Deciding Well-Definedness of First-Order, Object-Creating Operations over Tree-Structured

Deciding Well-Definedness of First-Order, Object-Creating Operations over Tree-Structured Data

Stijn Vansummeren
Limburgs Universitair Centrum
Universitaire Campus
B-3590 Diepenbeek.

April 28, 2005

Abstract

The well-definedness problem for a database query language consists of checking, given an expression and an input type, whether the semantics of the expression is defined for all inputs adhering to the input type. In this paper we study the well-definedness problem for a family of first-order, object-creating query languages which are evaluated in a tree-structured, list-based data model. We identify properties of base operations which can make the problem undecidable and give restrictions which are sufficient to ensure decidability. As a direct result, we obtain a large fragment of XQuery for which well-definedness is decidable.

1 Introduction

The operations of a general-purpose programming language such as C or Java are only defined on certain kinds of inputs. For example, if a is an array, then the array indexation $a[i]$ is only defined if i lies within the boundaries of the array. If, during the execution of a program, an operation is supplied with the wrong kind of input, then the output of the program is undefined. Indeed, the program may exit with a runtime error, or worse yet, it may compute the wrong output.

To detect such programming errors as early as possible, it is hence natural to ask whether we can solve the *well-definedness problem*: given an expression and an input type, decide whether the semantics of the expression is defined for all inputs adhering to the input type. Unfortunately, this problem is undecidable for any computationally complete programming language, by Rice's Theorem. Most programming languages therefore provide a *static type system* to detect programming errors [30, 34]. These systems

ensure "type safety" in the sense that every expression which passes the type system's tests is guaranteed to be well-defined. Due to the undecidability of the well-definedness problem, these systems are necessarily incomplete, i.e., there are expressions which are well-defined, but do not type-check. Such expressions are problematic from a programmer's point of view, as he must rewrite his code in order to get it to type-check. As such, a major quest in the theory of programming languages consists of finding static type systems for which the set of well-defined but ill-typed expressions is as small as possible.

Although the Holy Grail in this quest (i.e., a static type system which is both sound and complete) can never be found for general-purpose programming languages, this does not mean it cannot be found for smaller, specific-purpose programming languages. In this paper we therefore study the well-definedness problem for a family of *query languages* QL(B) which are evaluated in a tree-structured, list-based data model. Here, B is a set of *base operations* (such as an equality test, taking the children of a certain node in a tree, creating a new node, ...) and QL(B) is the query language obtained from B by adding variables, constants, conditional tests, let-bindings, and for-loops. Since base operations are free to create new nodes, every QL(B) is hence a first-order, object-creating query language.

Concretely, we study the well-definedness problem for such QL(B) in the presence of bounded-depth regular expression types. Regular expression types are based on regular tree languages [6, 11, 31, 32] and are widely used in general-purpose programming languages manipulating tree-structured data, such as XDuce [18, 19, 20], ℂDuce [15], and XQuery [5, 13]. The bounded-depth restriction is motivated by the fact that most tree-structured data (such as for example found in XML documents [39]) in practice has nesting depth at most five or six, and that unbounded-depth nesting is hence often not needed.

Specifically, we identify properties of base operations which can make the well-definedness problem undecidable and give corresponding restrictions which are sufficient to ensure decidability. In essence, we hence identify a broad class of QL(B) for which a sound and complete static type system does exist.

Our study is motivated by XQuery, the XML query language currently under development by the World Wide Web Consortium [5, 13]. Expressions in XQuery can be undefined. As an example, consider the following variation on one of the XQuery use cases [9]:

```
<bib> {
  for $b in $bib/book
  where $b/publisher = 'ACM'
  return element{$b/author}{$b/title}
} </bib>
```

This expression should create, for each book published by ACM, a node whose name equals the author of the book and whose child is the title of the book. If there is a book with more than one `author` node however, then the result of this expression is undefined because the XQuery specification requires that the first argument to the element constructor is a singleton list.

Since XQuery is in fact a full-fledged programming language, its well-definedness problem is of course undecidable. The typical XQuery expression does not use the whole of XQuery's computational power however. For example, sixty-two out of the seventy-seven XQuery uses cases [9] can be written as a "for-let-where-return" expression without recursive function definitions. It is hence natural to ask whether we can solve the well-definedness problem for such expressions.

