Static and Dynamic Semantics of NoSQL Languages

Véronique Benzaken[1] Giuseppe Castagna[2] Kim Nguyễn[1] Jérôme Siméon[3]

[1]LRI, Université Paris-Sud, Orsay, France,
[2]CNRS, PPS, Université Paris Diderot, Sorbonne Paris Cité, Paris, France
[3]IBM Watson Research, Hawthorne, NY, USA

Abstract

We present a calculus for processing semistructured data that spans differences of application area among several novel query languages, broadly categorized as "NoSQL". This calculus lets users define their own operators, capturing a wider range of data processing capabilities, whilst providing a typing precision so far typical only of primitive hard-coded operators. The type inference algorithm is based on semantic type checking, resulting in type information that is both precise, and flexible enough to handle structured and semistructured data. We illustrate the use of this calculus by encoding a large fragment of Jaql, including operations and iterators over JSON, embedded SQL expressions, and co-grouping, and show how the encoding directly yields a typing discipline for Jaql as it is, namely without the addition of any type definition or type annotation in the code.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Operational semantics; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure; H.3.3 [*Information Storage and Retrieval*]: Information Search and Retrieval—Query formulation

Keywords NoSQL, BigData Analytics, Jaql, Cloud Computing, Type Inference.

1. Introduction

The emergence of Cloud computing, and the ever growing importance of data in applications, has given birth to a whirlwind of new data models [17, 22] and languages. Whether they are developed under the banner of "NoSQL" [28, 33], for BigData Analytics [6, 16, 26], for Cloud computing [4], or as domain specific languages (DSL) embedded in a host language [19, 25, 30], most of them share a common subset of SQL and the ability to handle semistructured data. While there is no consensus yet on the precise boundaries of this class of languages, they all share two common traits: *(i)* an emphasis on sequence operations (*eg*, through the popular MapReduce paradigm) and *(ii)* a lack of types for both data and programs (contrary to, say, XML programming or relational databases where data schemas are pervasive). In [19, 20], Meijer argues that such languages can greatly benefit from formal foundations, and suggests comprehensions [8, 31, 32] as a unifying model. Although we agree with Meijer for the need to provide unified, formal foundations to those new languages, we argue that such foundations should account for novel features critical to various application domains that are not captured by comprehensions. Also, most of those languages provide limited type checking, or ignore it altogether. We believe type checking is essential for many applications, with usage ranging from error detection to optimization. But we understand the designers and programmers of those languages who are averse to any kind of type definition or annotation. In this article, we propose a calculus which is expressive enough to capture languages that go beyond SQL or comprehensions. We show how the calculus adapts to various data models while retaining a precise type checking that can exploit in a flexible way limited type information, information that is deduced directly from the structure of the program even in the absence of any explicit type declaration or annotation.

Example. We use Jaql [6, 16], a language over JSON [17] developed for BigData analytics, to illustrate how our proposed calculus works. Our reason for using Jaql is that it encompasses all the features found in the previously cited query languages and includes a number of original ones, as well. Like Pig [26] it supports sequence iteration, filtering, and grouping operations on non-nested queries. Like AQL [4] and XQuery [7], it features nested queries. Furthermore, Jaql uses a rich data model that allows arbitrary nesting of data (it works on generic sequences of JSON records whose fields can contain other sequences or records) while other languages are limited to flat data models, such as AQL whose data-model is similar to the standard relational model used by SQL databases (tuples of scalars and of lists of scalars). Lastly, Jaql includes SQL as an embedded sub-language for relational data. For these reasons, although in the present work we focus almost exclusively on Jaql, we believe that our work can be adapted without effort to a wide array of sequence processing languages.

The following Jaql program illustrates some of those features. It performs co-grouping [26] between one JSON input, containing information about departments, and one relational input containing information about employees. The query returns for each department its name and id, from the first input, and the number of high-income employees from the second input. A SQL expression is used to select the employees with income above a given value, while a Jaql filter is used to access the set of departments and the elements of these two collections are processed by the group expression (in Jaql "$" denotes the current element).

```
group
  (depts -> filter each x (x.size > 50))
        by g = $.depid  as ds,
  (SELECT * FROM employees WHERE income > 100)
        by g = $.dept as es
into { dept: g,
       deptName: ds[0].name,
       numEmps: count(es) };
```

The query blends Jaql expressions (*eg*, `filter` which selects, in the collection `depts`, departments with a `size` of more than `50` employees, and the grouping itself) with a SQL statement (select-

This work was partially supported by the ANR TYPEX project n. ANR-11-BS02-007 and by a visiting researcher grant of "Fondation Digiteo".

POPL '13, January 23–25, 2013, Rome, Italy.

ing employees in a relational table for which the salary is more than 100). Relations are naturally rendered in JSON as collections of records. In our example, one of the key difference is that field access in SQL requires the field to be present in the record, while the same operation in Jaql does not. Actually, field selection in Jaql is very expressive since it can be applied also to collections with the effect that the selection is recursively applied to the components of the collection and the collection of the results returned, and similarly for `filter` and other iterators. In other words, the expression `filter each x (x.size > 50)` above will work as much when `x` is bound to a record (with or without a `size` field: in the latter case the selection returns `null`), as when `x` is bound to a collection of records or of arbitrary nested collections thereof. This accounts for the semistructured nature of JSON compared to the relational model. Our calculus can express both, in a way that illustrates the difference in both the dynamic semantics and static typing.

In our calculus, the selection of all records whose *mandatory* field `income` is greater than 100 is defined as:

```
let Sel =
    `nil => `nil
  | ({income: x, .. } as y , tail) =>
       if x > 100 then (y,Sel(tail)) else Sel(tail)
```

(collections are encoded as lists *à la* Lisp) while the filtering among records or arbitrary nested collections of records of those where the (optional) `size` field is present and larger than 50 is:

```
let Fil =
    `nil => `nil
  | ({size: x, .. } as y,tail) =>
       if x > 50 then (y,Fil(tail)) else Fil(tail)
  | ((x,xs),tail) =>  (Fil(x,xs),Fil(tail))
  | (_,tail) => Fil(tail)
```

The terms above show nearly all the basic building blocks of our calculus (only composition is missing), building blocks that we dub *filters*. Filters can be defined recursively (*eg*, `Sel(tail)` is a recursive call); they can perform pattern matching as found in functional languages (the filter $p \Rightarrow f$ executes f in the environment resulting from the matching of pattern p); they can be composed in alternation ($f_1|f_2$ tries to apply f_1 and if it fails it applies f_2), they can spread over the structure of their argument (*eg*, (f_1,f_2) —of which `(x,Sel(tail))` is an instance— requires an argument of a product type and applies the corresponding f_i component-wise).

For instance, the filter `Fil` scans collections encoded as lists *à la* Lisp (*ie*, by right associative pairs with `` `nil `` denoting the empty list). If its argument is the empty list, then it returns the empty list; if it is a list whose head is a record with a `size` field (and possibly other fields matched by "`..`"), then it captures the whole record in `y`, the content of the field in `x`, the tail of the list in `tail`, and keeps or discards `y` (*ie*, the record) according to whether `x` (*ie*, the field) is larger than 50; if the head is also a list, then it recursively applies both on the head and on the tail; if the head of the list is neither a list, nor a record with a size field, then the head is discarded. The encoding of the whole grouping query is given in Section 5.1.

Our aim is not to propose yet another "NoSQL/cloud computing/bigdata analytics" query language, but rather to show how to *express* and *type* such languages via an encoding into our core calculus. Each such language can in this way preserve its execution model but obtain for free a formal semantics, a type inference system and, as it happens, a prototype implementation. The type information is deduced via the encoding (without the need of *any* type annotation) and can be used for early error detection and debugging purposes. The encoding also yields an executable system that can be used for rapid prototyping. Both possibilities are critical in most typical usage scenarios of these languages, where deployment is very expensive both in time and in resources. As observed by Meijer [19] the advent of big data makes it more important than ever for programmers (and, we add, for language and system designers) to have a single abstraction that allows them to process, transform, query, analyze, and compute across data presenting utter variability both in volume and in structure, yielding a "mind-blowing number of new data models, query languages, and execution fabrics" [19] . The framework we present here, we claim, encompasses them all. A long-term goal is that the compilers of these languages could use the type information inferred from the encoding and the encoding itself to devise further optimizations.

Types. Pig [26], Jaql [16, 27], AQL [4] have all been conceived by considering just the map-reduce execution model. The type (or, schema) of the manipulated data did not play any role in their design. As a consequence these languages are untyped and, when present, types are optional and clearly added as an afterthought. Differences in data model or type discipline are particularly important when embedded in a host language (since they yield the so-called impedance mismatch). The reason why types were/are disregarded in such languages may originate in an alleged tension between type inference and heterogeneous/semistructured data: on the one hand these languages are conceived to work with collections of data that are weakly or partially structured, on the other hand current languages with type inference (such as Haskell or ML) can work only on homogeneous collections (typically, lists of elements of the same type).

In this work we show that the two visions can coexist: we type data by semantic subtyping [15], a type system conceived for semistructured data, and describe computations by our *filters* which are untyped combinators that, thanks to a technique of weak typing introduced in [10], can polymorphically type the results of data query and processing with a high degree of precision. The conception of *filters* is driven by the schema of the data rather than the execution model and we use them (i) to capture and give a uniform semantics to a wide range of semi structured data processing capabilities, (ii) to give a type system that encompasses the types defined for such languages, if any, notably Pig, Jaql and AQL (but also XML query and processing languages: see Section 5.1), (iii) to infer the precise result types of queries written in these languages *as they are* (so without the addition of any explicit type annotation/definition or new construct), and (iv) to show how minimal extensions/modifications of the current syntax of these languages can bring dramatic improvements in the precision of the inferred types.

The types we propose here are extensible record types and heterogeneous lists whose content is described by regular expressions on types as defined by the following grammar:

Types	t	::=	v	(singleton)
		|	$\{\ell{:}t, \ldots, \ell{:}t\}$	(closed record)
		|	$\{\ell{:}t, \ldots, \ell{:}t\,\texttt{,..}\}$	(open record)
		|	$[r]$	(sequences)
		|	`int` | `char`	(base)
		|	`any` | `empty` | `null`	(special)
		|	$t\vert t$	(union)
		|	$t\backslash t$	(difference)

Regexp r ::= $\varepsilon \mid t \mid r\texttt{*} \mid r\texttt{+} \mid r\texttt{?} \mid r\,r \mid r\vert r$

where ε denotes the empty word. The semantics of types can be expressed in terms of sets of *values* (values are either constants —such as `1`, `2`, `true`, `false`, `null`, `'1'`, the latter denoting the character `1`—, records of values, or lists of values). So the singleton type v is the type that contains just the value v (in particular `null` is the singleton type containing the value `null`). The closed record type $\{\texttt{a:int}, \texttt{b:int}\}$ contains all record values with exactly two fields `a` and `b` with integer values, while the open record type $\{\texttt{a:int}, \texttt{b:int}\,\texttt{,..}\}$ contains all record values with *at least* two fields `a` and `b` with integer values. The sequence type $[r]$ is the set

of all sequences whose content is described by the *regular expression* r; so, for example `[char*]` contains all sequences of characters (we will use `string` to denote this type and the standard double quote notation to denote its values) while `[({a:int} {a:int})+]` denotes nonempty lists of even length containing record values of type `{a:int}`. The union type $s|t$ contains all the values of s and of t, while the difference type $s\backslash t$ contains all the values of s that are not in t. We shall use `bool` as an abbreviation of the union of the two singleton types containing true and false: `'true|'false`. `any` and `empty` respectively contain all and no values. Recursive type definitions are also used (see Section 2.2 for formal details).

