# A Calculus with Partially Dynamic Records for Typeful Manipulation of JSON Objects*

Atsushi Ohori[1,2], Katsuhiro Ueno[1,2], Tomohiro Sasaki[1,2], and Daisuke Kikuchi[1,3]

1 Research Institute of Electrical Communication, Tohoku University
Sendai, Miyagi, 980-8577, Japan
{ohori,katsu,tsasaki,kikuchi}@riec.tohoku.ac.jp

2 Graduate School of Information Sciences, Tohoku University
Sendai, Miyagi, 980-8579, Japan

3 Hitachi Solutions East Japan, Ltd.
Sendai, Miyagi, 980-0014, Japan
daisuke.kikuchi.hz@hitachi-solutions.com

## Abstract

This paper investigates language constructs for high-level and type-safe manipulation of JSON objects in a typed functional language. A major obstacle in representing JSON in a static type system is their heterogeneous nature: in most practical JSON APIs, a JSON array is a heterogeneous list consisting of, for example, objects having common fields and possibly some optional fields. This paper presents a typed calculus that reconciles static typing constraints and heterogeneous JSON arrays based on the idea of *partially dynamic records* originally proposed and sketched by Buneman and Ohori for complex database object manipulation. Partially dynamic records are dynamically typed records, but some parts of their structures are statically known. This feature enables us to represent JSON objects as typed data structures. The proposed calculus smoothly extends with ML-style pattern matching and record polymorphism. These results yield a typed functional language where the programmer can directly import JSON data as terms having static types, and can manipulate them with the full benefits of static polymorphic type-checking. The proposed calculus has been embodied in SML#, an extension of Standard ML with record polymorphism and other practically useful features. This paper also reports on the details of the implementation and demonstrates its feasibility through examples using actual Web APIs. The SML# version 3.1.0 compiler includes JSON support presented in this paper, and is available from Tohoku University as open-source software under a BSD-style license.

**1998 ACM Subject Classification** D.3.3 [Programming Languages] Language Constructs and Features - data types and structures

**Keywords and phrases** JSON, Type System, Polymorphic Record Calculus, SML#

**Digital Object Identifier** 10.4230/LIPIcs...18

* The first and the second authors have been partially supported by the JSPS KAKENHI Grant Number 25280019 under the title "Foundational research on implementation methods for the ML-style polymorphic programming language SML#". The second author has also been partially supported by the JSPS KAKENHI Grant Number 15K15964 under the title "A systematic approach for developing practical programming languages".

[1] **This is the author's version of the work; the definitive version will be published in the proceeding of the ECOOP conference, Rome, Italy, July 20–22, 2016. It is made available for your personal use, not for redistribution, until the open-access proceeding of ECOOP 2016 will be available.**

## 1 Introduction

JSON (JavaScript Object Notation) is a text format for serialized structured data. Owing to its simplicity and the human-readable nature, it has become a standard format for data exchanged over the Internet. Web servers and cloud systems, for example, now often provide *JSON API*s, which specify the details of their HTTP requests and responses as JSON formats. Safe and high-level manipulation of JSON objects is, therefore, becoming essential for a programming language to serve as a production language for modern Internet applications.

A common current practice is to provide a library to parse JSON strings and to represent the resulting abstract syntax trees in some data structures of the underlying programming language such as a recursive `datatype` of ML or `JSONObject` class with methods to access JSON attributes with keys. Some languages also provide a mechanism to automatically generate classes and types for given application-specific JSON structures. Examples include JSON schema [17] and the type provider of F# [27]. While these tools free the programmer from the tedious task of deserializing JSON data, from the perspective of a programming language, they are not ideal. Programming with JSON abstract syntax trees is inherently untyped. Type/class generations are meta-level and are not integrated parts of the type system of the language. These approaches cannot take full advantage of high-level and type-safe programming constructs available in advanced programming languages. Indeed, the original motivation for this work came from our experience of developing an ERP system with web interface jointly with a software company [22]. During this development, we had to write codes that dealt with JSON abstract syntax trees. This was not only tedious but also error prone. During this experience we became painfully aware of the need for high-level and type-safe support for JSON objects. The goal of the present paper is to develop programming language constructs for high-level and type-safe manipulation of JSON objects in such a way that they are part of the language type system.

To achieve our goal, we consider a typed functional language with labeled records and pattern matching as an ideal starting point. We note that JSON is a data structure constructed from atomic types (i.e., integers, floating-point numbers, booleans, and strings); "objects," which are named unordered collections of values; and "arrays," which are ordered sequences of values. All of these are common data structures available in a typed functional language. One would therefore expect that they can be directly mapped to typed data structures constructed from labeled records (representing JSON "objects") and lists (representing JSON "arrays"). One would then expect that JSON objects can be directly manipulated through field selection primitives for labeled records and a rich set of higher-order combinators for lists. If such a mapping was indeed possible, then programming with JSON would be a typeful and comfortable programming practice. For example, one would write a function to retrieve, from a JSON object obtained from a weather service on the Internet, a list of cities where the wind speed exceeds 20 mps in the following declarative and concise way:

```
fun highWind weatherMapData =
    foldl (fn ({name, wind, ...}, cities) =>
               if #speed wind > 20.0 then name::cities else cities)
          nil
          weatherMapData
```

Furthermore, one would expect that such a higher-order function with JSON objects is properly type-checked.

For a data structure constructed with records and lists, this is achieved with record polymorphism [20]. The above function, for example, is given the following static type:

$$\texttt{highWind} \ : \ \forall(t_1 :: \{\texttt{name} : t_2, \texttt{wind} : t_3\}, t_2, t_3 :: \{\texttt{speed} : \texttt{real}\}).\ t_1\ \texttt{list} \to t_2\ \texttt{list}$$

where $\forall(t_1 :: \{\texttt{name} : t_2, \texttt{wind} : t_3\}, \ldots)$ represents kinded abstraction of type variable $t_1$ whose possible instances are restricted to those record types that contain at least `name` and `wind` fields of type $t_2$ and $t_3$. This mechanism ensures type-safe manipulation of lists of records containing `name` and `wind` attributes. Since JSON objects often contain a large collection of complicated record structures, an analogous type-safe and high-level treatment is highly desirable. This mechanism of record polymorphism is an integrated part of SML# [26]. The SML# [26] compiler infers the following typing for the above function:

```
highWind : ['a#{name:'b, wind:'c},'b,'c#{speed:real}. 'a list -> 'b list]
```

This variant of ML provides us an ideal starting point for developing a practical polymorphic language with typeful JSON object manipulation support.

Apparently there are a number of technical issues to be sorted out before developing a satisfactory programming language that realizes this desired view. The major obstacles arise from the property that JSON is inherently heterogeneous. In many practical JSON APIs, objects are required to contain certain set of fields and may contain some optional fields. Moreover, this property is often implicit; objects with optional fields are simply those that contain them, and there is no explicit tag to indicate their existence or nonexistence. Because of this property, JSON arrays are heterogeneous in most cases. This is in sharp contrast with typed representation in ML, where an optional field is represented as `option` datatype, and collection types are always homogeneous. Designing a typed language that uniformly deals with heterogeneous JSON objects constitutes a challenge. In this paper, we provide one solution, and implement it in SML#.

We base our development on the observation made in [6] that a polymorphic record calculus with collection types (set types or list types) can be extended with *partially dynamic records* to form a programming language for high-level and type-safe manipulations of heterogeneous collections. A partially dynamic record type is a refinement of type *dynamic* proposed in [2]. In contrast with the standard dynamic type, however, it statically reveals some record fields. Combining this typing constraint with collection-type constructors, one can obtain a typing mechanism for manipulating heterogeneous persistent collections in an ML-style polymorphic language. In the proposal of [6], the idea and typing rules are sketched out, but their formal properties and implementation methods are not investigated. The goal of the present paper is to develop a typed calculus suitable for JSON data based on the idea of partially dynamic records, to establish type soundness, and to implement the calculus for practical use.

When we separate dynamic typing from partially dynamic records, then types of partially dynamic records intuitively correspond to super types of objects in an object-oriented language. Since a super type also represents a substructure common to all of its subtypes, in a type system with structural subtyping, heterogeneous collections can be represented as (uniform) collections of a super type of all the possible element types. With a suitable mechanism for dynamic object inspection (dynamic "down casting") for JSON data, an object-oriented type system with object subsumption would certainly be an alternative approach. However, subtyping would complicate polymorphism and type inference with records, and therefore its impact on an implementation method for ML-style functional languages remains to be investigated. Our proposal is suitable for ML-style functional languages and is readily implementable.

Our general motivation of developing static typing for semi-structured external data is shared with the authors of XDuce [14] and CDuce [3]. They aim at developing a typed language for XML processing. In [10], it has been shown that some of these features can be integrated in OCaml. In these systems, however, XML types do not have direct relationship to static types of existing programming languages. As a result, record-like XML structures are not related to labeled records or any other static data structures of the underlying programming language. In the OCaml+XDuce language [10], for example, XML types are opaque types in the OCaml type system. By contrast, our goal is to develop a language mechanism to manipulate external record-like data directly as labeled records in a static type system of ML. A major technical contribution of our research is not just a development of yet another type system for JSON, but seamless and direct integration of JSON structures in a static polymorphic programming language with labeled records.

The specific technical contributions of the present paper include the following:

- We have presented a type system of JSON data and have developed an algorithm to convert JSON data to explicitly typed ones. In the type system, any JSON data have a unique type. The type system contains *partial record types*, and the type reconstruction algorithm computes a partial record type for a heterogeneous JSON array by computing the partial record type that matches all the element types.
- We have developed a calculus with partially dynamic records, defined its operational semantics, and shown a type soundness theorem. By defining the runtime model of partially dynamic records as a pair of a typed JSON object and its *static view*, this calculus serves as a programming language that integrates with JSON data.
- We have presented a compilation method to extend the calculus with a polymorphic primitive to convert a static complex value to dynamic JSON objects. We have also extended the core calculus with ML-style pattern matching for JSON data.
- We have fully implemented the calculus by extending the SML# compiler. Using the implementation, we have evaluated the implemented compiler with a number of examples including a JSON API in a real web service, and have demonstrated its practical feasibility.