We will show that XQuery's basic functions and operators [24] are in fact base operations. As such, "for-let-where-return" XQuery fits nicely into our family of studied query languages. The decidability of well-definedness for a large fragment of "for-let-where-return" XQuery immediately follows as we show that, in the absence of automatic coercions, the various axis movements, node constructors, value and node comparisons, and node-name and text-content inspections satisfy our restrictions. In contrast, well-definedness for this fragment with automatic coercions is undecidable.

We note that the above decidability result cannot be obtained simply by using the existing XQuery static type system [13] on this restricted fragment. Indeed, consider the following expression which is trivially well-defined since the then-branch of the if-test will never be executed.

```
if false then element{()}{()} else ()
```

In the general setting for which the XQuery type system is designed, it is undecidable to check that an expression always evaluates to true. The XQuery type system is therefore "conservative" in the sense that it requires both branches of an if-test to type-check. Since the then-branch in our example is always undefined, it cannot type-check, and hence the whole expression is ill-typed. This example clearly illustrates that in order to solve well-definedness one also has to solve "satisfiability". We will see, however, that this alone is not sufficient to solve well-definedness.

In a companion paper [37] we study the well-definedness problem for the Nested Relational Calculus (NRC), a well-known query language for the complex object data model [1, 8, 38]. Although both the NRC and QL(B) are first-order languages, this does not mean that their well-definedness problems are the same. Indeed, QL(B) operates on a tree-structured, list-based data model, has object identity, and can create new objects, whereas the NRC operates on a set-based data model without object identity. Moreover, regular expression types are capable of specifying both lower-bound

and upper-bound constraints on the input, while the complex object types for which we study well-definedness in the NRC can only specify upper-bound constraints. For example, it is possible to give a regular expression type which only recognizes those inputs which contain at least three and at most five atomic data values. Such a complex object type does not exist, however.

As a result of these differences we will show that the presence of base operations which are undefined on non-singleton inputs has no impact on the decidability of the well-definedness problem for QL(B), whereas such base operations already cause the well-definedness problem for the positive-existential fragment of the NRC to become undecidable [37]. That such operations are not problematic with regard to well-definedness for QL(B) is entirely due to its list-based data model. Indeed, we will show that well-definedness for the positive-existential NRC equipped with such a base operation, interpreted in a list-based data model, is decidable.

Related work. A problem reminiscent to the well-definedness problem is *semantic type-checking*: check that the output of a given expression is always in a given output type for every input adhering to a given input type. This problem has already been studied extensively for XML-related query languages [2, 3, 25, 26, 29, 35]. Most of these approaches restrict themselves to the setting where the set of possible tree labels is finite. Semantic type-checking can then be realized by a reduction to the satisfiability problem of monadic second-order logic over trees [36], which is known to be decidable. In contrast, Alon et al. [2, 3] study the problem in the presence of an infinite set of *data values*, which often causes the problem to become undecidable. This is the setting which is closest to ours, as we study the well-definedness problem in the presence of such data values. We will show however, that there are QL(B) for which well-definedness is decidable, but semantic type-checking is not.

We will also see that in order to solve well-definedness it is necessary (but not sufficient) to solve the satisfiability problem (i.e., non-empty output of a well-defined expression on at least one input). This problem has also been studied extensively for XML-related query languages both in the setting of a finite set of labels [10, 16, 28, 33] and an infinite set of data values [23].

Organization. This paper is further organized as follows. We introduce our data model in Section 2. In Section 3 we introduce the notion of a base operation and show how to extend these to a query language QL(B). In Section 4 we state the well-definedness problem and introduce bounded-depth regular expression types. In Sections 5, 6, and 7 we identify several properties of base operations which may render the well-definedness problem undecidable and propose corresponding restrictions on base operations.

We show that these restrictions are sufficient to ensure decidability in Section 8. In that section we also show that the well-definedness problem for the positive-existential NRC equipped with a base operation which is undefined on non-singleton inputs is decidable if we interpret this language in a list-based data model. We conclude in Section 9.