These types can express all the types of Pig, Jaql and AQL, all XML types, and much more. So for instance, AQL includes only homogeneous lists of type t, that can be expressed by our types as `[ t* ]`. In Jaql's documentation one can find the type `[ long(value=1), string(value="a"), boolean* ]` which is the type of arrays whose first element is 1, the second is the string "a" and all the other are booleans. This can be easily expressed in our types as `[1 "a" bool*]`. But while Jaql only allows a limited use of regular expressions (Kleene star can only appear in tail position) our types do not have such restrictions. So for example `[char* '@' char* '.' (('f' 'r')|('i' 't'))]` is the type of all strings (*ie*, sequences of chars) that denote email addresses ending by either `.fr` or `.it`. We use some syntactic sugar to make terms as the previous one more readable (*eg*, `[ .* '@' .* ('.fr'|'.it')]`). Likewise, henceforth we use `{a?:`t`}` to denote that the field `a` of type t is optional; this is just syntactic sugar for stating that either the field is undefined or it contains a value of type t (for the formal details of this encoding see the full version of this work available on line).

Coming back to our initial example, the filter `Fil` defined before expects as argument a collection of the following type:

```
type Depts = [ ( {size?: int, ..} | Depts )* ]
```

that is a, possibly empty, arbitrary nested list of records with an optional `size` field of type `int`: notice that it is important to specify the optional field and its type since a `size` field of a different type would make the expression `x > 50` raise a run-time error. This information is deduced just from the *structure* of the filter (since `Fil` does not contain any type definition or annotation).

We define a type inference system that rejects any argument of `Fil` that has not type `Depts`, and deduces for arguments of type `[({size: int, addr: string}| {sec: int} | Depts)+]` (which is a subtype of `Depts`) the result type `[({size: int, addr: string}|Depts)*]` (so it does not forget the field `addr` but discards the field `sec`, and by replacing `*` for `+` recognizes that the test may fail).

By encoding primitive Jaql operations into a formal core calculus we shall provide them a formal and clean semantics as well as precise typing. So for instance it will be clear that applying the following dot selection `[ [{a:3}] {a:5, b:true} ].a` the result will be `[ [3] 5 ]` and we shall be able to deduce that `_.a` applied to arbitrary nested lists of records with an optional integer `a` field (*ie*, of type t `= {a?:int} | [ t* ]`) yields arbitrary nested lists of `int` or `null` values (*ie*, of type u `= int | null | [ u* ]`).

Finally we shall show that if we accept to extend the current syntax of Jaql (or of some other language) by some minimal filter syntax (*eg*, the pattern filter) we can obtain a huge improvement in the precision of type inference.

Contributions. The main contribution of this work is the definition of a calculus that encompasses structural operators scattered over NoSQL languages and that possesses some characteristics that make it unique in the swarm of current semi-structured data processing languages. In particular it is parametric (though fully embeddable) in a host language; it uniformly handles both width and deep nested data recursion (while most languages offer just the former and limited forms of the latter); finally, it includes first-class arbitrary deep composition (while most languages offer this operator only at top level), whose power is nevertheless restrained by the type system.

An important contribution of this work is that it directly compares a programming language approach with the tree transducer one [13]. Our calculus implements transformations typical of top-down tree transducers but has several advantages over the transducer approach: (1) the transformations are expressed in a formalism immediately intelligible to any functional programmer; (2) our calculus, in its untyped version, is Turing complete; (3) its transformations can be statically typed (at the expenses of Turing completeness) without any annotation yielding precise result types (4) even if we restrict the calculus only to well-typed terms (thus losing Turing completeness), it still is strictly more expressive than well-known and widely studied deterministic top-down tree transducer formalisms.

The technical contributions are (i) the proof of Turing completeness for our formalism, (ii) the definition of a type system that copes with records with computable labels (iii) the definition of a static type system for filters and its correctness, (iv) the definition of a static analysis that ensures the termination (and the proof thereof) of the type inference algorithm with complexity bounds expressed in the size of types and filters and (iv) the proof that the terms that pass the static analysis form a language strictly more expressive than top-down tree transducers.

Outline. In Section 2 we present the syntax of the three components of our system. Namely, a minimal set of *expressions*, the calculus of *filters* used to program user-defined operators or to encode the operators of other languages, and the core *types* in which the types we just presented are to be encoded. Section 3 defines the operational semantics of filters and a declarative semantics for operators. The type system as well as the type inference algorithm are described in Section 4. In Section 5 we present how to handle a large subset of Jaql. Section 6 reports on some subtler design choices of our system and compare with related work. For space constraints, proofs, secondary results, encodings, some formal definitions (in particular the definition of the static analysis for termination and the interpretation of record values as *quasi-constant* functions), and further extensions can be found only in the full version available online.

2. Syntax

In this section we present the syntax of the three components of our system: a minimal set of *expressions*, the calculus of *filters* used to program user-defined operators or to encode the operators of other languages, and the core *types* in which the types presented in the introduction are to be encoded.

The core of our work is the definition of filters and types. The key property of our development is that filters can be grafted to any host language that satisfies minimal requirements, by simply adding filter application to the expressions of the host language. The minimal requirements of the host language for this to be possible are quite simple: it must have constants (typically for types `int`, `char`, `string`, and `bool`), variables, and either pairs or record values (not necessarily both). On the static side the host language must have at least basic and products types and be able to assign a type to expressions in a given type environment (*ie*, under some typing assumptions for variables). By the addition of filter applications, the host language can acquire or increase the capability to define polymorphic user-defined iterators, query and processing expressions, and be enriched with a powerful and precise type system.

2.1 Expressions

In this work we consider the following set of expressions

Definition 1 (expressions).

Exprs	e	::=	c	(constants)
		|	x	(variables)
		|	(e, e)	(pairs)
		|	$\{e{:}e, ..., e{:}e\}$	(records)
		|	$e + e$	(record concatenation)
		|	$e \setminus \ell$	(field deletion)
		|	$\mathsf{op}(e, ..., e)$	(built-in operators)
		|	fe	(filter application)

where f ranges over *filters* (defined later on), c over generic constants, and ℓ over *string* constants.

Intuitively, these expressions represent the syntax supplied by the host language —though only the first two and one of the next two are really needed— that we extend with (the missing expressions and) the expression of filter application. Expressions are formed by constants, variables, pairs, records, and operation on records: record concatenation gives priority to the expression on the right. So if in $r_1 + r_2$ both records contains a field with the same label, it is the one in r_2 that will be taken, while field deletion does not require the record to contain a field with the given label (though this point is not important). The metavariable op ranges over operators as well as functions and other constructions belonging to or defined by the host language. Among expressions we single out a set of *values*, intuitively the results of computations, that are formally defined as follows:

$$v \quad ::= \quad c \mid (v, v) \mid \{\ell{:}v;\ \ldots;\ \ell{:}v\}$$

We use `"foo"` for character string constants, `'c'` for characters, `1 2 3 4 5` and so on for integers, and backquoted words, such as `` `foo ``, for atoms (*ie*, user-defined constants). We use three distinguished atoms `` `nil ``, `` `true ``, and `` `false ``. Double quotes can be omitted for strings that are labels of record fields: thus we write {`name:"John"`} rather than {`"name":"John"`}. Sequences (*aka*, heterogeneous lists, ordered collections, arrays) are encoded *à la LISP*, as nested pairs where the atom `` `nil `` denotes the empty list. We use `[`$e_1\ \ldots\ e_n$`]` as syntactic sugar for $(e_1, \ldots, (e_n, \texttt{`nil})...)$.

2.2 Types

Definition 2 (types).

Types	t	::=	b	(basic types)
		|	v	(singleton types)
		|	$\mathbf{(}t\mathbf{,}t\mathbf{)}$	(products)
		|	$\mathbf{\{}\ell\mathbf{:}t, \ldots, \ell\mathbf{:}t\mathbf{\}}$	(closed records)
		|	$\mathbf{\{}\ell\mathbf{:}t, \ldots, \ell\mathbf{:}t\mathbf{,..\}}$	(open records)
		|	$t\mathbf{\|}t$	(union types)
		|	$t\mathbf{\&}t$	(intersection types)
		|	$\neg t$	(negation type)
		|	`empty`	(empty type)
		|	`any`	(any type)
		|	$\mu T.t$	(recursive types)
		|	T	(recursion variable)
		|	$\mathsf{Op}(t, ..., t)$	(foreign type calls)

where every recursion is guarded, that is, every type variable is separated from its binder by at least one application of a type constructor (*ie*, products, records, or Op).

Most of these types were already explained in the introduction. We have basic types (`int`, `bool`, ...) ranged over by b and singleton types v denoting the type that contains only the value v. Record types come in two flavors: closed record types whose values are records with exactly the fields specified by the type, and open record types whose values are records with *at least* the fields specified by the type. Product types are standard and we have a complete set of type connectives, that is, finite unions, intersections and negations. We use `empty`, to denote the type that has no values and `any` for the type of all values (sometimes denoted by "`_`" when used in patterns). We added a term for recursive types, which allows us to encode both the regular expression types defined in the introduction and, more generally, the recursive type definitions we used there. Finally, we use Op (capitalized to distinguish it from expression operators) to denote the host language's *type* operators (if any). Thus, when filter applications return values whose type belongs just to the foreign language (*eg*, a list of functions) we suppose the typing of these functions be given by some type operators. For instance, if succ is a user defined successor function, we will suppose to be given its type in the form Arrow(`int`,`int`) and, similarly, for its application, say apply(succ,3) we will be given the type of this expression (presumably `int`). Here Arrow is a type operator and apply an expression operator.

The denotational semantics of types as sets of values, that we informally described in the introduction, is at the basis of the definition of the subtyping relation for these types. We say that a type t_1 is a subtype of a type t_2, noted $t_1 \leq t_2$, if and only if the set of values denoted by t_1 is contained (in the set-theoretic sense) in the set of values denoted by t_2. For the formal definition and the decision procedure of this subtyping relation the reader can refer to the work on semantic subtyping [15].

2.3 Patterns

Filters are our core untyped operators. All they can do are three different things: (1) they can structurally decompose and transform the values they are applied to, or (2) they can be sequentially composed, or (3) they can do pattern matching. In order to define filters, thus, we first need to define patterns.

Definition 3 (patterns).

Patterns	p	::=	t	(type)
		|	x	(variable)
		|	$\mathbf{(}p\mathbf{,}p\mathbf{)}$	(pair)
		|	$\mathbf{\{}\ell\mathbf{:}p, \ldots, \ell\mathbf{:}p\mathbf{\}}$	(closed rec)
		|	$\mathbf{\{}\ell\mathbf{:}p, \ldots, \ell\mathbf{:}p\mathbf{,..\}}$	(open rec)
		|	$p\mathbf{\|}p$	(or/union)
		|	$p\mathbf{\&}p$	(and/intersection)

where the subpatterns forming pairs, records, and intersections have distinct capture variables, and those forming unions have the same capture variables.

