# 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, Univ 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.

## 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 [19, 24] and languages. Whether they are developed under the banner of "NoSQL" [30, 36], for BigData Analytics [6, 18, 28], for Cloud computing [4], or as domain specific languages (DSL) embedded in a host language [21, 27, 33], 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 [21, 22], Meijer argues that such languages can greatly benefit from formal foundations, and suggests comprehensions [8, 34, 35] 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 paper, 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, 18], a language over JSON [19] 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 [28] 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 [28] 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 (selecting 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

An extended abstract of this work appears in the proceedings of *POPL 13, 40th ACM Symposium on Principles of Programming Languages*, ACM Press, 2013.

`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 5.3.

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 [21] 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" [21] . 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 [28], Jaql [18, 29], 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 [16], 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 5.3), $(iii)$ 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) |
| | | \| | $\{\ell{:}t, \ldots, \ell{:}t\,,..\}$ | (open record) |
| | | \| | $[r]$ | (sequences) |
| | | \| | `int` \| `char` | (base) |
| | | \| | `any` \| `empty` \| `null` | (special) |
| | | \| | $t\|t$ | (union) |
| | | \| | $t\backslash t$ | (difference) |

**Regexp** $r ::= \varepsilon \mid t \mid r* \mid r+ \mid r? \mid r\,r \mid r|r$

where $\varepsilon$ 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 $\{\texttt{a:int}, \texttt{b:int}, ..\}$ 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})+]`

For the sake of precision, to comply with Jaql's semantics the last pattern should rather be `({..},tail) => Fil(tail)`: since field selection $e$`.size` fails whenever $e$ is not a record or a list, this definition would detect the possibility of this failure by a static type error.

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 definition see Appendix G).

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. 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 8 reports on some subtler design choices of our system. We compare related works in Section 9 and conclude in Section 10. In order to avoid blurring the presentation, proofs, secondary results, further encodings, and extensions are moved into a separate appendix.

## 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

The only exception are the "bags" types we did not include in order to focus on essential features.

**Definition 1 (expressions).**

| | | | | |
|---|---|---|---|---|
| **Exprs** | $e$ | ::= | $c$ | (constants) |
| | | \| | $x$ | (variables) |
| | | \| | $(e, e)$ | (pairs) |
| | | \| | $\{e\texttt{:}e, ..., e\texttt{:}e\}$ | (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 $\ell$ 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:

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

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

Expressions, in particular filter applications, are typed by the following set of types (typically only basic, product, recursive and —some form of— record types will be provided by the host language):

**Definition 2 (types).**

| | | | | |
|---|---|---|---|---|
| **Types** | $t$ | ::= | $b$ | (basic types) |
| | | \| | $v$ | (singleton types) |
| | | \| | $(t,t)$ | (products) |
| | | \| | $\{\ell\texttt{:}t, \ldots, \ell\texttt{:}t\}$ | (closed records) |
| | | \| | $\{\ell\texttt{:}t, \ldots, \ell\texttt{:}t\texttt{,..}\}$ | (open records) |
| | | \| | $t\texttt{\|}t$ | (union types) |
| | | \| | $t\texttt{\&}t$ | (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 [16].

## 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) |
| | | \| | $(p,p)$ | (pair) |
| | | \| | $\{\ell\texttt{:}p, \ldots, \ell\texttt{:}p\}$ | (closed rec) |
| | | \| | $\{\ell\texttt{:}p, \ldots, \ell\texttt{:}p\texttt{,..}\}$ | (open rec) |
| | | \| | $p\texttt{\|}p$ | (or/union) |
| | | \| | $p\texttt{\&}p$ | (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 $\Omega$) 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 $(p_1,p_2)$ 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 $p_1\&p_2$ 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 [16] 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 [16]):

**Theorem 4 (Accepted type [16]).** *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 $\lbag p \rbag$.*

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 $\lbag p \rbag$, 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) $\lbag p \rbag$, 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 [16]).** *There exists an algorithm that for every pattern p, and $t \leq \lbag p \rbag$ 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) |
| | | \| | $\mu X.f$ | (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 4.5 (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.

### 3.1 Big step semantics

We define a big step operational semantics for filters. The definition is given by the inference rules in Figure 1 for judgements 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 $\gamma$ yields an object $r$ where $r$ is either a value or $\Omega$. 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)

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

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

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

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

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

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

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

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

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

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

**Figure 1.** Dynamic semantics of filters

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 call, in which case variables may occurr in it (*cf.* rule **rec-call**). Environment $\delta$ 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 $\delta$ 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 (full proof in Appendix B):

**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 $\lambda$-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. □

### 3.2 Semantics of declarative filters

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 to encode co-grouping (or `groupby` on several input sequences) with a combination of `groupby` and filters (*cf.* Appendix H).

### 3.3 Syntactic sugar

Now that we have formally defined the semantics of filters we can use them to introduce some syntactic sugar.

***Expressions.*** 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:

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

$$\texttt{match}\ e\ \texttt{with}\ p_1 \Rightarrow e_1 \mid \ldots \mid p_n \Rightarrow e_n \stackrel{\text{def}}{=} (p_1 \Rightarrow e_1 \mid \ldots \mid p_n \Rightarrow e_n)e$$

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 $\ell$ 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

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

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 labelled by $\ell$, if present. Finally, we will also use for expressions, types, and patterns the syntactic sugar for lists used in the introduction. So, for instance, $[p_1\ p_2\ \ldots\ p_n]$ is matched by lists of $n$ elements provided that their $i$-th element matches $p_i$.

# 4. Type inference

In this section we describe a type inference algorithm for our expressions.

## 4.1 Typing of simple and foreign expressions

Variables, constants, and pairs are straightforwardly typed by

$$[\textsc{Vars}]\ \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad [\textsc{Constant}]\ \frac{}{\Gamma \vdash c : c} \qquad [\textsc{Prod}]\ \frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : (t_1\texttt{,}t_2)}$$

where $\Gamma$ denotes a typing environment that is a function from expression variables to types and $c$ denotes both a constant and the singleton type containing the constant. Expressions of the host language are typed by the `type` function which given a type environment and a foreign expression returns the type of the expression, and that we suppose to be given for each host language.

$$[\textsc{Foreign}]\ \frac{\Gamma \vdash e_1 : t_1 \quad \cdots \quad \Gamma \vdash e_n : t_n}{\Gamma \vdash \texttt{op}(e_1,\ldots,e_n) : \texttt{type}((\Gamma, x_1\texttt{:}t_1, \ldots, x_n\texttt{:}t_n), \texttt{op}(x_1,\ldots,x_n))}$$

Since the various $e_i$ can contain filter applications, thus unknown to the host language's type system, the rule [FOREIGN] swaps them with variables having the same type.

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.* the expression operator `apply` in Section 2.2).

## 4.2 Typing of records

The typing of records is novel and challenging because record expressions may contain string *expressions* in label position, such as in $\{e_1\texttt{:}e_2\}$, while in all type systems for record we are aware of, labels are never computed. It is difficult to give a type to $\{e_1\texttt{:}e_2\}$ 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 $\{e_1\texttt{:}e_2\}$, 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 $\{e_1\texttt{:}e_2\}$ 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 $\ell_i$ for some $i \in [1..n]$. Therefore $\{e_1\texttt{:}e_2\}$ 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].

$$[\textsc{Rcd-Fin}]\ \frac{\Gamma \vdash e : \ell_1|\cdots|\ell_n \qquad \Gamma \vdash e' : t}{\Gamma \vdash \{e\texttt{:}e'\} : \{\ell_1\texttt{:}t\}|\cdots|\{\ell_n\texttt{:}t\}}$$

$$[\textsc{Rcd-Inf}]\ \frac{\Gamma \vdash e : t \qquad \Gamma \vdash e' : t'}{\Gamma \vdash \{e\texttt{:}e'\} : \{\texttt{..}\}}\ \begin{array}{l} t \leq \texttt{string} \\ t \text{ is infinite} \end{array}$$

$$[\textsc{Rcd-Mul}]\ \frac{\Gamma \vdash \{e_1\texttt{:}e_1'\} : t_1 \quad \cdots \quad \Gamma \vdash \{e_n\texttt{:}e_n'\} : t_n}{\Gamma \vdash \{e_1\texttt{:}e_1', \ldots, e_n\texttt{:}e_n'\} : t_1 + \cdots + t_n}$$

$$[\textsc{Rcd-Conc}]\ \frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 + e_2 : t_1 + t_2}\ \begin{array}{l} t_1 \leq \{\texttt{..}\} \\ t_2 \leq \{\texttt{..}\} \end{array} \qquad [\textsc{Rcd-Del}]\ \frac{\Gamma \vdash e : t}{\Gamma \vdash e \setminus \ell : t \setminus \ell}\ t \leq \{\texttt{..}\}$$

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 CDuce [5, 15], which is a right-priority record concatenation defined to take into account undefined and unknown fields: for instance, $\{a\texttt{:int}, b\texttt{:int}\} + \{a\texttt{?:bool}\} = \{a\texttt{:int|bool}, b\texttt{:int}\}$; 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 $\{\texttt{..}\}$) we have $t + \{\texttt{..}\} = \{\texttt{..}\}$. 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 ... $\leq \{\texttt{..}\}$). See Appendix G 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.

## 4.3 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

$$[\textsc{Filter-App}]\ \frac{\Gamma \vdash e : t \qquad \Gamma \wr \varnothing \wr \varnothing \vdash_{fil} f(t) : s}{\Gamma \vdash fe : s}$$

The function `type` must be able to handle type environments with types of our system. It can do it either by subsuming variable with specific types to the types of the host language (*eg*, if the host language does not support singleton types then the singleton type `3` will be subsumed to `int`) or by typing foreign expressions by using our types.

relies on an auxiliary deduction system for judgments of the form $\Gamma \wr \Delta \wr M \vdash_{fil} f(t) : s$ that states that if in the environments $\Gamma, \Delta, M$ (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 $\lbag f \rbag$, 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 $\lbag f \rbag$ is the set of values defined by:

$$\begin{array}{lll}
\lbag e \rbag = \texttt{any} & & \lbag Xa \rbag = \texttt{any} \\
\lbag p \Rightarrow f \rbag = \lbag p \rbag \,\&\, \lbag f \rbag & & \lbag \boldsymbol{\mu} X.f \rbag = \lbag f \rbag \\
\lbag f_1 | f_2 \rbag = \lbag f_1 \rbag \,|\, \lbag f_2 \rbag & & \lbag \texttt{groupby}\ f \rbag = \texttt{[any*]} \\
\lbag (f_1,f_2) \rbag = (\lbag f_1 \rbag, \lbag f_2 \rbag) & & \lbag \texttt{orderby}\ f \rbag = \texttt{[any*]} \\
\lbag f_1;f_2 \rbag = \lbag f_1 \rbag & & \\
\lbag \{\ell_1 : f_1, .., \ell_n : f_n\ ,\ldots\} \rbag = \{\ell_1 : \lbag f_1 \rbag, .., \ell_n : \lbag f_2 \rbag\ ,\ldots\} & &
\end{array}$$

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 $v \notin \lbag f \rbag$. For every $\gamma$, $\delta$, if $\delta;\gamma \vdash_{eval} f(v) \rightsquigarrow r$, then $r \equiv \Omega$.*

The proof is a straightforward induction on the structure of the derivation, and is detailed in Appendix C. 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

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

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

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

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:

**Lemma 10 (Product decomposition).** *Let $t \in$ **Types** such that $t \leq \texttt{(any,any)}$. 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 $\texttt{rank}(t)$, 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 paper. We refer the interested reader to [15] and [23] for practical algorithms that compute such decompositions for any subtype of $\texttt{(any,any)}$ or of $\texttt{\{..\}}$. These notions of decomposition, rank and projection can be generalized to records:

**Lemma 11 (Record decomposition).** *Let $t \in$ **Types** such that $t \leq \texttt{\{..\}}$. 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 $\texttt{rank}(t)$, and use the notation $\boldsymbol{\rho}_\ell^j(t)$ for the type of label $\ell$ 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: $\Gamma$ : **Vars** $\rightarrow$ **Types** denoting *type environments* that associate term variables with their types; $\Delta$ : **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 $\Gamma \wr \Delta \wr M \vdash f(t) : t'$ stating that applying $f$ to an expression of type $t$ in the environments $\Gamma$, $\Delta$, $M$ 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 $\lbag f_1 \rbag$. 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 $\Delta$ 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.