The SML# version 3.1.0 includes JSON features presented in this paper, and is available from Tohoku University as open-source software under a BSD-style license. Using this compiler, Tohoku University and Hitachi Solutions East Japan, Ltd. plan to develop a web application framework to be used for development of a personal prescription notebook server for mobile devices.

The rest of the paper is organized as follows. In Section 2, we analyze static properties of JSON data, discuss technical issues in designing a typed functional language with JSON data, and describe our strategy. Section 3 presents the calculus and establishes type soundness. Section 4 describes implementation techniques and reports on our implementation. Section 5 shows some examples using our SML# compiler and demonstrates the feasibility of our proposal. Section 6 compares our proposal with existing work. Section 7 concludes the paper with suggestions for further investigation.

## 2 Analysis of JSON and our strategy

JSON data, as defined in [5], can be analyzed through the following abstract syntax:

$$j \quad ::= \quad c^b \mid \langle l = j, \ldots, l = j \rangle \mid [j, \ldots, j]$$

$c^b$ is a constant of atomic type $b$, which represents implicitly typed JSON literals. $\langle l_1 = j_1, \ldots, l_n = j_n \rangle$ is a JSON object, where $l$ ranges over a given set of labels (string literals). In the actual JSON format, an object is an unordered collection of colon-separated name-value pairs. In the presentation of the syntax of the calculus, we use record notation similar to ML records. In $\langle l_1 = j_1, \ldots, l_n = l_n \rangle$, the labels $\{l_1, \ldots, l_n\}$ are pairwise distinct. This property represents the unordered nature of fields in a JSON object. $[j_1, \ldots, j_n]$ is a JSON array consisting of a sequence of JSON values, and corresponds to a list or a vector (immutable array) in a functional language. In this paper, we represent them as lists. Vector views are equally possible.

There is no difficulty in parsing JSON strings and representing the resulting abstract syntax trees as data structures in a programming language. The following is a typical definition in ML:

```
datatype json =
    BOOL of bool | INT of int | REAL of real | STRING of string | NULL
  | OBJECT of (string * json) list
  | ARRAY of json list
```

Some ML compilers provide a library to parse JSON strings into a datatype similar to the above. Our goal is to map untyped runtime values (values of a universal type) as a typed term that can be directly manipulated by a typed ML program.

In the formal development, we omit the null object (`NULL` term above). As we shall briefly explain in Section 4, it can easily be introduced as a built-in constant of a built-in type.

For this data structure we have the following straightforward observation. If we restrict the `ARRAY` variant to be homogeneous, then the above data structure corresponds to runtime values of the following set of ML types:

$$\tau \quad ::= \quad b \mid \{l : \tau, \ldots, l : \tau\} \mid \tau \; \mathit{list}$$

It is a routine matter to define a type reconstruction function that first checks whether a given value of type `json` is typable, and if it is typable, then returns its type. We can then regard type `json` as an instance of type *dynamic* proposed in [2]. This simple observation yields a language having the following features:

- It contains an atomic type `json` (corresponding to *dynamic*) whose runtime values are typed JSON data.
- It provides a language construct of the form `_json` $e$ `as` $\tau$ that dynamically checks a typed JSON value whether or not its type component coincides with $\tau$, and if it is the case, then converts the typed JSON value to a value of static type $\tau$. This would raise a runtime exception of *DynamicTypeError* if the type-check fails. An equivalent statement `_typecase` $e$ `of` $\tau_1$ `=>` $e_1$ `|` $\cdots$ `|` $\tau_n$ `=>` $e_n$ `|` `_` `=>` $e_0$ can also be provided.
- It provides a primitive `import` $e$ to parse a JSON string denoted by $e$ to construct an abstract syntax tree of the JSON data and then to check whether it is typable or not. If it is typable then it creates a typed JSON value. This expression has type `json`.

All the above language constructs can be effectively defined in an ML-style type system, and this approach can straightforwardly yield an extension of ML. While this simple approach has some usefulness, it is not practical for serious applications with JSON. As we discussed in Introduction, most of practical JSON data are heterogeneous and are rejected by the type-checking process described above.

To analyze the problem, suppose a given JSON array contains objects having mandatory `name:string`, `wind:{speed:real,deg:real}`, `main:{temp:real}` fields, and optional `clouds:{all:int}` and `main:{pressure:int}` fields, such as the following:

```
[
 {"name":"Sendai", "main":{"temp":19.0},
  "wind":{"speed":7.6, "deg":170.0}},
 {"name":"Natori", "main":{"temp":13.0, "pressure":1010},
  "wind":{"speed":5.6, "deg":150.0}, "clouds":{"all":92}},
 ⋮
]
```

This is an example of commonly encountered heterogeneous JSON arrays. A typical program would extract the mandatory `name:string`, `main:{temp:real}`, and `wind:{speed:real}` fields. Since mandatory fields conceptually represent a static constraint on a list of values, we want a programming language for JSON objects to represent the constraint in its static type system so that the programmer can safely use polymorphic function on records with list combinators such as `foldr` and `map`, as suggested in Introduction,

One approach to reconcile the static constraint and heterogeneous JSON arrays is to introduce *partially dynamic records* presented in [6]. A partially dynamic record type, written as $\{\!\!\{l_1 : \tau_1, \ldots, l_n : \tau_n\}\!\!\}$, denotes a dynamically typed record about which it is statically known that its actual type contains the set of fields $l_1 : \tau_1, \ldots, l_n : \tau_n$. A heterogeneous JSON array can then be typed as a list of a partially dynamic record type. For example, the above term can be given a type of the form

$$\{\!\!\{\mathtt{name} : \mathtt{string},\ \mathtt{main} : \{\!\!\{\mathtt{temp} : \mathtt{real}\}\!\!\},\ \mathtt{wind} : \{\!\!\{\mathtt{speed} : \mathtt{real}, \mathtt{deg} : \mathtt{real}\}\!\!\}\}\!\!\}\ \mathtt{list}$$

indicating the fact that each element of the list is a record that contains at least `name:string`, `main:{temp:real}`, and `wind:{speed:real, deg:real}` fields, and possibly contains more fields. By integrating this typing feature into a static type system with record type, list type, and record polymorphism, we can obtain a typed language that supports high-level and typeful manipulation of JSON objects.

## 3 Definition of the calculus

This section defines the calculus, establishes its type soundness, and describes the necessary extensions to integrate it into an ML-style language with record polymorphism.

### 3.1 Typed JSON Objects

We first define a type system for JSON data. We continue to use the abstract syntax of JSON data that we defined in Section 2.

The set of JSON types is given by the following syntax.

$$\pi \quad ::= \quad b \mid \{l : \pi, \ldots, l : \pi\} \mid \{\!\!\{l : \pi, \ldots, l : \pi\}\!\!\} \mid \pi\ \mathit{list} \mid \mathit{json}$$

In addition to JSON object types of the form $\{l_1 : \pi_1, \ldots, l_n : \pi_n\}$ and JSON array types of the form $\pi$ *list*, we introduce two forms of partial types. One is a partial record type $\{\!\!\{l_1 : \pi_1, \ldots, l_n : \pi_n\}\!\!\}$, which represents JSON objects that contain at least the set of fields $l_1 : \pi_1, \ldots, l_n : \pi_n$. The other is the type *json* for which no structure is known.

To model the above interpretation of these partial types in a type system, we define the following ordering on the set of types:

- $\pi \leq \mathit{json}$ for any $\pi$.
- $\{l_1 : \pi_1, \ldots, l_n : \pi_n, \ldots\} \leq \{\!\!\{l_1 : \pi'_1, \ldots, l_n : \pi'_n\}\!\!\}$ if $\pi_1 \leq \pi'_1, \ldots, \pi_n \leq \pi'_n$.

$$
\begin{array}{rcl}
[\![b]\!] & = & \{c^b \mid c^b \text{ is a constant literal of type } b\} \\
[\![\{l_1 : \pi_1, \ldots, l_n : \pi_n\}]\!] & = & \{\langle l_1 = j_1, \ldots, l_n = j_n\rangle \mid j_1 \in [\![\pi_1]\!], \ldots, j_n \in [\![\pi_n]\!]\} \\
[\![\{\!|l_1 : \pi_1, \ldots, l_n : \pi_n|\!\}]\!] & = & \{\langle l_1 = j_1, \ldots, l_n = j_n, \ldots\rangle \mid j_1 \in [\![\pi_1]\!], \ldots, j_n \in [\![\pi_n]\!]\} \\
[\![\pi\ \mathit{list}]\!] & = & \{[j_1, \ldots, j_n] \mid j_1 \in [\![\pi]\!], \ldots, j_n \in [\![\pi]\!]\} \\
[\![\mathit{json}]\!] & = & \text{the set of all syntactically well formed json terms}
\end{array}
$$

**Figure 1** A syntactic model of JSON types

- $\{\!|l_1 : \pi_1, \ldots, l_n : \pi_n, \ldots|\!\} \leq \{\!|l_1 : \pi_1', \ldots, l_n : \pi_n'|\!\}$ if $\pi_1 \leq \pi_1', \ldots, \pi_n \leq \pi_n'$.
- $\pi\ \mathit{list} \leq \pi'\ \mathit{list}$ if $\pi \leq \pi'$.