Patterns are essentially types in which capture variables (ranged over by x, y, ...) may occur in every position that is not under a negation or a recursion. A pattern is used to match a value. The matching of a value v against a pattern p, noted v/p, either fails (noted Ω) or it returns a substitution from the variables occurring in the pattern, into values. The substitution is then used as an environment in which some expression is evaluated. If the pattern is a type, then the matching fails if and only if the pattern is matched against a value that does not have that type, otherwise it returns the empty substitution. If it is a variable, then the matching always succeeds and returns the substitution that assigns the matched value to the variable. The pair pattern $\mathbf{(}p_1\mathbf{,}p_2\mathbf{)}$ succeeds if and only if it is matched against a pair of values and each sub-pattern succeeds on the corresponding projection of the value (the union of the two substitutions is then returned). Both record patterns are similar to the product pattern with the specificity that in the open record pattern "**..**" matches all the fields that are not specified in the pattern. An intersection pattern $p_1\mathbf{\&}p_2$ succeeds if and only if both patterns succeed (the union of the two substitutions is then

returned). The union pattern $p_1|p_2$ first tries to match the pattern p_1 and if it fails it tries the pattern p_2.

For instance, the pattern $(\texttt{int\&}x,y)$ succeeds only if the matched value is a pair of values (v_1, v_2) in which v_1 is an integer —in which case it returns the substitution $\{x/v_1, y/v_2\}$— and fails otherwise. Finally notice that the notation "p `as` x" we used in the examples of the introduction, is syntactic sugar for $p\&x$.

This informal semantics of matching (see [15] for the formal definition) explains the reasons of the restrictions on capture variables in Definition 3: in intersections, pairs, and records all patterns must be matched and, thus, they have to assign distinct variables, while in union patterns just one pattern will be matched, hence the same set of variables must be assigned, whichever alternative is selected.

The strength of patterns is their connections with types and the fact that the pattern matching operator can be typed *exactly*. This is entailed by the following theorems (both proved in [15]):

Theorem 4 (Accepted type [15]). *For every pattern p, the set of all values v such that $v/p \neq \Omega$ is a type. We call this set the* accepted type *of p and note it by $\lfloor p \rceil$.*

The fact that the exact set of values for which a matching succeeds is a type is not obvious. It states that for every pattern p there exists a syntactic type produced by the grammar in Definition 2 whose semantics is exactly the set of all and only values that are matched by p. The existence of this syntactic type, which we note $\lfloor p \rceil$, is of utmost importance for a precise typing of pattern matching. In particular, given a pattern p and a type t contained in (*ie*, subtype of) $\lfloor p \rceil$, it allows us to compute the *exact* type of the capture variables of p when it is matched against a value in t:

Theorem 5 (Type environment [15]). *There exists an algorithm that for every pattern p, and $t \leq \lfloor p \rceil$ returns a type environment $t/p \in \mathbf{Vars}(p) \to \mathbf{Types}$ such that $(t/p)(x) = \{(v/p)(x) \mid v : t\}$.*

2.4 Filters

Definition 6 (filters). A *filter* is a term generated by:

Filters	f	::=	e	(expression)
		|	$p \Rightarrow f$	(pattern)
		|	(f,f)	(product)
		|	$\{\ell{:}f, \ldots, \ell{:}f\,,..\}$	(record)
		|	$f\|f$	(union)
		|	$\boldsymbol{\mu}X.f$	(recursion)
		|	Xa	(recursive call)
		|	$f;f$	(composition)
		|	o	(declarative operators)
Operators	o	::=	`groupby` f	(filter grouping)
		|	`orderby` f	(filter ordering)
Arguments	a	::=	x	(variables)
		|	c	(constants)
		|	(a,a)	(pairs)
		|	$\{\ell{:}a, ..., \ell{:}a\}$	(record)

such that for every subterm of the form $f;g$, no recursion variable is free in f.

Filters are like transducers, that when applied to a value return another value. However, unlike transducers they possess more "programming-oriented" constructs, like the ability to test an input and capture subterms, recompose an intermediary result from captured values and a composition operator. We first describe informally the semantics of each construct.

The expression filter e always returns the value corresponding to the evaluation of e (and discards its argument). The filter $p \Rightarrow f$ applies the filter f to its argument in the environment obtained by matching the argument against p (provided that the matching does not fail). This rather powerful feature allows a filter to perform two critical actions: *(i)* inspect an input with regular pattern-matching before exploring it and *(ii)* capture part of the input that can be reused during the evaluation of the subfilter f. If the argument application of f_i to v_i returns v_i' then the application of the product filter (f_1,f_2) to an argument (v_1, v_2) returns (v_1', v_2'); otherwise, if any application fails or if the argument is not a pair, it fails. The record filter is similar: it applies to each specified field the corresponding filter and, as stressed by the "$..$", leaves the other fields unchanged; it fails if any of the applications does, or if any of the specified fields is absent, or if the argument is not a record. The filter $f_1|f_2$ returns the application of f_1 to its argument or, if this fails, the application of f_2. The semantics of a recursive filter is given by standard unfolding of its definition in recursive calls. The only real restriction that we introduce for filters is that recursive calls can be done only on arguments of a given form (*ie*, on arguments that have the form of values where variables may occur). This restriction in practice amounts to forbid recursive calls on the result of another recursively defined filter (all other cases can be easily encoded). The reason of this restriction is technical, since it greatly simplifies the analysis of Section 4.4 (which ensures the termination of type inference) without hampering expressiveness: filters are Turing complete even with this restriction (see Theorem 7). Filters can be composed: the filter $f_1;f_2$ applies f_2 to the result of applying f_1 to the argument and fails if any of the two does. The condition that in every subterm of the form $f;g$, f does not contain free recursion variables is not strictly necessary. Indeed, we could allow such terms. The point is that the analysis for the termination of the typing would then reject all such terms (apart from trivial ones in which the result of the recursive call is not used in the composition). But since this restriction *does not* restrict the expressiveness of the calculus (Theorem 7 proves Turing completeness with this restriction), then the addition of this restriction is just a design (rather than a technical) choice: we prefer to forbid the programmer to write recursive calls on the left-hand side of a composition, than systematically reject all the programs that use them in a non-trivial way.

Finally, we singled out some specific filters (specifically, we chose `groupby` and `orderby`) whose semantics is generally specified in a declarative rather than operational way. These do not bring any expressive power to the calculus (the proof of Turing completeness, Theorem 7, does not use these declarative operators) and actually they can be encoded by the remaining filters, but it is interesting to single them out because they yield either simpler encodings or more precise typing.

3. Semantics

The operational semantics of our calculus is given by the reduction semantics for filter application and for the record operations. Since the former is the only novelty of our work, we save space and omit the latter, which are standard anyhow.

We define a big step operational semantics for filters. The definition is given by the inference rules in Figure 1 for judgments of the form $\delta;\gamma \vdash_{eval} f(a) \rightsquigarrow r$ and describes how the evaluation of the application of filter f to an argument a in an environment γ yields an object r where r is either a value or Ω. The latter is a special value which represents a runtime error: it is raised by the rule **(error)** either because a filter did not match the form of its argument (*eg*, the argument of a filter product was not a pair) or because some pattern matching failed (*ie*, the side condition of **(patt)** did not hold). Notice that the argument a of a filter is always a value v unless the filter is the unfolding of a recursive

(expr) $$\frac{}{\delta;\gamma \vdash_{eval} e(v) \rightsquigarrow r} \quad r = \texttt{eval}(\gamma, e)$$

(prod) $$\frac{\delta;\gamma \vdash_{eval} f_1(v_1) \rightsquigarrow r_1 \quad \delta;\gamma \vdash_{eval} f_2(v_2) \rightsquigarrow r_2}{\delta;\gamma \vdash_{eval} (f_1,f_2)(v_1, v_2) \rightsquigarrow (r_1, r_2)} \quad \begin{array}{l}\text{if } r_1 \neq \Omega \\ \text{and } r_2 \neq \Omega\end{array}$$

(patt) $$\frac{\delta;\gamma\ ,\ v/p \vdash_{eval} f(v) \rightsquigarrow r}{\delta;\gamma \vdash_{eval} (p \Rightarrow f)(v) \rightsquigarrow r} \quad \text{if } v/p \neq \Omega$$

(comp) $$\frac{\delta;\gamma \vdash_{eval} f_1(v) \rightsquigarrow r_1 \quad \delta;\gamma \vdash_{eval} f_2(r_1) \rightsquigarrow r_2}{\delta;\gamma \vdash_{eval} (f_1;f_2)(v) \rightsquigarrow r_2} \quad \text{if } r_1 \neq \Omega$$

(union1) $$\frac{\delta;\gamma \vdash_{eval} f_1(v) \rightsquigarrow r_1}{\delta;\gamma \vdash_{eval} (f_1|f_2)(v) \rightsquigarrow r_1} \quad \text{if } r_1 \neq \Omega$$

(union2) $$\frac{\delta;\gamma \vdash_{eval} f_1(v) \rightsquigarrow \Omega \quad \delta;\gamma \vdash_{eval} f_2(v) \rightsquigarrow r_2}{\delta;\gamma \vdash_{eval} (f_1|f_2)(v) \rightsquigarrow r_2}$$

(rec) $$\frac{\delta, (X \mapsto f);\gamma \vdash_{eval} f(v) \rightsquigarrow r}{\delta;\gamma \vdash_{eval} (\boldsymbol{\mu} X.f)(v) \rightsquigarrow r}$$

(rec-call) $$\frac{\delta;\gamma \vdash_{eval} (\delta(X))(a) \rightsquigarrow r}{\delta;\gamma \vdash_{eval} (Xa)(v) \rightsquigarrow r}$$

(error) $$\frac{}{\delta;\gamma \vdash_{eval} f(a) \rightsquigarrow \Omega} \quad \text{if no other rule applies}$$

(recd) $$\frac{\delta;\gamma \vdash_{eval} f_1(v_1) \rightsquigarrow r_1 \quad \cdots \quad \delta;\gamma \vdash_{eval} f_n(v_n) \rightsquigarrow r_n}{\delta;\gamma \vdash_{eval} \{\ell_1{:}f_1, ..., \ell_n{:}f_n\texttt{,..}\}(\{\ell_1{:}v_1, ..., \ell_n{:}v_n, ..., \ell_{n+k}{:}v_{n+k}\}) \rightsquigarrow \{\ell_1{:}r_1, ..., \ell_n{:}r_n, ..., \ell_{n+k}{:}v_{n+k}\}} \quad \text{if } \forall i, r_i \neq \Omega$$

Figure 1. Dynamic semantics of filters

call, in which case variables may occur in it (*cf.* rule **rec-call**). Environment δ is used to store the body of recursive definitions.

The semantics of filters is quite straightforward and inspired by the semantics of patterns. The *expression* filter discards its input and evaluates (rather, asks the host language to evaluate) the expression e in the current environment (**expr**). It can be thought of as the right-hand side of a branch in a `match_with` construct.

The *product* filter expects a pair as input, applies its sub-filters component-wise and returns the pair of the results (**prod**). This filter is used in particular to express sequence mapping, as the first component f_1 transforms the element of the list and f_2 is applied to the tail. In practice it is often the case that f_2 is a recursive call that iterates on arbitrary lists and stops when the input is `'nil`. If the input is not a pair, then the filter fails (rule (**error**) applies).