### 4.4 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

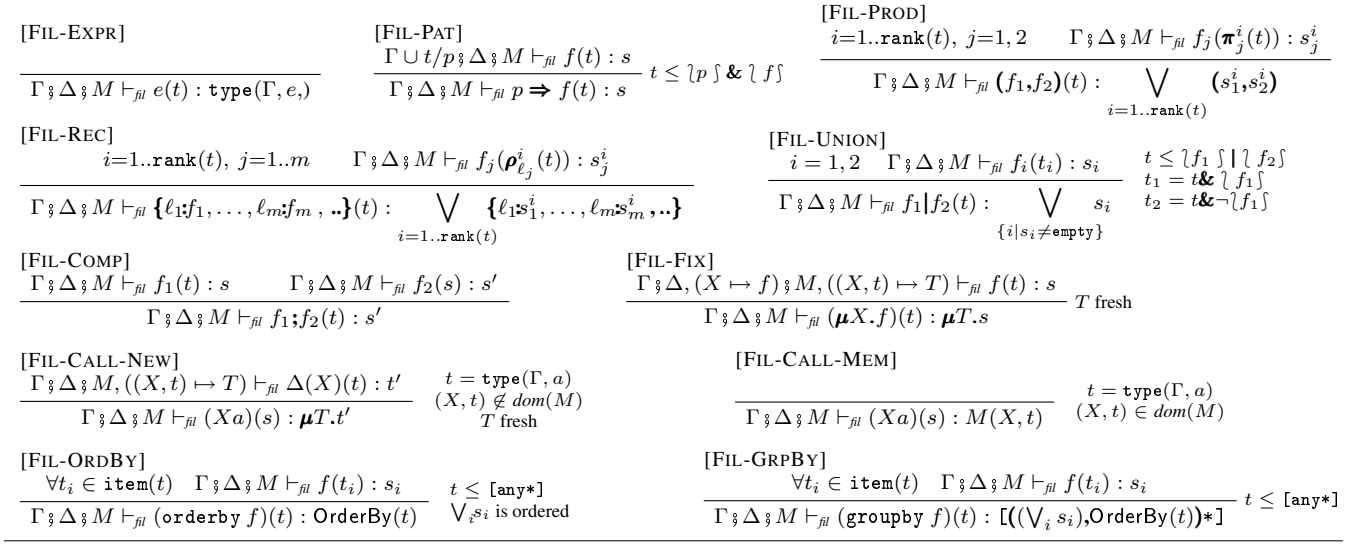


**Figure 2.** Type inference algorithm for filter application

compromise between precision and decidability. First we define an auxiliary function over sequence types:

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

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

The first and second line in the definition ensure that $\mathtt{item}()$ 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 $\mathtt{item}()$ 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 $\mathtt{item}()$ 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 $\mathsf{OrderBy}$ 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 Appendix E.

### 4.5 Soundness, termination, and complexity

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

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

whose full proof is given in Appendix C. It is easy to write a filter for which the type inference algorithm, that is the deduction of $\vdash_{\text{fil}}$, does not terminate: $\boldsymbol{\mu} X.x \Rightarrow X\mathtt{(}x\mathtt{,}x\mathtt{)}$. The deduction of $\Gamma \wr \Delta \wr M \vdash_{fil} f(t) : s$ 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 $\Gamma \wr \Delta \wr M \vdash_{fil} f(t) : s$ terminates. For space reasons the formal definition of *Check*$(f)$ is relegated to Appendix A, 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 $\Gamma \between \varnothing \between \varnothing \vdash_{fil} f(t) : s$ is in 2-EXPTIME. Furthermore, if $t$ is given as a non-deterministic tree automaton (NTA) then $\Gamma \between \varnothing \between \varnothing \vdash_{fil} f(t) : s$ is in EXPTIME, where the size of the problem is $|f| \times |t|$.*

(for proofs see Appendix A for termination and Appendix D for complexity). This complexity result is in line with those of similar formalisms. For instance in [20], 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 paper (excepted those in Appendix B) 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 $\iota_x$ the abstract subtree bound to the variable $x$, then the recursive call will be executed on the abstract argument $(\iota_x,\iota_z)$. So in the unfolding of the recursive call $x$ is bound to $\iota_x$, whereas $y$ and $z$ are bound to two distinct subtrees of $\iota_z$. 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 (*cf.* Appendix F) that typeable filters are strictly more expressive than top-down tree transducers with regular look-ahead, a formalism for tree transformations introduced in [14]. The intuition about this result can be conveyed by and example. 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. Then 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 [18], 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)
```

## 5.1 Built-in filters

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:

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

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))
```

## 5.2 Mapping

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

$$\begin{array}{lcl} [\![\texttt{c}]\!] & = & c \\ [\![x]\!] & = & x \\ [\![\$]\!] & = & \$ \\ [\![\{e_1:e'_1,\ldots,e_n:e'_n\}]\!] & = & \{[\![e_1]\!]:[\![e'_1]\!],\ldots,[\![e_n]\!]:[\![e'_n]\!]\} \\ [\![e.\texttt{l}]\!] & = & [\![e]\!].l \\ [\![\texttt{op}(e_1,\ldots,e_n)]\!] & = & \texttt{op}([\![e_1]\!],\ldots,[\![e_n]\!]) \\ [\![[e_1,\ldots,e_n]]\!] & = & ([\![e_1]\!],\ldots([\![e_n]\!],\texttt{`nil})\ldots) \\ [\![e \texttt{ -> } k]\!] & = & [\![e]\!];[\![k]\!]_{\texttt{F}} \end{array}$$

Jaql core expressions are mapped to filters as follows:

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

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.

### 5.3 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 3.3, 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 is given in Appendix H.

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 ex-

pressiveness, 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 [30] 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. This is shown in the section that follows.

## 6. JSON, XML, Regex

There exist various attempts to integrate JSON and XML. For instance JSONiq [30] is a query language designed to allow XML and JSON to be used in the same query. The motivation is that JSON and XML are both widely used for data interchange on the Internet. In many applications, JSON is replacing XML in Web Service APIs and data feeds, while more and more applications support both formats. More precisely, JSONiq embeds JSON into an XML query language (XQuery), but it does it in a stratified way: JSONiq does not allow XML nodes to contain JSON objects and arrays. The result is thus similar to OCamlDuce, the embedding of ℂDuce's XML types and expressions into OCaml, with the same drawbacks.

Our type system is derived from the type system of ℂDuce, whereas the theory of filters was originally designed to use ℂDuce as an host language. As a consequence XML types and expressions can be seamlessly integrated in the work presented here, without any particular restriction. To that end it suffices to use for XML elements and types the same encoding used in the implementation of ℂDuce, where an XML element is just a triple formed by a tag (here, an expression), a record (whose labels are the attributes of the element), and a sequence (of characters and or other XML elements) denoting its content. So for instance the following element

```
<product system="US-size">
  <number>557</number>
  <name>Blouse</name>
</product>
```

is encoded by the following triple:

```
("product" , { system : "US-size" } ,
   [
     ("number" , {} , [ 557 ])
     ("name", {}, "Blouse")
   ]
)
```

and this latter, with the syntactic sugar defined for ℂDuce, can be written as:

```
<product system="US-size">[
  <number>[ 557 ]
  <name>[ Blouse ]
]
```

Clearly in our system there are no restrictions in merging and nesting JSON and XML and no further extension is required to our system to define XML query and processing expressions. Just, the introduction of syntactic sugar to make the expressions readable, seems helpful:

$$\begin{array}{rcl} e & ::= & \texttt{<}e\ e\texttt{>}e \\ f & ::= & \texttt{<}f\ f\texttt{>}f \end{array}$$

The system we introduced here is already able to reproduce (and type) the same transformations as in JSONiq, but without the restrictions and drawbacks of the latter (this is why we 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 JSONiq). For instance, the example given in the JSONiq draft to show how to render in Xhtml the following JSON data:

```
{
  "col labels" : ["singular", "plural"],
  "row labels" : ["1p", "2p", "3p"],
  "data" :
     [
        ["spinne", "spinnen"],
        ["spinnst", "spinnt"],
        ["spinnt", "spinnen"]
     ]
}
```

can be encoded in the filters presented in this work (with the new syntactic sugar) as:

```
{ "col labels" : cl ,
  "row labels" : rl ,
  "data" : dl
} =>
 <table border="1" cellpadding="1" cellspacing="2">[
   <tr>[ <th>[ ] !(cl; Transform[ x -> <th>x ]) ]
   !(rl; Transform[ h ->
          <tr>[ <th>h !(dl; Transform[ x -> <td>x ]) ]
        ]
   )
 ]
```

(where `!` expands a subsequence in the containing sequence). The resulting Xhtml document is rendered in a web browser as:

| | singular | plural |
|---|---|---|
| 1p | spinne | spinnen |
| 2p | spinnst | spinnt |
| 3p | spinnt | spinnen |

Similarly, Jaql built-in libraries include functions to convert and manipulate XML data. So for example as it is possible in Jaql to embed SQL queries, so it is possible to evaluate XPath expressions, by the function `xpath()` which takes two arguments, an XML document and a string containing an xpath expression—*eg*, `xpath(read(seq(("conf/addrs.xml")),"content/city")` —. Filters can encode forward XPath expressions (see [23]) and precisely type them. So while in the current implementation there is no check of the type of the result of an external query (nor for

XPath or for SQL) and the XML document is produced independently from Jaql, by encoding (forward) XPath into filters we can not only precisely type calls to Jaql's `xpath()` function but also feed them with documents produced by Jaql expressions.

Finally, the very same regular expressions types that are used to describe heterogeneous sequences and, in particular, the content of XML elements, can be used to type regular expressions. Functions working on regular expressions (regex for short) form, in practice, yet another domain specific language that is embedded in general purpose languages in an untyped or weakly typed (typically, every result is of type `string`) way. Recursive patterns can straightforwardly encode regexp matching. Therefore, by combining the pattern filter with other filters it is possible to encode any regexp library, with the important advantage that, as stated by Theorem 4, the set of values (respectively, strings) accepted by a pattern (respectively, by a regular expression), can be precisely computed and can be expressed by a type. So a function such as the built-in Jaql's function `regex_extract()`, which extracts the substrings that match a given regex, can be easily implemented by filters and precisely typed by our typing rules. Typing will then amount to intersect the type of the string (which can be more precise than just `string`) with the type accepted by the pattern that encodes the regex at issue.

## 7. Programming with filters

Up to now we have used the filters to encode operators hard-coded in some languages in order to type them. Of course, it is possible to embed the typing technology we introduced directly into the compilers of these language so as to obtain the flexible typing that characterizes our system. However, an important aspect of filters we have ignored so far is that they can be used directly by the programmer to define user-defined operators that are typed as precisely as the hard-coded ones. Therefore a possibility is extend existing NoSQL languages by adding to their expressions the filter application $f\,e$ expression.