The reflexive transitive closure of this relation yields a partial ordering on the set of JSON types. This ordering is the converse of the one used in [6], where the ordering is originally introduce to model the amount of information. Here, we use the notation that is familiar in object-oriented type systems [1] for readability. Note, however, that this ordering is not used to define subsumption relation on terms, but to compute the common substructure of element types in a heterogeneous JSON array. The join (least upper bound) of $\pi_1$ and $\pi_2$ with respect to $\leq$ is written $\pi_1 \sqcup \pi_2$, which denotes the largest common substructure of $\pi_1$ and $\pi_2$.

The typing rules for JSON data are given below:

$$\vdash c^b \ : \ b$$

$$\frac{\vdash j_1 \ : \ \pi_1 \quad \cdots \quad \vdash j_n \ : \ \pi_n}{\vdash \langle l_1 = j_1, \ldots, l_n = j_n\rangle \ : \ \{l_1 : \pi_1, \ldots, l_n : \pi_n\}}$$

$$\frac{\vdash j_1 \ : \ \pi_1 \quad \cdots \quad \vdash j_n \ : \ \pi_n \quad \pi = \pi_1 \sqcup \cdots \sqcup \pi_n}{\vdash [j_1, \ldots, j_n] \ : \ \pi\ \mathit{list}}$$

This type system has a straightforward syntactic model. Figure 1 defines the semantics $[\![\pi]\!]$ of JSON type $\pi$. Using this, we define the semantic typing relation, denoted by $\models j : \pi$ if and only if $j \in [\![\pi]\!]$. We can then show the following simple soundness property of the type system:

▶ Proposition 1. If $\vdash j : \pi$ then $\models j : \pi$.

We note that, under this (intended) model, the ordering of JSON types corresponds to set inclusion, i.e., it is the case that if $\pi_1 \leq \pi_2$, then $[\![\pi_1]\!] \subseteq [\![\pi_2]\!]$. Note also that the subsumption rule

$$\frac{\vdash j \ : \ \pi \quad \pi \leq \pi'}{\vdash j \ : \ \pi'}$$

is sound with respect to the above simple semantics. If we added this rule to our JSON type system, it would correspond to the type system of simple objects with collection types, and our typing judgment corresponds to minimal typing. As we have noted earlier, subsumption is not used in our system. JSON terms are manipulated through polymorphic record operations, and its polymorphic record typing subsumes the object-oriented subsumption rule. To understand this, suppose we access the `name` field of an object of type `{name:string, age:int}`. Instead of applying subsumption rule to obtain a record type `{name:string}`, we give the `name`-field accessor a polymorphic type so that it can be applicable to any object containing a `name` filed. This approach computes more accurate typing than subtyping.

Therefore we have defined the JSON type system as a relation to deduce a unique (most specific) typing. Moreover, with the type *json*, any JSON term has a unique minimal type in our semantics. Indeed, the following property is easily shown:

▶ Proposition 2. For any $j$, there is a unique $\pi$ such that $\vdash j : \pi$. Moreover, if $j \in [\![\pi']\!]$, then $\pi \leq \pi'$.

We define the typed JSON term as a pair $(j : \pi)$ such that $\vdash j : \pi$. We write $\mathit{infer}(j)$ for the typed JSON term $(j : \pi)$ of $j$.

## 3.2 The Calculus with Partially Dynamic Records

The idea underlying our calculus for the static manipulation of JSON data is to consider typed JSON objects defined above as runtime values of partially dynamic record types sketched out in [6]. Based on this general idea, this section develops the typed calculus.

The set of terms of the calculus is given below:

$$e \quad ::= \quad c^b \mid x \mid \lambda x.e \mid e\ e \mid \{l = e, \ldots, l = e\} \mid e.l \mid e :: e \mid \mathsf{nil} \mid j \mid (e\ \mathsf{as}\ \pi\ \mathsf{else}\ e)$$

This is a variant of the typed lambda calculus with records and lists, extended with partially dynamic JSON objects and a dynamic type-checking construct. $\{l = e, \ldots, l = e\}$ is a labeled record in Standard ML, and $e.l$ is record field selection that selects the $l$ field from the record $e$. The operations on lists can be given as primitive functions, and we omit them. $j$ introduces a JSON object as a value of dynamic type. $(e_1\ \mathsf{as}\ \pi\ \mathsf{else}\ e_2)$ dynamically checks whether the type component $\pi'$ of a typed JSON object $e_1$ is smaller than or equal to $\pi$ ($\pi' \leq \pi$). If it is the case, then it coerces the JSON data to a value of type $\pi$; otherwise, it evaluates $e_2$. When $\pi$ is a partial record type, then this term denotes a value of a partially dynamic record, or equivalently a partially static JSON object.

This calculus intends to model a polymorphic programming language with JSON manipulation. An actual language requires additional standard features including recursion, recursive datatypes, pattern matching, and exception. There is no difficulty in extending the calculus with these features. In particular, exception is useful in manipulating dynamic objects and we shall implicitly assume that exception mechanism is available in writing examples.

The set of types is given below:

$$\tau \quad ::= \quad b \mid \tau \to \tau \mid \{l : \tau, \ldots, l : \tau\} \mid \{\!|l : \tau, \ldots, l : \tau|\!\} \mid \tau\ \mathit{list} \mid \mathit{json}$$

Note that this set of types subsumes all the JSON types (ranged over by $\pi$). $\{l : \tau, \ldots, l : \tau\}$ is a labeled record type, and $\tau\ \mathit{list}$ is a list type; when they appear in a typing judgment $\Gamma \vdash e : \tau$, they denote ML records and ML lists. They also denote JSON objects and JSON arrays when they appear as type specifications $\pi$ in a term $(e\ \mathsf{as}\ \pi\ \mathsf{else}\ e)$.

Let $\Gamma$ range over the set of type assignment, which is a finite function from variables to types. $\mathit{dom}(\Gamma)$ is the domain of $\Gamma$. For a typing assignment $\Gamma$, $\Gamma\{x : \tau\}$ is $\Gamma'$ such that $\mathit{dom}(\Gamma') = \mathit{dom}(\Gamma) \cup \{x\}$, $\Gamma'(x) = \tau$, and $\Gamma'(y) = \Gamma(y)$ for any $y \in \mathit{dom}(\Gamma)$ such that $x \neq y$. The type system is defined as a set of rules to derive the typing relation of the form $\Gamma \vdash e : \tau$ indicating that expression $e$ has type $\tau$ under type assignment $\Gamma$. The set of typing rules of the calculus is given in Figure 2.

The observant reader may have noticed that in the pure calculus defined above, there is no closed term having a partially dynamic record type, since $(e_1\ \mathsf{as}\ \pi\ \mathsf{else}\ e_2)$ requires $\mathsf{else}$

$$\Gamma \vdash c^b : b \qquad \Gamma \vdash \mathsf{nil} : \tau\ \mathit{list} \quad (\text{for any } \tau) \qquad \Gamma \vdash j : \mathit{json}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma\{x : \tau\} \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'} \qquad \frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau\ \mathit{list}}{\Gamma \vdash e_1 :: e_2 : \tau\ \mathit{list}} \qquad \frac{\Gamma \vdash e_i : \tau_i \quad (1 \le i \le n)}{\Gamma \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}$$

$$\frac{\Gamma \vdash e : \{l_1 : \tau_1, \ldots, l_i : \tau_i, \ldots, l_n : \tau_n\} \quad (1 \le i \le n)}{\Gamma \vdash e.l_i : \tau_i}$$

$$\frac{\Gamma \vdash e : \{\!|l_1 : \tau_1, \ldots, l_i : \tau_i, \ldots, l_n : \tau_n|\!\} \quad (1 \le i \le n)}{\Gamma \vdash e.l_i : \tau_i}$$

$$\frac{\Gamma \vdash e_1 : \mathit{json} \quad \Gamma \vdash e_2 : \pi}{\Gamma \vdash (e_1\ \mathsf{as}\ \pi\ \mathsf{else}\ e_2) : \pi} \qquad \frac{\Gamma \vdash e_1 : \{\!|l_1 : \tau_1, \ldots, l_n : \tau_n|\!\} \quad \Gamma \vdash e_2 : \pi}{\Gamma \vdash (e_1\ \mathsf{as}\ \pi\ \mathsf{else}\ e_2) : \pi}$$

**Figure 2** Type system for the calculus with partially dynamic records

$$\begin{aligned}
\mathcal{R}(c^b, b) &= c^b \\
\mathcal{R}(j, \mathit{json}) &= \mathit{infer}(j) \\
\mathcal{R}([j_1, \ldots, j_n], \pi\ \mathit{list}) &= \mathcal{R}(j_1, \pi) :: \cdots :: \mathcal{R}(j_n, \pi) :: \mathsf{nil} \\
\mathcal{R}(\langle l_1 = j_1, \ldots, l_n = j_n\rangle, \{l_1 : \pi_1, \ldots, l_n : \pi_n\}) &= \{l_1 = \mathcal{R}(j_1, \pi_1), \ldots, l_n = \mathcal{R}(j_n, \pi_n)\} \\
\mathcal{R}(\langle l_1 = j_1, \ldots, l_i = j_i, \ldots, l_n = j_n\rangle, \{\!|l_1 : \pi_1, \ldots, l_i : \pi_i|\!\}) &= \\
(\mathit{infer}(\langle l_1 = j_1, \ldots, l_i = j_i, \ldots, l_n = j_n\rangle)&, \{l_1 = \mathcal{R}(j_1, \pi_1), \ldots, l_i = \mathcal{R}(j_i, \pi_i)\})
\end{aligned}$$

**Figure 3** Dynamic coercion of JSON data

term $e_2$. As we shall explain later in Section 4, a typical use of as construct is of the form ($j$ as $\pi$ else *DynamicTypeError*), where *DynamicTypeError* is a term to raise an exception when $j$'s runtime type does not match with type $\pi$. This term has type $\pi$ for any $\pi$.