The *record* filter expects as input a record value with *at least* the same fields as those specified by the filter. It applies each sub-filter to the value in the corresponding field leaving the contents of other fields unchanged (**recd**). If the argument is not a record value or it does not contain all the fields specified by the record filter, or if the application of any subfilter fails, then the whole application of the record filter fails.

The *pattern* filter matches its input value v against the pattern p. If the matching fails so the filter does, otherwise it evaluates its sub-filter in the environment augmented by the substitution v/p (**patt**).

The *alternative* filter follows a standard first-match policy: If the filter f_1 succeeds, then its result is returned (**union-1**). If f_1 fails, then f_2 is evaluated against the input value (**union-2**). This filter is particularly useful to write the alternative of two (or more) *pattern* filters, making it possible to conditionally continue a computation based on the shape of the input.

The *composition* allows us to pass the result of f_1 as input to f_2. The composition filter is of paramount importance. Indeed, without it, our only way to iterate (deconstruct) an input value is to use a *product* filter, which always rebuilds a pair as result.

Finally, a *recursive* filter is evaluated by recording its body in δ and evaluating it (**rec**), while for a *recursive call* we replace the recursion variable by its definition (**rec-call**).

This concludes the presentation of the semantics of non-declarative filters (*ie*, without groupby and orderby). These form a Turing complete formalism (complete proof in the full version):

Theorem 7 (Turing completeness). *The language formed by constants, variables, pairs, equality, and applications of non-declarative filters is Turing complete.*

Proof (sketch). We can encode untyped call-by-value λ-calculus by first applying continuation passing style (CPS) transformations and encoding CPS term reduction rules and substitutions via filters. Thanks to CPS we eschew the restrictions on composition. □

To conclude the presentation of the semantics we have to define the semantics of groupby and orderby. We prefer to give the semantics in a declarative form rather than operationally in order not to tie it to a particular order (of keys or of the execution):

Groupby: `groupby` f applied to a sequence $[v_1 \ldots v_m]$ reduces to a sequence $[\ (k_1, l_1) \ldots (k_n, l_n)\ ]$ such that:

1. $\forall i,\ 1 \leq i \leq m,\ \exists j,\ 1 \leq j \leq n,$ s.t. $k_j = f(v_i)$
2. $\forall j,\ 1 \leq j \leq n,\ \exists i,\ 1 \leq i \leq m,$ s.t. $k_j = f(v_i)$
3. $\forall j,\ 1 \leq j \leq n, l_j$ is a sequence: $[\ v_1^j \ldots\ v_{n_j}^j]$
4. $\forall j,\ 1 \leq j \leq n, \forall k,\ 1 \leq k \leq n_j, f(v_k^j) = k_j$
5. $k_i = k_j \Rightarrow i = j$
6. $l_1, \ldots, l_n$ is a partition of $[v_1 \ldots\ v_m\ ]$

Orderby: `orderby` f applied to $[v_1 \ldots v_n]$ reduces to $[v'_1 \ldots v'_n]$ such that:

1. $[v'_1 \ldots v'_n]$ is a permutation of $[v_1 \ldots v_n]$,
2. $\forall i, j$ s.t. $1 \leq i \leq j \leq n, f(v_i) \leq f(v_j)$

Since the semantics of both operators is deeply connected to a notion of equality and order on values of the host language, we give them as "built-in" operations. However we will illustrate how our type algebra allows us to provide very precise typing rules, specialized for their particular semantics. It is also possible (see full version) to encode co-grouping (or `groupby` on several input sequences) with a combination of `groupby` and filters.

Syntactic sugar. The reader may have noticed that the productions for expressions (Definition 1) do not define any destructor (*eg*, projections, label selection, ...), just constructors. The reason is that destructors, as well as other common expressions, can be encoded by filter applications:

$$\begin{array}{ll}
e.\ell & \stackrel{\text{def}}{=} (\{\ell{:}x\texttt{,..}\} \Rightarrow x)e \\
\texttt{fst}(e) & \stackrel{\text{def}}{=} ((x\texttt{,any}) \Rightarrow x)e \\
\texttt{snd}(e) & \stackrel{\text{def}}{=} ((\texttt{any},x) \Rightarrow x)e \\
\texttt{let}\ p\texttt{=}e_1\ \texttt{in}\ e_2 & \stackrel{\text{def}}{=} (p \Rightarrow e_2)e_1 \\
\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 & \stackrel{\text{def}}{=} (\texttt{'true} \Rightarrow e_1|\texttt{'false} \Rightarrow e_2)e \\
\texttt{match}\ e\ \texttt{with}\ p_1 \Rightarrow e_1 | \ldots | p_n \Rightarrow e_n & \\
\qquad & \stackrel{\text{def}}{=} (p_1 \Rightarrow e_1| \ldots |p_n \Rightarrow e_n)e
\end{array}$$

These are just a possible choice, but others are possible. For instance in Jaql dot selection is overloaded: when $\_.\ell$ is applied to a record, Jaql returns the content of its ℓ field; if the field is absent or the argument is `null`, then Jaql returns `null` and fails if the argument is not a record; when applied to a list ('array' in Jaql terminology) it recursively applies to all the elements of the list. So Jaql's "$\_.\ell$" is precisely defined as

$$\boldsymbol{\mu} X.(\{\ell{:}x\texttt{,..}\} \Rightarrow x\ \texttt{|}\ (\{\texttt{..}\}|\texttt{null}) \Rightarrow \texttt{null}\ \texttt{|}\ (h,t) \Rightarrow (Xh,Xt))$$

Besides the syntactic sugar above, in the next section we will use $t_1 + t_2$ to denote the record type formed by all field types in t_2 and all the field types in t_1 whose label is not already present in t_2. Similarly $t \setminus \ell$ will denote the record types formed by all field types in t apart from the one labeled by ℓ, if present. Finally, we will also use for expressions, types, and patterns the syntactic sugar for lists used in the introduction. So, for instance, $[p_1\ p_2\ ...\ p_n]$ is matched by lists of n elements provided that their i-th element matches p_i.

4. Type inference

The type inference system assign types to *expressions*. Variables, constants, and pairs are typed by standard rules, while we suppose that the typing of foreign expressions is provided by the host language.[1] So we omit the corresponding rules (they can be found in the full version). The core of our type system starts with records.

4.1 Typing of records

The typing of records is novel and challenging because record expressions may contain string *expressions* in label position, such as in $\{e_1\!:\!e_2\}$, while in all type systems for record we are aware of, labels are never computed. It is difficult to give a type to $\{e_1\!:\!e_2\}$ since, in general, we do not statically know the value that e_1 will return, and which is required to form a record type. All we can (and must) ask is that this value will be a string. To type a record expression $\{e_1\!:\!e_2\}$, thus, we distinguish two cases according to whether the type t_1 of e_1 is finite (*ie*, it contains only finitely many values, such as, say, `Bool`) or not. If a type is finite, (finiteness of regular types seen as tree automata can be decided in polynomial time [11]), then it is possible to write it as a finite union of values (actually, of singleton types). So consider again $\{e_1\!:\!e_2\}$ and let t_1 be the type of e_1 and t_2 the type of e_2. First, t_1 must be a subtype of `string` (since record labels are strings). So if t_1 is finite it can be expressed as $\ell_1|\cdots|\ell_n$ which means that e_1 will return the string ℓ_i for some $i \in [1..n]$. Therefore $\{e_1\!:\!e_2\}$ will have type $\{\ell_i : t_2\}$ for some $i \in [1..n]$ and, thus, the union of all these types, as expressed by the rule [RCD-FIN] below. If t_1 is infinite instead, then all we can say is that it will be a record with some (unknown) labels, as expressed by rule [RCD-INF].

$$\text{[RCD-FIN]}\quad \frac{\Gamma \vdash e : \ell_1|\cdots|\ell_n \qquad \Gamma \vdash e' : t}{\Gamma \vdash \{e\!:\!e'\} : \{\ell_1\!:\!t\}|\cdots|\{\ell_n\!:\!t\}}$$

$$\text{[RCD-INF]}\quad \frac{\Gamma \vdash e : t \qquad \Gamma \vdash e' : t'}{\Gamma \vdash \{e\!:\!e'\} : \{..\}} \quad \begin{array}{l} t \leq \texttt{string} \\ t \text{ is infinite} \end{array}$$

$$\text{[RCD-MUL]}\quad \frac{\Gamma \vdash \{e_1\!:\!e'_1\} : t_1 \quad \cdots \quad \Gamma \vdash \{e_n\!:\!e'_n\} : t_n}{\Gamma \vdash \{e_1\!:\!e'_1, \ldots, e_n\!:\!e'_n\} : t_1 + \cdots + t_n}$$

$$\text{[RCD-CONC]}\quad \frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 + e_2 : t_1 + t_2} \; \begin{array}{l} t_1 \leq \{..\} \\ t_2 \leq \{..\} \end{array} \qquad \text{[RCD-DEL]}\quad \frac{\Gamma \vdash e : t}{\Gamma \vdash e \setminus \ell : t \setminus \ell} \; t \leq \{..\}$$

Records with multiple fields are handled by the rule [RCD-MUL] which "merges" the result of typing single fields by using the type operator $+$ as defined in $\mathbb{C}$Duce [5, 14], which is a right-priority record concatenation defined to take into account undefined and unknown fields: for instance, $\{a\!:\!\texttt{int}, b\!:\!\texttt{int}\} + \{a\texttt{?}\!:\!\texttt{bool}\} = \{a\!:\!\texttt{int}|\texttt{bool}, b\!:\!\texttt{int}\}$; unknown fields in the right-hand side may override known fields of the left-hand side, which is why, for instance, we have $\{a\!:\!\texttt{int}, b\!:\!\texttt{bool}\} + \{b\!:\!\texttt{int}, ..\} = \{b\!:\!\texttt{int}, ..\}$; likewise, for every record type t (*ie*, for every t subtype of $\{..\}$) we have $t + \{..\} = \{..\}$. Finally, [RCD-CONC] and [RCD-DEL] deal with record concatenation and field deletion, respectively, in a straightforward way: the only constraint is that all expressions must have a record type (*ie*, the constraints of the form $... \leq \{..\}$). See the full version for formal definitions of all these type operators.

Notice that these rules do not ensure that a record will not have two fields with the same label, which is a run-time error. Detecting such an error needs sophisticated type systems (*eg*, dependent types) beyond the scope of this work. This is why in the rule [RCD-MUL] we used type operator "+" which, in case of multiple occurring labels, since records are unordered, corresponds to randomly choosing one of the types bound to these labels: if such a field is selected, it would yield a run-time error, so its typing can be ambiguous. We can fine tune the rule [RCD-MUL] so that when all the t_i are finite unions of record types, then we require to have pairwise disjoint sets of labels; but since the problem would still persist for infinite types we prefer to retain the current, simpler formulation.

4.2 Typing of filter application

Filters are not first-class: they can be applied but not passed around or computed. Therefore we do not assign types to filters but, as for any other expression, we assign types to *filter applications*. The typing rule for filter application

$$\text{[FILTER-APP]}\quad \frac{\Gamma \vdash e : t \qquad \Gamma\,;\varnothing\,;\varnothing \vdash_{\mathit{fil}} f(t) : s}{\Gamma \vdash fe : s}$$

relies on an auxiliary deduction system for judgments of the form $\Gamma\,;\Delta\,;M \vdash_{\mathit{fil}} f(t) : s$ that states that if in the environments Γ, Δ, M (explained later on) we apply the filter f to a value of type t, then it will return a result of type s.