The next problem is to decide how far to go in the definition of filters. A complete integration, that is taking for $f$ all the definitions given so far, is conceivable but might disrupt the execution model of the host language, since the user could then define complex iterators that do not fit map-reduce or the chosen distributed compilation policy. A good compromise could be to add to the host language only filters which have "local" effects, thus avoiding to affect the map-reduce or distributed compilation execution model. The minimal solution consists in choosing just the filters for patterns, unions, and expressions:

$$f \;::=\; e \;\mid\; p \texttt{ => } f \;\mid\; f\texttt{|}f$$

Adding such filters to Jaql (we use the "=>" arrow for patterns in order to avoid confusion with Jaql's "->" pipe operator) would not allow the user to define powerful operators, but their use would already dramatically improve type precision. For instance we could define the following Jaql expression

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

(with the convention that a filter occurring as an expression denotes 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. As an example consider the following (extended) Jaql grouping expression:

```
group e by({town: ("Roma"|"Pisa"), ..} => "Italia"
          |{town: "Paris", ..} => "France"
          | _ => "?")
```

if $e$ has type `[{town:string, addr:string}*]`, then the type inferred by our system for this groupby expression is

```
[( ("Italia", [{town:"Roma"|"Pisa", addr:string}+])
  |("France", [{town:"Paris", addr:string}+])
  |("?", [{town:string/("Roma"|"Pisa"|"Paris"),
           addr:string}+])
 )*]
```

which precisely associates every key to the type of the elements it groups.

Finally, in order to allow a modular usage of filters, adding just filter application to the expression of the foreign language does not suffice: parametric filter definitions are also needed.

$$e ::= f e \;\mid\; \texttt{let filter}\, F\texttt{[}F_1,\ldots,F_n\texttt{]} = f\ \texttt{in}\ e$$

However, on the line of what we already said about the disruption of the execution model, recursive parametric filter definitions should be probably disallowed, since a compilation according to a map-reduce model would require to disentangle recursive calls.

## 8. 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 $(x,y) \Rightarrow (f_1x, f_2y)$ and similarly for records. The point is that $f_1x$ and $f_2y$ are expressions —and thus their pair is a filter— only if the $f_i$'s are closed (*ie*, wihtout 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

Syntactically we could write $\mu X.\texttt{`nil} \Rightarrow \texttt{`nil}|(h,t) \Rightarrow (h,(Xt)v)$ where $v$ is any value, but then this would not pass type-checking since the expression $(Xt)v$ must be typeable without knowing the $\Delta$ environment (cf. rule [FILTER-APP] at the beginning of Section 4.3). We purposedly stratified the system in order to avoid mutual recursion between filters and expressions.

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:*** The definition of records is redundant (both for types and patterns). Instead of the current definition we could have used just $\{\ell{:}t\}$, $\{\}$, and $\{..\}$, since the rest can be encoded by intersections. For instance, $\{\ell_1{:}t_1,\ldots,\ell_n{:}t_n\,,..\} = \{\ell_1{:}t\}\&...\&\{\ell_n{:}t_n\}\&\{..\}$. We opted to use the redundant definition for the sake of clarity. 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 paper, 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 7 keeps track of the `c` field) and therefore open records suffice.

***Record selection:*** Some languages —typically, the dynamic ones such as Javascript, Ruby, Python— allow the label of a field selection to be computed by an expression. We considered the definition of a fine-grained rule to type expressions of the form $e_1.e_2$: whenever $e_2$ is typed by a finite unions of strings, the rule would give a finite approximation of the type of the selection. However, such an extension would complex the definition of the type system, just to handle few interesting cases in which a finite union type can be deduced. Therefore, we preferred to omit its study and leave it for future work.

## 9. 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 [32] or in Smalltalk [12]. Also, monads or comprehensions [8, 34, 35] 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, 25, 26] and more recently [27] 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, 23] (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 [23]) is novel and much needed to encode JSON transformations.

## 10. 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.

As such we accounted for both components that, according to Landin, constitute the design of a language: operators and data structures. But while Landin considers the design of terms and types as independent activities we, on the contrary, advocate an approach in which the design of former is *driven* by the form of latter. Although other approaches are possible, we tried to convey the idea that this approach is nevertheless the only one that yields a type system whose precision, that we demonstrated all the work long, is comparable only to the precision obtained with hard-coded (as opposed to user-defined) operators. As such, our 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.

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 provide an early 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 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. Bottom-up and Top-down Tree Transformations – A comparison. *Theory of Computing Systems*, 9(2):198–231, 1975.

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

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

[16] 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.

[17] H. Hosoya. *Foundations of XML Processing: The Tree Automata Approach*. Cambridge University Press, 2010.

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

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

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

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

[22] 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.

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

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

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

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

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

[28] 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.

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

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

[31] A. Sabry and P. Wadler. A reflection on call-by-value. In *ICFP '96*. ACM, 1996.

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

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

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

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

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


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".

# Appendix

## A. Termination analysis algorithm

In order to deduce the result type of the application of a filter to an expression, the type inference algorithm abstractly executes the filter on the type of the expression. As explained in Section 4, the algorithm essentially analyzes what may happen to the original input type (and to its subtrees) after a recursive call and checks that, in all possible cases, every subsequent recursive call will only be applied to subtrees of the original input type. In order to track the subtrees of the original input type, we use an infinite set $\mathbf{Ids} = \{\iota_1, \iota_2, \ldots\}$ of *variable identifiers* (ranged over by, possible indexed, $\iota$). These identifiers are used to identify the subtrees of the original input type that are bound to some variable after a recursive call. Consider for instance the recursive filter:

$$\boldsymbol{\mu} X.\mathbf{(}x_1\mathbf{,(}x_2\mathbf{,}x_3\mathbf{))} \Rightarrow X\mathbf{(}x_1\mathbf{,}x_3\mathbf{)} \mid \_ \Rightarrow \texttt{`nil}.$$

The algorithm records that at the first call of this filter each variable $x_i$ is bound to a subtree $\iota_i$ of the input type. The recursive call $X\mathbf{(}x_1\mathbf{,}x_3\mathbf{)}$ is thus applied to the "abstract argument" $\mathbf{(}\iota_1\mathbf{,}\iota_3\mathbf{)}$. If we perform the substitutions for this call, then we see that $x_1$ will be bound to $\iota_1$ and that $x_2$ and $x_3$ will be bound to two subtrees of the $\iota_3$ subtree. We do not *care* about $x_2$ since it is not used in any recursive call. What is important is that both $x_1$ and $x_3$ are bound to subtrees of the original input type (respectively, to $\iota_1$ and to a subtree of $\iota_3$) and therefore, for this filter, the type inference algorithm will terminate for every possible input type.

The previous example introduces the two important concepts that are still missing for the definition of our analysis:

1. Recursive calls are applied to *symbolic arguments*, such as $\mathbf{(}\iota_1\mathbf{,}\iota_3\mathbf{)}$, that are obtained from arguments by replacing variables by variable identifiers. Symbolic arguments are ranged over by $A$ and are formally defined as follows:

| | | | | |
|---|---|---|---|---|
| **Symb Args** | $A$ | ::= | $\iota$ | **(variable identifiers)** |
| | | \| | $c$ | **(constants)** |
| | | \| | $\mathbf{(}A\mathbf{,}A\mathbf{)}$ | **(pairs)** |
| | | \| | $\mathbf{\{}\ell\mathbf{:}A, \ldots, \ell\mathbf{:}A\mathbf{\}}$ | **(records)** |
| | | \| | $\bot$ | **(indeterminate)** |

2. We said that we "care" for some variables and disregard others. When we analyze a recursive filter $\boldsymbol{\mu} X.f$ the variables we care about are those that occur in arguments of recursive calls of $X$. Given a filter $f$ and a filter recursion variable $X$ the set of these variables is formally defined as follows

$$\mathit{Mark}_X(f) = \{x \mid Xa \sqsubseteq f \text{ and } x \in \mathsf{vars}(a)\}$$

where $\sqsubseteq$ denotes the subtree containment relation and $\mathsf{vars}(e)$ is the set of expression variables that occur free in $e$. With an abuse of notation we will use $\mathsf{vars}(p)$ to denote the capture variables occurring in $p$ (thus, $\mathsf{vars}(fe) = \mathsf{vars}(f) \cup \mathsf{vars}(e)$ and $\mathsf{vars}(p \Rightarrow f) = \mathsf{vars}(f) \setminus \mathsf{vars}(p)$, the rest of the definition being standard).

As an aside notice that the fact that for a given filter $f$ the type inference algorithm terminates on all possible input types *does not* imply that the execution of $f$ terminates on all possible input values. For instance, our analysis correctly detects that for the filter $\boldsymbol{\mu} X.\mathbf{(}x_1\mathbf{,}x_2\mathbf{)} \Rightarrow X\mathbf{(}x_1\mathbf{,}x_2\mathbf{)|}\_ \Rightarrow \_$ type inference terminates on all possible input types (by returning the very same input type) although the application of this same filter never terminates on arguments of the form $\mathbf{(}v_1\mathbf{,}v_2\mathbf{)}$.

We can now formally define the two phases of our analysis algorithm.

***First phase.*** The first phase is implemented by the function $\mathit{Trees}\_(\_,\_)$. For every filter recursion variable $X$, this function explores a filter and does the following two things:

1. It builds a substitution $\boldsymbol{\sigma}$ : **Vars** $\rightarrow$ **Ids** from expression variables to variables identifiers, thus associating each capture variable occurring in a pattern of the filter to a fresh identifier for the abstract subtree of the input type it will be bound to.
2. It uses the substitution $\boldsymbol{\sigma}$ to compute the set of symbolic arguments of recursive calls of $X$.

In other words, if $n$ recursive calls of $X$ occur in the filter $f$, then $\mathit{Trees}_X(\boldsymbol{\sigma}, f)$ returns the set $\{A_1, ..., A_n\}$ of the $n$ symbolic arguments of these calls, obtained under the hypothesis that the free variables of $f$ are associated to subtrees as specified by $\boldsymbol{\sigma}$. The formal definition is as follows:

$$\begin{array}{lcll}
\mathit{Trees}_X(\boldsymbol{\sigma}, e) & = & \varnothing & \\
\mathit{Trees}_X(\boldsymbol{\sigma}, p \Rightarrow f) & = & \mathit{Trees}_X(\boldsymbol{\sigma} \cup \bigcup\limits_{x_i \in \mathsf{vars}(p)} \{x_i \mapsto \iota_i\}, f) & (\iota_i \text{ fresh}) \\
\mathit{Trees}_X(\boldsymbol{\sigma}, \mathbf{(}f_1\mathbf{,}f_2\mathbf{)}) & = & \mathit{Trees}_X(\boldsymbol{\sigma}, f_1) \cup \mathit{Trees}_X(\boldsymbol{\sigma}, f_2) & \\
\mathit{Trees}_X(\boldsymbol{\sigma}, f_1\mathbf{|}f_2) & = & \mathit{Trees}_X(\boldsymbol{\sigma}, f_1) \cup \mathit{Trees}_X(\boldsymbol{\sigma}, f_2) & \\
\mathit{Trees}_X(\boldsymbol{\sigma}, \boldsymbol{\mu} X.f) & = & \varnothing & \\
\mathit{Trees}_X(\boldsymbol{\sigma}, \boldsymbol{\mu} Y.f) & = & \mathit{Trees}_X(\boldsymbol{\sigma}, f) & (X \neq Y) \\
\mathit{Trees}_X(\boldsymbol{\sigma}, Xa) & = & \{a\boldsymbol{\sigma}\} & \\
\mathit{Trees}_X(\boldsymbol{\sigma}, f_1\mathbf{;}f_2) & = & \mathit{Trees}_X(\boldsymbol{\sigma}, f_2) & \\
\mathit{Trees}_X(\boldsymbol{\sigma}, of) & = & \mathit{Trees}_X(\boldsymbol{\sigma}, f) & (o = \texttt{groupby}\ , \texttt{orderby}\ ) \\
\mathit{Trees}_X(\boldsymbol{\sigma}, \mathbf{\{}\ell_i\mathbf{:}f_i\ ,\ \mathbf{..\}}_{i \in I}) & = & \bigcup\limits_{i \in I} \mathit{Trees}_X(\boldsymbol{\sigma}, f_i) &
\end{array}$$

where $a\boldsymbol{\sigma}$ denotes the application of the substitution $\boldsymbol{\sigma}$ to $a$.

The definition above is mostly straightforward. The two important cases are those for the pattern filter where the substitution $\boldsymbol{\sigma}$ is updated by associating every capture variable of the pattern with a fresh variable identifier, and the one for the recursive call where the symbolic argument is computed by applying $\boldsymbol{\sigma}$ to the actual argument.