We construct a big-step operational semantics in the style of natural semantics [18]. The set of runtime values, ranged over by $v$, is given below:

$$v \quad ::= \quad c^b \mid \mathit{Cls}(E, x, e) \mid \{l = v, \ldots, l = v\} \mid v :: v \mid \mathsf{nil} \mid (j : \pi) \mid (j : \pi, v) \mid \mathit{Wrong}$$

$\mathit{Cls}(E, x, e)$ is a function closure. $(j : \pi)$ is a typed JSON value defined in Subsection 3.1. $(j : \pi, v)$ is a pair of a typed JSON value and its *view* $v$. *Wrong* indicates runtime failure.

A typed JSON value $(j : \pi)$ is a runtime model of type *json*. A typed JSON value with static view $(j : \pi, v)$ is a runtime model of partially dynamic JSON terms, i.e., those terms whose types contain partial record types. These (partially) dynamic values are type-checked at runtime by the construct ($e_1$ as $\pi$ else $e_2$), as explained above. Figure 3 gives an algorithm $\mathcal{R}$, which takes a typed JSON value and a type and returns a value of that type.

We write $(j : \pi) \approx v$ if $v$ is identical to $(j : \pi)$ or is of the form $(j : \pi, v')$ for some $v'$. Let $E$ range over the set of value assignments, which is a finite function from variables to values. Figure 4 shows the set of evaluation rules that derives the evaluation relation of the form $E \vdash e \Downarrow v$ indicating that $e$ evaluates to $v$ under value assignment $E$. We note that the entire set of rules should be taken with the following implicit rules yielding *Wrong*: if any of the conditions in the premises are not satisfied or evaluation of any subterm yields *Wrong*, then the entire evaluation yields *Wrong*.

$$E \vdash c^b \Downarrow c^b \qquad E \vdash \mathsf{nil} \Downarrow \mathsf{nil} \qquad E \vdash j \Downarrow \mathit{infer}(j) \qquad E \vdash \lambda x.e \Downarrow \mathit{Cls}(E, x, e)$$

$$\frac{E(x) = v}{E \vdash x \Downarrow v} \qquad \frac{E \vdash e_1 \Downarrow \mathit{Cls}(E', x, e) \quad E \vdash e_2 \Downarrow v' \quad E'\{x : v'\} \vdash e \Downarrow v}{E \vdash e_1\ e_2 \Downarrow v}$$

$$\frac{E \vdash e_1 \Downarrow v_1 \quad E \vdash e_2 \Downarrow v_2}{E \vdash e_1 :: e_2 \Downarrow v_1 :: v_2} \qquad \frac{E \vdash e_1 \Downarrow v_1 \quad \cdots \quad E \vdash e_n \Downarrow v_n}{E \vdash \{l_1 = e_1, \ldots, l_n = e_n\} \Downarrow \{l_1 = v_1, \ldots, l_n = v_n\}}$$

$$\frac{E \vdash e \Downarrow \{l_1 = v_1, \ldots, l_i = v_i, \ldots, l_n = v_n\} \quad (1 \le i \le n)}{E \vdash e.l_i \Downarrow v_i}$$

$$\frac{E \vdash e \Downarrow (j : \pi, \{l_1 = v_1, \ldots, l_i = v_i, \ldots, l_n = v_n\}) \quad (1 \le i \le n)}{E \vdash e.l_i \Downarrow v_i}$$

$$\frac{E \vdash e_1 \Downarrow v \quad (j : \pi') \approx v \quad \pi' \le \pi}{E \vdash (e_1\ \mathsf{as}\ \pi\ \mathsf{else}\ e_2) \Downarrow \mathcal{R}(j, \pi)} \qquad \frac{E \vdash e_1 \Downarrow v \quad (j : \pi') \approx v \quad \pi' \not\le \pi \quad E \vdash e_2 \Downarrow v'}{E \vdash (e_1\ \mathsf{as}\ \pi\ \mathsf{else}\ e_2) \Downarrow v'}$$

**Figure 4** Operational semantics of the calculus

## 3.3 Soundness

We first define value typing, denoted by $\models v : \tau$, indicating that runtime value $v$ has type $\tau$ as the following relation:

- $\models c^b : b$
- $\models \mathit{Cls}(E, x, e) : \tau_1 \to \tau_2$ iff there is some $\Gamma$ such that $\models E : \Gamma$ and $\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2$.
- $\models \{l_1 = v_1, \ldots, l_n = v_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$ iff $\models v_i : \tau_i$ for each $1 \le i \le n$.
- $\models \mathsf{nil} : \tau\ \mathit{list}$ for any $\tau$.
- $\models v_1 :: v_2 : \tau\ \mathit{list}$ iff $\models v_1 : \tau$ and $\models v_2 : \tau\ \mathit{list}$.
- $\models (j : \pi) : \mathit{json}$ iff $\vdash j : \pi$ in the JSON type system.
- $\models (j : \pi, v) : \{\!|l_1 : \pi_1, \ldots, l_n : \pi_n|\!\}$ iff $\vdash j : \pi$ in the JSON type system, $\pi \le \{\!|l_1 : \pi_1, \ldots, l_n : \pi_n|\!\}$, and $\models v : \{l_1 : \pi_1, \ldots, l_n : \pi_n\}$.

We write $\models E : \Gamma$ if $\mathit{dom}(E) = \mathit{dom}(\Gamma)$ and $\models E(x) : \Gamma(x)$ for any $x \in \mathit{dom}(E)$.

The following property is easily shown from the semantics of JSON typing:

▶ **Lemma 1.** *If $\vdash j : \pi$ and $\pi \le \pi'$, then $\models \mathcal{R}(j, \pi') : \pi'$.*

We then have the following:

▶ **Theorem 2.** *If $\Gamma \vdash e : \tau$, $\models E : \Gamma$ and $E \vdash e \Downarrow v$, then $\models v : \tau$.*

*Proof.* This is proved by induction on the size of derivations of $E \vdash e \Downarrow v$. Here we only show a few cases involving JSON values.

Case $e = j$. We have $\models \mathit{infer}(j) : \mathit{json}$ by the definition of $\mathit{infer}(j)$.

Case $e = (e_1\ \mathsf{as}\ \pi\ \mathsf{else}\ e_2)$. From the evaluation rules, we have $E \vdash e_1 \Downarrow v_1$. From the typing rules, we have either $\Gamma \vdash e_1 : \mathit{json}$ or $\Gamma \vdash e_1 : \{\!|\cdots|\!\}$. By the induction hypothesis, we have either $\models v_1 : \mathit{json}$ or $\models v_1 : \{\!|\cdots|\!\}$. From the definition of value typing, $v_1$ is either $(j : \pi')$ or $(j : \pi', v')$ for some $j$, $\pi'$, and $v'$ such that $\vdash j : \pi'$. In both cases, $(j : \pi') \approx v_1$ holds. If $\pi' \le \pi$, we have $\models \mathcal{R}(j, \pi) : \pi$ by Lemma 1. If $\pi' \not\le \pi$, we have $\Gamma \vdash e_2 : \pi$ and $E \vdash e_2 \Downarrow v'$ from the typing and evaluation rules. By the induction hypothesis, we have $\models v' : \pi$. ◻

The calculus so far defined is a minimal one with a monomorphic type system. In the rest of this section, we describe some extensions to develop a polymorphic functional language.

## 3.4 Extension to polymorphism

There is no difficulty in extending the monomorphic type system of the calculus defined above to ML-style polymorphism. Because the runtime models of both dynamic type *json* and partial record types $\{\!|l_1 : \pi_1, \ldots, l_n : \pi_n|\!\}$ are JSON data, it is natural to restrict the target type $\pi$ of ($e_1$ `as` $\pi$ `else` $e_2$) to be a monomorphic JSON type. With this restriction, the type system straightforwardly extends to an ML-style polymorphic type system.

Moreover, since partially dynamic JSON data are viewed as ML records, the calculus nicely blends with record polymorphism. Indeed, we can extend a polymorphic record calculus [20] to JSON data and partial record field selections without much difficulty. In our implementation we shall report in Section 4, a partial record field selection is written as a function

```
fn x => #foo (view x)
```

which has a record-polymorphic type

```
['a#{foo: 'b}, 'b. 'a dyn -> 'b]
```

indicating the fact that it accepts any partially dynamic record that has at least a `foo` field.

## 3.5 Serializing ML values as JSON data

The calculus defined above captures how to deal with given JSON data in a statically typed language. In our calculus, we have included JSON term $j$ as a new syntactic category. Our implicit assumption underlying this design is that JSON term $j$ is an abstract syntax of the JSON format. From a practical perspective, this corresponds to including a primitive to parse a string representation of JSON data written by the programmer or received through an I/O primitive. This is what we have done in our implementation.

The opposite direction is of course necessary; we also want to serialize values in the language to generate JSON data in a type-consistent way. This problem can be stated as a problem to define a construct `toJson`($e$) that produces a JSON value $j$ such that (`toJson`($e$) `as` $\pi$ `else` $\perp$) is equivalent to $e$. Moreover, we would like `toJson`() to work polymorphically. This is a sub-problem of type-dependent value printing.

One approach is to add primitives to introduce term of *dynamic* type in [2] and to format a dynamic value. However, there is a subtle technical issue concerning the introduction of *dynamic* type. In practical implicitly typed polymorphic languages in the ML family, only the compiler knows the type information; therefore, $\pi$ does not exist at runtime. Recursive data types make things more difficult since the recursive structure of a data type is not necessarily determined from its name; for example, *list* is just a type name and hence does not indicate its recursive structure consisting of cons and nil. We have partially solved this problem for monomorphic ground types and have implemented a special construct that embeds type representation in object codes. This enables us to *reify* a type of expression. Using this, SML# implements the following primitive:

```
dynamic : ['a. 'a -> dynamic]
```

The current limitation is that this primitive can reify the type information only when the type of the argument to this primitive is monomorphic. Details of this technique is outside the scope of this paper, and we will present them elsewhere.