2 Data model

Intuitively, every value in our data model is a finite list of atomic data values and nodes. Nodes are grouped in "stores" (lists of trees). We distinguish between nodes that define the structure of a tree (called *element nodes*) and nodes that hold actual data information (called *text nodes*). This tree-structured, list-based data model can be used to encode the traditional relational data model (as we show in Section 5), a list-based version of the complex object data model (as we show in Section 8.2), and the tree-structured data found in XML documents. For example, Figure 1(a) depicts a store where the first tree represents the XML fragment in Figure 1(b) and the second tree represents the XML fragment in Figure 1(c). Here we use circles to depict element nodes and boxes to depict text nodes. In fact, our data model is quite close to the one employed by XQuery. Indeed, XQuery expressions do not operate directly on XML text, but on instances of the XQuery data model [14]. Every value in this data model is a list of items, where every item is an atomic value or a node. There are seven node kinds, the most prominent being the element, attribute, and text nodes. Nodes are grouped in lists of trees. Granted, we distinguish fewer node kinds than XQuery, but this is done solely for simplicity. If desired, we could add additional node types without sacrificing any of our results.

We note that every item in the XQuery data model also carries a *type annotation*. Examples of such annotations are `integer` (for atoms) and `element of type Bibliography` (for element nodes). Potentially, these type annotations can also be `untypedAtomic` (for atoms) or `untyped` (for nodes) indicating that the item was not validated against a schema. XQuery uses the type annotations of validated inputs during (1) static and dynamic type-checking[1] and (2) the evaluation of type-tests (such as *instance-of* and *typeswitch*). In our context, such type annotations are irrelevant however. Indeed, the aim of this paper is to study well-definedness, which is more fundamental than static or dynamic type-checking. Furthermore, we do not consider type-tests, as these are already known to quickly turn the well-definedness problem undecidable [37]. Values in our data model therefore correspond to unvalidated values in the XQuery data model. All references to the semantics of XQuery should hence also be understood to mean "the

[1]Static type-checking is an optional feature in XQuery. All XQuery processors have to perform dynamic type-checking however.

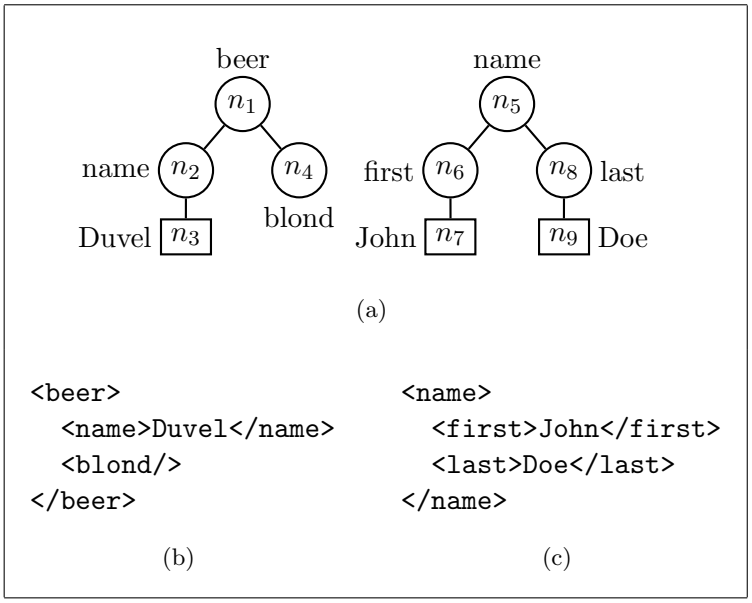


Figure 1: A store and the XML fragments it represents.

semantics of XQuery when the input is unvalidated".

2.1 Atoms and nodes

Formally, we assume to be given a recursively enumerable set $\mathcal{A} = \{a, b, \dots\}$ of *atoms*, which contains the booleans `true` and `false`. In practice $\mathcal{A}$ will also contain the other usual data values such as integers, strings, and so on. We further assume to be given an infinite set $\mathcal{N} = \{n, m, \dots\}$ of *nodes*, disjoint with $\mathcal{A}$, which is partitioned into a recursively enumerable infinite set $\mathcal{N}^e$ of *element nodes* and a recursively enumerable infinite set $\mathcal{N}^t$ of *text nodes*. Elements of $\mathcal{A} \cup \mathcal{N}$ are called *items*.

2.2 Stores

Nodes are given an interpretation inside a *store*, which is essentially a list of ordered node-labeled trees.[2] Formally, a store Σ is a tuple $(V, E, \lambda, <, \prec)$ where

- V is a finite set of nodes;
- E is the *edge relation*: a binary relation on V such that (V, E) is an acyclic directed graph where every node has in-degree at most one and text nodes have out-degree zero (hence (V, E) is composed of trees);

[2]The notion of a store was first developed for the XQuery data model [17, 22].