To define this auxiliary deduction system, which is the core of our type analysis, we first need to define $\wr f \int$, the type accepted by a filter f. Intuitively, this type gives a necessary condition on the input for the filter not to fail:

Definition 8 (Accepted type). Given a filter f, the *accepted type* of f, written $\wr f \int$ is the set of values defined by:

$$\begin{array}{lll@{\qquad}lll}
\wr e \int & = & \texttt{any} & \wr Xa \int & = & \texttt{any} \\
\wr p \Rightarrow f \int & = & \wr p \int \,\&\, \wr f \int & \wr \boldsymbol{\mu} X.f \int & = & \wr f \int \\
\wr f_1|f_2 \int & = & \wr f_1 \int \,|\, \wr f_2 \int & \wr \texttt{groupby}\ f \int & = & [\texttt{any*}] \\
\wr (f_1,f_2) \int & = & (\wr f_1 \int, \wr f_2 \int) & \wr \texttt{orderby}\ f \int & = & [\texttt{any*}] \\
\wr f_1;f_2 \int & = & \wr f_1 \int & & & \\
\multicolumn{6}{l}{\wr \{\ell_1\!:\!f_1,..,\ell_n\!:\!f_n, ..\} \int \;=\; \{\ell_1\!:\wr f_1 \int,..,\ell_n\!:\wr f_2 \int, ..\}}
\end{array}$$

It is easy to show that an argument included in the accepted type is a necessary (but not sufficient, because of the cases for composition and recursion) condition for the evaluation of a filter not to fail:

Lemma 9. *Let f be a filter and v be a value such that $v \notin \wr f \int$. For every γ, δ, if $\delta;\gamma \vdash_{\mathit{eval}} f(v) \rightsquigarrow r$, then $r \equiv \Omega$.*

The last two auxiliary definitions we need are related to product and record types. In the presence of unions, the most general form for a product type is a finite union of products (since intersections distribute on products). For instance consider the type

$$(\texttt{int},\texttt{int})|(\texttt{string},\texttt{string})$$

This type denotes the set of pairs for which either both projections are `int` or both projections are `string`. A type such as

$$(\texttt{int}|\texttt{string},\texttt{int}|\texttt{string})$$

is less precise, since it also allows pairs whose first projection is an `int` and second projection is a `string` and *vice versa*. We see that it is necessary to manipulate finite unions of products (and similarly for records), and therefore, we introduce the following notations:

[1] Notice that our expressions, whereas they include *filter* applications, do not include applications of expressions to expressions. Therefore if the host language provides function definitions, then the applications of the host language must be dealt as foreign expressions, as well (*cf.* `apply` in §2.2).

Lemma 10 (Product decomposition). *Let $t \in$* **Types** *such that $t \leq$* **(any,any)**. *A* product decomposition *of t, denoted by $\boldsymbol{\pi}(t)$ is a set of types:*

$$\boldsymbol{\pi}(t) = \{(t_1^1, t_2^1), \ldots, (t_1^n, t_2^n)\}$$

such that $t = \bigvee_{t_i \in \boldsymbol{\pi}(t)} t_i$. For a given product decomposition, we say that n is the rank *of t, noted $\mathtt{rank}(t)$, and use the notation $\boldsymbol{\pi}_i^j(t)$ for the type t_i^j.*

There exist several suitable decompositions whose details are out of the scope of this article. We refer the interested reader to [14] and [21] for practical algorithms that compute such decompositions for any subtype of **(any,any)** or of **{..}**. These notions of decomposition, rank and projection can be generalized to records:

Lemma 11 (Record decomposition). *Let $t \in$* **Types** *such that $t \leq$* **{..}**. *A* record decomposition *of t, denoted by $\boldsymbol{\rho}(t)$ is a finite set of types $\boldsymbol{\rho}(t)$=$\{r_1, \ldots, r_n\}$ where each r_i is either of the form $\{\ell_1^i : t_1^i, \ldots, \ell_{n_i}^i : t_{n_i}^i\}$ or of the form $\{\ell_1^i : t_1^i, \ldots, \ell_{n_i}^i : t_{n_i}^i, ..\}$ and such that $t = \bigvee_{r_i \in \boldsymbol{\rho}(t)} r_i$. For a given record decomposition, we say that n is the* rank *of t, noted $\mathtt{rank}(t)$, and use the notation $\boldsymbol{\rho}_\ell^j(t)$ for the type of label ℓ in the j^{th} component of $\boldsymbol{\rho}(t)$.*

In our calculus we have three different sets of variables. The set **Vars** of term variables, ranged over by $x, y, ...$, introduced in patterns and used in expressions and in arguments of calls of recursive filters. The set **RVars** of term recursion variables, ranged over by $X, Y, ...$ and that are used to define recursive filters. The set **TVars** of type recursion variables, ranged over by $T, U, ...$ used to define recursive types. In order to use them we need to define three different environments: Γ : **Vars** $\rightarrow$ **Types** denoting *type environments* that associate term variables with their types; Δ : **RVars** $\rightarrow$ **Filters** denoting *definition environments* that associate each filter recursion variable with the body of its definition; M : **RVars** $\times$ **Types** $\rightarrow$ **TVars** denoting *memoization environments* which record that the call of a given recursive filter on a given type yielded the introduction of a fresh recursion type variable. Our typing rules, thus work on judgments of the form $\Gamma ; \Delta ; M \vdash f(t) : t'$ stating that applying f to an expression of type t in the environments Γ, Δ, M yields a result of type t'. This judgment can be derived with the set of rules given in Figure 2.

These rules are straightforward, when put side by side with the dynamic semantics of filters, given in Section 3. It is clear that this type system simulates *at the level of types* the computations that are carried out by filters on values at runtime. For instance, rule [FIL-EXPR] calls the typing function of the host language to determine the type of an expression e. Rule [FIL-PROD] applies a product filter recursively on the first and second projection for each member of the product decomposition of the input type and returns the union of all result types. Rule [FIL-REC] for records is similar, recursively applying sub-filters label-wise for each member of the record decomposition and returning the union of the resulting record types. As for the pattern filter (rule [FIL-PAT]), its subfilter f is typed in the environment augmented by the mapping t/p of the input type against the pattern (*cf.* Theorem 5). The typing rule for the union filter, [FIL-UNION] reflects the first match policy: when typing the second branch, we know that the first was not taken, hence that at runtime the filtered value will have a type that is in t but not in $\wr f_1 \wr$. Notice that this is *not* ensured by the definition of accepted type — which is a rough approximation that discards grosser errors but, as we stressed right after its definition, is not sufficient to ensure that evaluation of f_1 will not fail— but by the type system itself: the premises *check* that $f_1(t_1)$ is well-typed which, by induction, implies that f_1 will never fail on values of type t_1 and, ergo, that these values will never reach f_2. Also, we discard from the output type the contribution of the branches that cannot be taken, that is, branches whose accepted type have an empty intersection with the input type t. Composition (rule [FIL-COMP]) is straightforward. In this rule, the restriction that f_1 is a filter with no open recursion variable ensures that its output type s is also a type without free recursion variables and, therefore, that we can use it as input type for f_2. The next three rules work together. The first, [FIL-FIX] introduces for a recursive filter a fresh recursion variable for its output type. It also memoize in Δ that the recursive filter X is associated with a body f and in M that for an input filter X and an input type t, the output type is the newly introduced recursive type variable. When dealing with a recursive call X two situations may arise. One possibility is that it is the first time the filter X is applied to the input type t. We therefore introduce a fresh type variable T and recurse, replacing X by its definition f. Otherwise, if the input type has already been encountered while typing the filter variable X, we can return its memoized type, a type variable T. Finally, Rule [FIL-ORDBY] and Rule [FIL-GRPBY] handle the special cases of `groupby` and `orderby` filters. Their typing is explained in the following section.

4.3 Typing of `orderby` and `groupby`

While the "structural" filters enjoy simple, compositional typing rules, the ad-hoc operations `orderby` and `groupby` need specially crafted rules. Indeed it is well known that when transformation languages have the ability to compare data values type-checking (and also type inference) becomes undecidable (*eg*, see [2, 3]). We therefore provide two typing approximations that yield a good compromise between precision and decidability. First we define an auxiliary function over sequence types:

Definition 12 (Item set). Let $t \in$ **Types** such that $t \leq$ `[any*]`. The *item set* of t denoted by $\mathtt{item}(t)$ is defined by:

$$\begin{array}{ll} \mathtt{item}(\mathtt{empty}) & = \varnothing \\ \mathtt{item}(t) & = \mathtt{item}(t \,\&\, (\mathtt{any},\mathtt{any})) \quad \text{if } t \not\leq (\mathtt{any},\mathtt{any}) \\ \mathtt{item}(\bigvee_{1 \leq i \leq \mathtt{rank}(t)} (t_i^1, t_i^2)) & = \bigcup_{1 \leq i \leq \mathtt{rank}(t)} (\{t_i^1\} \cup \mathtt{item}(t_i^2)) \end{array}$$

The first and second line in the definition ensure that `item`() returns the empty set for sequence types that are not products, namely for the empty sequence. The third line handles the case of non-empty sequence type. In this case t is a finite union of products, whose first components are the types of the "head" of the sequence and second components are recursively the types of the tails. Note also that this definition is well-founded. Since types are regular trees the number of distinct types accumulated by `item`() is finite. We can now defined typing rules for the `orderby` and `groupby` operators.

`orderby` f*:* The `orderby` filter uses its argument filter f to compute a key from each element of the input sequence and then returns the same sequence of elements, sorted with respect to their key. Therefore, while the types of the elements in the result are still known, their order is lost. We use `item`() to compute the output type of an `orderby` application:

$$\mathsf{OrderBy}(t) = \mathtt{[}(\bigvee_{t_i \in \mathtt{item}(t)} t_i) * \mathtt{]}$$

`groupby` f*:* The typing of `orderby` can be used to give a rough approximation of the typing of `groupby` as stated by rule [FIL-GRPBY]. In words, we obtain a list of pairs where the key component is the result type of f applied to the items of the sequence, and use OrderBy to shuffle the order of the list. A far more precise typing of `groupby` that keeps track of the relation between list elements and their images via f is given in the full version.