On top of this dynamic primitive, we introduce the following two primitives:

```
toDynamic : ['a#json. 'a -> dynamic]
dynamicToJson : dynamic -> {json:void dyn, string:string}
```

`toDynamic` is an alias to `dynamic` whose argument type is constrained to JSON type $\pi$ through the kind constraint `'a#json` that restricts instance of `'a` to JSON types. `dynamicToJson` converts dynamic value to dynamic json object and its string representation. Using them, we can write a code

```
let
  val x = [{name = "Joe", score = {math = 76, lang = 89}},
           {name = "Bob", score = {math = 96, lang = 94}}]
in
  print (#string (dynamicToJson (toDynamic x)))
end
```

that produces the following output in JSON format:

```
[
 {"name":"Joe", "score":{"lang":89, "math":76}},
 {"name":"Bob", "score":{"lang":94, "math":96}}
]
```

## 3.6 Pattern matching with JSON

Another useful extension is to integrate JSON manipulation constructs ($e_1$ as $\pi$ else $e_2$) with ML-style pattern matching. This integration relieves the programmer from writing type annotations, and enables the programmer to use familiar programming idioms with ML-style patterns. This extension can be provided by syntactic elaboration in a language with ML-style pattern matching. Here we outline the elaboration.

We assume that the core calculus is extended with the ML-style case statement

case $exp$ of $pat_1$ => $exp_1$ | $\cdots$ | $pat_n$ => $exp_n$

with an extension that record patterns match with partially dynamic JSON objects as well as ML records. We then introduce the following case statement for JSON data

jsonCase $e$ of $jp_1$ => $exp_1$ | $\cdots$ | $jp_n$ => $exp_n$

that performs a case analysis on dynamic or partially dynamic JSON value $e$ against the set of JSON patterns $jp_1, \ldots, jp_n$. The set of JSON patterns is given below:

$$\begin{array}{lcll}
jp & ::= & c^b & \text{(constants of type } b\text{)} \\
& | & x : \pi & \text{(any JSON value of } \pi\text{)} \\
& | & \{l = jp, \ldots, l = jp\} & \text{(JSON object)} \\
& | & \{l = jp, \ldots, l = jp, \ldots\} & \text{(partial JSON object)} \\
& | & \mathsf{nil} : \pi\ list & \text{(empty JSON array)} \\
& | & jp :: x & \text{(JSON array)}
\end{array}$$

This definition is similar to that of Standard ML pattern language, except that variables and nil must be type-annotated to ensure that the pattern corresponds to a unique JSON type.

The above jsonCase expression is translated to a combination of JSON primitives of the core calculus and case expressions of ML. To define the translation scheme, we define the type part $Ty(jp)$ of $jp$ and the pattern part $Pat(jp)$ of $jp$ in Figure 5. JSON case expression of the form

$$
\begin{array}{rcl}
Pat(c^b) & = & c^b \\
Pat(x : \pi) & = & x \\
Pat(\{l_1 = jp_1, \ldots, l_n = jp_n\}) & = & \{l_1 = Pat(jp_1), \ldots, l_n = Pat(jp_n)\} \\
Pat(\{l_1 = jp_1, \ldots, l_n = jp_n, \ldots\}) & = & \{\!|l_1 = Pat(jp_1), \ldots, l_n = Pat(jp_n)|\!\} \\
Pat(\mathsf{nil} : \pi\ list) & = & \mathsf{nil} \\
Pat(jp_1 :: x) & = & Pat(jp_1) :: x \\
\\
Ty(c^b) & = & b \\
Ty(x : \pi) & = & \pi \\
Ty(\{l_1 = jp_1, \ldots, l_n = jp_n\}) & = & \{l_1 : Ty(jp_1), \ldots, l_n : Ty(jp_n)\} \\
Ty(\{l_1 = jp_1, \ldots, l_n = jp_n, \ldots\}) & = & \{\!|l_1 : Ty(jp_1), \ldots, l_n : Ty(jp_n)|\!\} \\
Ty(\mathsf{nil} : \pi\ list) & = & \pi\ list \\
Ty(jp_1 :: x) & = & Ty(jp_1)\ list
\end{array}
$$

**Figure 5** The type-part and the pattern-part of JSON pattern

$$\mathsf{jsonCase}\ \ e\ \ \mathsf{of}\ \ jp_1\ \ \texttt{=>}\ \ e_1\ \ \texttt{|}\ \cdots\ \texttt{|}\ \ jp_n\ \ \texttt{=>}\ \ e_n$$

is translated to the following nested case analysis term $E_1$:

$$
\begin{array}{rcl}
E_1 & = & \mathsf{case}\ \ (e\ \ \mathsf{as}\ \ Ty(jp_1)\ \ \mathsf{else}\ \ E_2)\ \mathsf{of}\ \ Pat(jp_1)\ \ \texttt{=>}\ \ e_1\ \ \texttt{|}\ \ \texttt{\_}\ \ \texttt{=>}\ \ E_2 \\
E_2 & = & \mathsf{case}\ \ (e\ \ \mathsf{as}\ \ Ty(jp_2)\ \ \mathsf{else}\ \ E_3)\ \mathsf{of}\ \ Pat(jp_2)\ \ \texttt{=>}\ \ e_2\ \ \texttt{|}\ \ \texttt{\_}\ \ \texttt{=>}\ \ E_3 \\
& \vdots & \\
E_n & = & \mathsf{case}\ \ (e\ \ \mathsf{as}\ \ Ty(jp_n)\ \ \mathsf{else}\ \ DynamicTypeError) \\
& & \mathsf{of}\ \ Pat(jp_n)\ \ \texttt{=>}\ \ e_n\ \ \texttt{|}\ \ \texttt{\_}\ \ \texttt{=>}\ \ MatchError
\end{array}
$$

where *MatchError* and *DynamicTypeError* are terms to raise exceptions. *DynamicTypeError* indicates that dynamic type-checking fails. *MatchError* indicates that list or constant pattern matching in $jp_i$ fails.

## 4 Implementation

We have implemented all the features presented in the previous section in the SML# compiler version 3.1.0, which is available from: `http://www.pllab.riec.tohoku.ac.jp/smlsharp/`. The JSON features are supported both in the separate compilation mode and in the interactive session. The following is a very simple actual interactive session.

```
$ smlsharp
SML# 3.1.0 ··· for x86_64-pc-linux-gnu with LLVM 3.7.1
# open JSON;
  ··· output message on JSON structure being opened.
# import "{\"name\":\"SML#\", \"version\":\"3.1\"}";
val it = _ : void dyn
```

```
# _jsoncase it of {name=x:string, ...} => x;
val it = "SML#" : string
```

SML# interpreter prompts the user by printing "`#`". `open JSON` makes the primitives defined in the `JSON` structure, such as `import`, available at the top-level. The user can also write `JSON.import` without opening `JSON` structure. Section 5 shows more examples.

Through our effort of extending the full-fledged and complex compiler with JSON support, we have observed that we can implement the required dynamic typing systematically and efficiently if we provide a mechanism for the compiler to access user-level library codes. Based on this observation, we have successfully completed our implementation using user-level SML# codes with relatively small amount of modification to the compiler.

The implementation consists of three components. The first is the library for JSON object manipulation written as user-level codes. The second is typed elaboration that compiles ($e$ as $\pi$ else $e$) using type information. The third is syntactic elaboration that transforms jsonCase expression. The compiler performs the second and the third transformation using the JSON support library through the mechanism to access user-level codes. The compiler support is necessary for two reasons: (1) in both translations, the generated codes are type-varying and therefore the translation codes are not typable, and (2) the typed elaboration requires compile-time static type information.

We report on its details in the following three steps. First, we explain the strategy of typed elaboration that implements the *json* type with its introduction and elimination. Second, we extend the typed elaboration with the partially dynamic records. Last, we describe the techniques we used in implementing the jsonCase translation presented in Subsection 3.6.

## 4.1 Implementation of *json* type

Implementation of the *json* type involves the introduction of the JSON term, denoted by $j$ in the calculus, and the implementation of the ($e_1$ as $\pi$ else $e_2$) statement that eliminates type *json*. We first review the semantics structure defined in the formal calculus with our strategy to implement them, and then describe the details of our implementation.

A semantic object ($j : \pi$) of type *json* is a pair of a JSON term and a JSON type. The structure of $j$ and $\pi$ are represented as ML datatypes. The introduction of a JSON term $j$ of type *json* is implemented as an ML function that parses a given JSON string to obtain a JSON term $j$, infers its JSON type $\pi$ such that $\vdash j : \pi$, and then returns the pair of the parse result and $\pi$.

Those JSON terms are dynamically type-checked and converted to ML value through the construct ($e_1$ as $\pi$ else $e_2$). We provide the syntax

`_json` $e$ `as` $\pi$

for ($e_1$ as $\pi$ else *DynamicTypeError*) where else term $e_2$ is fixed to the one that raise exception. To implement this, we need to *reify* $\pi$. For this purpose, a datatype representation of JSON type $\pi$ is defined in the user-level library and the compiler generates user-level code representing $\pi$ in that representation. This code generation are done by searching for appropriate user-level codes in the library from the given context and inserting references to them. Then, the compiler translates `_json` $e$ `as` $\pi$ into the user-level code that extracts the JSON-type part $\pi'$ from the result of evaluation of $e$, checks whether $\pi' \leq \pi$ holds, and if it succeeds then converts the JSON term of $e$ to a runtime value of type $\pi$. The resulting code constitutes a composition of user-level functions in the library, each of which performs one of the above steps. The same searching mechanism as the reification is used to generate this composition.