***Second phase.*** The second phase is implemented by the function $\mathit{Check}()$. The intuition is that $\mathit{Check}(f)$ must "compute" the application of $f$ to all the symbolic arguments collected in the first phase and then check whether the variables occurring in the arguments of recursive calls (*ie*, the "marked" variables) are actually bound to subtrees of the original type (*ie*, they are bound either to variable identifiers or to subparts of variable identifiers). If we did not have filter composition, then this would be a relatively easy task: it would amount to compute substitutions (by matching symbolic arguments against patterns), apply them, and finally verify that the marked variables satisfy the sought property. Unfortunately, in the case of a composition filter $f_1\mathbf{;}f_2$ the analysis is more complicated than that. Imagine that we want to check the property for the application of $f_1\mathbf{;}f_2$ to a symbolic argument $A$. Then to check the property for the recursive calls occurring in $f_2$ we must compute (or at least, approximate) the set of the symbolic arguments that will be produced by the application of $f_1$ to $A$ and that will be thus fed into $f_2$ to compute the composition. Therefore $\mathit{Check}(f)$ will be a function that rather than returning just true or false, it will return a set of symbolic arguments that are the result of the execution of $f$, or it will fail if any recursive call does not satisfy the property for marked variables.

More precisely, the function $\mathit{Check}()$ will have the form

$$\mathit{Check}_{\mathcal{V}}(\boldsymbol{\sigma}, f, \{A_1, ..., A_n\})$$

where $\mathcal{V} \subseteq$ **Vars** stores the set of marked variables, $f$ is a filter, $\boldsymbol{\sigma}$ is a substitution for (at least) the free variables of $f$ into **Ids**, and $A_i$ are symbolic arguments. It will either fail if the marked variables do not satisfy the property of being bound to (subcomponents of) variable identifiers or return an over-approximation of the result of applying $f$ to all the $A_i$ under the hypothesis $\boldsymbol{\sigma}$. This approximation is either a set of new symbolic arguments, or $\bot$. The latter simply indicates that $\mathit{Check}()$ is not able to compute the result of the application, typically because it is the result of some expression

- $\mathit{Check}_{\mathcal{V}}(\boldsymbol{\sigma}, f, \{A_1, ..., A_n\}) = \bigcup_{i=1}^{n} \mathit{Check}_{\mathcal{V}}(\boldsymbol{\sigma}, f, A_i)$
- **If** $(\rceil A \lceil \& \lfloor\, f \rfloor = \varnothing)$ **then** $\mathit{Check}_{\mathcal{V}}(\boldsymbol{\sigma}, f, A) = \varnothing$ **otherwise:**

$$
\begin{array}{rcl}
\mathit{Check}_{\mathcal{V}}(\boldsymbol{\sigma}, e, A) & = & \begin{cases} \{a\boldsymbol{\sigma}\} & \text{if } e \equiv a \\ \bot & \text{otherwise} \end{cases} \\[2ex]
\mathit{Check}_{\mathcal{V}}(\boldsymbol{\sigma}, f_1\mathbf{;}f_2, A) & = & \mathit{Check}_{\mathcal{V}}(\boldsymbol{\sigma}, f_2, \mathit{Check}_{\mathcal{V}}(\boldsymbol{\sigma}, f_1, A)) \\[2ex]
\mathit{Check}_{\mathcal{V}}(\boldsymbol{\sigma}, Xa, A) & = & \bot \\[2ex]
\mathit{Check}_{\mathcal{V}}(\boldsymbol{\sigma}, \boldsymbol{\mu}X\mathbf{.}f, A) & = & \begin{cases} \mathit{fail} & \text{if } \mathit{Check}_{\mathit{Mark}_X(f)}(\boldsymbol{\sigma}, f, A) = \mathit{fail} \\ \bot & \text{otherwise} \end{cases} \\[2ex]
\mathit{Check}_{\mathcal{V}}(\boldsymbol{\sigma}, f_1\mathbf{|}f_2, A) & = & \begin{cases} \mathit{Check}_{\mathcal{V}}(\boldsymbol{\sigma}, f_2, A) & \text{if } \lfloor\, f_1 \,\rfloor \mathbf{\&} \rceil A \lceil = \varnothing \\ \mathit{Check}_{\mathcal{V}}(\boldsymbol{\sigma}, f_1, A) & \text{if } \rceil A \lceil \setminus \lfloor f_1 \rfloor = \varnothing \\ \mathit{Check}_{\mathcal{V}}(\boldsymbol{\sigma}, f_1, A) \cup \mathit{Check}_{\mathcal{V}}(\boldsymbol{\sigma}, f_2, A) & \text{otherwise} \end{cases} \\[2ex]
\mathit{Check}_{\mathcal{V}}(\boldsymbol{\sigma}, \mathbf{(}f_1\mathbf{,}f_2\mathbf{)}, A) & = & \begin{cases} \mathit{Check}_{\mathcal{V}}(\boldsymbol{\sigma}, f_1, A_1) \times \mathit{Check}_{\mathcal{V}}(\boldsymbol{\sigma}, f_2, A_2) & \text{if } A \equiv (A_1, A_2) \\ \mathit{Check}_{\mathcal{V}}(\boldsymbol{\sigma}, f_1, \iota_1\ ) \times \mathit{Check}_{\mathcal{V}}(\boldsymbol{\sigma}, f_2, \iota_2) & \text{if } A \equiv \iota \quad \text{(where } \iota_1 \text{ and } \iota_2 \text{ are fresh)} \\ \mathit{fail} & \text{if } A \equiv \bot \end{cases} \\[2ex]
\mathit{Check}_{\mathcal{V}}(\boldsymbol{\sigma}, \{\ell_i\mathbf{:}f_i\ \mathbf{..}\}_{1\le i\le n}, A) & = & \begin{cases} \bigcup_{B_i \in \mathit{Check}_{\mathcal{V}}(\boldsymbol{\sigma}, f_i, A_i)} \{\ell_1\mathbf{:}B_1, ..., \ell_n\mathbf{:}B_n, ..., \ell_{n+k}\mathbf{:}A_{n+k}\ \mathbf{..}\} & \text{if } A \equiv \{\ell_i\mathbf{:}f_i\ \mathbf{..}\}_{1\le i\le n+k} \\ \bigcup_{B_i \in \mathit{Check}_{\mathcal{V}}(\boldsymbol{\sigma}, f_i, \iota_i)} \{\ell_1\mathbf{:}B_1, ..., \ell_n\mathbf{:}B_n, ..., \ell_{n+k}\mathbf{:}A_{n+k}\ \mathbf{..}\} & \text{if } A \equiv \iota \quad \text{(where all } \iota_i \text{ are fresh)} \\ \mathit{fail} & \text{if } A \equiv \bot \end{cases} \\[2ex]
\mathit{Check}_{\mathcal{V}}(\boldsymbol{\sigma}, p \Rightarrow f, A) & = & \begin{cases} \mathit{fail} & \text{if } A \not\backslash p = \mathit{fail} \\ \mathit{Check}_{\mathcal{V}}(\boldsymbol{\sigma} \cup (A \not\backslash p), f, A) & \text{otherwise} \end{cases}
\end{array}
$$

**Figure 3.** Definition of the second phase of the analysis

belonging to the host language (*eg*, the application of a function) or because it is the result of a recursive filter or of a recursive call of some filter.

The full definition of $\mathit{Check}_{\_}(\_,\_,\_)$ is given in Figure 3. Let us comment the different cases in detail.

$$\mathit{Check}_{\mathcal{V}}(\boldsymbol{\sigma}, f, \{A_1, ..., A_n\}) = \bigcup_{i=1}^{n} \mathit{Check}_{\mathcal{V}}(\boldsymbol{\sigma}, f, A_i)$$

simply states that to compute the filter $f$ on a set of symbolic arguments we have to compute $f$ on each argument and return the union of the results. Of course, if any $\mathit{Check}_{\mathcal{V}}(\boldsymbol{\sigma}, f, A_i)$ fails, so does $\mathit{Check}_{\mathcal{V}}(\boldsymbol{\sigma}, f, \{A_1, ..., A_n\})$. The next case is straightforward: if we know that an argument of a given form makes the filter $f$ fail, then we do not perform any check (the recursive call of $f$ will never be called) and no result will be returned. For instance, if we apply the filter $\{\ell\mathbf{:}f\ \mathbf{..}\}$ to the symbolic argument $(\iota_1\mathbf{,}\iota_2)$, then this will always fail so it is useless to continue checking this case. Formally, what we do is to consider $\rceil A \lceil$, the set of all values that have the form of $A$ (this is quite simply obtained from $A$ by replacing `any` for every occurrence of a variable identifier or of $\bot$) and check whether in it there is any value accepted by $f$, namely, whether the intersection $\rceil A \lceil \& \lfloor\, f \rfloor$ is empty. If it is so, we directly return the empty set of symbolic arguments, otherwise we have to perform a fine grained analysis according to the form of the filter.

If the filter is an expression, then there are two possible cases: either the expression has the form of an argument ($\equiv$ denotes syntactic equivalence), in which case we return it after having applied the substitution $\boldsymbol{\sigma}$ to it; or it is some other expression, in which case the function says that it is not able to compute a result and returns $\bot$.

When the filter is a composition of two filters, $f_1\mathbf{;}f_2$, we first call $\mathit{Check}_{\mathcal{V}}(\boldsymbol{\sigma}, f_1, A)$ to analyze $f_1$ and compute the result of applying $f_1$ to $A$, and then we feed this result to the analysis of $f_2$:

$$\mathit{Check}_{\mathcal{V}}(\boldsymbol{\sigma}, f_1\mathbf{;}f_2, A) = \mathit{Check}_{\mathcal{V}}(\boldsymbol{\sigma}, f_2, \mathit{Check}_{\mathcal{V}}(\boldsymbol{\sigma}, f_1, A))$$

When the filter is recursive or a recursive call, then we are not able to compute the symbolic result. However, in the case of a recursive filter we must check whether the type inference terminates on it. So we mark the variables of its recursive calls, check whether its definition passes our static analysis, and return $\bot$ only if this analysis —*ie*, $\mathit{Check}_{\mathit{Mark}_X(f)}(\boldsymbol{\sigma}, f, A)$— did not fail. Notice that $\bot$ is quite different from failure since it allows us to compute the approximation as long as the filter after the composition does not bind (parts of) the result that are $\bot$ to variables that occur in arguments of recursive calls. A key example is the case of the filter `Filter[`$F$`]` defined in Section 5.1. If the parameter filter $F$ is recursive, then without $\bot$ `Filter[`$F$`]` would be rejected by the static analysis. Precisely, this is due to the composition $Fx\mathbf{;}(\texttt{`false} \Rightarrow Xl \mid \cdots)$. When $F$ is recursive the analysis supposes that $Fx$ produces $\bot$, which is then passed over to the pattern. The recursive call $Xl$ is executed in the environment $\bot/\texttt{`true}$, which is empty. Therefore

$$
\begin{array}{rcl}
A/x & = & \begin{cases} \{x \mapsto A\} & \text{if } A \equiv v \text{ or } A \equiv \iota \text{ or } x \notin \mathcal{V} \\ \mathit{fail} & \text{otherwise} \end{cases} \\[2ex]
A/(p_1,p_2) & = & \begin{cases} (A_1/p_1) \cup (A_2/p_2) & \text{if } A \equiv (A_1,A_2) \\ (\iota_1/p_1) \cup (\iota_2/p_2) & \text{if } A \equiv \iota \quad \text{(where } \iota_1, \iota_2 \text{ are fresh)} \\ \mathit{fail} & \text{if } A \equiv \bot \\ \Omega & \text{otherwise} \end{cases} \\[2ex]
A/p_1 \& p_2 & = & A/p_1 \cup A/p_2 \\[2ex]
A/p_1 | p_2 & = & \begin{cases} A/p_1 & \text{if } A/p_1 \neq \Omega \\ A/p_2 & \text{otherwise} \end{cases} \\[2ex]
A/\{\ell_1{:}p_1, ..., \ell_n{:}p_n\} & = & \begin{cases} \bigcup_{i \leq i \leq n} A_i/p_i & \text{if } A \equiv \{\ell_1{:}A_1, ..., \ell_n{:}A_n\} \\ \bigcup_{i \leq i \leq n} \iota_i/p_i & \text{if } A \equiv \iota \quad \text{(where } \iota_i \text{ are fresh)} \\ \mathit{fail} & \text{if } A \equiv \bot \\ \Omega & \text{otherwise} \end{cases} \\[2ex]
A/\{\ell_1{:}p_1, ..., \ell_n{:}p_n, ..\} & = & \begin{cases} \bigcup_{i \leq i \leq n} A_i/p_i & \text{if } A \equiv \{\ell_1{:}A_1, ..., \ell_n{:}A_n, ..., \ell_{n+k}{:}A_{n+k}\} \\ \bigcup_{i \leq i \leq n} \iota_i/p_i & \text{if } A \equiv \iota \quad \text{(where } \iota_i \text{ are fresh)} \\ \mathit{fail} & \text{if } A \equiv \bot \\ \Omega & \text{otherwise} \end{cases} \\[2ex]
A/t & = & \varnothing
\end{array}
$$

**Figure 4.** Pattern matching with respect to a set $\mathcal{V}$ of marked variables

the result of the $Fx$ call, whatever it is, cannot affect the termination of the subsequent recursive calls. Even though the composition uses the result of $Fx$, the form of this result cannot be such as to make typechecking of `Filter[`$F$`]` diverge.