4.4 Soundness, termination, and complexity

The soundness of the type inference system is given by the property of subject reduction for filter application

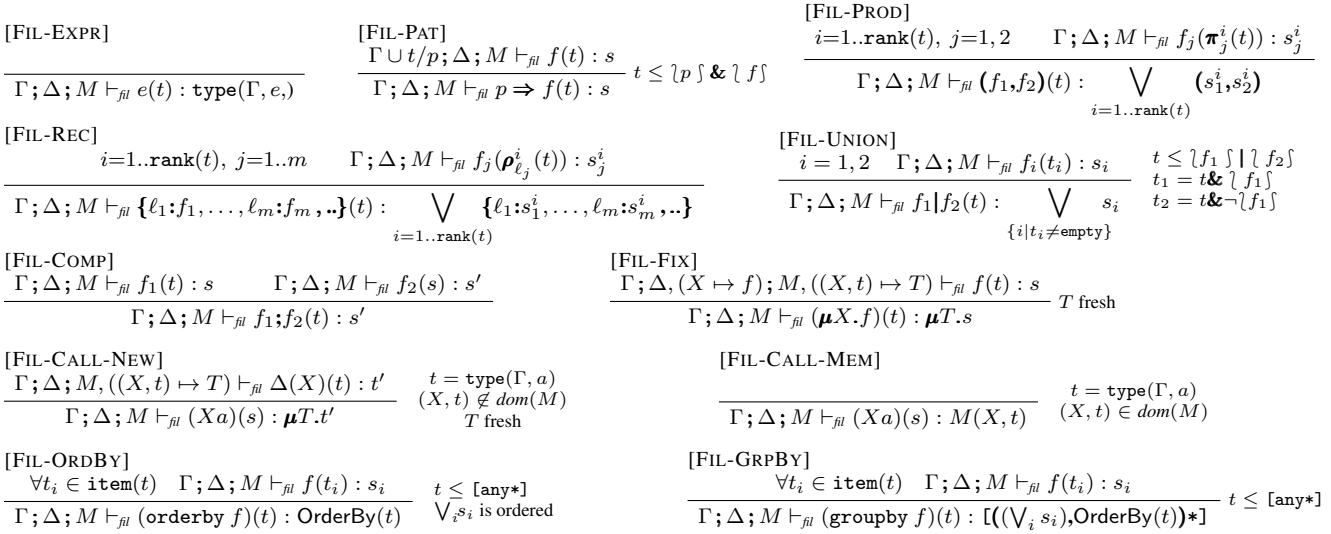

Figure 2. Type inference algorithm for filter application

Theorem 13 (subject reduction). *If $\varnothing;\varnothing;\varnothing \vdash_{fil} f(t) : s$, then for all $v : t$, $\varnothing;\varnothing \vdash_{eval} f(v) \rightsquigarrow r$ implies $r : s$.*

whose proof is given in the full version. It is easy to write a filter for which the type inference algorithm, that is the deduction of $\vdash_{\text{fil}}$, does not terminate: $\boldsymbol{\mu}X.x \Rightarrow X(x,x)$. The deduction of $\Gamma;\Delta;M \vdash_{fil} f(t) : s$ simulates an (abstract) execution of the filter f on the type t. Since filters are Turing complete, then in general it is not possible to decide whether the deduction of $\vdash_{\text{fil}}$ for a given filter f will terminate for every input type t. For this reason we define a static analysis *Check*(f) for filters that ensures that if f passes the analysis, then for every input type t the deduction of $\Gamma;\Delta;M \vdash_{fil} f(t) : s$ terminates. For space reasons the formal definition of *Check*(f) is available in only the full version, but its behavior can be easily explained. Imagine that a recursive filter f is applied to some input type t. The algorithm tracks all the recursive calls occurring in f; next it performs one step of reduction of each recursive call by unfolding the body; finally it checks in this unfolding that if a variable occurs in the argument of a recursive call, then it is bound to a type that is a subtree of the original type t. In other words, the analysis verifies that in the execution of the derivation for $f(t)$ every call to s/p for some type s and pattern p always yields a type environment where variables used in recursive calls are bound to subtrees of t. This implies that the rule [FIL-CALL-NEW] will always memoize for a given X, types that are obtained from the arguments of the recursive calls of X by replacing their variables with a subtree of the original type t memoized by the rule [FIL-FIX]. Since t is regular, then it has finitely many distinct subtrees, thus [FIL-CALL-NEW] can memoize only finitely many distinct types, and therefore the algorithm terminates.

More precisely, the analysis proceeds in two passes. In the first pass the algorithm tracks all recursive filters and for each of them it (i) marks the variables that occur in the arguments of its recursive calls, (ii) assigns to each variable an abstract identifier representing the subtree of the input type to which the variable will be bound at the initial call of the filter, and (iii) it returns the set of all types obtained by replacing variables by the associated abstract identifier in each argument of a recursive call. The last set intuitively represents all the possible ways in which recursive calls can shuffle and recompose the subtrees forming the initial input type. The second phase of the analysis first abstractly reduces by one step each recursive filter by applying it on the set of types collected in the first phase of the analysis and then checks whether, after this reduction, all the variables marked in the first phase (*ie*, those that occur in arguments of recursive calls) are still bound to subtrees of the initial input type: if this checks fails, then the filter is rejected.

It is not difficult to see that the type inference algorithm converges if and only if for every input type there exists a integer n such that after n recursive calls the marked variables are bound only to subtrees of the initial input type (or to something that does not depend on it, of course). Since deciding whether such an n exists is not possible, our analysis checks whether for all possible input types a filter satisfies it for $n{=}1$, that is to say, that at every recursive call its marked variables satisfy the property; otherwise it rejects the filter.

Theorem 14 (Termination). *If Check(f), then for every type t the deduction of $\Gamma;\varnothing;\varnothing \vdash_{fil} f(t) : s$ is in 2-EXPTIME. Furthermore, if t is given as a non-deterministic tree automaton (NTA) then $\Gamma;\varnothing;\varnothing \vdash_{fil} f(t) : s$ is in EXPTIME, where the size of the problem is $|f| \times |t|$.*

This complexity result is in line with those of similar formalisms. For instance in [18], it is shown that type-checking non deterministic top-down tree transducers is in EXPTIME when the input and output types are given by a NTA.

All filters defined in this article pass the analysis. As an example consider the filter `rotate` that applied to a list returns the same list with the first element moved to the last position (and the empty list if applied to the empty list):

$$\boldsymbol{\mu}X.\ (\ (x,(y,z)) \Rightarrow (y,X(x,z)) \quad | \quad w \Rightarrow w\ )$$

The analysis succeeds on this filter. If we denote by ι_x the abstract subtree bound to the variable x, then the recursive call will be executed on the abstract argument (ι_x,ι_z). So in the unfolding of the recursive call x is bound to ι_x, whereas y and z are bound to two distinct subtrees of ι_z. The variables in the recursive call, x and z, are thus bound to subtrees of the original tree (even though the argument of the recursive call is *not* a subtree of the original tree), therefore the filter is accepted . In order to appreciate the precision of the inference algorithm consider the type `[int+ bool+]`, that is, the type of lists formed by some integers (at least one) followed by some booleans (at least one). For the application of `rotate` to an argument of this type our algorithm *statically* infers the most precise type, that is, `[int* bool+ int]`. If we apply it once more the inferred type is `[int* bool+ int int]|[bool* int bool]`.

Generic filters are Turing complete. However, requiring that *Check*$()$ holds —meaning that the filter is typeable by our system— restricts the expressive power of our filters by preventing them

from *recomposing* a new value before doing a recursive call. For instance, it is not possible to typecheck a filter which reverses the elements of a sequence. Determining the exact class of transformations that typeable filters can express is challenging. However it is possible to show (see the full version for the proof) that typeable filters are strictly more expressive than top-down tree transducers with regular look-ahead, a formalism for tree transformations introduced in [13]. For an intuition of this result consider the tree:

$$a(u_1(\ldots(u_n()))v_1(\ldots(v_m())))$$

that is, a tree whose root is labeled a with two children, each being a monadic tree of height n and m, respectively. It is not possible to write a top-down tree transducer with regular look-ahead that creates the tree

$$a(u_1(\ldots(u_n(v_1(\ldots v_m()))))))$$

which is just the concatenation of the two children of the root, seen as sequences, a transformation that can be easily programmed by typeable filters. The key difference in expressive power comes from the fact that filters are evaluated with an *environment* that binds capture variables to sub-trees of the input. This feature is essential to encode sequence concatenation and sequence flattening —two pervasive operations when dealing with sequences— that cannot be expressed by top-down tree transducers with regular look-ahead.

5. Jaql

In this Section, we show how filters can be used to capture some popular languages for processing data on the Cloud. We consider Jaql [16], a query language for JSON developed by IBM. We give translation rules from a subset of Jaql into filters.

Definition 15 (Jaql expressions). We use the following simplified grammar for Jaql (where we distinguish simple expressions, ranged over by e, from "core expressions" ranged over by k).

```
e ::= c                                                    (constants)
    | x                                                    (variables)
    | $                                                    (current value)
    | [e, ..., e]                                          (arrays)
    | { e:e, ..., e:e }                                    (records)
    | e.l                                                  (field access)
    | op(e, ...,e)                                         (function call)
    | e -> k                                               (pipe)
k ::= filter (each x )? e                                  (filter)
    | transform (each x)? e                                (transform)
    | expand ((each x)? e)?                                (expand)
    | group ((each x)? by x=e (as x)?)? into e             (grouping)
```

In order to ease the presentation we extend our syntax by adding "filter definitions" (already informally used in the introduction) to filters and "filter calls" to expressions:

$$e ::= \texttt{let filter}\ F[F_1,\ldots,F_n] = f\ \texttt{in}\ e \quad \textbf{(filter defn.)}$$
$$f ::= F[f,\ldots,f] \quad \textbf{(call)}$$

where F ranges over *filter names*. The mapping for most of the language we consider rely on the following built-in filters.

```
let filter Filter [F] = μX.
      ‘nil ⇒ ‘nil
    | ((x, xs),tl) ⇒ (X(x, xs),X(tl))
    | (x,tl) ⇒ Fx ;(‘true ⇒ (x, X(tl))|‘false ⇒ X(tl))

let filter Transform [F] = μX.
      ‘nil ⇒ ‘nil
    | ((x, xs),tl) ⇒ (X(x, xs),X(tl))
    | (x,tl) ⇒ (Fx, X(tl))

let filter Expand = μX.
      ‘nil ⇒ ‘nil
    | (‘nil,tl) ⇒ X(tl)
    | ((x, xs),tl) ⇒ (x,X(xs, tl))
```

Jaql expressions are mapped to our expressions as follows (where `$` is a distinguished expression variable interpreting Jaql's `$`):

$$\begin{array}{lcl}
[\![\texttt{c}]\!] & = & c \\
[\![x]\!] & = & x \\
[\![\texttt{\$}]\!] & = & \texttt{\$} \\
[\![\{e_1:e_1',\ldots,e_n:e_n'\}]\!] & = & \{[\![e_1]\!]:[\![e_1']\!],\ldots,[\![e_n]\!]:[\![e_n']\!]\} \\
[\![e.\texttt{l}]\!] & = & [\![e]\!].l \\
[\![\texttt{op}(e_1,\ldots,e_n)]\!] & = & \texttt{op}([\![e_1]\!],\ldots,[\![e_n]\!]) \\
[\![[e_1,\ldots,e_n]]\!] & = & ([\![e_1]\!],\ldots([\![e_n]\!],\text{`}\texttt{nil})\ldots) \\
[\![e \texttt{ -> } k]\!] & = & [\![e]\!];[\![k]\!]_{\mathsf{F}}
\end{array}$$