- $\lambda : V \to \mathcal{A}$ is the *labeling function* which associates each node in V with its *label*;[3]
- $<$ is the *sibling order*: a strict partial order on V that compares exactly the different children of a common node:

$$(n < n') \vee (n' < n) \Leftrightarrow \exists m \in V : E(m, n) \wedge E(m, n');$$

and

- $\prec$ is the *root order*: a strict total order on the roots (i.e., the nodes with in-degree zero).

As an example, Figure 1(a) depicts the store $(V, E, \lambda, <, \prec)$ where

$$\begin{aligned} V &= \{n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8, n_9\} \\ E &= \{(n_1, n_2), (n_1, n_4), (n_2, n_3), (n_5, n_6), (n_5, n_8), (n_6, n_7), (n_8, n_9)\} \\ < &= \{(n_2, n_4), (n_6, n_8)\} \\ \prec &= \{(n_1, n_5)\}, \end{aligned}$$

and where λ is defined by

$$\begin{array}{lll} \lambda(n_1) := \text{beer} & \lambda(n_2) := \text{name} & \lambda(n_3) := \text{Duvel} \\ \lambda(n_4) := \text{blond} & \lambda(n_5) := \text{name} & \lambda(n_6) := \text{first} \\ \lambda(n_7) := \text{John} & \lambda(n_8) := \text{last} & \lambda(n_9) := \text{Doe}. \end{array}$$

Document order Using the sibling and root order, we define the *document order* $\ll$ on Σ which intuitively equals the left-to-right, pre-order traversal of a list of trees. As an example, for the store in Figure 1(a) we have $n_i \ll n_j$ if, and only if, i is smaller than j.

Formally, the document order $\ll$ is the strict total order on V such that (1) if $E(n, n')$ then $n \ll n'$, and (2) if $m < n$ or $m \prec n$, $E^*(m, m')$, and $E^*(n, n')$, then $m' \ll n'$. Here we write E^* for the reflexive transitive closure of E.

Terminology We will use the standard terminology for trees on stores. That is, if $E(m, n)$ then m is the *parent* of n and n is a *child* of m. A node $n \in V$ is a *root node* of Σ if it has in-degree zero. We write *roots*(Σ) for the set of all root nodes in Σ. If Σ has at most one root node, then we say that Σ is a *tree*. Note that the empty store is hence also a tree. For convenience we will denote the empty store by $\varnothing$.

[3]The label of a text node is called the node's *content* in XQuery terminology.

Concatenation Two stores Σ and Σ' are *disjoint* when $V_\Sigma \cap V_{\Sigma'} = \emptyset$. If Σ and Σ' are disjoint stores then the *concatenation of* Σ and Σ', denoted by $\Sigma \circ \Sigma'$, is the store with node set $V_\Sigma \cup V_{\Sigma'}$, edges $E_\Sigma \cup E_{\Sigma'}$, labeling function $\lambda_\Sigma \cup \lambda_{\Sigma'}$, sibling order $<_\Sigma \cup <_{\Sigma'}$, and root order

$$\prec_\Sigma \cup \prec_{\Sigma'} \cup \ \mathit{roots}(\Sigma) \times \mathit{roots}(\Sigma').$$

Clearly, all stores can be written as a concatenation of trees.

Sub-trees Finally, if n is a node in Σ, then the *sub-tree of* Σ *rooted at* n, denoted by $\Sigma|_n$, is the store with nodes $V' = \{m \mid E^*(n, m)\}$, edges $E \cap (V' \times V')$, labeling function $\lambda|_{V'}$, sibling order $< \cap (V' \times V')$, and the empty root order.

2.3 Values

A *value-tuple* of arity p is a tuple $(\Sigma; s_1, \ldots, s_p)$ where Σ is a store and every s_j is a finite list of atoms and nodes in Σ. A *value* is a value-tuple of arity one. We write $\mathcal{V}_p$ for the set of all value-tuples with arity p and abbreviate $\mathcal{V}_1$ by $\mathcal{V}$.

We denote the empty list by $\langle\rangle$, non-empty lists by for example $\langle a, b, c\rangle$, and the concatenation of two lists s_1 and s_2 by $s_1 \circ s_2$. In addition, we will write $s(j)$ for the j-th item of a list s, $|s|$ for the width of s, and $\mathit{rng}(s)$ for the set of items occurring in s.

