is the data representation model used by the different programming languages or technologies. Complex software applications already suffer enough from the ways standard tools represent data — the same information is often stored in multiple representations such as XML documents, objects in an object-oriented language, and records in a relational database. The conversions between these forms may consume significant development effort.

Such a situation will certainly not improve if a part of the system is implemented in a functional language which uses e.g. S-expressions — unless the part is relatively isolated, mapping between S-expression and some of the other representations will be required. This observation led us to the following idea: if the functional tool to be integrated should not introduce additional discrepancy, then it must use one of the data representation models already supported by the system.

Because JSON is convincingly gaining momentum as a widely used data-interchange format, if it proves suitable for functional computation, then it could provide foundations for functional tools that integrate more smoothly.

To explore the suitability of JSON, we first identified the general properties of a potential solution and incorporated them into the approach named Jsonya. It defines how functional computation can be applied in a heterogeneous system, includes a suitable JSON-based data model (published in another paper [41]), and outlines the required properties of the functional computation itself. Following the approach we defined Jsonya/fn, a simple, low-level, homoiconic (i.e. both represented in and operating with the same data model) pure functional programming language. The most challenging part was finding a simple yet powerful evaluation algorithm, which provides maximum benefits at low implementation cost. To test the feasibility of our approach, we created several different implementations in Java and JavaScript, and used the architecture of a recently developed large global application to explore what effect our approach and language would have on the development of similar systems.

The experimental implementations showed that Jsonya/fn is relatively easy to implement in a way that features homoiconicity, automatic order of initialisation, and automatic structural and computational memoisation. Some of the important features are that computations are consistent regardless of the place

of execution, and that homoiconicity also applies to environments and closures, which are represented as regular Jsonya values. Such properties may be useful in various contexts in global applications, including distribution of computation. While the lack of high-level languages translating to Jsonya/fn currently prevents Jsonya from being applied in industrial setting, our results suggest that it has good prospects for achieving this, and more importantly, that further exploration of JSON-based functional computation is likely to be a good investment.

To the best of our knowledge this is the first systematic attempt to explore JSON as a foundation for functional computation. The main contributions of this paper are the following:

- Exploration of some of the notable attempts to apply functional programming in Internet applications and the data models they use (Section 2).
- Analysis of how JSON can be used as a basis for functional computation resulting in the Jsonya approach, which identifies the application areas and the properties of the data model and the computations (Section 3).
- The homoiconic JSON-based low-level functional language Jsonya/fn, which features automatic memoisation of computations and structures, closures and environments with self-sufficient representations as regular values, and automatic order of computations (Section 4).
- Assessment of the feasibility of fully implementing Jsonya by adding a high-level language translating to Jsonya/fn based on multiple implementations and analysis of a real-world application (Section 5).

2. Related Data Models. In this section we look at various data models used to apply functional programming in the context of global applications. By *data model*, also often called *information model* or *metamodel*, we refer to the means provided by the environment, e.g. a programming language, to represent or model information. This terminology has been defined more thoroughly in the context of Model-driven engineering [16], but for the needs of our paper the informal definition will suffice.

Data models are often neglected in favour of other aspects, such as syntax and processing details, and considered to follow intuitively from the latter. When certain ambiguities occur, e.g. whether the order in which the fields of a Java class are declared is important, it is often easy to resolve them by establishing a convention. In a wider context, however, such negligence can have a more negative effect. For example XML, for which data interchange is one of the main goals,

is agreed upon only at the syntax level and has multiple different interpretations (metamodels) [45] in use.

Functional programming approaches have always been attempted against the challenges of the industrial software development. The difficulties with global applications, however, led to some significant advances in the application of functional instruments. Because of this, functional programming has gained a reputation for being able to solve certain problems better than the traditional approaches.

To give a background, we identified some attempts to provide the means for one or more of the following:

- server-side request handling for the generation of dynamic web pages or web service responses;
- code reuse between parts of the system of different nature;
- simpler communication between system parts, e.g. without the object-relational impedance mismatch problem;
- scalability, e.g. via distributed computing.

The list of approaches and attempts is far from complete but we believe it gives a wide enough overview of the used data models.

2.1. Algebraic Data Types. Probably the majority of the contemporary functional tools use data models based on the algebraic data types. In them, a set of primitive data types is provided together with means to define new types in terms of constructors wrapping zero or more arguments. This approach allows languages to provide many useful features, including pattern matching, static type checking, and Hindley-Milner [22, 34, 9] type inference.

Dynamic Web Page Generation. Because a large part of server-side programming consists of the generation of HTML or XML-based content, modelling them as algrabraic data types is important. As "there is a natural fit between the XML document domain and Haskell tree datatypes" [44], multiple researchers have provided solutions for HTML and XML modelling successfully utilising static checking to guarantee some form of validity [25, 43]. Similar results have been achieved for other languages too, including Standad ML [13] and Curry [19]. Instruments which provide higher-level abstractions for HTML interface have also been presented [20, 39].

To further facilitate server-side programming, various utilities can be added; Haskell is an example of a functional language with a remarkably rich web

programming toolkit[1]. Such instruments can vary from simple libraries helping with CGI interaction to complete frameworks featuring integrated web servers, data persistence, templating and more. Haskell's type system has influenced their design in the same way it has influenced any other software library; while these tools exhibit some very interesting properties, their relevance to Haskell's data model is beyond the scope of this paper.

Extending to Multiple Layers. Other tools, however, attempt to go beyond the processing of form input and the generation of HTML. No matter how elegantly the information is modelled in the server-side, all this beauty evaporates when it has to be encoded and sent to a non-functional layer of the system, such as the database or a JavaScript client. The information needs to be serialised in a form supported by some protocol (e.g. XML), and neither its representation, nor the surrounding operations can be reused.

To resolve this, various approaches allow the same functional representation or manipulations to be used in more than one layer of the system. Happstack[2] and Yesod[3] are two notable frameworks extending beyond the middle layer. Both allow the same types to be reused in the database, define convenient means to process requests and generate responses, and provide some support in the generation of JavaScript code.

None of the above approaches, however, attempts to treat JavaScript above the syntactical level. In contrast, ML5 by Murphy et al. [36] is a language based on Standard ML which compiles to both bytecode for the server and JavaScript for the client. Its type system associates each value with a particular world, and its runtime system handles marshalling and unmarshalling between the client and the server worlds. Hanus [21] also explores JavaScript generation from a functional language, allowing constraints specified as Curry functions to be used for both client- and server-side validation. WebSharper[4] also generates JavaScript from a language with algebraic data types in an attempt to provide a web framework allowing the creation of both client front ends and server-side backends using only F#.

Merging All Layers. A more radical way to completely eliminate the impedance mismatch problem is to unify all layers into a single language. One such approach is Links [6]. The language compiles to JavaScript for the client and to SQL for the database. It is statically checked; a notable feature

[1] `http://www.haskell.org/haskellwiki/Web/Frameworks`
[2] `http://happstack.com/`
[3] `http://www.yesodweb.com/`
[4] `http://websharper.com/home`

is the integration of XML both as a primitive type and at the syntax level.

Ur/Web, an extension on top of Ur [4], also attempts to provide an all-layer solution, but the focus is more on metaprogramming and practical use of dependent types. It provides means for construction of queries and transactions in the context of an SQL database. For the client-side it compiles to JavaScript and features page generation in a functional reactive programming style. A syntax for XML literals is included in the language. The type system is based on ML, but a lot more is added on top of it.

Analysis. The solutions based on algebraic data types we reviewed solve the four problems we defined earlier to a different extent. Their data models are suitable for processing web requests and building dynamic (e.g. HTML) responses. Most tools have a means to facilitate the transfer of information to other parts of the system but reuse of code and representation is achieved by taking over the whole layer. With the exception of ML5, the listed solutions do not explicitly address scalability.

2.2. Based on S-expressions. Originating from Lisp, S-expressions are remarkably powerful for their simplicity. Most often they are used to represent lists written like `(a (b c) d)` and are built on top of only two constructs: atoms and ordered pairs. Lisp-based languages use them for both source code and data structure representation. Because they can be parsed easily and are much shorter than alternatives like XML, S-expressions have often been considered suitable for a data-interchange format. One notable example is the SXML format by Kiselyov [28], which represents the XML Infoset in the form of S-expressions. Tools for it, similar to their XML counterparts, such as SXSLT [29] are also present.