In actual implementation, JSON types are represented by the following definition:

```
datatype jsonTy =
    BOOLty | INTty | REALty | STRINGty | NULLty
  | ARRAYty of jsonTy
  | RECORDty of (string * jsonTy) list
  | PARTIALRECORDty of (string * jsonTy) list
  | JSONty
```

To represent unordered record fields, the list of `RECORDty` and `PARTIALRECORDty` are sorted in alphabetical order. `NULLty` is the type of the null value of JSON, whose treatments we omitted in the formal development in Section 3.

We implement the *json* type as a type-annotated JSON term rather than the pair of a JSON term and JSON type as mentioned above, so that the JSON type of any subterms of JSON are computed once JSON is read. The following shows the definition of the type-annotated JSON and type *json* whose name in implementation is `dyn`:

```
datatype json =
    BOOL of bool | INT of int | REAL of real | STRING of string | NULL
  | ARRAY of json list * jsonTy
  | OBJECT of (string * json) list
datatype dyn = DYN of json
```

In addition to the standard JSON term representation, it is sufficient to add a type annotation to the JSON array; the other component carries type information. The data constructor `DYN` is hidden from the user; hence the `dyn` type can be seen as atomic types from the user.

We provide a function of the following signature for the user to read JSON data as a value of type `dyn`:

```
val import : string -> dyn
```

This function parses JSON in the given string and computes the element type of every array appearing in the given JSON.

We define the following user-level functions for the compiler to use during compilation:

```
val getJson : dyn -> json
val checkTy : json * jsonTy -> unit
val checkInt : json -> int
val checkString : json -> int
 ...
val checkArray : json -> json list
```

`getJson` $e$ returns the internal JSON term of internal type `json`. `checkTy(e,` $\pi$`)` extracts the type $\pi'$ of JSON term $e$ and if $\pi' \not\leq \pi$ then it raises `RuntimeTypeError` exception. `checkInt`, `checkString`, `checkArray`, etc., are coercion functions defined for atomic types `int`, `string`, etc., and for type constructors `array` and others. Each coercion function checks the JSON type of a given JSON term and converts it to the corresponding ML value.

Using these user-level functions, we define the following two code generation functions in the compiler: `coerceJson(`$e$`,`$\pi$`)` to call appropriate `check` function to obtain an ML value from a JSON term $e$, and `tyToJsonTy(`$\pi$`)` to convert a static type $\pi$ to a term of type `jsonTy`.

These definitions are straightforward except in the case of partially dynamic records we will mention in the next subsection. `_json` $e$ `as` $\pi$ is compiled by the following code generation function:

```
fun compileJson (e, ty) =
    let
      val jsonExp = App (J.getJson(), [e])
      val viewExp = coerceJson (jsonExp, ty)
      val viewTy = tyToJasonTy ty
      val checkExp = App (J.checkTy(), [jsonExp, viewTy])
    in
      Seq [checkExp, Typed (viewExp, ty)]
    end
```

In this code, `App`, `Seq`, and `Typed` generates application term, sequencing term, and type annotated term, respectively. `J.getJson()` and `J.checkTy()` are references to the user-level functions of the same name through the searching mechanism mentioned above.

## 4.2 Implementation of partially dynamic records

The approach presented in the previous subsection is extended to partially dynamic records. In the calculus presented in Section 3, a partially dynamic record is regarded as a value of type dynamic coupled with its view. To deal with partially dynamic records and dynamic values uniformly, we extend `dyn` type to `'a dyn` where `'a` is the type of the view. We additionally introduce a new type `void` indicating that no view is available; an attempt to view a term of type `void dyn` will result in `AttemptToReturnVOIDValue` runtime exception. With these extensions, *json* is represented as `void dyn`, and $\{\!|l_1 : \tau_1, \ldots, l_n : \tau_n|\!\}$ is represented as `{`$l_1$`:`$\tau_1$`, ..., `$l_n$`:`$\tau_n$`} dyn`. For example, to coerce a JSON object $e$ to a list of partially dynamic records of type $\{\!|$`name` : `string`$|\!\}$, the programmer can write the following:

```
_json e as {name: string} dyn list
```

then an ML list of partially dynamic records is obtained by generating the view of each element of the list.

One concern of this approach is performance; the calculus produces a view for a partially dynamic record every time a partially dynamic record is generated. Naive implementation would waste time and memory by producing unused views when reading a large JSON array consisting of a variety of JSON objects. To avoid this overhead, we delay the generation of views until the view is really needed. The definition of `dyn` is eventually as follows:

```
datatype 'a dyn = DYN of (json -> 'a) * json
```

This `'a` is to be instantiated to either a record type or `void`. The function of type `json -> 'a` produces an ML record of type `'a` as a view of the given JSON term. Partially dynamic records are then converted to ML records by the following function:

```
val view : 'a dyn -> 'a = fn DYN (f, x) => f x
```

The case of `coerceJson` for partially dynamic record types is given below:

```
fun coerceJson (jsonExp, DYNty argTy) =
    let
    val funExp = Fn (fn x => coerceJson (Var x, argTy))
    val jsonTy = tyToJsonTy argTy
  in
    App (J.makeCoerce(), [jsonExp, jsonTy, funExp])
  end
```

`Fn` is a compiler primitive to convert a meta-level function to the corresponding object-level function term. `makeCoerce : json -> jsonTy -> (json -> 'a) -> 'a dyn` is a user-level function that takes json term $j$, its type, and a coercion function, and generates a partially dynamic term.

## 4.3 Implementation of pattern matching with JSON

We have implemented the following syntax that embodies jsonCase presented in Subsection 3.6:

$$\texttt{\_jsoncase}\ e\ \texttt{of}\ jp_1\ \texttt{=>}\ e_1\ \texttt{|}\ \cdots\ \texttt{|}\ jp_n\ \texttt{=>}\ e_n$$

with several shorthands available in ML's pattern language. Moreover, along with the syntax of JSON, we allow any string literals to occur in $jp$ as labels of record fields. An example of string literals as record labels appears in Subsection 5.2.

The elaboration scheme presented in Subsection 3.6 cannot straightforwardly incorporate our implementation strategy owing to the following issues: (1) the ML record pattern does not match with partially dynamic records since partially dynamic records have their own data constructors, (2) the construction of the view of a partially dynamic record is delayed by a function, and (3) the code duplication in the present elaboration scheme increases the code size exponentially. A standard solution to these issues would be to develop a match compilation algorithm for jsonCase, similarly to ML's pattern matching compilation.

Instead of taking the full-fledged approach of developing a match compiler, we adopt the following light-weight strategy. We translate `_jsoncase` into a nested `case` expression that interleaves pattern matching with view construction of partially dynamic records. Let $m$ be the number of (top-level) partially dynamic record patterns of the form $\{l_j^1 = p_j^1, \ldots, l_j^k = p_j^k, \ldots\}$ $(1 \leq j \leq m)$ occurring in $jp_i$. The result $\mathcal{T}(e, jp_i\ \texttt{=>}\ e_i)$ of translation of match $jp_i\ \texttt{=>}\ e_i$ is the nested `case` expression of the form

$$\texttt{case}\ e\ \texttt{of}\ \overline{jp_i}\ \texttt{=>}\ E_1\ \texttt{|}\ \texttt{\_}\ \texttt{=>}\ \texttt{raise}\ M$$

where $\overline{jp_i}$ is the pattern obtained by replacing the $j$-th partially dynamic record pattern with a fresh variable $x_j$, $M$ is the exception indicating the fact that the other cases should be tried. The main expression $E_1$ is generated by the following cascading equations:

$$\begin{array}{rcl}
E_1 & = & \texttt{case view}\ x_1\ \texttt{of}\ \{l_1^1\texttt{=}x_1^1\texttt{,}\ \ldots\texttt{,}\ l_1^k\texttt{=}x_1^k\}\ \texttt{=>} \\
& & \quad \mathcal{T}(x_1^1, jp_1^1\ \texttt{=>}\ \mathcal{T}(\cdots\ \mathcal{T}(x_1^k, jp_1^k\ \texttt{=>}\ E_2)\cdots)\cdots) \\
& & \texttt{|}\ \texttt{\_}\ \texttt{=>}\ \texttt{raise}\ M \\
& \vdots & \\
E_m & = & \texttt{case view}\ x_m\ \texttt{of}\ \{l_m^1\texttt{=}x_m^1\texttt{,}\ \ldots\texttt{,}\ l_m^l\texttt{=}x_m^l\}\ \texttt{=>} \\
& & \quad \mathcal{T}(x_m^1, jp_m^1\ \texttt{=>}\ \mathcal{T}(\cdots\ \mathcal{T}(x_m^1, jp_m^l\ \texttt{=>}\ e_i)\cdots)\cdots) \\
& & \texttt{|}\ \texttt{\_}\ \texttt{=>}\ \texttt{raise}\ M
\end{array}$$

The fallback to the next match is realized by handling the exception $M$. The entire `_jsoncase` is translated into the following expression:

```
let
  val $x$ = $e$
  exception $M$
in
  $\mathcal{T}$(_json $x$ as $Ty(jp_1)$, $jp_1$ => $e_1$) handle $M$ =>
       ⋮
  $\mathcal{T}$(_json $x$ as $Ty(jp_n)$, $jp_n$ => $e_n$) handle $M$ =>
  raise Match
end
```

In addition to patterns presented in Subsection 3.6, we have implemented a pattern for heterogeneous lists of the following form:

$$jp \quad ::= \quad \cdots \mid jp_1 \texttt{ :: } \cdots \texttt{ :: } jp_n \texttt{ :: } x$$

This pattern matches with the first $n$ elements of the given JSON array and binds $x$ to the rest of the array. In contrast to ML's list pattern, the type of each element $jp_i$ of a heterogeneous list pattern may differ. The matching with this heterogeneous list pattern is performed by coercing the given JSON term to the term of type `void dyn list`, taking $n$ elements from the head of the list, and then matching the $i$-th element with $jp_i$. To realize this, our implementation translates the following case:

```
_jsoncase $e_1$ of $jp_1$ :: ⋯ :: $jp_n$ :: $x$ => $e_2$
```

into the following expression:

```
case (_json $e_1$ as void dyn list) of
  $x_1$::⋯::$x_n$::$x$ =>
    _jsoncase $x_1$ of $jp_1$ =>
        ⋮
      _jsoncase $x_n$ of $jp_n$ => $e_2$
```

where $x_1, \ldots, x_n$ are fresh variables.