Jaql core expressions are mapped to filters as follows:

$$\begin{array}{lcl}
[\![\texttt{filter } e]\!]_{\mathsf{F}} & = & [\![\texttt{filter each \$ } e]\!]_{\mathsf{F}} \\
[\![\texttt{filter each } x\ e]\!]_{\mathsf{F}} & = & \texttt{Filter [}x \Rightarrow [\![e]\!]\texttt{]} \\
[\![\texttt{transform } e]\!]_{\mathsf{F}} & = & [\![\texttt{transform each \$ } e]\!]_{\mathsf{F}} \\
[\![\texttt{transform each } x\ e]\!]_{\mathsf{F}} & = & \texttt{Transform [}x \Rightarrow [\![e]\!]\texttt{]} \\
[\![\texttt{expand each } x\ e]\!]_{\mathsf{F}} & = & [\![\texttt{expand}]\!]_{\mathsf{F}};[\![\texttt{transform each } x\ e]\!]_{\mathsf{F}} \\
[\![\texttt{expand}]\!]_{\mathsf{F}} & = & \texttt{Expand} \\
[\![\texttt{group into } e]\!]_{\mathsf{F}} & = & [\![\texttt{group by } y\texttt{=true into } e]\!]_{\mathsf{F}} \\
[\![\texttt{group by } y\texttt{=}e_1 \texttt{ into } e_2]\!]_{\mathsf{F}} & = & [\![\texttt{group each \$ by } y\texttt{=}e_1 \texttt{ into } e_2]\!]_{\mathsf{F}} \\
[\![\texttt{group each } x \texttt{ by } y\texttt{=}e_1 \texttt{ into } e_2]\!]_{\mathsf{F}} & = & \\
\multicolumn{3}{r}{[\![\texttt{group each } x \texttt{ by } y = e_1 \texttt{ as \$ into } e_2]\!]_{\mathsf{F}}} \\
[\![\texttt{group each } x \texttt{ by } y = e_1 \texttt{ as } g \texttt{ into } e_2]\!]_{\mathsf{F}} & = & \\
\multicolumn{3}{r}{\texttt{groupby } x \Rightarrow [\![e_1]\!]\texttt{; Transform [}(y,g) \Rightarrow [\![e_2]\!]\texttt{]}}
\end{array}$$

This translation defines the (first, in our knowledge) formal semantics of Jaql. Such a translation is *all* that is needed to define the semantics of a NoSQL language and, as a bonus, endow it with the type inference system we described *without requiring any modification of the original language*. No further action is demanded since the machinery to exploit it is all developed in this work.

As for typing, every Jaql expression is encoded into a filter for which type-checking is ensured to terminate: *Check*() holds for `Filter[]`, `Transform[]`, and `Expand` (provided it holds also for their arguments) since they only perform recursive calls on recombinations of subtrees of their input; by its definition, the encoding does not introduce any new recursion and, hence, it always yields a composition and application of filters for which *Check*() holds.

5.1 Examples

To show how we use the encoding, let us encode the example of the introduction. For the sake of the concision we will use filter definitions (rather than expanding them in details). We use `Fil` and `Sel` defined in the introduction, `Expand` and `Transform[]` defined at the beginning of the section, the encoding of Jaql's field selection as defined in Section 3, and finally `Head` that returns the first element of a sequence and a family of recursive filters `Rgrp`i with $i \in \mathbb{N}^+$ both defined below:

```
let filter Head = ‘nil => null | (x,xs) => x

let filter Rgrpi = ‘nil => ‘nil
                 | ((i,x),tail) => (x , Rgrpi tail)
                 | _ => Rgrpi tail
```

Then, the query in the introduction is encoded as follows

```
1  [employees depts];
2  [Sel Fil];
3  [Transform[x =>(1,x)]  Transform[x =>(2,x)]];
4  Expand;
5  groupby ( (1,$)=>$.dept | (2,$)=>$.depid );
6  Transform[(g,l)=>(
7            [(l; Rgrp1) (l; Rgrp2)];
8            [es ds] =>
9                   { dept: g,
10                    deptName: (ds ; Head).name),
11                  numEmps: count(es) } )]
```

In words, we perform the selection on employees and filter the departments (lines 1-2); we tag each element by `1` if it comes from employees, and by `2` if it comes from departments (line 3); we merge the two collections (line 4); we group the heterogeneous list according to the corresponding key (line 5); then for each element of the result of grouping we capture in `g` the key (line 6), split the group into employees and depts (line 7), capture each subgroup into the corresponding variable (*ie*, `es` and `ds`) (line 8) and return the expression specified in the query after the "`into`" (lines 8-10). The general definition of the encoding for the co-grouping can be found in the full version.

Let us now illustrate how the above composition of filters is typed. Consider an instance where:

- `employees` has type `[ `*Remp*`* ]`, where *Remp* ≡ `{ dept: int, income:int, ..}`
- `depts` has type `[ (`*Rdep* `|` *Rbranch*`)* ]`, where *Rdep* ≡ `{depid:int, name: string, size: int}` *Rbranch* ≡ `{brid:int, name: string}` (this type is a subtype of `Dept` as defined in the introduction)

The global input type is therefore (line 1)

`[ [ `*Remp*`* ] [ (`*Rdep* `|` *Rbranch*`)* ] ]`

which becomes, after selection and filtering (line 2)

`[ [ `*Remp*`* ] [ `*Rdep*`* ] ]`

(note how all occurrences of *Rbranch* are ignored by `Fil`). Tagging with an integer (line 3) and flattening (line 4) yields

`[ (1,`*Remp*`)* (2,`*Rdep*`)* ]`

which illustrates the precise typing of products coupled with singleton types (*ie*, `1` instead of `int`). While the `groupby` (line 5) introduces an approximation the dependency between the tag and the corresponding type is kept

`[ (int, [ ((1,`*Remp*`) | (2,`*Rdep*`) )+ ]) * ]`

Lastly the transform is typed exactly, yielding the final type

`[ {dept:int, deptName:string|null, numEmps:int }* ]`

Note how `null` is retained in the output type (since there may be employees without a department, then `Head` may be applied to an empty list returning `null`, and the selection of `name` of `null` returns `null`). For instance suppose to pipe the Jaql grouping defined in the introduction into the following Jaql expression, in order to produce a printable representation of the records of the result

```
transform each x (
 (x.deptName)@":"@(to_string x.dep)@":"@(x.numEmps))
```

where `@` denotes string concatenation and `to_string` is a conversion operator (from any type to string). The composition is ill-typed for three reasons: the field `dept` is misspelled as `dep`, `x.numEmps` is of type `int` (so it must be applied to `to_string` before concatenation), and the programmer did not account for the fact that the value stored in the field `deptName` may be `null`. The encoding produces the following lines to be appended to the previous code:

```
12 Transform[ x =>
13  (x.deptName)@":"@(to_string x.dep)@":"@(x.numEmps)]
```

in which all the three errors are detected by our type system. A subtler example of error is given by the following alternative code

```
12 Transform[
13    { dept : d, deptName: n&String, numEmps: e } =>
14       n @ ":" @ (to_string d) @ ":" @ (to_string e)
15  | { deptName: null, .. } => ""
16  | _ => "Invalid department" ]
```

which corrects all the previous errors but adds a new one since, as detected by our type system, the last branch can be never selected. As we can see, our type-system ensures soundness, forcing the programmer to handle exceptional situations (as in the `null` example above) but is also precise enough to detect that some code paths can never be reached.

In order to focus on our contributions we kept the language of types and filters simple. However there already exists several contributions on the types and expressions used here. Two in particular are worth mentioning in this context: recursive patterns and XML.

Definition 3 defines patterns inductively but, alternatively, we can consider the (possibly infinite) regular trees *coinductively* generated by these productions and, on the lines of what is done in ℂDuce, use the recursive patterns so obtained to encode regular expressions patterns (see [5]). Although this does not enhance expressiveness, it greatly improves the writing of programs since it makes it possible to capture distinct subsequences of a sequence by a single match. For instance, when a sequence is matched against a pattern such as `[ (int as x | bool as y | _)* ]`, then `x` captures (the list of) all integer elements (capture variables in regular expression patterns are bound to lists), `y` captures all Boolean elements, while the remaining elements are ignored. By such patterns, co-grouping can be encoded without the `Rgrp`. For instance, the transform in lines 6-11 can be more compactly rendered as:

```
6  Transform[(g,[ ((1,es)|(2,ds))* ]) =>
7                       { dept: g,
8                         deptName: (ds;Head).name,
9                         numEmps: count(es) }]
```

For what concerns XML, the types used here were originally defined for XML, so it comes as a no surprise that they can seamlessly express XML types and values. For example ℂDuce uses the very same types used here to encode both XML types and elements as triples, the first element being the tag, the second a record representing attributes, and the third a heterogeneous sequence for the content of the element. Furthermore, we can adapt the results of [10] to encode forward XPath queries in filters. Therefore, it requires little effort to use the filters presented here to encode languages such as JSONiq [28] designed to integrate JSON and XML, or to precisely type regular expressions, the import/export of XML data, or XPath queries embedded in Jaql programs. The description of these encodings can be found in the long version of this article, where we also argue that it is better to extend NoSQL languages with XML primitives directly derived from our system rather than to use our system to encode languages such as JSONiq. As a matter of fact, existing approaches tend to juxtapose XML and JSON operators thus yielding to stratified (*ie*, not tightly integrated) systems which have several drawbacks (*eg*, JSONiq does not allow XML nodes to contain JSON objects and arrays). Such restrictions are absent from our approach since both XML and JSON operators are encoded in the same basic building blocks and, as such, can be freely nested and combined.

5.2 Extensions

Hitherto we used filters only to encode primitive operators of some NoSQL languages, in particular Jaql. However, it is possible to *add* filters to other languages, so as to have user-defined operators typed as precisely as primitive ones. From a linguistic point of view this is a no-brainer: it suffices to add filter application to the expressions of the host language. However, such an extension can be problematic from a computational viewpoint, since it may disrupt the execution model, especially for what concerns aspects of parallelism and distribution. A good compromise is to add only filters that have "local" effects, which can already bring dramatic increases in expressiveness and type precision without disrupting the distributed compilation model. For instance, one can add just pattern and union filters as in the following (extended) Jaql program:

```
transform ( {a:x,..} as y => {y.*, sum:x+x} | y => y )
```

(with the convention that a filter occurring as an expression de-

notes its application to the current argument `$`). With this syntax, our inference system is able to deduce that feeding this expression with an argument of type `[{a?:int, c:bool}*]` returns a result of type `[({a:int, c:bool, sum:int} | {c:bool})*]`. This precision comes from the capacity of our inference system to discriminate between the two branches of the filter and deduce that a `sum` field will be added only if the `a` field is present. Similarly by using pattern matching in a Jaql "`filter`" expression, we can deduce that `filter ( int => true | _ => false )` fed with any sequence of elements always returns a (possibly empty) list of integers. An even greater precision can be obtained for grouping expressions when the generation of the key is performed by a filter that discriminates on types: the result type can keep a precise correspondence between keys and the corresponding groups.

6. Commentaries

Finally, let us explain some subtler design choices for our system.

Filter design: The reader may wonder whether products and record *filters* are really necessary since, at first sight, the filter (f_1,f_2) could be encoded as $(x,y) \Rightarrow (f_1 x, f_2 y)$ and similarly for records. The point is that $f_1 x$ and $f_2 y$ are expressions —and thus their pair is a filter— only if the f_i's are closed (*ie*, without free term recursion variables). Without an explicit product filter it would not be possible to program a filter as simple as the identity map, $\mu X.\texttt{`nil} \Rightarrow \texttt{`nil} | (h,t) \Rightarrow (h,Xt)$ since Xt is not an expression (X is a free term recursion variable). Similarly, we need an explicit record filter to process recursively defined record types such as $\mu X.(\{\texttt{head:int}, \texttt{tail:}X\} | \texttt{`nil})$.