Being able to use the same structure for both source code and in-memory operations as well as data interchange and persistence opens interesting possibilities in the context of Internet-centric applications. The PLT Scheme server [30] (now Racket) uses continuations to allow web page navigation flow to be defined in the same way as in a regular interactive console application. Not surprisingly, the HTML response is elegantly modelled with S-expressions. For validation Nørmark [37] provides a robust solution by synthesising mirror functions for each tag based on a DTD.

Clojure[5] is a Lisp-based language compiling to Java Virtual Machine bytecode, with strong focus on multi-threading. Its S-expressions are enhanced with additional constructs to also support vectors and maps. Compilers to other tar-

[5] `http://clojure.org/`

gets such as the .NET CLR[6] and JavaScript [32] are available, thus allowing the same functional computations to be applied to different system parts, including browser-based clients.

In this category, it is also necessary to mention Curl [24]. It aims to be a single language covering all aspects of web application development, and in its recent commercially developed versions supports programming for the server side, web, desktop and mobile clients. The language is intended to be easy to use for everything ranging from text formatting to complex business logic. The language is heavily influenced by Lisp, but has deviated from it to the point where it is more object-oriented than functional. It is homoiconic and highly extensible via its powerful macro facility.

The final example in this category is Hop [40], a dialect of Lisp for the programming of interactive web applications. It allows both the client- and the server-side, as well as client-server communication to be defined with the same, S-expression-based syntax. For certain application types, such as interactive pages that do not need database storage, all system aspects are covered. HTML is defined as s-expressions, and JavaScript is generated from Hop as needed. Interestingly, Hop actually uses JSON for the communication between the JavaScript and the server.

The presented examples show that the power and elegance of S-expressions makes them a suitable data model for wide range of applications; yet, with few exceptions, their popularity in mainstream Internet programming is limited. Whether the programmers are scared of the brackets or there is some other reason for this is beyond our competence; we can only conclude that being simple and powerful does not guarantee wide acceptance.

2.3. XML-based. XML is the most popular format for data interchange, with a remarkably rich set of supporting technologies. It can be parsed and generated from a wide variety of languages, and is used as the basis of numerous file formats and protocols.

XML (or more precisely, some of its data models) is used as a basis for functional computations in two ways. The fist one is to define a programming language in XML-based syntax, thus achieving a homoiconic language which is both written in, and operating with XML. XSLT 1.0 [5] and XSLT 2.0 [27] are the most popular such languages. Focused on transforming XML documents they are often incorporated in larger systems for the generation of XHTML web pages or server side transformations. XSLT 1.0 is implemented by a variety of XML toolsets, including all popular web browsers.

[6] `https://github.com/richhickey/clojure-clr`

The syntax of XSLT is however, not entirely defined by XML constructs. XSLT incorporates XPath expressions, which are simply included as strings. A complete XML-only implementation of these expression would dramatically increase the size of the programs.

Because of this, the majority of the functional tools dedicated to XML manipulation use custom syntax. There is a whole family of typed languages which use XML as values and DTD or other schema technology as types. Examples of this family are XMλ [33], XDuce [23], CDuce [2], and OCamlDuce [15]. The W3C recommendation XQuery [3] also uses custom non-XML syntax to express computations, but is dynamically typed and targets to be a query language implemented in various environments.

Another interesting example is MashMaker [14], which allows the creation of web mashups via a functional language that can be textually or graphically edited. The language has some specific features such as storing data and code in a single tree, and allowing extensions to that tree to be defined. The data representation however is not exactly XML, but a slightly modified data model based on it.

XML is a very powerful basis for a data model and can be used to represent various kinds of information. It has, however, two limitations: its verbosity for certain applications, such as expressing computations, and its dissimilarity to the data models of most programming languages, which introduce overhead and make it to be perceived as too heavy.

2.4. Other. Many authors, including the creator of JavaScript [11], consider JavaScript to be a functional programming language. For that reason, frameworks encouraging functional programming techniques in JavaScript are not uncommon. In a similar fashion the document-oriented databases CouchDB[7] and MongoDB[8], whose data models are based on JSON, use JavaScript for the definition of Map/Reduce views. In both databases JavaScript functions are not decomposed in any way, but simply represented as strings containing their source code.

Other notable examples are the dynamically typed languages Erlang and Oz. The first enjoys popularity because its Actor Model approach to multithreading makes it suitable for certain scalable application. QHTML [12] is an attempt to take advantage of the multiple programming paradigms supported by the second, allowing the programmer to define both the server side and the client side in Oz, treating HTML documents like traditional GUI elements.

[7] `http://couchdb.apache.org/`

[8] `http://www.mongodb.org/`

A seemingly more developed solution is Opa, a platform allowing web applications to be written in a single functional language advertised as concise, secure, scalable, easy-to-deploy and open source[9]. It allows client, server and database code to all be written in a type-safe manner. Instead of constructive values the language provides list, record and sum types with convenient manipulation and pattern matching syntax. Additional valuable features are syntax for HTML generation and support for distribution of server instances.

All of these approaches attempt to solve various problems for the development of Internet applications. None of them, however, allows for gradual application of functional computation in a large system. Either the whole system is attacked at once, which makes taking advantage of existing technologies harder, or the solution is focused on a specific part of the system.

3. Jsonya. To enable the gradual integration of functional technologies into the development of global applications, the technologies being introduced need to be aligned with the current industrial trends. Because of its popularity, functional programming on top of a JSON is likely to interoperate more easily with other technologies, even if they use a less interchangeable data model. The other data-interchange-friendly options, XML and S-expressions, compared to JSON, stand at the extremes: although XML is still dominant it is much more complex than JSON, which is advertised as "The Fat-Free Alternative to XML"[10], while S-expressions, although much simpler, remain unknown to most of the developers outside the functional world.

In this section we analyse JSON as a basis for functional computations and gradually define our approach by identifying the principles that follow from the the use of JSON itself in the context of the global applications.

3.1. Motivation and Approach. To better illustrate the ideas behind the approach, we will describe the main utilisation that motivated it. In large heterogeneous software systems, any two communicating software items implemented in sufficiently different technologies (e.g. different programming languages that are not link-compatible or easily interoperable by other means) usually suffer from some of the following:

- the same problem domain information has to be modelled in both software items;

[9] `http://opalang.org/`

[10] `http://www.json.org/fatfree.html`

- routines which map between the model and a (third) transferable representations are needed;
- if the same operations for the modelled information are needed by both items, then either a remote invocation mechanism must be used, or the operations must be implemented at both places.

The above problems are addressed by the many contemporary instruments targeting inter-process communication, mapping between data representations, translation and generation of source code, etc. Most of these solutions, however, are not fully automatic, are available only for a limited number of programming languages, and often introduce additional configuration overhead.

To overcome the mentioned problems, we propose the following approach:

- A (transferable) JSON representation of the information used by both items is defined.
- The computations needed by both software items are defined in a high-level language and translated to an intermediate pure functional language on top of JSON.
- The software items do not build representations of the same problem domain information; instead they directly use the transferable representation from a generic JSON object model, or if feasible, from tiny wrappers on top of it.
- The software items use the functions defined in the intermediate functional language via a tiny embedded interpreter instead of reimplementing them.
- Communication between items is done by sending and receiving JSON which encodes information and/or functions.

This approach, is related to the Common Technical Representation approach, which we published earlier [42]. CTR advocates focusing on textual representations which are compatible to various data models, widely applicable, simple and declarative. The current paper may be seen as a CTR instance using JSON as a representation format and focused on functional programming.

3.2. Data Model. Before focusing on the intermediate language we need to resolve one problem standing in our way to it — the lack of an established data model for JSON. Like XML, JSON is defined only at syntax level, but unlike XML, which suffers from having multiple data models [45], we were not aware of any popular programming-language-independent JSON data model. For computations to behave consistently regardless of the host technology, in a separate paper, we presented Jsonya/dm [41] — a strictly defined data model for JSON.

In this section we are going to summarise it briefly with regard to the needs of this article.