## 5 Evaluations through realistic examples

In this section, we demonstrate the feasibility and the usefulness of our approach through realistic examples using existing Web APIs.

### 5.1 Heterogeneous record collections in JSON

We first go through the basic usage of our JSON extension using simple examples.

A typical usage of JSON is to represent a collection of objects. The following example imports a JSON array of objects, coerces it to a list of records, and extracts their `name` fields:

```
val J = "[{\"name\":\"Joe\", \"age\":21, \"grade\":1.1},
         {\"name\":\"Sue\", \"age\":31, \"grade\":2.0},
         {\"name\":\"Bob\", \"age\":41, \"grade\":3.9}]"
fun getNames l = map #name l
val j = import J
val vl = _json j as {name:string, age:int, grade:real} list
val nl = getNames vl
```

```
# val J = "[{\"name\":\"Joe\", \"age\":21, \"grade\":1.1},\
          \ {\"name\":\"Sue\", \"age\":31, \"grade\":2.0},\
          \ {\"name\":\"Bob\", \"age\":41, \"grade\":3.9}]";
val J =
  "[{\"name\":\"Joe\", \"age\":21, \"grade\":1.1},
    {\"name\":\"Sue\", \"age\":31, \"grade\":2.0},
    {\"name\":\"Bob\", \"age\":41, \"grade\":3.9}]" : string
# fun getNames l = map #name l;
val getNames = fn : ['a#{name: 'b}, 'b. 'a list -> 'b list]
# val j = import J;
val j = _ : void dyn
# val vl = _json j as {name:string, age:int, grade:real} list;
val vl =
  [
   {age = 21,grade = 1.1,name = "Joe"},
   {age = 31,grade = 2.0,name = "Sue"},
   {age = 41,grade = 3.9,name = "Bob"}
  ] : {age: int, grade: real, name: string} list
# val nl = getNames vl;
val nl = ["Joe","Sue","Bob"] : string list
```

**Figure 6** Interactive SML# session with a simple example

Figure 6 shows the actual output of an interactive session of this program, where the `JSON` structure has been already opened at the top-level. The SML# compiler type-checks this example and generates the expected results. Some explanations of the compiler output are in order.

- For `getNames`, the compiler infers the polymorphic type `['a#{name:'b}, 'b. 'a list -> 'b list]`.
- The result `j` of `import`ing the JSON string `J` is given the type `void dyn`, which is the representation of type *json* in our implementation.
- The result `vl` of coercing `j` to type `{name:string, age:int, grade:real} list` is a list of records of that type, as expected.
- For the resulting records bound to `vl`, the polymorphic function `getName` is safely applied to yield a list of `string` bound to `nl`.

In writing a JSON string as an ML string constant, control characters such as `"` need to be escaped to `\"`. This is not a big problem, given the fact that JSON objects are usually obtained from external servers or are generated by a program, and there are few occasions to define a JSON object as a string constant. The above artificial example is there for explanation purposes. In writing the JSON string examples below, we omit the out-most `"` and escape characters.

As we discussed in Introduction, collections of JSON objects in typical web services are usually heterogeneous; some fields of each object in the collection may be optional. Suppose a JSON API requires that a collection of JSON objects have mandatory `name` and `age` fields, and optional `grade` and `nickname` fields. Let $J'$ be the following JSON term compliant with this format:

```
[
 {"name":"Alice", "age":10, "nickname":"Allie"},
 {"name":"Dinah", "age":3, "grade":2.0},
 {"name":"Puppy", "age":7}
]
```

In our scheme, we can type such a heterogeneous collection with partially dynamic records as follows:

```
val j' = import J'
val vl' = _json j' as {name:string, age:int} dyn list
val nl' = getNames (map view vl')
```

While the inferred type of `j'` is `void dyn` as in the previous example, the SML# compiler infers the following typing for `vl'`:

```
val vl' = _ : {name:string, age:int} dyn list
```

This shows that a JSON object containing a heterogeneous collection is typed as a list of partially dynamic records. We note that the record-polymorphic function `getName` can be applied to the resulting partially dynamic records.

When some optional fields are significant, then we can *enlarge* the type of each object separately. The following function picks up either a `nickname` or `name` for each record in the given list depending on the existence of `nickname` field.

```
fun getFriendlyName vl =
    map (fn x => _jsoncase x of {nickname=y:string, ...} => y
                              | _ => #name (view x))
        vl
```

The pattern `{nickname=y:string, ...}` enlarges the partial record type of $x$ to $\{\!|\texttt{nickname} : \texttt{string}, \ldots |\!\}$ and binds the `nickname` field to $y$ if the enlargement succeeds. This type of enlargement does not affect the type of $x$. Owing to SML#'s record polymorphism, `getFriendlyName` has a record-polymorphic type

```
['a#{name:string}. 'a dyn list -> string]
```

indicating that this function can be applied to any partial record that has at least a `name` field of `string` type.

## 5.2 Partial records in the real world

As we briefly mentioned above, collections of partially dynamic records frequently appear in JSON, typically in communication among web services. Examples include the OAuth authorization protocol [13], Twitter search API [28], and Google Maps API Web services [12]. Most other popular web services also provide JSON interfaces.

This subsection demonstrates the benefits of our proposal using a sample in the real world. For this purpose, we choose OpenWeatherMap [23], which is simple but elaborate enough for our purpose of realistic evaluations. OpenWeatherMap provides free access to weather data over the Internet. JSON is adopted by OpenWeatherMap as a format for sending weather data to the client.

By sending a specific HTTP request to OpenWeatherMap's web server with some parameters, the user can obtain a variety of collections of weather data as an JSON array of JSON objects. Each of the objects consists of fields for weather parameters measured by a weather station such as temperature, wind speed, and amount of precipitation. The concrete form of the weather data may vary for several reasons: the precipitation amount may be absent if it does not rain, the geographic details of cities may be omitted if the user requests data for a particular city, and a record may contain extra system-reserved parameters that are undocumented and would be ignored by clients. Owing to such flexibility of the data structure, the response data is inherently heterogeneous and partial. The following JSON data is an example of the server response consisting of current weather parameters of several cities in Japan:

```
{"list":[
 {"id":2111149, "name":"Sendai-shi", "sys":{···},
  "coord":{"lon":140.87, "lat":38.27}, "weather":[{"main":"Rain",···}],
  "main":{"temp":273.538,···}, "clouds":{"all":92}, "rain":{"3h":1},
  "wind":{"speed":0.45,···}, "dt":1424968138},
 {"id":1857910, "name":"Kyoto", "sys":{···},
  "coord":{"lon":135.75, "lat":35.02}, "weather":[{"main":"Clear",···}],
  "main":{"temp":275.887,···}, "clouds":{"all":8},
  "wind":{"speed":5.21,···}, "dt":1424968420},
 ···],···}
```

In this example, the record of `Sendai-shi` has a `rain` field but `Kyoto` does not, owing to clear weather in `Kyoto`.

A convenient way to read this weather data in SML# is to write the partial record type that specifies only the significant fields. User $A$, who is interested only in the names and temperatures of cities, would write the following code:

```
_json e as {list:{name:string, main:{temp:real} dyn} dyn list} dyn
```

while another user $B$, who is interested in other parameters, would write different types of nested partially dynamic records. In our language, the user can control the structure of JSON data to be retrieved in a comfortable, flexible, and type-safe manner. This maximizes the flexibility of programming with JSON objects.

Partially dynamic records and record polymorphism are useful in dealing with this kind of heterogeneous data collection. Consider the following function that calculates the average temperature:

```
fun avgTemp l =
    foldl (op +) 0.0 (map (#temp o view o #main o view) l)
    / real (length l)
```

This function has a record-polymorphic type

```
['a#{main:'b dyn}, 'b#{temp:real}. 'a dyn list -> real]
```

indicating that `avgTemp` is independent of the detail of the structure of weather data. Thus the user $A$ and others who are interested at least in temperature can share this function. Another way to implement this computation is to use `_jsoncase` on dynamic values.

```
fun avgTemp' l =
  foldl (op +) 0.0
    (map (fn x => _jsoncase x of {main={temp:real,...},...} => temp) l)
  / real (length l)
```

This function returns the same value as `avgTemp` but has type `['a. 'a dyn list -> real]`, which is different from that of `avgTemp`. This new typing indicates that `avgTemp'` can be applied to any dynamic data. While `avgTemp'` is more flexible than `avgTemp`, `avgTemp'` may raise a runtime exception owing to dynamic coercion or match failure, whereas `avgTemp` never fails at runtime. The user may choose an appropriate style of programming for dynamic values according to his/her intention.

The flexibility provided by `_jsoncase` is particularly useful when we want to write codes to access an optional field. The following example lists the names of the cities where it rains more heavily than a given threshold.

```
fun rainyCities t cityListInJson =
    List.mapPartial (fn x => if (_jsoncase x of
                                   {rain={"3h"=y:int}, ...} => y > t
                                 | _ => false)
                             then SOME (#name (view x))
                             else NONE)
                    cityListInJson
```

In this example, `{"3h"=⋯}` is the record pattern that matches with JSON objects consisting of a single `3h` field. Again, this function has a record-polymorphic type, which precisely represents the behavior of the function, similar to the example of `getFriendlyName` in Section 5.1.