For the union filter $f_1|f_2$ there are three possible cases: $f_1$ will always fail on $A$, so we return the result of $f_2$; or $f_2$ will never be executed on $A$ since $f_1$ will never fail on $A$, so we return just the result of $f_1$; or we cannot tell which one of $f_1$ or $f_2$ will be executed, and so we return the union of the two results.

If the filter is a product filter $(f_1,f_2)$, then its accepted type $\lbag(f_1,f_2)\rbag$ is a subtype of $(\mathtt{any},\mathtt{any})$. Since we are in the case where $\lceil A \rceil \& \lbag(f_1,f_2)\rbag$ is not empty, then $A$ can have just two forms: either it is a pair or it is a variable identifier $\iota$. In the former case we check the filters component-wise and return as result the set-theoretic product of the results. In the latter case, recall that $\iota$ represents a subtree of the original input type. So if the application does not fail it means that each subfilter will be applied to a subtree of $\iota$. We introduce two fresh variable identifiers to denote these subtrees of $\iota$ (which, of course are subtrees of the original input type, too) and, as in the previous case, apply the check component-wise and return the set-theoretic product of the results. The case for the record filter is similar to the case of the product filter.

Finally, the case of pattern filter is where the algorithm checks that the marked variables are indeed bound to subtrees of the initial input type. What *Check*() does is to match the symbolic argument against the pattern and update the substitution $\boldsymbol{\sigma}$ so as to check the subfilter $f$ in an environment with the corresponding assignments for the capture variables in $p$. Notice however that the pattern matching $A \overset{\mathcal{V}}{\not\backslash} p$ receives as extra argument the set $\mathcal{V}$ of marked variables, so that while computing the substitutions for the capture variables in $p$ it also checks that all capture variables that are also marked are actually bound to a subtree of the initial input type. $A \overset{\mathcal{V}}{\not\backslash} p$ is defined in Figure 4 (to enhance readability we omitted the $\mathcal{V}$ index since it matters only in the first case and it does not change along the definition). The check for marked variables is performed in the first case of the definition: when a symbolic argument $A$ is matched against a variable $x$, if the variable is not marked ($x \notin \mathcal{V}$), then the substitution $\{x \mapsto A\}$ is returned; if it is marked, then the matching (and thus the analysis) does not fail only if the symbolic argument is either a value (so it does not depend on the input type) or it is a variable identifier (so it will be exactly a subtree of the original input type). The other cases of the definition of $A \overset{\mathcal{V}}{\not\backslash} p$ are standard. Just notice that in the product and record patterns when $A$ is a variable identifier $\iota$, the algorithm creates fresh new variable identifiers to denote the new subtrees in which $\iota$ will be decomposed by the pattern. Finally, the reader should not confound $\Omega$ and *fail*. The former indicates that the argument does not match the value, while the latter indicates that a marked variable is not bound to a value or to exactly a subtree of the initial input type (notice that the case $\Omega$ cannot happen in the definition of *Check*() since $A \overset{\mathcal{V}}{\not\backslash} p$ is called only when $\lceil A \rceil$ is contained in $\lbag p \rbag$).

***The two phases together.*** Finally we put the two phases together. Given a recursive filter $\boldsymbol{\mu} X.f$ we run the analysis by marking the variables in the recursive calls of $X$ in $f$, running the first phase and then and feeding the result to the second phase: $\mathit{Check}_{\mathit{Mark}_X(f)}(\varnothing, f, \mathit{Trees}_X(\varnothing, f))$. If this function does not fail then we say that $\boldsymbol{\mu} X.f$ passed the check:

$$\mathit{Check}(\boldsymbol{\mu} X.f) \iff \mathit{Check}_{\mathit{Mark}_X(f)}(\varnothing, f, \mathit{Trees}_X(\varnothing, f)) \neq \mathit{fail}$$

For filters that are not recursive at toplevel it suffices to add a dummy recursion: $\mathit{Check}(f) \stackrel{\text{def}}{=} \mathit{Check}(\boldsymbol{\mu} X.f)$ with $X$ fresh.

**Theorem 16 (Termination).** *If $\mathit{Check}(f)$ then the type inference algorithm terminates for $f$ on every input type $t$.*

In order to prove the theorem above we need some auxiliary definitions:

**Definition 17 (Plinth [15]).** A plinth $\boldsymbol{\beth} \subset \mathbf{Types}$ is a set of types with the following properties:

- $\boldsymbol{\beth}$ is finite
- $\boldsymbol{\beth}$ contains `any`, `empty` and is closed under Boolean connectives (&,|,¬)
- for all types $t = (t_1,t_2)$ in $\boldsymbol{\beth}$, $t_1 \in \boldsymbol{\beth}$ and $t_2 \in \boldsymbol{\beth}$
- for all types $t = \{\ell_1{:}t_1, \ldots, \ell_n{:}t_n, (..)\}$ in $\boldsymbol{\beth}$, $t_i \in \boldsymbol{\beth}$ for all $i \in [1..n]$.

We define the plinth of $t$, noted $\boldsymbol{\beth}(t)$, as the smallest plinth containing $t$.

Intuitively, the plinth of $t$ is the set of types that can be obtained by all possible boolean combination of the subtrees of $t$. Notice that $\boldsymbol{\beth}(t)$ is always defined since our types are regular: they have finitely many distinct subtrees, which (modulo type equivalence) can thus be combined in finitely many different ways.

**Definition 18 (Extended plinth and support).** Let $t$ be a type and $f$ a filter. The *extended plinth* of $t$ and $f$, noted $\hat{\boldsymbol{\beth}}(f,t)$ is defined as $\boldsymbol{\beth}(t \vee \bigvee_{v \sqsubseteq f} v)$ (where $v$ ranges over values).

The *support* of $t$ and $f$, noted as $\mathit{Support}(f,t)$, is defined as

$$\mathit{Support}(f,t) \stackrel{\text{def}}{=} \hat{\boldsymbol{\beth}}(f,t) \cup \bigcup_{A \in \mathit{Trees}_\varnothing(\varnothing,f)} \{A\sigma \mid \sigma : \mathbf{Ids}(A) \to \hat{\boldsymbol{\beth}}(f,t)\}$$

The extended plinth of $t$ and $f$ is the set of types that can be obtained by all possible boolean combination of the subtrees of $t$ and of values that occur in the filter $f$. The intuition underlying the definition of support of $t$ and $f$ is that it includes all the possible types of arguments of recursive calls occurring in $f$, when $f$ is applied to an argument of type $t$.

Lastly, let us prove the following technical lemma:

**Lemma 19.** *Let $f$ be a filter such that $\mathit{Check}(f)$ holds. Let $t$ be a type. For every derivation $D$ (finite or infinite) of*

$$\Gamma \wr \Delta \wr M \vdash_{\mathit{fil}} f(t) : s \quad ,$$

*for every occurrence of the rule* [FIL-CALL-NEW]

$$\frac{\Gamma' \wr \Delta' \wr M', ((X,t'') \mapsto T) \vdash_{\mathit{fil}} \Delta'(X)(t'') : t'}{\Gamma' \wr \Delta' \wr M' \vdash_{\mathit{fil}} (X\ a)(s) : \boldsymbol{\mu} T.t'} \quad \begin{array}{c} t'' = \mathtt{type}(\Gamma', a) \\ (X,t'') \notin \mathit{dom}(M') \\ T \text{ fresh} \end{array}$$

*in $D$, for every $x \in \mathtt{vars}(a)$, $\Gamma'(x) \in \hat{\boldsymbol{\beth}}(f,t)$ (or equivalently, $\mathtt{type}(\Gamma', a) \in \mathit{Support}(f,t)$)*

*Proof.* By contradiction. Suppose that there exists an instance of the rule for which $x \in \mathtt{vars}(a) \wedge x \notin \hat{\boldsymbol{\beth}}(f,t)$. This means that $\Gamma'(x)$ is neither a singleton type occurring in $f$ nor a type obtained from $t$ by applying left projection, right projection or label selection ($\star$). But since $\mathit{Check}(f)$ holds, $x$ must be bound to either an identifier or a value $v$ (see Figure 4) during the computation of $\mathit{Check}_{\mathit{Mark}_X(f)}(\varnothing, f, \mathit{Trees}_X(\varnothing, f))$. Since identifiers in $\mathit{Check}(f)$ are only introduced for the input parameter or when performing a left projection, right projection or label selection of another identifier, this ensure that $x$ is never bound to the result of an expression whose type is not in $\hat{\boldsymbol{\beth}}(f,t)$, which contradicts ($\star$). □

We are now able to prove Theorem 16. Let $t$ be a type and $f$ a filter define

$$\mathcal{D}(f,t) = \{(X,t') \mid X \sqsubseteq f, t' \in \mathit{Support}(f,t)\}$$

Notice that since $\mathit{Support}(f,t)$ is finite and there are finitely many different recursion variables occuring in a filter, then $\mathcal{D}(f,t)$ is finite, too. Now let $f$ and $t$ be the filter and type mentioned in the statement of Theorem 16 and consider the (possibly infinite) derivation of $\Gamma \wr \Delta \wr M \vdash_{\mathit{fil}} f(t) : s$. Assign to every judgment $\Gamma' \wr \Delta' \wr M' \vdash_{\mathit{fil}} f'(t') : s'$ the following weight

$$\mathit{Wgt}(\Gamma' \wr \Delta' \wr M' \vdash_{\mathit{fil}} f'(t') : s') = (|\mathcal{D}(f,t) \backslash \mathit{dom}(M')|, \mathit{size}(f'))$$

where $|S|$ denotes the cardinality of the set $S$ (notice that in the definition $S$ is finite), $\mathit{dom}(M')$ is the domain of $M'$, that is, the set of pairs $(X,t)$ for which $M'$ is defined, and $\mathit{size}(f')$ is the depth of the syntax tree of $f'$.

Notice that the set of all weights lexicographically ordered form a well-founded order. It is then not difficult to prove that every application of a rule in the derivation of $\Gamma \wr \Delta \wr M \vdash_{\mathit{fil}} f(t) : s$ strictly decreases *Wgt*, and therefore that this derivation must be finite. This can be proved by case analysis on the applied rule, and we must distinguish only three cases:

[FIL-FIX] Notice that in this case the first component of *Wgt* either decreases or remains constant, and the second component strictly decreases.