The definition of JSON states 7 kinds of *values*: *object*, *array*, *number*, *string*, *false*, *null* and *true* (alternatively *true* and *false* are assumed to be the two values of the primitive type *boolean*). *Objects* are collections of *name–value* pairs, where each name is a string; *arrays* are ordered lists of values; *numbers* as literals of decimal digits having integer and optionally exponential and fractional parts; *strings* are defined as quoted literals of unicode characters; *false*, *null* and *true* are defined by their literal names. Here is an example JSON code containing all kinds of values:

```
{
    "string": "Some text with a new line\n",
    "number": 18.33,
    "object": {"a": 1, "b": 2},
    "array": [0, 1, [], {}, "item"],
    "literals": [true, false, null]
}
```

The way JSON is defined seems straightforward; it does not appear that a formal description is necessary. A closer look, however, reveals several uncertainties — it is not always obvious which aspects of the representation are essential and which are not (e.g. the white space between the values). Here are some of the more significant ambiguities:

- **Object order**: is {`"a": 1, "b": 2`} distinguishable from {`"b": 2, "a": 1`}?
- **Empty collections**: are {} and `[]` distinguishable?
- **Number representation**: Are `30`, `30.0`, and `30.00`, or `0` and `-0` distinguishable? How large can the numbers be?
- **Empty values**: are `false`, `0`, `""` and `null` distinguishable?

To some readers the answers to these questions may seem obvious, and the need to ask them exaggerated. However, an analysis of some of the JSON parsing libraries linked from `http://json.org/` showed that for each of these questions, libraries whose implementors have assumed different answers can be found. Since such variations are not acceptable for our goal, we pick a stricter interpretation of JSON for the basis of our data model.

Elements. After parsing, all available information is represented in a hierarchy of elements of one of the seven kinds which directly correspond to the

seven kinds of values in JSON. This way the kinds are as close to the syntax as possible — they are not intended to serve as a complete type system and to associate semantics with the values (e.g. that *true* and *false* are *booleans*). Since a JSON file contains exactly one root value, which is either an object or an array, parsing yields exactly one element of the appropriate kind. These elements are the fundamental building blocks of information — they are not constructed using a smaller data primitive, at least not in an observable way.

Order of Object Members. Order of object members is inessential, e.g. {`"a": 1, "b": 2`} and {`"b": 2, "a": 1`} encode the same object. We pick this interpretation because of performance and complexity considerations — efficiently implementing a map without ordering is simpler. For persistent data structures, although not impossible to implement, we were not able to find suitable ordered associative array implementations providing $O(\log n)$ run-time complexity for searches and modifications.

This unordered interpretation actually threatens the desired computational consistency. For example the ECMAScript Standard states "The mechanics and order of enumerating the properties (...) is not specified" [10, p. 92], but the JavaScript implementations of some popular web browsers iterate the fields in the order in which they were assigned. Thus, a developer unaware that this behaviour is implementation specific may accidentally write unportable code.

To enforce consistent computations we need to either disallow ordered iteration of object members, or specify a concrete order of traversal. While the former seems interesting, it is too exotic for our needs: to process data, special (unordered) techniques may be needed: e.g. commutative functions and special versions of the fundamental higher-order functions.

As the majority of the efficient persistent associative array implementations are based on sorted structures, such as balanced binary trees or skip lists, we pick the lexicographical order of the names (keys) as the only way in which object members can be enumerated[11].

Numbers. Number elements represent values which can be written as finite decimal numbers. Two number elements are indistinguishable if and only if they encode the same number. Therefore `30`, `30.0` and `3e1` are indistinguishable from each other, and approximate or special values are not supported (e.g. $\frac{1}{3}$ or $+\infty$). There are no limits on the precision or range of the numbers, they

[11] Actually, we need to be even more specific as the way different languages compare strings is affected by the representation they use. Most notably, languages using UTF8 or UCS4 may give different results than languages using UTF16 for strings with character outside of the Basic Multilingual Plane.

can be as large as the memory constrains of the particular environment allow.

This rather radical choice was seen as the only way to ensure both consistent computation and practical applicability. Binary floating point numbers are very efficient when a hardware implementation is available, but are hard to reason about and different implementation may provide slightly different results [35]. On the other hand, for many practical applications, e.g. monetary values, decimal numbers are essential [7].

We do not impose any limit on the numbers; particular environments may fail to execute an operation if a number is too large for them, but should not silently approximate or truncate the result. When necessary software engineers can negotiate limits for some problem domain values in order to ensure that all modules are capable of processing them.

Other Choices. We do not allow objects to have multiple members with the same name and consider such JSON code illegal. The kind of each element is observable, so elements of different kinds are always distinguishable from one another, thus {}, `[]`, `""`, `0`, `null` and `false` are all distinct values.

Elements do not have observable identity, only values, i.e. there is no way to detect whether two elements are the same or not other than by their observable properties. This implies that all elements are immutable and forces computations to be pure.

Observable Properties. To summarise, elements have the following observable properties, and no other:

- for each element, its kind (one of *object*, *array*, *decimal*, *string*, *false*, *null* or *true*);
- for *object* elements, a lexicographically sorted list of the names (keys) of its members;
- for *object* elements, by given string, the value of the member with that name;
- for *array* elements, the number of items they have (their size);
- for *array* elements, by given zero-based index, the value of the item at that index;
- for *number* elements, the decimal number they represent;
- for *string* elements, the text (sequence of unicode characters) they represent.

It turns out, that these 7 properties provide all the necessary information for computations, and in fact can be used to design an appropriate set of primitive

selector functions. They also suggest that the building blocks for our functional computations are rather high-level compared to other functional languages.

3.3. Design Considerations. Before we define the language itself, we are going to outline some of its important characteristics. To more easily ensure computational consistency and make reasoning easier, all functions should be **pure**. To allow programs to be easily communicated through various communication channels, the language should be **homoiconic**, i.e. the primary representations of the programs should also be in Jsonya/dm.

Since JSON is structured as a tree without any support for references, sharing data would require some non-trivial high-level solution. In order to avoid complication, automatic **structural sharing** (memoisation) would be valuable. Going further in this direction, it is natural to think about **automatic memoisation** of computations too; although not so crucial as structural sharing, memoising computations may have a positive impact on performance [18].

The language, should be relatively **low level**, more like an intermediate language. Being defined in JSON, additional usability would require the use of micro languages, which would make automatic processing harder and would still not make it as elegant as a specially designed syntax (see the difference between XSLT 2.0 and XQuery [31]). However, because higher-level tools may not be available soon, the language should still be human-readable/writable.

The language, should be relatively **low level**, more like an intermediate language. Being defined in JSON, additional usability would require the use of micro languages, which would make automatic processing harder and will still not make it as elegant as a specially designed syntax (see the difference between XSLT 2.0 and XQuery [31]). However, because higher-level tools may not be available soon, the language should still be human-readable/writable.

The language should be closer to the **lazy evaluation** model than to the strict evaluation model. This decision is significant, as with strict evaluation, the object members must always be computed in a specified order; since the only allowed order is the lexicographical one and not the one in which the members are written, this behaviour will not be very intuitive or useful. However, in lazy programs errors may be delayed, and exposed only when specific parts of the results are accessed, thus causing potential consistency problems. To avoid such complications, lazy-like evaluation should be limited to the internal computations; invocations from the host language should either succeed by returning a complete JSON-based result, or fail.

Finally, to not complicate the language, all functions should take **exactly one input parameter** and return exactly one element as a result. This approach

does not limit program expression as multi-argument functions can be simulated by currying. It is followed by some of the most popular high-level functional languages such as Haskell and ML, which provide convenient syntax for currying instead of functions with more than one parameter. Some low-level functional languages also take the same approach in order to stay closer to formalisms such as lambda calculus and combinatory logic. Examples include Peyton Jones's FLIC [38], the partial evaluator of Gomard and Jones [17] and Joy and Axford's GCODE [26]. More importantly, instead of defining multi-argument functions we can just make the construction of array and object elements more convenient, and in future consider providing destructuring assignments, similar to the ones supported by some JavaScript implementations[12] [13].