Another typical factor that introduces heterogeneousness in practical JSON is number literal. Along with the nature of JavaScript, an integer value (without a fraction part) of JSON may be interpreted as a floating-point value: `5` of JSON usually means either `5` of integers or `5.0` of floating-point numbers. This is also the case for OpenWeatherMap according to its documentation. To deal with this, one would read JSON numbers as `void dyn` as follows:

```
val j = _json e as ⋯main:{temp:void dyn} dyn⋯
```

and write the following function that coerces JSON number as `real`:

```
fun getNumAsReal x =
    _jsoncase x of
      (n:int) => Real.fromInt n
    | (r:real) => r
    | _ => 0.0 / 0.0  (* not a number *)
```

We believe that the above examples demonstrate strongly that our JSON extension is beneficial and useful in practical software development with JSON objects including applications with web services.

During the evaluations, we also found a few issues worth considering for further improvements of JSON programming. One is the treatment of JSON numbers. We note that `getNumAsReal` example above is not entirely satisfactory in the spirit of typeful programming since the types of both `j` and `getNumAsReal` are much looser than the user's intention. One possible refinement to our framework is to introduce an additional ad-hoc ordering relation `int` $\leq$ `real` on JSON types. Another more accurate solution would be to introduce a new type `num` with the ordering relations `int` $\leq$ `num` and `real` $\leq$ `num`. Another possible refinement is to merge `_jsoncase` with ML `case` so that the user can interleave partially dynamic record patterns and ML value patterns. For instance, in the example of `rainyCities`, if the user could write a mixture of the pattern for partially dynamic records `{rain={3h:int}, ...}` and the field selection `#name` at the argument position of `fn`, it would be a more intuitive and convenient shorthand for ML programmers. One approach is to redesign the ML pattern language to integrate JSON patterns and refine the ML pattern matching algorithm with JSON cases. We believe both refinements are feasible, and we would like to investigate them in future.

On the whole, we conclude that our proposal provides a promising basis for the seamless integration of JSON manipulation in ML.

## 6 Related works

There are a number of libraries for serializing/deserializing JSON objects in statically typed languages such as YoJson [16], Aeson [24], and Atdgen [15]. There are also meta-level tools and frameworks to generate application-specific classes and types for processing JSON objects such as `type_conv` of Camlp4 [8] and `JsonProvider` of F# 3.0 [27]. In these approaches, however, JSON objects and their associated functions are not part of the type system of the underlying programming languages. As a result, they do not achieve type-safe and seamless programming with JSON objects.

In the general perspective of developing typing discipline for implicitly typed data formats, our work is related to a number of works that investigate type structures of XML and RDF. XDuce and CDuce type systems [14, 3] represent static structures of XML based on regular expression types. Frisch et al. [11] investigates semantic subtyping as a basis for CDuce-like type systems. These type systems are more powerful than ours, and a number of features such recursive structures and (tag-less) unions are cleanly represented. Perhaps owing to their flexibility, however, regular expression types are not related to static data structures in a type system of a polymorphic programming language. As we have mentioned in Introduction, our goal and contribution is to represent JSON structures using labeled record types in ML so that they can be directly manipulated in ML using polymorphic record operations.

Regarding JSON objects, Colazzo et al. [7] recently presented a type inference algorithm for large JSON data-sets using union types and subtyping. Compared with the partial record types we have used, their approach could potentially find more accurate types for a large collection of heterogeneous JSON objects. It is, however, not immediately obvious whether their JSON type system can be integrated in a static type system of a programming language.

Our approach is based on partially dynamic records [6], whose central idea is to represent a type of a heterogeneous collection as a record structure common to all components in the collection. From this perspective, it is related to gradual typing proposed by Siek and Taha [25]. Their approach uses subtyping and dynamic typing. This combination can represent dynamically typed heterogeneous collections in a statically typed language. Largely based on this observation, Bierman et al. [4] gave a formal account for the core of TypeScript. This approach is also adopted in Flow [9]. As we commented in the Introduction, however, subtyping would complicate polymorphism and type inference, and its impact on the implementation method remains to be investigated. Kiselyov et al. [19] presented an implementation technique to encode heterogeneous lists by exploiting Haskell type classes and their extensions including multi-parameter classes and functional dependencies. Its type theoretical property is, however, not well investigated, and its type theoretical properties and their relationship to ours are not clear.

## 7 Conclusions and further investigation

JSON is a widely accepted format for data exchange over the Internet, especially among web services. In this paper, we have developed a typed programming language for seamless and high-level manipulation of JSON data. The major obstacle to the seamless introduction of JSON manipulation to a typed language is the heterogeneous nature of JSON. We have presented a typed calculus to deal with heterogeneous JSON objects based on partially dynamic records, and have established its type soundness. The proposed calculus has been fully implemented as an extension to the SML# compiler. In the extended language, the programmer can directly import a JSON object as a term of static data structure composed of labeled records and lists with the full benefits of static polymorphic type-cheking. The implementation also provides `_jsoncase` syntax for ML-style pattern matching. We have demonstrated the feasibility and the benefits of our approach through examples that interact with actual web services.

The important work we are now planning to conduct is to put our proposed language into serious software development in industry, and to evaluate its benefits in software development. From a more technical perspective, we plan to investigate programming language support for manipulating JSON-based databases. We would like to have a declarative query language for a large and complex JSON data, and would like to integrate it into the SML# compiler. A possible approach toward such an integration is to extend the type system proposed in this paper and the integration of SQL in a programming language with the record polymorphism we reported in [21]. Another interesting future work is to extend our technique to data formats similar to JSON, such as RDF and XML.

## Acknowledgments

The authors thank the members of the project “Research and Development on Highly Functional and Highly Available Information Storage Technology” sponsored by the Ministry of Education, Culture, Sports, Science and Technology in Japan for discussions.

The authors also thank the anonymous reviewers for careful reading of the paper and a number of constructive comments.

## References

1 M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

2 M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. *ACM Transcations on Programming Languages and Systems*, 13(2):237–268, 1991.

3 V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general-purpose language. In *Proceedings of the ACM International Conference on Functional Programming*, pages 51–63, 2003.

4 G. Bierman, M. Abadi, and M. Torgersen. Understanding TypeScript. In *Proceedings of the European conference on Object-Oriented Programming*, pages 257–281. Springer Berlin Heidelberg, 2014.

5 T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159 (Proposed Standard), 2014. URL: `http://www.ietf.org/rfc/rfc7159.txt`.

6 P. Buneman and A. Ohori. Polymorphism and type inference in database programming. *ACM Transactions on Database Systems*, 21(1):30–74, 1996.

7 D. Colazzo, G. Ghelli, and C. Sartiani. Typing massive JSON datasets. In *Proceedings of the International Workshop on Cross-model Language Design and Implementation (XLDI)*, Copenhagen, Denmark, 2012.

8 J. Dimino. Camlp4, 2014. URL: `https://opam.ocaml.org/packages/camlp4/camlp4.4.03.0/`.

9 Flow | a static typechecker for JavaScript. URL: `http://flowtype.org`.

10 A. Frisch. OCaml + XDuce. In *Proceedings of the ACM International Conference on Functional Programming*, pages 192–200, 2006.

11 A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM*, 55(4), 2008.

12 Google Maps APIs | Google Developers. URL: `https://developers.google.com/maps/`.

13 D. Hardt. The OAuth 2.0 Authorization Framework. RFC 6749, 2012. URL: `http://www.ietf.org/rfc/rfc6749.txt`.

14 H. Hosoya and B. Pierce. XDuce: A statically typed XML processing language. *ACM Trans. Internet Technol.*, 3(2):117–148, 2003.

15 M. Jambon. Atdgen, 2010. URL: `https://github.com/mjambon/atdgen`.

16 M. Jambon. Yojson: JSON library for OCaml, 2010-2012. URL: `https://github.com/mjambon/yojson`.

17 JSON schema. URL: `http://json-schema.org`.

18 G. Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer Verlag, 1987.

19 O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 96–107, 2004.

20 A. Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, 1995. A preliminary summary appeared at ACM POPL, 1992 under the title “A compilation method for ML-style polymorphic record calculi.”.

21 A Ohori and K. Ueno. Making Standard ML a practical database programming language. In *Proceedings of the ACM International Conference on Functional Programming*, pages 307–319, 2011.

22 A. Ohori, K. Ueno, K. Hoshi, S. Nozaki, T. Sato, T. Makabe, and Y. Ito. SML# in industry: A practical ERP system development. In *Proceedings of the ACM International Conference on Functional Programming*, pages 167–173, 2014.

23 OpenWeatherMap.org. OpenWeatherMap, 2012-2016. URL: `http://openweathermap.org`.

24 B. O’Sullivan. Aeson: Fast JSON parsing and encoding, 2011-2014. URL: `https://hackage.haskell.org/package/aeson`.

25 J. Siek and W. Taha. Gradual typing for objects. In *Proceedings of the European conference on Object-Oriented Programming*, pages 2–27, 2007.

26 SML#, 2006-2016. URL: `http://www.riec.tohoku.ac.jp/smlsharp/`.

27 D. Syme, K. Battocchi, K. Takeda, D. Malayeri, J. Fisher, J. Hu, T. Liu, B. McNamara, D. Quirk, M. Taveggia, W. Chae, U. Matsveyeu, and T. Petricek. F#3.0 - strongly-typed language support for internet-scale information sources. Technical Report MSR-TR-2012-101, Microsoft Research, 2012.

28 The Search API | Twitter Developers. URL: `https://dev.twitter.com/rest/public/search`.