[FIL-CALL-NEW] In this case Lemma 19 ensures that the first component of the *Wgt* of the premise strictly decreases. Since this is the core of the proof let us expand the rule. Somewhere in the derivation of $\Gamma \wr \Delta \wr M \vdash_{\mathit{fil}} f(t) : s$ we have the following rule:

[FIL-CALL-NEW]
$$\frac{\Gamma' \wr \Delta' \wr M', ((X,t'') \mapsto T) \vdash_{\mathit{fil}} \Delta'(X)(t'') : t'}{\Gamma' \wr \Delta' \wr M' \vdash_{\mathit{fil}} (X a)(s) : \boldsymbol{\mu} T.t'} \quad \begin{array}{c} t'' = \mathtt{type}(\Gamma', a) \\ (X,t'') \notin \mathit{dom}(M') \\ T \text{ fresh} \end{array}$$

and we want to prove that $(\mathcal{D}(f,t) \setminus \mathit{dom}(M')) \supsetneq (\mathcal{D}(f,t) \setminus (\mathit{dom}(M') \cup (X,t''))$ and the containment must be strict to ensure that the measure decreases. First of all notice that $(X,t'') \notin \mathit{dom}(M')$, since it is a side condition for the application of the rule. So in order to prove that containment is strict it suffices to prove that $(X,t'') \in \mathcal{D}(t,f)$. But this is a consequence of Lemma 19 which ensures that $t'' \in \mathit{Support}(f,t)$, whence the result.

[FIL-*] With all other rules the first component of *Wgt* remains constant, and the second component strictly decreases.

### A.1 Improvements

Although the analysis performed by our algorithm is already fine grained, it can be further refined in two simple ways. As explained in Section 4, the algorithm checks that after *one step* of reduction the capture variables occurring in recursive calls are bound to subtrees of the initial input type. So a possible improvement consists to try to check this property for a higher number of steps, too. For instance consider the filter:

$$\begin{array}{l} \boldsymbol{\mu} X.(\text{`nil},x) \Rightarrow X(\{\ell{:}x\}) \\ \quad \mid (y,\_) \Rightarrow X((\text{`nil},(y,y))) \\ \quad \mid \_ \Rightarrow \text{`nil} \end{array}$$

This filter does not pass our analysis since if $\iota_y$ is the identifier bound to the capture variable $y$, then when unfolding the recursive call in the second branch the $x$ in the first recursive call will be bound to $(\iota_y,\iota_y)$. But if we had pursued our abstract execution one step further we would have seen that $(\iota_y,\iota_y)$ is not used in a recursive call and, thus, that type inference terminates. Therefore, a first improvements is to modify $\mathit{Check}()$ so that it does not stop

```
type M = (K,V) | (V,V,K)
type V = { var : string } | { lambda2 : (string, string, M) }
type K = { var : string } | { lambda1 : (string, M) }

let filter Eval =
  | ( { lambda2 : (x, k, m) }, v , h ) -> m ; Subst[x,v] ; Subst[k,h]; Eval
  | ( { lambda1 : (x, m) }, v ) -> m ; Subst[x,v] ; Eval
  | x -> x

let filter Subst[c,F] =
  | ( Subst[c,F] ,  Subst[c,F] ,  Subst[c,F] )
  | ( Subst[c,F] ,  Subst[c,F] )
  | { var : c }  ->  F
  | { lambda2 : (x&¬c, k&¬c, m) }  ->  { lambda2 : (x, k, m;Subst[c,F]) }
  | { lambda1 : (x&¬c, m) }  ->  { lambda1 : (x, m;Subst[c,F]) }
  | x  ->  x
```

**Figure 5.** Filter encoding of $\lambda_{\text{cps}}$

just after one step of reduction but tries to go down $n$ steps, with $n$ determined by some heuristics based on sizes of the filter and of the input type.

A second and, in our opinion, more promising improvement is to enhance the precision of the test $\rceil A \lceil \& \lfloor f \rfloor = \varnothing$ in the definition of *Check*, that verifies whether the filter $f$ fails on the given symbolic argument $A$. In the current definition the only information we collect about the type of symbolic arguments is their structure. But further type information, currently unexploited, is provided by patterns. For instance, in the following (admittedly stupid) filter

$$\begin{array}{l}\boldsymbol{\mu} X.(\texttt{int},x) \Rightarrow x \\ \quad \mid y\&\texttt{int} \Rightarrow X((y,y)) \\ \quad \mid z \Rightarrow z\end{array}$$

if in the first pass we associate $y$ with $\iota_y$, then we know that $(\iota_y,\iota_y)$ has type $(\texttt{int},\texttt{int})$. If we record this information, then we know that in the second pass $(\iota_y,\iota_y)$ will always match the first pattern, and so it will never be the argument of a further recursive call. In other words, there is at most one nested recursive call. The solution is conceptually simple (but yields a cumbersome formalization, which is why we chose the current formulation) and amounts to modify *Trees*$(,)$ so that when it introduces fresh variables it records their type information with them. It then just suffices to modify the definition of $\rceil A \lceil$ so as it is obtained from $A$ by replacing every occurrence of a variable identifier by its type information (rather than by `any`) and the current definition of *Check*$()$ will do the rest.

## B. Proof of Turing completeness (Theorem 7)

In order to prove Turing completeness we show how to define by filters an evaluator for untyped (call-by-value) $\lambda$-calculus. If we allowed recursive calls to occur on the left-hand side of composition, then the encoding would be straightforward: just implement $\beta$ and context reductions. The goal however is to show that the restriction on compositions does not affect expressiveness. To that end we have to avoid context reductions, since they require recursive calls before composition. To do so, we first translate $\lambda$-terms via Plotkin's call-by-value CPS translation and apply Steele and Rabbit's administrative reductions to them obtaining terms in $\lambda_{\text{cps}}$. The latter is isomorphic to cbv $\lambda$-calculus (see for instance [31]) and defined as follows.

$$\begin{array}{rcl} M & ::= & KV \mid VVK \\ V & ::= & x \mid \lambda x.\lambda k.M \\ K & ::= & k \mid \lambda x.M \end{array}$$

with the following reduction rules (performed at top-level, without any reduction under context).

$$\begin{array}{rcl} (\lambda x.\lambda k.M)VK & \longrightarrow & M[x := V][k := K] \\ (\lambda x.M)V & \longrightarrow & M[x := V] \end{array}$$

In order to prove Turing completeness it suffices to show how to encode $\lambda_{\text{cps}}$ terms and reductions in our calculus. For the sake of readability, we use mutually recursive types (rather than their encoding in $\mu$-types), we use records (though pairs would have sufficed), we write just the recursion variable $X$ for the filter `x ->` $X$`x`, and use $\neg t$ to denote the type `any`$\backslash t$. Term productions are encoded by the recursive types given at the beginning of Figure 5.

Next we define the evaluation filter `Eval`. In its body it calls the filter `Subst[`$c$`,F]` which implements the capture free substitution and, when $c$ denotes a constant, is defined as right below.

It is straightforward to see that the definitions in Figure 5 implement the reduction semantics of CPS terms. Of course the definition above would not pass our termination condition. Indeed while `Subst` would be accepted by our algorithm, `Eval` would fail since it is not possible to ensure that the recursive calls of `Eval` will receive from `Subst` subtrees of the original input type. This is expected: while substitution always terminate they may return trees that are not subtrees of the original term.

## C. Proof of subject reduction (Theorem 13)

We first give the proof of Lemma 9, which we restate:

Let $f$ be a filter and $v$ be a value such that $v \notin \lfloor f \rfloor$. Then for every $\gamma$, $\delta$, if $\delta;\gamma \vdash_{\mathit{eval}} f(v) \rightsquigarrow r$, $r \equiv \Omega$.

*Proof.* By induction on the derivation of $\delta;\gamma \vdash_{\mathit{eval}} f(v) \rightsquigarrow r$, and case analysis on the last rule of the derivation:

**(expr):** here, $\lfloor e \rfloor = \texttt{any}$ for any expression $e$, therefore this rule cannot be applied (since $\forall v, v \in \texttt{any}$).

**(prod):** assume $v \notin \lfloor (f_1,f_2) \rfloor \equiv (\lfloor f_1 \rfloor, \lfloor f_2 \rfloor)$. then either $v \notin (\texttt{any},\texttt{any})$, in which case only rule **(error)** applies and therefore $r \equiv \Omega$. Or $v \equiv (v_1, v_2)$ and $v_1 \notin \lfloor f_1 \rfloor$. Then by induction hypothesis on the first premise, $\delta;\gamma \vdash_{\mathit{eval}} f_1(v_1) \rightsquigarrow r_1$ and $r_1 \equiv \Omega$, which contradicts the side condition $r_1 \neq \Omega$ (and

We were a little bit sloppy in the notation and used a filter parameter as a pattern. Strictly speaking this is not allowed in filters and, for instance, the branch `{ var : c } -> F` in `Subst` should rather be written as `{ var : y } -> ( (y=c); (true -> F | false -> { var : y }) )`. Similarly, for the other two cases.

similarly for the second premise). Therefore this rule cannot be applied to evaluate $(f_1,f_2)$.

**(patt):** Similarly to the previous case. If $v \notin \lfloor p \Rightarrow f \rfloor = \lfloor p \rfloor \& \lfloor f \rfloor$ then either $v \notin \lfloor p \rfloor$ (which contradicts $v/p \neq \Omega$) or $v \notin \lfloor f \rfloor$ and by induction hypotheis, $\delta;\gamma, v/p \vdash_{eval} f(v) \rightsquigarrow \Omega$.

**(comp):** If $v \notin \lfloor f_1;f_2 \rfloor \equiv \lfloor f_1 \rfloor$, then by induction hypothesis on the first premise, $r_1 \equiv \Omega$, which contradicts the side condition $r_1 \neq \Omega$. Therefore only rule **(error)** can be applied here.

**(recd):** Similar to product type: either $v$ is not a record and therefore only the rule **(errro)** can be applied, or one of the $r_i = \Omega$ (by induction hypothesis) which contradicts the side condition.

**(union1) and (union2):** since $v \notin \lfloor f_1 | f_2 \rfloor$, this means that $v \notin \lfloor f_1 \rfloor$ *and* $v \notin \lfloor f_2 \rfloor$. Therefore if rule **(union1)** is chosen, by induction hypothesis, $r_1 \equiv \Omega$ which contradicts the side condition. If rule **(union2)** is chosen, then by induction hypothesis $r_2 \equiv \Omega$ which gives $r \equiv \Omega$.

**(rec-call)** trivially true, since it cannot be that $v \notin \texttt{any}$

**(rec)** we can apply straightfowardly the induction hypothesis on the premise and have that $r \equiv \Omega$.

**(groupby) and (orderby):** If $v \notin f$ then the only rule that applies is **(error)** which gives $r \equiv \Omega$.

□

We are now equipped to prove the subject reduction theorem which we restate in a more general manner:

For every $\Gamma, \Delta, M, \gamma, \delta$ such that $\forall x \in dom(\gamma), x \in dom(\Gamma) \wedge \gamma(x) : \Gamma(x)$, if $\Gamma \wp \Delta \wp M \vdash_{fil} f(t) : s$, then for all $v : t$, $\delta;\gamma \vdash_{eval} f(v) \rightsquigarrow r$ implies $r : s$.

The proof is by induction on the derivation of $\delta;\gamma \vdash_{eval} f(v) \rightsquigarrow r$, and by case analysis on the rule. Beside the basic case, the only rules which are not a straightforward application of induction are the rules **union1** and **union2** that must be proved simulatneously. Other cases being proved by a direct application of the induction hypothesis, we only detail the case of the product constructor.