4. Jsonya/fn. In this section we define Jsonya/fn, the language which conforms to the aforementioned approach. While we have defined it rather concretely and with particular applications in mind, it should be considered more as a proof of concept than as a product ready for industrial use as is.

4.1. Main Idea. Following the outlined approach we created a JSON-based, untyped language, which is relatively close to the untyped lambda calculus, but has very different variable binding rules as a consequence of the unorderedness of the objects. The following snippet shows the very simple increment function:

```
{
    "tag": "object",
    "fields": {
        "inc": {
            "tag": "function",
            "summary": "Computes x + 1.",
            "input": "x",
            "body": {
                "tag": "call",
                "function": {"tag": "get",
                    "path": ["sum"]},
                "parameter": {
                    "tag": "array",
                    "items": [
                        {"tag": "get", "path": ["x"]},
```

[12] http://wiki.ecmascript.org/doku.php?id=proposals:destructuring_assignment

[13] https://developer.mozilla.org/en-US/docs/JavaScript/New_in_JavaScript/1.7

```
                        {"tag": "quote", "value": 1}
                    ]
                }
            }
        },
        "inc7": {
            "tag": "call",
            "function": {"tag": "get",
                "path": ["inc"]},
            "parameter": {"tag": "quote", "value": 7}
        }
    }
}
```

This rather trivial example is actually an object definition. Assuming that the function `sum` is defined in the environment, its evaluation will result in an object like the following:

```
{
    "inc": {
        "tag": "closure",
        "summary": "Computes x + 1.",
        "input": "x",
        "body": {...},
        "environment": {...}
    },
    "inc7": 8
}
```

From this it should be visible that: Jsonya/fn is very verbose; a Jsonya/fn program actually is a Jsonya element defining another Jsonya element; functions and closures are regular Jsonya elements, not some kind of special values.

What is not directly shown is that:

- The `environment` field of the `increment` closure includes `sum`, `inc` and `inc7`, and is 68 JSON lines.
- `sum` is not a built-in function but a regular function which invokes a built-in; built-in functions are not values but intrinsic elements of the interpreter referred to by string identifiers.
- The order in which `inc` and `inc7` are listed is irrelevant; single or mutual recursions do not require special constructs.

4.2. Verbosity. The code of the increment function may seem too large even for a low-level language, so one may rightfully ask whether it would be usable at all. The two main concerns are that programs will be too large to be processed efficiently and nearly impossible to write and edit by hand. Fortunately, both issues can be addressed by a more appropriate representation and a translator between it and Jsonya/fn. Indeed, if we look carefully at the above program, we will notice that the most overhead in terms of size comes because of the object representation of names and values. If we substitute the objects with arrays and remove the comments, we may yield something like:

```
["object", {
    "inc": ["function", "x",
        ["call",
            ["get", ["sum"]],
            ["array",
                ["get", ["x"]],
                ["quote", 1]]]],
    "inc7": ["call",
        ["get", ["inc"]],
        ["quote", 7]]}]
```

which looks like LISP with some additional syntax noise. This program is much shorter, while all we did was to substitute the named constructs with positional ones. We can save even more if we introduce a shorthand to access variables like `"$x"` or if we use a custom syntax instead of JSON-based one. Going this way, however, we deviate from the original target, which is convenient programmatic processing.

Jsonya/fn represents programs in a way natural for an object-based in-memory representation. Because this may not suit all purposes, languages for the specific purposes should be designed when needed, while Jsonya/fn should be used as a central canonical representation, similarly to the way prescribed in the CTR approach [42].

4.3. Syntax. A Jsonya/fn program is usually stored in a JSON file whose root element defines an expression. Jsonya/fn expressions are defined by object *expression construct* elements which are built on top of other expressions or Jsonya/dm values. Each *expression construct* object has a field `tag` which determines its type. There are exactly 8 expression constructs:

- `"object"` — constructs an object. Contains a member object `fields`, which holds the expressions of the resulting members.

- `"array"` — constructs an array. Contains a member array `items`, which holds the expressions of the resulting members.
- `"quote"` — quotes a value. Contains a member field `value` holding the value to be returned verbatim.
- `"get"` — gets a value from the environment. Contains a member array of strings `path` holding the name of the variable followed by a sequence of built-in names to be applied to it; see Section 4.4.
- `"call"` — invokes a function with a given parameter. The fields `function` and `parameter` contain the appropriate expressions.
- `"internal"` — invokes an internal (built-in). Contains the member string `name` and the member expression `parameter`; see Section 4.4.
- `"if"` — an if-then-else expression. Contains a member expression `condition` which must evaluate to either *true* or *false*; based on it the result is the value of either the `then` or the `else` expression.
- `"function"` — a lambda expression. Contains a member string `input` holding the name of the input variable and an expression `body`; evaluates to a closure object; see Section 4.5.

An illustration of the syntax based on JSON is given in Figure **??**; however, note that Jsonya/fn is defined on the data model level, so defining it in terms of textual (JSON-based) syntax is inaccurate and given for illustrative purpose only[14]. Most notably, the order of fields is not relevant, and the objects may contain additional fields.

4.4. Built-ins. In order to keep the data model simple and not pollute it with special values, we decided that built-in functions should not be first-class values in it (or values at all). This does not limit the programmer from passing them indirectly — either by their name or, via a wrapper function which invokes them — which is more flexible for library implementations, as changing a function is easier than changing a built-in.

Consistency Among Nodes. The main benefit of not having built-ins as first class values is that this allows all the information, including the function definitions, the result of the execution and closures, to be easily interchangeable as JSON. To achieve full mobility and execution consistency, however, all parties must observe the following rules:

[14]Alternatively, we could have used a tool like JSON-schema (`http://json-schema.org/`) but such a definition would be less readable and still inexact

⟨*expression*⟩ ::= ⟨*object*⟩ | ⟨*array*⟩ | ⟨*quote*⟩ | ⟨*get*⟩ | ⟨*call*⟩ | ⟨*internal*⟩ | ⟨*if*⟩ | ⟨*function*⟩

⟨*object*⟩ ::= `{"tag": "object", "fields":` ⟨*object of expr.*⟩ `}`

⟨*array*⟩ ::= `{"tag": "array", "items":` ⟨*array of expr.*⟩ `}`

⟨*quote*⟩ ::= `{"tag": "quote", "value":` ⟨*any value*⟩ `}`

⟨*get*⟩ ::= `{tag": "get", "path":` ⟨*array of strings*⟩ `}`

⟨*call*⟩ ::= `{"tag": "call", "function":` ⟨*expression*⟩ `, parameter:` ⟨*expression*⟩ `}`

⟨*internal*⟩ ::= `{"tag": "internal", "name":` ⟨*string*⟩ `}`

⟨*if*⟩ ::= `{"tag": "if", "condition":` ⟨*expression*⟩ `, "then":` ⟨*expression*⟩`",`
`"else":` ⟨*expression*⟩ `}`

⟨*function*⟩ ::= `{"tag": "function", "input":` ⟨*string*⟩`, "body":` ⟨*expression*⟩ `}`

Fig. 1. Pseudo-grammar of Jsonya/fn

- All built-ins must behave as pure functions and be unambiguously specified without allowing result variation.
- For each built-in a name (identifier) highly-unlikely to collide globally must be picked. Web addresses or UUIDs can be used.
- Built-ins cannot change; if the built-in is to be modified, e.g. to fix a bug or to improve its design, then a new built-in with a different identifier must be introduced, and the old onemust be removed or documented as deprecated instead.

Because at this point Jsonya/fn deliberately does not provide means to check whether a built-in is present or not, if the above rules are observed, then all successful executions of a Jsonya/fn programs will produce the same result regardless of the place of execution.

Selectors. To take advantage of the JSON-based structure, the core language must provide means to both easily construct composite elements (i.e. *objects* and *arrays*), which is done by the appropriate expressions, and to access their sub-elements. The latter is achieved with a family of built-ins which we call *selectors*. They are designed in accordance with the observable properties identified in Section 3.2:

- `"@kind"` — gets the kind of the element. Returns a string with one of the values `"object"`, `"array"`, `"decimal"`, `"string"`, `"false"`, `"true"` or `"null"`.