Likewise, one can wonder why we put in filters only the "open" record variant that copy extra fields and not the closed one. The reason is that if we want a filter to be applied only to records with exactly the fields specified in the filter, then this can be simply obtained by a pattern matching. So the filter $\{\ell_1\texttt{:}f_1, \ldots, \ell_n\texttt{:}f_n\}$ (*ie*, without the trailing "`..`") can be simply introduced as syntactic sugar for $\{\ell_1\texttt{:any}, \ldots, \ell_n\texttt{:any}\} \Rightarrow \{\ell_1\texttt{:}f_1, \ldots, \ell_n\texttt{:}f_n\texttt{,..}\}$

Constructors: The syntax for constructing records and pairs is exactly the same in patterns, types, expressions, and filters. The reader may wonder why we did not distinguish them by using, say, $\times$ for product types or $=$ instead of `:` in record values. This, combined with the fact that values and singletons have the same syntax, is a critical design choice that greatly reduces the confusion in these languages, since it makes it possible to have a unique representation for constructions that are semantically equivalent. Consider for instance the pattern $(x,(3,\texttt{`nil}))$. With our syntax $(3,\texttt{`nil})$ denotes both the product type of two singletons 3 and `` `nil ``, or the value $(3,\texttt{`nil})$, or the singleton that contains this value. According to the interpretation we choose, the pattern can then be interpreted as a pattern that matches a product or a pattern that matches a value. If we had differentiated the syntax of singletons from that of values (*eg*, $\{v\}$) and that of pairs from products, then the pattern above could have been written in five different ways. The point is that they all would match exactly the same sets of values, which is why we chose to have the same syntax for all of them.

Record types: In order to type records with computed labels we distinguished two cases according to whether the type of a record label is finite or not. Although such a distinction is simple, it is not unrealistic. Labels with singleton types cover the (most common) case of records with statically fixed labels. The dynamic choice of a label from a statically known list of labels is a usage pattern seen in JavaScript when building an object which must conform to some interface based on a run-time value. Labels with infinite types cover the fairly common usage scenario in which records are used as dictionaries: we deduce for the expression computing the label the type `string`, thus forcing the programmer to insert some code that checks that the label is present before accessing it.

The rationale behind the typing of records was twofold. First and foremost, in this work we wanted to avoid type annotations at all costs (since there is not even a notion of schema for JSON records and collections —only the notion of basic type is defined— we cannot expect the Jaql programmer to put any kind of type information in the code). More sophisticated type systems, such as dependent types, would probably preclude type reconstruction: dependent types need a lot of annotations and this does not fit our requirements. Second, we wanted the type-system to be simple yet precise. Making the finite/infinite distinction increases typing precision at no cost (we do not need any extra machinery since we already have singleton types). Adding heuristics or complex analysis just to gain some precision on records would have blurred the main focus of our article, which is not on typing records but on typing *transformations* on records. We leave such additions for future work.

Record polymorphism: The type-oriented reader will have noticed that we do not use row variables to type records, and nevertheless we have a high degree of polymorphism. Row variables are useful to type functions or transformations since they can keep track of record fields that are not modified by the transformation. In this setting we do not need them since we do not type transformations (*ie*, filters) but just the application of transformations (filters are not first-class terms). We have polymorphic typing via filters (see how the first example given in Section 5.2 keeps track of the `c` field) and therefore open records suffice.

Related work: In the (nested) relational (and SQL) context, many works have studied the integration of (nested)-relational algebra or SQL into general purpose programming languages. Among the first attempts was the integration of the relational model in Pascal [29] or in Smalltalk [12]. Also, monads or comprehensions [8, 31, 32] have been successfully used to design and implement query languages including a way to embed queries within host languages. Significant efforts have been done to equip those languages with type systems and type checking disciplines [1, 9, 23, 24] and more recently [25] for integration and typing aspects. However, these approaches only support homogeneous sequences of records in the context of specific classes of queries (practically equivalent to a nested relational algebra or calculus), they do not account for records with computable labels, and therefore they are not easily transposable to a setting where sequences are heterogeneous, data are semi-structured, and queries are much more expressive.

While the present work is inspired and stems from previous works on the XML iterators, targeting NoSQL languages made the filter calculus presented here substantially different from the one of [10, 21] (dubbed XML filters in what follows), as well in syntax as in dynamic and static semantics. In [10] XML filters behave as some kind of top-down tree transducers, termination is enforced by heavy syntactic restrictions, and a *less* constrained use of the composition makes type inference challenging and requires sometimes cumbersome type annotations. While XML filters are allowed to operate by composition on the *result* of a recursive call (and, thus, simulate bottom-up tree transformations), the absence of explicit arguments in recursive calls makes programs understandable only to well-trained programmers. In contrast, the main focus of the current work was to make programs immediately intelligible to any functional programmer and make filters effective for the typing of sequence transformations: sequence iteration, element filtering, one-level flattening. The last two are especially difficult to write with XML filters (and require type annotations). Also, the integration of filters with record types (absent in [10] and just sketched in [21]) is novel and much needed to encode JSON transformations.

7. Conclusion

Our work addresses two very practical problems, namely the typing of NoSQL languages and a comprehensive definition of their semantics. These languages add to list comprehension and SQL operators the ability to work on heterogeneous data sets and are based on JSON (instead of tuples). Typing precisely each of these features using the best techniques of the literature would probably yield quite a complex type-system (mixing row polymorphism for records, parametric polymorphism, some form of dependent typing,...) and we are skeptical that this could be achieved without using any explicit type annotation. Therefore we explored the formalization of these languages from scratch, by defining a calculus and a type system. The thesis we defended is that all operations typical of current NoSQL languages, as long as they operate structurally (*ie*, without resorting on term equality or relations), amount to a combination of more basic bricks: our filters. On the structural side, the claim is that combining recursive records and pairs by unions, intersections, and negations suffices to capture all possible structuring of data, covering a palette ranging from comprehensions, to heterogeneous lists mixing typed and untyped data, through regular expressions types and XML schemas. Therefore, our calculus not only provides a simple way to give a formal semantics to, reciprocally compare, and combine operators of different NoSQL languages, but also offers a means to equip these languages, in they current definition (*ie*, without any type definition or annotation), with precise type inference. This type inference yields and surpasses in precision systems using parametric polymorphism and row variables. The price to pay is that *transformations* are not first class: we do not type filters but just their applications. However, this seems an advantageous deal in the world of NoSQL languages where "selects" are never passed around (at least, not explicitly), but early error detection is critical, especially in the view of the cost of code deployment.[2]

The result are filters, a set of untyped terms that can be easily included in a host language to complement in a typeful framework existing operators with user-defined ones. The requirements to include filters into a host language are so minimal that every modern typed programming language satisfies them. The interest resides not in the fact that we can add filter applications to any language, rather that filters can be used to define a smooth integration of calls to domain specific languages (*eg*, SQL, XPath, Pig, Regex) into general purpose ones (*eg*, Java, C#, Python, OCaml) so as both can share the same set of values and the same typing discipline. Likewise, even though filters can provide a prototyping platform for queries, they cannot currently be used as a final compilation stage for NoSQL languages: their operations rely on a Lisp-like encoding of sequences and this makes the correspondence with optimized bulk operations on lists awkward. Whether we can derive an efficient compilation from filters to map-reduce (recovering the bulk semantics of the high-level language) is a challenging open question. Future plans include practical experimentation of our technique: we intend to benchmark our type analysis against existing collections of Jaql programs, gauge the amount of code that is ill typed and verify on this how frequently the programmer adopted defensive programming to cope with the potential type errors.

References

[1] A. Albano, G. Ghelli, and R. Orsini. Fibonacci: A programming language for object databases. *The VLDB Journal*, 4:403–444, 1995.

[2] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. XML with data values: typechecking revisited. In *PODS '01*. ACM, 2001.

[3] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. Typechecking XML views of relational databases. *ACM Trans. Comput. Logic*, 4:315–354, July 2003.

[4] A. Behm *et al.* Asterix: towards a scalable, semistructured data platform for evolving-world models. *DAPD*, 29(3):185–216, 2011.

[5] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-friendly general purpose language. In *ICFP '03*. ACM, 2003.

[6] K. Beyer *et al*. Jaql: A scripting language for large scale semistructured data analysis. *PVLDB*, 4(12):1272–1283, 2011.

[7] S. Boag, D. Chamberlain, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML query language, W3C rec., 2007.

[8] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, 1994.

[9] P. Buneman, R. Nikhil, and R. Frankel. A Practical Functional Programming System for Databases. In *Proc. Conference on Functional Programming and Architecture*. ACM, 1981.

[10] G. Castagna and K. Nguyễn. Typed iterators for XML. In *ICFP '08*. ACM, 2008.

[11] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, C. Löding, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. http://www.grappa.univ-lille3.fr/tata, 2007.

[12] G. Copeland and D. Maier. Making Smalltalk a database system. In *ACM SIGMOD Conf.*, 1984.

[13] J. Engelfriet. Top-down tree transducers with regular look-ahead. *Mathematical Systems Theory*, 10(1):289–303, Dec. 1976.

[14] A. Frisch. *Théorie, conception et réalisation d'un langage de programmation adapté à XML.* PhD thesis, Université Paris 7 Denis Diderot, 2004.

[15] 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):1–64, 2008.

[16] Jaql. `http://code.google.com/p/jaql`.

[17] JavaScript Object Notation (JSON). `http://json.org/`.

[18] W. Martens and F. Neven. Typechecking top-down uniform unranked tree transducers. In *ICDT '03*. Springer, 2002.

[19] E. Meijer. The world according to LINQ. *ACM Queue*, 9(8):60, 2011.

[20] E. Meijer and G. Bierman. A co-relational model of data for large shared data banks. *Communications of the ACM*, 54(4):49–58, 2011.

[21] K. Nguyễn. *Language of Combinators for XML: Conception, Typing, Implementation*. PhD thesis, Université Paris-Sud 11, 2008.

[22] Odata. `http://www.odata.org/`.

[23] A. Ohori and P. Buneman. Type Inference in a Database Programming Language. In *LISP and Functional Programming*, 1988.

[24] A. Ohori, P. Buneman, and V. Tannen. Database Programming in Machiavelli –a Polymorphic Language with Static Type Inference. In *ACM SIGMOD Conf.*, 1989.

[25] A. Ohori and K. Ueno. Making standard ML a practical database programming language. In *ICFP '11*, 2011.

[26] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *ACM SIGMOD Conf.*, 2008.

[27] F. Özcan *et al*. Emerging trends in the enterprise data analytics: connecting Hadoop and DB2 warehouse. In *ACM SIGMOD Conf.*, 2011.

[28] J. Robie (editor). JSONiq. `http://jsoniq.org`.

[29] J. Schmidt and M. Mall. Pascal/R Report. Technical Report 66, Fachbereich Informatik, université de Hamburg, 1980.

[30] Squeryl: A Scala ORM and DSL for talking with Databases with minimum verbosity and maximum type safety. `http://squeryl.org/`.

[31] V. Tannen, P. Buneman, and L. Wong. Naturally embedded query languages. In *ICDT*, pages 140–154, 1992.

[32] P. Trinder and P. Wadler. Improving list comprehension database queries. In *4th IEEE Region 10 Conference (TENCON)*, 1989.

[33] Unql. `http://www.unqlspec.org/`.

[2] Only filter-encoded operators are not first class: if the host language provides, say, higher-order functions, then they stay higher-order and are typed by embedding the host type system, if any, via "foreign type calls".