**(expr):** we suppose that the host languages enjoys subject reduction, hence $e(v) = r : s$.

**(prod):** Here, we know that $t \equiv \bigvee_{i \leq \texttt{rank}(t)} (t_1^i, t_2^i)$. Since $v$ is a value, and $v : t$, we obtain that $v \equiv (v_1, v_2)$. Since $(v_1, v_2)$ is a value, $\exists i \leq \texttt{rank}(t)$ such that $(v_1, v_2) : (t_1^i, t_2^i)$. The observation that $v$ is a value is crucial here, since in general given a type $t' \leq t$ with $t' \equiv \bigvee_{j \leq \texttt{rank}(t')} ({t'_1}^j, {t'_2}^j)$ it is not true that $\exists i, j$ s.t. $({t'_1}^j, {t'_2}^j) \leq ({t_1}^i, {t_2}^i)$. However this property holds for singleton types and therefore for values. We have therefore that $v_1 : t_1^i$ and $v_2 : t_1^i$. Furthermore, since we suppose that a typing derivation exists and the typing rules are syntax directed, then $\Gamma \wp \Delta \wp M \vdash_{fil} f(t_1^i) : s_1^i$ must be a used to prove our typing judgement. We can apply the induction hypothesis and deduce that $\delta;\gamma \vdash_{eval} f(v_1) \rightsquigarrow r_1$ and similarly for $v_2$. We have therefore that the filter evaluates to $(r_1, r_2)$ which has type $(s_1^i, s_2^i) \leq \bigvee_{j \leq \texttt{rank}(s)} ({s_1}^i, {s_2}^i)$ which proves this case.

**(union1) and (union2):** both rules must be proved together. Indeed, given a filter $f_1|f_2$ for which we have a typing derivation for the judgement $\Gamma \wp \Delta \wp M \vdash_{fil} f_1|f_2(t) : s$, either $v : t\&\lfloor f_1 \rfloor$ and we can apply the induction hypothesis and therefore, $\Gamma \wp \Delta \wp M \vdash_{fil} f_1|(t\&\lfloor f_1 \rfloor) : s_1$ and if $\delta;\gamma \vdash_{eval} f(v_1) \rightsquigarrow r_1$ (case **union1**) $r_1 : s_1$. However it might be that $v \notin t\&\lfloor f_1 \rfloor$. Then by Lemma 9 we have that $\delta;\gamma \vdash_{eval} f(v_1) \rightsquigarrow \Omega$ (case **union2**) and we can apply the induction hypothesis on the second premise, which gives us $\delta;\gamma \vdash_{eval} f(v_1) \rightsquigarrow r_2$ which allows us to conclude.

**(error):** this rule can never occur. Indeed, if $v : t$ and $\Gamma \wp \Delta \wp M \vdash_{fil} f(t) : s$, that means in particular that $t \leq \lfloor f \rfloor$ and therefore that all of the side conditions for the rules **prod**, **pat** and **recd** hold.

## D. Complexity of the typing algorithm

*Proof.* For clarity we restrict ourselves to types without record constructors but the proof can straihgtforwardly be extended to them. First remark that our types are isomorphic to alternating tree automata (ATA) with intersection, union and complement (such as those defined in [11, 17]). From such an ATA $t$ it is possible to compute a non-deterministic tree automaton $t'$ of size $O(2^{|t|})$. When seeing $t'$ in our type formalism, it is a type generated by the following grammar:

$$\begin{array}{lll} \tau & ::= & \mu X.\tau \mid \tau|const \mid const \\ const & ::= & (\tau,\tau) \mid atom \\ atom & ::= & X \mid b \end{array}$$

where $b$ ranges over negation and intersections of basic types. Intuitively each recursion variable $X$ is a state in the NTA, a finite union of products is a set of non-deterministict transitions whose right-hand-side are each of the products and *atom* productions are leaf-states. Then it is clear than if $Check(f)$ holds, the algorithm considers at most $|f| \times |t'|$ distinct cases (thanks to the memoization set $M$). Furthermore for each rule, we may test a subtyping problem (*eg*, $t \leq \lfloor f \rfloor$), which itself is EXPTIME thus giving the bound □

## E. Precise typing of `groupby`

The process of inferring a precise type for `groupby` $f(t)$ is decomposed in several steps. First we have to compute the set $\mathcal{D}(f)$ of discriminant domains for the filter $f$. The idea is that these domains are the types on which we know that $f$ may give results of different types. Typically this corresponds to all possible branchings that the filter $f$ can do. For instance, if $\{t_1, ..., t_n\}$ are pairwise disjoint types and $f$ is the filter $t_1 \Rightarrow e_1 | \cdots | t_n \Rightarrow e_n$, then the set of discriminant domains for the filter $f$ is $\{t_1, ..., t_n\}$, since the various $s_i$ obtained from $\varnothing \wp \varnothing \wp \varnothing \vdash_{fil} f(t_i) : s_i$ may all be different, and we want to keep track of the relation between a result type $s_i$ and the input type $t_i$ that produced it. Formally $\mathcal{D}(f)$ is defined as follows:

$$\begin{array}{lll} \mathcal{D}(e) & = & \{\texttt{any}\} \\ \mathcal{D}((f_1,f_2)) & = & \mathcal{D}(f_1) \times \mathcal{D}(f_2) \\ \mathcal{D}(p \Rightarrow f) & = & \{\lfloor p \rfloor \,\&\, t \mid t \in \mathcal{D}(f)\} \\ \mathcal{D}(f_1|f_2) & = & \mathcal{D}(f_1) \cup \{t \backslash \lfloor f_1 \rfloor \mid t \in \mathcal{D}(f_2)\} \\ \mathcal{D}(\mu X.f) & = & \mathcal{D}(f) \\ \mathcal{D}(Xa) & = & \{\texttt{any}\} \\ \mathcal{D}(f_1;f_2) & = & \mathcal{D}(f_1) \\ \mathcal{D}(of) & = & \{\texttt{[any*]}\} \qquad (o = \texttt{groupby}\,,\texttt{orderby}\,) \\ \mathcal{D}(\{\ell_i{:}f_i\,,\,..\}_{i\in I}) & = & \bigcup_{t_i \in \mathcal{D}(f_i)} \{\ell_i{:}t_i\,,\,..\}_{i\in I} \end{array}$$

Now in order to have a high precision in typing we want to compute the type of the intermediate list of the groupby on a set of disjoint types. For that we define a normal form of a set of types that given a set of types computes a new set formed of pairwise disjoint types whose union covers the union of all the original types.

$$\mathcal{N}(\{t_i \mid i \in S\}) = \bigcup_{\varnothing \subset I \subseteq S} (\bigcap_{i \in I} t_i \setminus \bigcup_{j \in S \setminus I} t_j)$$

Recall that we are trying to deduce a type for `groupby` $f(t)$, that is for the expression `groupby` $f$ when applied to an argument of type $t$. Which types shall we use as input for our filter $f$ to compute the type of the intermediate result? Surely we want to have all the

types that are in the $\texttt{item}(t)$. This however is not enough since we would not use the information of the descriminant domains for the filter. For instance if the filter gives two different result types for positive and negative numbers and the input is a list of integers, we want to compute two different result types for positive and negatives and not just to compute the result of the filter application on generic integers (indeed $\texttt{item}(\texttt{[int*]}) = \{\texttt{int}\}$). So the idea is to add to the set that must be normalized all the types of the set of discriminant types. This however is not precise enough, since these domains may be much larger than the item types of the input list. For this reason we just take the part of the domain types that intersects at least some possible value of the list. In other terms we consider the normal form of the following set

$$\texttt{item}(t) \cup \{\sigma_i \wedge \bigvee_{t_i \in \mathcal{D}(f)} t_i \mid \sigma_i \in \texttt{item}(t)\}$$

The idea is then that if $\{t_i\}_{i\in I}$ is the normalized set of the set above, then the type of grouby is the type $\texttt{[}\bigvee_{i\in I}\texttt{(}f(t_i)\texttt{,[}t_i\texttt{+])*]}$ with the optimization that we can replace the * by a + if we know that the input list is not empty.

## F. Comparison with top-down tree transducers

We first show that every Top-down tree transducers with regular look-ahead can be encoded by a filter. We use the definition of [14] for top-down tree transducers:

**Definition 20 (Top-down Tree transducer with regular look-ahead).** A *top-down tree transducer with regular look-ahead* (TDTTR) is a 5-tuple $< \Sigma, \Delta, Q, Q_d, R >$ where $\Sigma$ is the input alphabet, $\Delta$ is the output alphabet, $Q$ is a set of states, $Q_d$ the set of initial states and $R$ a set of rules of the form:

$$q(a(x_1, \ldots, x_n)), D \to b(q_1(x_{i_1}), \ldots, q_m(x_{i_m}))$$

where $a \in \Sigma_n$, $b \in \Delta_m$, $q, q_1, \ldots, q_m \in Q$, $\forall j \in 1..m, i_j \in 1..n$ and $D$ is a mapping from $\{x_1, \ldots, x_n\}$ to $2^{T_\Sigma}$ ($T_\Sigma$ being the set of regular tree languages over alphabet $\Sigma$). A TDTTR is *deterministic* if $Q_d$ is a singleton and the left-hand sides of the rules in $R$ are pairwise disjoint.

Since our filters encode programs, it only make sense to compare filters and *deterministic* TDTTR. We first show that given such a *deterministic* TDTTR we can write an equivalent filter $f$ and, furthermore, that $\mathit{Check}(f)$ does not fail. First we recall the encoding of ranked labeled trees into the filter data-model:

**Definition 21 (Tree encoding).** Let $t \in T_\Sigma$. We define the tree-encoding of $t$ and we write $[\![t]\!]$ the value defined inductively by:

$$\begin{array}{lcll} [\![a]\!] & = & \texttt{(`a,[])} & \forall a \in \Sigma_0 \\ [\![a(t_1, \ldots, t_n)]\!] & = & \texttt{(`a,[ } [\![t_1]\!] \texttt{ ... } [\![t_n]\!] \texttt{ ])} & \forall a \in \Sigma_n \end{array}$$

where the list notation $\texttt{[ } v_1 \texttt{ ... } v_n \texttt{ ]}$ is a short-hand for $\texttt{(}v_1\texttt{, ..., (}v_n\texttt{, `nil))}$. We generalize this encoding to tree languages and types. Let $S \subseteq T_\sigma$, we write $[\![S]\!]$ the set of values such that $\forall t \in S, [\![t]\!] \in [\![S]\!]$.

In particular it is clear than when $S$ is regular, $[\![S]\!]$ is a type.

**Lemma 22 (TDTTR $\to$ filters).** *Let $T =< \Sigma, \Delta, Q, \{q_0\}, R >$ be a deterministic TDTTR. There exists a filter $f_T$ such that:*

$$\forall t \in \mathit{dom}(T), [\![T(t)]\!] \equiv f_T[\![t]\!]$$

*Proof.* The encoding is as follows. For every state $q_i \in Q$, we will introduce a recursion variable $X_i$. Formally, the translation is performed by the function $\mathit{TR} : 2^Q \times Q \to$ **Filters** defined as:

$$\begin{array}{lcll} \mathit{TR}(S, q_i) & = & x \Rightarrow X_i\ x & \text{if } q_i \in S \\ \mathit{TR}(S, q_i) & = & \mu X_i.(f_1 | \ldots | f_n) & \text{if } q_i \notin S \end{array}$$

where every rule $r_j \in R$

$$r_j \equiv q_i(a_j(x_1, \ldots, x_n)), D_j \to b_j(q_{j_1}(x_{j_{k_1}}), \ldots, q_{j_m}(x_{j_{k_m}}))$$

is translated into:

$$\texttt{(`}a_j\texttt{, [ } x_1\texttt{\&}[\![D_j(x_1)]\!] \texttt{ ... } x_n\texttt{\&}[\![D_j(x_n)]\!] \texttt{ ])} \Rightarrow \texttt{`}b_j\texttt{,[]} \quad \text{if } b_j \in \Delta_0$$

$$\texttt{(`}a_j\texttt{, [ } x_1\texttt{\&}[\![D_j(x_1)]\!] \texttt{ ... } x_n\texttt{\&}[\![D_j(x_n)]\!] \texttt{ ])} \Rightarrow \texttt{(`}b_j\texttt{,(}x_{j_{k_1}}\texttt{;}\mathit{TR}(S', q_{j_1})\texttt{,(} \ldots \texttt{,(}x_{j_{k_m}}\texttt{;}\mathit{TR}(S', q_{j_m})\texttt{,`nil))))} \quad \text{where } S' = S \cup \{q_i\} \text{ otherwise}$$