- `"@keys"` — gets the names of the members of an object. Returns a sorted[15] array of strings.
- `"@size"` — gets the number of items of an array element. Returns a single number.
- `".`*`name`*`"` where *name* is a field name — gets the field with that *name* from the object. This is actually a family of built-ins; all names starting with '`.`' are reserved for this.
- `"#`*`index`*`"` where *index* is a decimal representation of a zero-based index of an array item — gets the item. Also a family of built-ins, names starting with '`#`' are reserved for this.

The reason to define the last two selectors as families of built-ins is to allow means for future static reasoning, without putting additional expression constructs. Host-language variants of the same selectors, with the addition of two for the value of a string or a number, are completely sufficient interface for accessing the information from the host technology.

4.5. Environments and Closures.

It is important to define the entities which play a crucial role in the interpretation. After evaluation, each `function` construct results in a `closure` with the following structure:

```
{
   "tag": "closure",
   "input": input-name,
   "body": body-expression,
   "environment": environment
}
```

Input-name and *body-expression* are simply copied from the given `function`. The only new element is the *environment*.

Environment objects hold the definitions of all variables accessible in a given evaluation context, i.e. variables which the *body-expression* can use. Environments are also first class values, explicitly represented as Jsonya elements. This allows them to be easily passed around (locally or remotely) and facilitates metaprogramming. An environment object has a structure similar to the following:

```
{
    "tag": "environment",
```

[15] Our current implementations sort in UTF16 order, but this may be changed in future.

```
    "parent": parent-environment,
    "locals": {
        "name_1" : expr_1,
        "name_2" : expr_2,
        ...
        "name_n" : expr_n,
    }
}
```

As one would expect, the `locals` field contains the variables and *parent-environment* is an environment from which variables should be looked up if not present locally, or `null`. What may surprise some is that the variables are not stored as values but rather as unevaluated expressions. This provides several benefits for the interpretation algorithm, while one can still inject values by simply putting them in `quote` expressions. Most notably, this allows various lazy-like evaluation schemes to be implemented without having to break out of the Jsonya/dm data model (e.g. to provide lazy memory cells).

4.6. Basic Interpreter. Now we are ready to define the basic interpretation algorithm. The following pseudocode defines the `eval` function, which takes two elements — the environment into which the expression to be evaluated and the expressions itself. This function needs to be implemented in the host programming language that will use Jsonya/fn:

```
EVAL(ENV, EXP)
switch exp.tag
case "object":
   localEnv = sub-environment of env, with exp.fields as locals
   return object containing
     k : eval(localEnv, exp.fields[k]) for each field name k
case "array":
     return array containing eval(env, exp.items[i])
         for each item index i
case "get":
     p = exp.path
     varEnv = locate the parent of env with p[0]
     val = eval(varEnv, varEnv.locals[p[0]])
     return val with all internals from p[1..] applied to it
case "call":
     closure = eval(env, exp.function);
     parameter = eval(env, exp.parameter);
```

```
     callEnv = closure.env
         with {"tag": "quote", "value": parameter}
         added to its locals
     return eval(callEnv, closure.body);
case "internal":
     parameter = eval(env, exp.parameter);
     return internal(exp.name, parameter);
case "if":
     condition = eval(env, exp.condition);
     if condition
     then return eval(env, exp.then);
     else return eval(env, exp.else);
case "quote":
     return exp.value;
case "function":
     return exp patched with
         "tag" : "closure",
         "environment": env
end
```

What we do is to recursively evaluate each expression depending on its kind. Many of the types of expressions are straightforward. The most interesting ones are the `object`, `function` and `call` expressions.

For the evaluation of the `object` expressions a new sub-environment which includes the expressions of all object fields is created. Then this environment is used for the evaluation of each object.

The `function` expressions are evaluated trivially. The only important thing is that the same environment that is given for the evaluation is stored in the closure.

The `call` expression is the most interesting one. First it gets the closure that is to be invoked and constructs a new environment which is based on the captured one, but with the value of the `parameter` assigned to the variable whose name is given by `input`. Then it evaluates the `body` of the closure in the created environment.

With the exception of the data model and the implementations of the internals (which we assume to be implemented in the function `internal`), this is basically what is needed to interpret Jsonya/fn.

4.7. Unorderedness and Laziness. Let us again consider the unorderedness of the objects. The presented algorithm can evaluate correctly the

following Jsonya/fn program:

```
{
    "tag": "object",
    "fields": {
        "a": {"tag": "get", "path": ["c"]},
        "b": {"tag": "quote", "value": 5},
        "c": {"tag": "get", "path": ["b"]},
    }
}
```

As expected, the result of it is the object {"a": 5, "b": 5, "c": 5}. The algorithm manages to compute the values in the appropriate order, as a lazy functional language would do. Based on this, one will expect that order of initialisation is not something to worry about any more.

The following example however (written in JavaScript-like notation[16] to spare 40 lines of code) shows this is not completely the case:

```
{
    "a": {
        "x": 5,
        "y": b.z + x  //10
    },
    "b": {
        "z": a.x,  //5
        "t": a.y + z  //15
    },
}
```

If we look at the values of the fields `x`, `y`, `z`, and `t`, they do not have cyclic dependencies, so if the interpreter is smart enough, their values can be computed correctly. Unfortunately this does not happen in the above algorithm, as it treats values as indivisible entities. When `y` refers to `b.z`, it would cause the whole `b` to be computed, which will need `a.x` or `a.y`, which will in turn need `a`, resulting in an infinite recursion.

Luckily, this deficiency can be resolved relatively easily. If we focus on cross-referencing objects, i.e. objects whose members mutually refer to each other's, then we can identify the reason for this infinite recursion as the implementation of the `get` expression. Even though we may need just a part of an

[16] It would not work in JavaScript either.

object it always evaluates the whole object. To resolve this we introduce the following function which evaluates only part of a value, and we also modify the `"get"` case of the `eval` function to use it:

```
evalPart(env, exp, path)
if path.length and exp.tag == "object"
    and path[0][0] == '.' then
  localEnv = sub-environment of env, with exp.fields  as locals
  // evaluate only the important field
  key = path[0][1..];
  return lazyEval(localEnv, exp.fields[key], path[1..]);
elseif path.lenght and exp.tag == "array"
    and path[0][0] == '#' then
  index = parse path[0][1..];
  return lazyEval(env, exp.items[index], path[1..]);
else
  return eval(env, exp) with all path elements applied to it;
end
eval(env, exp)
...
case "get":
  p = exp.path;
  e = locate the parent of env containing p[0]
  return lazyEval(e, e.locals[p[0]], p[1..]);
...
```

As can be seen, this enhancement is not very complicated, but it solves the particular problem. However, this solution is also not complete. If `a` and `b` were returned from a function instead of accessed directly from the environment, evaluation would still recurse infinitely.

It is not hard to step further and resolve this case too. In fact, we can transfer more cases to the `evalPart` function and modify some of the `eval` cases. In the implementation of the `"call"` expressions we can enclose the parameter expression in a closure and inject a call to this closure in the inner environment, instead of passing the value, thus achieving a call-by-name semantics. We can also allow, by incorporating some of the internals in the `evalPart` function, certain properties of composite element, e.g. `@size`, to be computed without computing its members.

At this point, however, we are not confident that going beyond the automatic order of variable initialisation, e.g. making Jsonya/fn a full call-by-need

implementations is reasonable as we are targeting consistent fail-fast behaviour.

4.8. Memoisation. Consider the following example of the straightforward implementation of the Fibonacci function:

```
{
  "tag": "object",
  "fields": {
    "sum": { calls built-in-numeric-sum },
    "less": { calls built-in-numeric-less },
    "fibonacci": {
      55 lines of code doing roughly the following:
      if less [x, 2]
      then x
      else sum [
        fibonacci sum [x. -1],
        fibonacci sum [x. -2]
      ]
    },
      "fib7": { calls fibonacci 7},
      "fib17": { calls fibonacci 17},
  }
}
```

Fibonacci numbers have often been used to illustrate the need of memoisation or the benefits of tail recursion, so it is not hard to notice that this implementation runs in exponential time. While we certainly need to address its performance, there are also some other issues, which are less obvious.

One such problem is the size of the resulting program. The resulting object will have five members: `sum`, `less`, `fibonacci`, `fib7` and `fib17`. While the last two will simply have the values 13 and 1597, the first three will be closures. Although all reside in the same environment, each will have a copy of it, containing the expressions of all five members. In this case, the result in expanded form[17] is 698 lines, compared to the 179 lines of the input. In general, the size of the result can grow quadratically.

Structural Memoisation. Before focusing on the JSON representation, we need to first address the in-memory behaviour. A careful examination of the `eval` and `evalPart` functions would reveal that the environments will actually be shared — when object members are evaluated, the same environment is passed

[17] Each value on separate line, plus one extra closing line for non-empty objects and arrays.

to each sibling. Because of this, the result will occupy much less space in memory than in JSON.

If the same value is computed by different means, e.g. two function calls return the same value, it will not be shared. As described in Section 3.3, we want to step further and enforce structural sharing even in this case. To do this, in the host language we need to do the following:

- A suitable hash function capable of computing hash values of all Jsonya/dm values must be defined. It must be sophisticated enough to keep the collision probability (i.e. two distinct Jsonya/dm values with the same hash in the same system node) negligible.
- The object model needs to be enhanced so that upon the construction of each Jsonya/dm value its hash value is associated with it.
- A value cache (e.g. a hash map) between hash values and Jsonya/dm values needs to be allocated and the functions creating Jsonya/dm values are modified to check the cache and reuse values there.
- Depending on the capabilities of the host technology, a cleanup strategy such as least-recently-used or random-replacement or weak references can be employed.

It is not crucial that full structural sharing is achieved; for our needs it is sufficient if it captures often occurring repetitions. The associated hash value, however, is also important for another reason — it allows us to compare values of arbitrary size in constant time.

The same approach and hash function can also be used to reduce the size of the JSON representation. An alternative, but still JSON-based, format which includes (e.g. string based) references will make the size of the file representation similar to the in-memory one. The only drawback is that serialisation and deserialisation would require additional processing step. In this paper, we omit the definition of such a format.

Computational Memoisation. With the introduced structural sharing, the memory layout of the values becomes the smallest possible directed acyclic graph. This, however, is not observable — the values are still treated as though they were tree-based. In particular, the fact that each of the five closures from the Fibonacci example will point to the same environment will not save them from having to perform the same computation with it. If several closures were using `fib17`, each of them would compute its value for itself. This behaviour may lead to much more obscure performance problems than the computation time of the Fibonacci number itself.

One reason for this is that our algorithm effectively does tree reduction instead of graph reduction, and so the techniques we apply are not appropriate from a performance standpoint. Fortunately, we can easily resolve this and other problems by simply memoising the `eval` and the `evalPart` functions using the hash values already defined. As with structural memoisation, a replacement strategy may be implemented in order to manage memory consumption.

The effects of this memoisation are significant — the evaluator will not evaluate the same expression in the same environment twice. Because hash values depend on the whole structure of the Jsonya/dm values, if the same expression occurs multiple times in the same environment, its computation will be reused. This way, variables like `fib17` will not be computed more than once.

Regarding the computation of the Fibonacci number, the way we evaluate the `"call"` expressions is by evaluating their body expression in an environment containing the passed parameter. Thus, if the same closure is invoked with the same argument twice, the second invocation will reuse the value of the first. This in turn means that the Fibonacci example will be automatically memoised.

4.9. Other Properties. So far we have shown the most significant properties of the language, namely the explicit representation of closures and environments, the unorderedness of the values and automatic order of evaluation, the automatic structural and computation memoisation.

Another interesting property is that recursion, both single and mutual, works without the need of a fixpoint combinator or other special constructs. This is a side effect of the way we pass environments — effectively each regularly defined closure contains the function it was defined from.

One limitation that the Jsonya/fn language has is the lack of error recovery. Each failure terminates the program, and can be observed only in the host language. Although this is done to ensure computational consistency, we hope to find a better solution in future.

5. Feasibility. Jsonya/fn's structural and computational memoisation, automatic order of initialisation and treatment of closures may tackle certain causes of complexity when an appropriate programming style is used. However, let us go back to the problem that we claimed to provide a solution for, namely what data model should be used in order to apply functional computations in global applications. We argue that JSON-based data model like Jsonya/dm is a valuable option, and particularly that:

- Jsonya can provide benefits to global Internet applications when integrated gradually;

- the costs of the gradual integration of Jsonya in global Internet applications is outweighed by the benefits.

Because the presented approach currently lacks some necessary tools, we have to make certain assumptions. Specifically, we assume that an appropriate high-level functional programming language that compiles to Jsonya/fn will be present in addition to it, and assess the feasibility of utilising the resulting toolset. We believe that such a language is not hard to develop, but in order not to speculate, our claim becomes: If a high-level function programming language that compiles to Jsonya/fn is present, then the resulting toolset can be gradually integrated into the development of global Internet application with benefits outweighing the costs.

5.1. Integration Cost. The success of any JSON-focused instrument, including Jsonya/fn, depends on how easily applicable in various environments it is. JSON Schema [46] is an example of a tool implemented for variety of languages, including C, Java, .NET, ActionScript, Haskell, Python, Ruby, PHP and JavaScript[18]. Some of the implementations, however, only implement part of the functionality, yet none of them is shorter than a few hundred lines of code. More complete implementations such as JSV by Gary Court[19] spawn thousands of lines.

Fully implementing a tool like JSON Schema requires a serious effort. Because of this, it is important to assess how hard the implementation of Jsonya/fn for each target platform will be. In addition to required effort, the size of the implementation is also important, especially if it is to be included in web pages or mobile applications. To find out, we implemented the described Jsonya/fn interpreter in Java and JavaScript similarly to the shown pseudo-code:

- The simple version of the interpreter, which did not handle crossreferences, was implemented in less than 300 lines of code in both languages (including reasonable number of comments, but in the case of Java without counting the JSON parser and data model which were provided by Jackson[20]).
- Implementing the full algorithm, i.e. supporting crossreferences and memoisation, added less than 100 additional lines in both implementations.

However, not everything went smoothly. The main problems we encountered were:

[18] According to `http://json-schema.org/implementations.html`
[19] https://github.com/garycourt/JSV
[20] `http://jackson.codehaus.org/`

- The data model of JavaScript was not very convenient because we were not able to associate the hash values without polluting the Jsonya/dm model. To work around this, all Jsonya/dm values were wrapped in an additional object.
- Because JavaScript lacks native support for large decimal numbers, to fully implement Jsonya/fn, an external library is needed. Because such libraries are usually thousands of lines of code, in certain cases it may be reasonable to restrict the supported numeric range instead.
- Because of the statically typed nature of Java, the invocation of Jsonya/fn functions and the definition of built-ins felt unnatural. To provide convenient, type-safe means for this, we attempted two approaches, which were successful but complex. The first one used inheritance and was about 1500 lines; the second used dynamic proxies to synthesise objects from annotations and was about 3700 lines.

Based on these results, we may assume that implementing a conforming Jsonya/fn interpreter is relatively easy. Our minimal implementations are comparable in size to the famous Lisp metacircular evaluator as described by Abelson et al. [1], which is about 400 lines of code. A quick search shows that interpreters of other simple languages are also in the range of a few hundred lines of code.

In fact, if the aforementioned JSON-schema was implemented in Jsonya/fn (by using the hypothetical high-level language) it would have to be written only once, even if it is thousands of lines of code; then even if a Jsonya/fn interpreter is not available for a certain technology, implementing a minimal conforming interpreter and using the Jsonya/fn implementation would still be cheaper than implementing the whole schema validation algorithm.

This last example is also a significant motivation for the development of Jsonya/fn. JSON is young and its toolset is relatively immature. We believe that our approach can help the situation to be improved.

5.2. Benefits. Finally, we will look at the architecture of a real global application and define where Jsonya/fn could be applied and how. The software system, whose name we cannot disclose due to legal reasons, allows users to create a specific type of content, as well as to access the content created by other users by browsing or searching it. The content is not statically displayed but rendered by an appropriate means; browsing and searching is also implemented via specific visualisations and interactions. Users can provide feedback about the content of others, as well as to observe what other users do, similarly to other socially-enabled systems.