The fact that $\forall t \in \mathit{dom}(T), [\![T(t)]\!] \equiv \mathit{TR}(\varnothing, q_0)[\![t]\!]$ is proved by a straightforward induction on $t$. The only important point to remark is that since $T$ is deterministic, there is exactly one branch of the alternate filter $f_1 | \ldots | f_n$ that can be selected by pattern matching for a given input $v$. For as to why $\mathit{Check}(f_T)$ holds, it is sufficient to remark that each recursive call is made on a strict subtree of the input which guarantees that $\mathit{Check}(f_T)$ returns true. □

**Lemma 23 (TDTTR $\not=$ filters).** *Filters are strictly more expressive than TDTTRs.*

*Proof.* Even if we restrict filters to have the same domain as TDTTRs (meaning, we used a fixed input alphabet $\Sigma$ for atomic types) we can define a filter that cannot be expressed by a TDTTR. For instance consider the filter:

$$\begin{array}{rl} y \Rightarrow (\mu X.\texttt{(`a,`nil)} & \Rightarrow y \\ \texttt{|(`b,`nil)} & \Rightarrow y \\ \texttt{|(`a,[}x\texttt{])} & \Rightarrow (\texttt{`a, [ } X\ x \texttt{ ])} \\ \texttt{|(`b,[}x\texttt{])} & \Rightarrow (\texttt{`b, [ } X\ x \texttt{ ])} \end{array}$$

This filter works on monadic trees of the form $u_1(\ldots u_{n-1}(u_n)\ldots)$ where $u_i \in \{a, b\}$, and essentially replaces the leaf $u_n$ of a tree $t$ by a copy of $t$ itself. This cannot be done by a TDTTR. Indeed, TDTTR have only two ways to "remember" a subtree and copy it. One is of course by using variables; but their scope is restricted to a rule and therefore an arbitrary subtree can only be copied at a *fixed distance* of its original position. For instance in a rule of the form $q(a(x)), D \to a(q_1(x), b(b(q_1(x))))$, assuming than $q_1$ copies its input, the copy of the original subtree is two-levels down from its next sibling but it cannot be arbitrary far. A second way to copy a subtree is to remember it using the states. Indeed, states can encode the knowledge that the TDTTR has accumulated along a path. However, since the number of states is finite, the only thing that a TDTTR can do is copy a fixed path. For instance for any given $n$, there exists a TDTTR that performs the transformation above, for trees of height $n$ (it has essentially $2^n - 1$ states which remember every possible path taken). For instance for $n = 2$, the TDTTR is:

$$\begin{array}{lcl} q_0(a(x)), D & \to & a(q_1(x)) \\ q_1(a()), D & \to & a(a) \\ q_1(b()), D & \to & a(b) \\ q_0(b(x)), D & \to & b(q_2(x)) \\ q_2(a()), D & \to & b(a) \\ q_2(b()), D & \to & b(b) \end{array}$$

where $\Sigma = \Delta = \{a, b\}$, $Q = \{q0, q1, q2\}$, $Q_d = \{q_0\}$ and $D = \{x \mapsto T_\Sigma\}$. It is however impossible to write a TDTTR that replaces a leaf with a copy of the whole input, for inputs of arbitrary size. A similar example is used in [13] to show that TDTTR and bottom-up tree transducers are not comparable. □

## G. Operators on record types.

We use the theory of records defined for ℂDuce. We summarize here the main definitions. These are adapted from those given in Chapter 9 of Alain Frisch's PhD thesis [15] where the interested

reader can also find detailed definitions of the semantic interpretation of record types and of the subtyping relation it induces, the modifications that must be done to the algorithm to decide it, finer details on pattern matching definition and compilation techniques for record types and expressions.

Let $Z$ denote some set, a function $r : \mathcal{L} \to Z$ is *quasi-constant* if there exists $z \in Z$ such that the set $\{\ell{\in}\mathcal{L} \mid r(\ell) \neq z\}$ is finite; in this case we denote this set by $\mathtt{dom}(r)$ and the element $z$ by $\mathtt{def}(r)$. We use $\mathcal{L} \twoheadrightarrow Z$ to denote the set of quasi-constant functions from $\mathcal{L}$ to $Z$ and the notation $\{\ell_1 = z_1, \ldots, \ell_n = z_n, \_ = z\}$ to denote the quasi-constant function $r : \mathcal{L} \twoheadrightarrow Z$ defined by $r(\ell_i) = z_i$ for $i = 1..n$ and $r(\ell) = z$ for $\ell \in \mathcal{L} \setminus \{\ell_1, \ldots, \ell_n\}$. Although this notation is not univocal (unless we require $z_i \neq z$), this is largely sufficient for the purposes of this section.

Let $\perp$ be a distinguished constant, then the sets $\mathtt{string} \twoheadrightarrow \mathbf{Types} \cup \{\perp\}$ and $\mathtt{string} \twoheadrightarrow \mathbf{Values} \cup \{\perp\}$ denote the set of all record types expressions and of all record values, respectively. The constant $\perp$ represents the value of the fields of a record that are "undefined". To ease the presentation we use the same notation both for a constant and the singleton type that contains it: so when $\perp$ occurs in $\mathcal{L} \twoheadrightarrow \mathbf{Values} \cup \{\perp\}$ it denotes a value, while in $\mathtt{string} \twoheadrightarrow \mathbf{Types} \cup \{\perp\}$ it denotes the singleton type that contains only the value $\perp$.

Given the definitions above, it is clear that the record types in Definition 2 are nothing but specific notations for some quasi-constant functions in $\mathtt{string} \twoheadrightarrow \mathbf{Types} \cup \{\perp\}$. More precisely, the open record type expression $\mathbf{\{}\ell_1\mathbf{:}t_1, \ldots, \ell_n\mathbf{:}t_n\mathbf{,..\}}$ denotes the quasi-constant function $\{\ell_1 = t_1, \ldots, \ell_n = t_n, \_ = \mathtt{any}\}$ while the closed record type expression $\mathbf{\{}\ell_1\mathbf{:}t_1, \ldots, \ell_n\mathbf{:}t_n\mathbf{\}}$ denotes the quasi-constant function $\{\ell_1 = t_1, \ldots, \ell_n = t_n, \_ = \perp\}$. Similarly, the optional field notation $\mathbf{\{}..., \ell\mathtt{?}\mathbf{:}t, ...\mathbf{\}}$ denotes the record type expressions in which $\ell$ is mapped either to $\perp$ or to the type $t$, that is, $\{..., \ell = t \mid \perp, ...\}$.

Let $t$ be a type and $r_1, r_2$ two record type expressions, that is $r_1, r_2 : \mathtt{string} \twoheadrightarrow \mathbf{Types} \cup \{\perp\}$. The *merge* of $r_1$, and $r_2$ with respect to $t$, noted $\oplus_t$ and used infix, is the record type expression defined as follows:

$$(r_1 \oplus_t r_2)(\ell) \stackrel{\text{def}}{=} \begin{cases} r_1(\ell) & \text{if } r_1(\ell)\mathbf{\&}t \leq \mathtt{empty} \\ (r_1(\ell) \setminus t)\mathbf{|}r_2(\ell) & \text{otherwise} \end{cases}$$

Recall that by Lemma 11 a *record type* (*ie*, a subtype of $\{..\}$) is equivalent to a finite union of *record type expressions* (*ie*, quasi-constant functions in $\mathtt{string} \twoheadrightarrow \mathbf{Types} \cup \{\perp\}$). So the definition of *merge* can be easily extended to all record types as follows

$$(\bigvee_{i\in I} r_i) \oplus_t (\bigvee_{j\in J} r'_j) \stackrel{\text{def}}{=} \bigvee_{i\in I, j\in J} (r_i \oplus_t r'_j)$$

Finally, all the operators we used for the typing of records in the rules of Section 4.2 are defined in terms of the merge operator:

$$t_1 + t_2 \stackrel{\text{def}}{=} t_2 \oplus_\perp t_1 \quad (1)$$

$$t \setminus \ell \stackrel{\text{def}}{=} \{\ell = \perp, \_ = c_0\} \oplus_{c_0} t \quad (2)$$

where $c_0$ is any constant different from $\perp$ (the semantics of the operator does not depend on the choice of $c_0$ as long as it is different from $\perp$).

Notice in particular that the result of the concatenation of two record type expressions $r_1 + r_2$ may result for each field $\ell$ in three different outcomes:

1. if $r_2(\ell)$ does not contain $\perp$ (*ie*, the field $\ell$ is surely defined), then we take the corresponding field of $r_2$: $(r_1 + r_2)(\ell) = r_2(\ell)$
2. if $r_2(\ell)$ is undefined (*ie*, $r_2(\ell) = \perp$), then we take the corresponding field of $r_1$: $(r_1 + r_2)(\ell) = r_1(\ell)$
3. if $r_2(\ell)$ *may* be undefined (*ie*, $r_2(\ell) = t\mathbf{|}\perp$ for some type $t$), then we take the union of the two corresponding fields since it can results either in $r_1(\ell)$ or $r_2(\ell)$ according to whether the record typed by $r_2$ is undefined in $\ell$ or not: $(r_1 + r_2)(\ell) = r_1(\ell)\mathbf{|}(r_2(\ell) \setminus \perp)$.

This explains all the examples we gave in the main text. In particular, $\mathbf{\{}a\mathbf{:}\mathtt{int}, b\mathbf{:}\mathtt{int}\mathbf{\}} + \mathbf{\{}a\mathtt{?}\mathbf{:}\mathtt{bool}\mathbf{\}} = \mathbf{\{}a\mathbf{:}\mathtt{int}\mathbf{|}\mathtt{bool}, b\mathbf{:}\mathtt{int}\mathbf{\}}$ since "$a$" *may* be undefined in the right hand-side record while "$b$" *is* undefined in it, and $\mathbf{\{}a\mathbf{:}\mathtt{int}\mathbf{\}} + \mathbf{\{..\}} = \mathbf{\{..\}}$, since "$a$" in the right hand-side record is defined (with $a \mapsto \mathtt{any}$) and therefore has priority over the corresponding definition in the left hand-side record.

## H. Encoding of Co-grouping

As shown in Section 5.2, our `groupby` operator can encode JaQL's `group each` $x = e$ `as` $y$ `by` $e'$, where $e$ computes the grouping key, and for each distinct key, $e'$ is evaluated in the environment where $x$ is bound to the key and $y$ is bound to the sequence of elements in the input having that key. Co-grouping is expressed by:

```
group
   l_1 by x = e_1 as y_1
          ⋮
   l_n by x = e_n as y_n
into e
```

Co-grouping is encoded by the following composition of filters:

```
[ [[l_1]] ... [[l_n]] ];
[ Transform[x ⇒ (1,x)] ... Transform[x ⇒ (n,x)] ];
Expand;
groupby ((1,$) ⇒ [[e_1]] | ... |(n,$) ⇒ [[e_n]]);
Transform[ (x,l) ⇒ ([(l;Rgrp1)...(l; Rgrpn)];
                    [y_1, ..., y_n] ⇒ [[e]] )]
```

where

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

Essentially, the co-grouping encoding takes as argument a sequence of sequences of values (the $n$ sequences to co-group). Each of these $n$ sequence, is tagged with an integer $i$. Then we flatten this sequences of tagged values. We can on this single sequence apply our `groupby` operator and modify each key selector $e_i$ so that it is only applied to a value tagged with integer $i$. Once this is done, we obtain a sequence of pairs $(k, l)$ where $k$ is the commen grouping key and $l$ a sequence of tagged values. We only have to apply the auxiliary `Regrp` filter which extracts the $n$ subsequences from $l$ (tagged with $1..n$) and removes the integer tag for each value. Lastly we can call the recombining expression $e$ which has in scope $x$ bound to the current grouping key and $y_1, \ldots, y_n$ bound to each of the sequences of values from input $l_1, \ldots, l_n$ whose grouping key is $x$.