The development of the system started in 2010 and is still ongoing; the first publicly accessible version was released in early 2011. We were involved in the development of the system from the beginning of the project to until early 2012. The system needs to handle up to one million registered users, to be accessible from all popular desktop web browsers, as well as from Android and iOS devices. It also needs to have a (relatively simple) administrative interface.

All of these goals were reached. The server side was implemented in Java on top of the Google App Engine, using the non-relational Datastore and Blobstore to persist the information. It was accessed via a set of HTTP-based web services written in Java. Most of them used JSON to process input and return results, but some used the GWT RPC protocol. The web based front-end was implemented mostly in Java and translated to JavaScript with GWT. Parts of the web interface were implemented directly in JavaScript and parts had to be implemented in Adobe Flash and ActionScript. The mobile front ends were implemented as native Android and iPhone applications which communicate to the server with JSON though HTTP. Parts of the Java code from the server were reused in the Android application.

By using the architecture of the system and our experience during its development, we identified the following benefits that the availability of a technology like Jsonya/fn would have have brought:

- The problem domain model would have been reused in all four programming languages employed. Certain bugs caused by incompatibilities would have been avoided.
- GWT would very likely not have been used; although it allowed us to reuse Java code in the client side its limitations were too severe.
- Many computations for the model, including validation logic, would have been reused through Jsonya/fn, thus avoiding several client/server bugs.
- Certain security bugs caused by unnecessarily sending more information (whole objects) because of the used JSON serialisation tool would have been avoided.
- The synchronisation between client and server related to offline operation would have been implemented more easily, as with Jsonya/fn we would have been able to design a simpler solution which applies the same modification operations both locally and remotely.

The above benefits would be available (to a lesser extent) if some of the system parts were left intact, i.e. implemented without Jsonya/fn.

The main drawbacks we identified from the application of the hypothetical toolset were:

- Some of the developers would have had to learn another programming language.
- The data model of the database is not compatible with Jsonya; database operations would still have to be implemented manually.
- Unless a more sophisticated type-safe implementation is applied, interactions between Jsonya/fn and Java or Objective C would feel unnatural.

We believe that despite these drawbacks, the benefits outweigh the costs. The analysed system, although relatively large, is not unique in having these complexities; other global applications we have worked on also suffered from similar problems. Of course, to really validate these predictions, a high-level functional language needs to be implemented on top of Jsonya/fn, and the approach needs to actually be applied; we hope that in future we will have the opportunity, and the resources, to do so.

6. Discussion. The automatic memoisation that Jsonya/fn features is capable of drastically optimising certain programs, but does not come without its cost — many operations require a cache lookup, which for certain programs will more often fail than not. Because of this, a well written functional program evaluated in a strict interpreter may outperform an equivalent Jsonya/fn program by a constant factor. In exactly what situations the benefits of memoisation will outweigh the cost requires more evaluation.

While structural sharing may be easily implemented in the interpreter, thus Jsonya/dm values containing a lot of duplication would consume small amount of memory when loaded, saving them back to (flat) JSON can cause their size to explode. To resolve this either a JSON-based file format which employs automatic sharing need to be developed, or these structures need to be processed appropriately before serialisation.

Although the syntax of Jsonya/fn is relatively stable, various additions and modifications have been considered, and may be applied in the future. A `let` expression construct, which defines variables in a local scope, could allow local variables that are not part of the resulting objects to be defined. However, this can be simulated relatively easily by exploiting the `call` construct with a `parameter` defining the local variables and an anonymous `function` which contains the resulting expression. Another option would be to allow a `where`

field to be defined to all (or most) constructs, but this does not seem necessary at this point.

We also presented several properties of the language for which we did not evaluate thoroughly the benefits or drawbacks. The automatic order of initialisation which was illustrated by resolving the crossreference example was one of them. Our experience shows, that in the development of large software systems, the order of initialisation can become very complicated. Taking this burden off developers' shoulders would be a valuable addition, but this may vary depending on the specific technology. Another feature which we did not assess fully was the potential for code mobility based on the way closures are represented. Automatic code distribution (and scalability in general) was one of the goals which influenced large part of the design of Jsonya/fn, yet how exactly it can be achieved remains a topic for future research.

Other important issues that need to be further explored in future are the efficient implementation of the Jsonya/dm data model, which is the heart of the whole approach and the main requirement for consistent computations, and the form of the hypothetical high-level functional language which translates to Jsonya/fn. Whether it is suitable to translate some of the popular functional languages, or a new one has to be developed, remains to be seen.

7. Conclusion. The analysis of the popular functional tools applied in the context of Internet applications shows that the choice of data model is an important one. JSON-based data model can be used to provide smooth integration of functional computation in global applications. The JSON-based approach Jsonya incorporates the data model Jsonya/dm and the intermediate homoiconic functional language Jsonya/fn, which is represented as and works with Jsonya/dm values. The language is defined following the main principles of JSON. Built-ins are handled in such a way that environments and closures are also represented explicitly as Jsonya/dm values, thus achieving a stronger form of homoiconicity. The described implementation of the language features automatic structural sharing, computational memoisation and automatic order of evaluation. The assessment of the feasibility of fully implementing the approach shows that it can be beneficial to large heterogeneous Internet-centric applications.

It is our hope that in the near future Jsonya will reach the point when it can be used to in the development of large Internet-centric applications, but more importantly, we hope to have convinced the readers that JSON exhibits very interesting properties as a foundation for functional programming and deserves to be explored further.

REFERENCES

[1] Abelson H., G. Sussman, J. Sussman, A. Perlis. *Structure and interpretation of computer programs*, **2**(1985), MIT Press Cambridge, MA.

[2] Benzaken V., G. Castagna, A. Frisch. CDuce: an XML-centric general-purpose language. *SIGPLAN Not.*, **38** (2003), No 9, 51–63. doi: 10.1145/944746.944711. http://doi.acm.org/10.1145/944746.944711

[3] Chamberlin D., J. Robie, D. Florescu, S. Boag, J. Siméon, M. F. Fernández. XQuery 1.0: An XML query language. W3C recommendation, W3C, Jan. 2007. http://www.w3.org/TR/2007/REC-xquery-20070123/.

[4] Chlipala A. Ur: statically-typed metaprogramming with type-level record computation. In: Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation, PLDI '10, New York, NY, USA, ACM, 2010, 122–133.

http://doi.acm.org/10.1145/1806596.1806612

[5] Clark J. XSL transformations (XSLT) version 1.0. W3C recommendation, W3C, Nov. 1999. http://www.w3.org/TR/1999/REC-xslt-19991116.

[6] Cooper E., S. Lindley, P. Wadler, J. Yallop. Links: Web programming without tiers. In: Formal Methods for Components and Objects (Eds F. de Boer, M. Bonsangue, S. Graf, W.-P. de Roever), Lecture Notes in Computer Science, Vol. **4709**, Springer Berlin/Heidelberg, ISBN 978-3-540-74791-8, 2007, 266–296.

[7] Cowlishaw M. F. Decimal floating-point: Algorism for computers. In: Proceedings of the 16th IEEE Symposium on Computer Arithmetic (ARITH-16'03), ISBN 0-7695-1894-X, ARITH '03, IEEE Computer Society, Washington, DC, USA, 2003, 104–111. http://dl.acm.org/citation.cfm?id=786450.786615

[8] Crockford D. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational), 2006. http://www.ietf.org/rfc/rfc4627.txt

[9] Damas L., R. Milner. Principal type-schemes for functional programs. In: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '82, ISBN 0-89791-065-6, ACM, New York, NY, USA, 1982, 207–212. http://doi.acm.org/10.1145/582153.582176

[10] ECMA. ECMA-262: ECMAScript Language Specification. 5.1 edition, June 2011. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

[11] Eich B. Javascript at ten years. *ACM SIGPLAN Notices*, **40** (2005), ACM, 129–129.

[12] El-Ansary S., D. Grolaux, P. Van Roy, M. Rafea. Overcoming the multiplicity of languages and technologies for web-based development using a multi-paradigm approach. In: Multiparadigm Programming in Mozart/Oz (Ed. P. Van Roy), Lecture Notes in Computer Science, Vol. **3389** , Springer Berlin/Heidelberg, ISBN 978-3-540-25079-1, 2005, 113–124. http://dx.doi.org/10.1007/978-3-540-31845-3_10

[13] Elsman, M., K. Larsen. Typing XHTML web applications in ML.In: *Practical Aspects of Declarative Languages* (Ed. B. Jayaraman), Lecture Notes in Computer Science, Vol. **3057** Springer Berlin/Heidelberg, ISBN 978-3-540-22253-8, 2004, 224–238.

[14] Ennals R., D. Gay. User-friendly functional programming for web mashups. *SIGPLAN Not.*, **42** (2007), No 9, ISSN 0362-1340, 223–234. http://doi.acm.org/10.1145/1291220.1291187

[15] Frisch. A. OCaml + XDuce. *SIGPLAN Not.*, **41** (2006), No 9, ISSN 0362-1340, 192–200. http://doi.acm.org/10.1145/1160074.1159829

[16] Gaevi D., D. Djuri, V. Devedi, D. Gaevic, D. Djuric, V. Devedic. Model driven engineering. In: Model Driven Engineering and Ontology Development, Lecture Notes in Computer Science, Springer Berlin/Heidelberg, ISBN 978-3-642-00282-3, 2009, 125–155.

[17] Gomard C. K., N. D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, **1**, ISSN 1469-7653, 21–69. http://dx.doi.org/10.1017/S0956796800000058

[18] Hall M., J. Mayfield. Improving the performance of ai software: Payoffs and pitfalls in using automatic memoization. In: Proceedings of the Sixth International Symposium on Artificial Intelligence, Megabyte, Sept. 1993, 178–184.

[19] Hanus M. High-level server side web scripting in Curry. In: Practical Aspects of Declarative Languages (Ed. I. Ramakrishnan), Lecture Notes in Computer Science, Vol. **1990**, Springer Berlin/Heidelberg, ISBN 978-3-540-41768-2, 2001, 76–92.

[20] Hanus M. Type-oriented construction of web user interfaces. In: Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming, PPDP '06, ISBN 1-59593-388-3, ACM, New York, NY, USA, 2006, 27–38. http://doi.acm.org/10.1145/1140335.1140341

[21] Hanus M. Putting declarative programming into the web: Translating Curry to JavaScript. In: Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming, PPDP '07, ACM, ISBN 978-1-59593-769-8, New York, NY, USA, 2007, 155–166. http://doi.acm.org/10.1145/1273920.1273942

[22] Hindley R. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, **146** (1969), 29–60.

[23] Hosoya H., B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Trans. Internet Technol.*, **3** (2003), No 2, ISSN 1533-5399, 117–148. http://doi.acm.org/10.1145/767193.767195

[24] Hostetter M., D. Kranz, C. Seed, C. Terman, S. Ward. Curl: a gentle slope language for the web. *World Wide Web J.*, **2** (1997), No 2, ISSN 1085-2301, 121–134. http://dl.acm.org/citation.cfm?id=275062.275073

[25] Jones M. P. Type classes with functional dependencies. In: Proceedings of the 9th European Symposium on Programming Languages and Systems, ESOP '00, ISBN 3-540-67262-1, London, UK, 2000, 230–244. http://dl.acm.org/citation.cfm?id=645394.651909

[26] Joy M., T. Axford. GCODE: a revised standard for a graph representation for functional programs.*SIGPLAN Not.*, **26** (1991), No 1, ISSN 0362-1340, 133–139. http://doi.acm.org/10.1145/122203.122214.

[27] Kay M. XSL transformations (XSLT) version 2.0. W3C recommendation, W3C, Jan. 2007. http://www.w3.org/TR/2007/REC-xslt20-20070123/.

[28] Kiselyov O. SXML specification. *SIGPLAN Not.*, **37** (2002), No 6, ISSN 0362-1340, 52–58. http://doi.acm.org/10.1145/571727.571736

[29] Kiselyov O., S. Krishnamurthi. SXSLT: Manipulation language for XML. In: Practical Aspects of Declarative Languages (Eds V. Dahl and P. Wadler), Lecture Notes in Computer Science, Vol. **2562**, Springer Berlin/Heidelberg, ISBN 978-3-540-00389-2, 2003, 256–272. http://dx.doi.org/10.1007/3-540-36388-2_18

[30] Krishnamurthi, S., P. Hopkins, J. McCarthy, P. Graunke, G. Pettyjohn, M. Felleisen. Implementation and use of the PLT Scheme web server. *Higher-Order and Symbolic Computation*, **20** (2007), No 4, 431–460.

[31] Marchal B. Working XML: Comparing XSLT 2.0 and XQuery. IBM developerWorks, Apr. 2006. `http://www.ibm.com/developerworks/xml/library/x-wxxm34/`

[32] McGranaghan M. ClojureScript: Functional programming for JavaScript platforms. *Internet Computing, IEEE*, **15** (2011), No 6, 97–102.

[33] Meijer E., M. Shields. XMlambda – a functional language for constructing and manipulating XML documents. Technical report, submitted to USENIX 2000 Technical Conference.

[34] Milner R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, **17** (1978), No 3, ISSN 0022-0000, 348–375. doi: 10.1016/0022-0000(78)90014-4 `http://www.sciencedirect.com/science/article/pii/0022000078900144`

[35] Monniaux D. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, **30** (2008), No 12, ISSN 0164-0925, 1–41.

[36] Murphy VII T., K. Crary, R. Harper. Type-safe distributed programming with ML5. In: Trustworthy Global Computing(Eds G. Barthe, C. Fournet), Lecture Notes in Computer Science, Vol. **4912**, Springer Berlin/Heidelberg, ISBN 978-3-540-78662-7, 2008, 108–123.

[37] Nørmark K. Web programming in Scheme with LAML. *Journal of Functional Programming*, **15** (2005), No 1, 53–65.

[38] Peyton Jones S. L. FLIC—a functional language intermediate code. *SIGPLAN Not.*, **23** (1988), No 8, ISSN 0362-1340, 30–48. `http://doi.acm.org/10.1145/47907.47910`

[39] Plasmeijer R., P. Achten. iData for the world wide web programming interconnected web forms. In:Functional and Logic Programming (Eds Hagiya M., P. Wadler) , Lecture Notes in Computer Science, Vol. **3945**, Springer Berlin/Heidelberg, ISBN 978-3-540-33438-5, 2006, 242–258.

[40] Serrano M., E. Gallesio, F. Loitsch. Hop, a language for programming the web 2.0. In: Proceedings of the First Dynamic Languages Symposium, Oct. 2006.

[41] Sredkov M. Jsonya/dm: A univocal JSON interpretation. In: Proceedings of the 8th Central and Eastern European Software Engineering Conference in Russia (CEE-SECR), 2012, Moscow, Russia (in print).

[42] Sredkov M. Common textual representation of software items. In: Proceedings of the 6th South East European Doctoral Student Conference: Infusing Research and Knowledge in South-East Europe (Eds D. Dranidis, A. Kapoulas, A. Vivas), Thessaloniki, Greece, Sept. 2011, ISBN 978-960-9416-04-7, 350–357.

[43] Thiemann P. A typed representation for HTML and XML documents in Haskell. *J. Funct. Program.*, **12** (2002), ISSN 0956-7968, 435–468. doi: 10.1017/S0956796802004392. `http://dl.acm.org/citation.cfm?id=968417.968423`

[44] Wallace M., C. Runciman. Haskell and XML: generic combinators or type-based translation? In: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming, ICFP '99, ISBN 1-58113-111-9, ACM, NY, USA, 1999, 148–159. `http://doi.acm.org/10.1145/317636.317794`

[45] Wilde E., R. J. Glushko. Document design matters. *Commun. ACM*, **51** (2008), ISSN 0001-0782, 43–49.

[46] Zyp K. A JSON media type for describing the structure and meaning of JSON documents, Nov. 2010. `http://tools.ietf.org/html/draft-zyp-json-schema-03`

Faculty of Mathematics and Informatics
Sofia University
5, James Bourchier Blvd
1164 Sofia, Bulgaria
e-mail: `msredkov@fmi.uni-sofia.bg`

Received December 14, 2012
Final Accepted April 29, 2013