2.4 Renamings

A *renaming* ρ is a permutation of $\mathcal{A} \cup \mathcal{N}$ that is the identity on the booleans and maps atoms to atoms, element nodes to element nodes, and text nodes to text nodes. A *node-renaming* is a renaming that is the identity on atoms. Renamings are extended to sets, tuples, and lists in the canonical way:

$$\begin{aligned} \rho(S) &= \{\rho(v) \mid v \in S\} \\ \rho((v_1, \ldots, v_p)) &= (\rho(v_1), \ldots, \rho(v_p)) \\ \rho(\langle v_1, \ldots, v_p\rangle) &= \langle\rho(v_1), \ldots, \rho(v_p)\rangle \end{aligned}$$

Note that in particular ρ is thus also extended to stores and value-tuples.

Two value-tuples v and v' are *isomorphic*, denoted by $v \equiv v'$, when there exists a renaming ρ such that $\rho(v) = v'$. Two value-tuples v an v' are *node-isomorphic*, denoted by $v \equiv_{node} v'$, when there exists a node-renaming such that $\rho(v) = v'$.

2.5 Conventions

We will use the following conventions throughout this paper. We will abbreviate tuples such as $t_1, \ldots, t_p$ by $\vec{t}$, write $[k, l]$ for the subset $\{i \mid k \leq i \leq l\}$

of the natural numbers, and write $|S|$ for the cardinality of a set S. Furthermore, if g is a function from set S to set T, then we write $dom(g)$ for S and $rng(g)$ for $\{g(i) \mid i \in S\}$. If S' is a subset of S then we write $g|_{S'}$ for the function from S' to T which equals g on S. Finally, if g is injective and $j \in rng(g)$, then we write $g^{-1}(j)$ for the unique element $i \in S$ for which $g(i) = j$.

3 Syntax and semantics

3.1 Base operations

A *base operation* of arity p is a relation $R \subseteq \mathcal{V}_p \times \mathcal{V}$ which is

1. *Computable*: it is effectively decidable, given a value-tuple v, whether there exists a w such that $R(v, w)$, and if so, such a w is effectively computable from v.

2. *Store-increasing*: R only relates value-tuples $(\Sigma; \vec{s})$ to values of the form $(\Sigma \circ \Sigma'; s')$ with Σ' possibly empty. Hence, R can add trees to a store, but cannot modify existing trees.

3. *Node-generic*: for every node-renaming ρ we have $R(v, w)$ if, and only if, $R(\rho(v), \rho(w))$. As such, R can only interpret nodes by the information given in the input store. Furthermore, nodes that are added to the input store are chosen non-deterministically.

4. a *Semi-function*: R is a function up to node-isomorphism, i.e., if $R(v, w)$ and $R(v, z)$, then $w \equiv_{node} z$.

5. *Reachable-only*: R only uses information of those trees in the input store whose nodes are mentioned in one of the input lists. That is, for all list-tuples $\vec{s}$, all (possibly empty) trees $\Theta_1, \ldots, \Theta_k, \Theta'_1, \ldots, \Theta'_k$ such that $\Theta_j = \Theta'_j$ if a node of Θ_j is mentioned in $\vec{s}$, and all stores Σ disjoint with $\Theta_1, \ldots, \Theta_k, \Theta'_1, \ldots, \Theta'_k$, we have

$$R((\Theta_1 \circ \cdots \circ \Theta_k; \vec{s}), (\Theta_1 \circ \cdots \circ \Theta_k \circ \Sigma; s')) \Leftrightarrow R((\Theta'_1 \circ \cdots \circ \Theta'_k; \vec{s}), (\Theta'_1 \circ \cdots \circ \Theta'_k \circ \Sigma; s')).$$

We write $R(v)$ for the set of all values w for which $R(v, w)$ holds. The first four properties above capture the notion of a "determinate" transformation from the theory of object-creating queries [1]. As such, $R(v)$ is finitely representable and this representation can effectively be computed from v.

Many of the basic functions and operators found in programming and query languages are in fact base operations. Since our study was motivated by XQuery, we clarify this claim by some of XQuery's basic functions and operators.

- XQuery's concatenation operator is the binary base operation that relates $(\Sigma; s, s')$ to $(\Sigma; s \circ s')$. Although this operator is denoted by a comma in XQuery, we will denote it by *concat*.

- XQuery's *children* axis is a unary base operation that relates $(\Sigma; s)$, with s a list of nodes, to $(\Sigma; s')$ where s' is the unique list containing the children of nodes in s in document order. Formally this means that
$$rng(s') = \{n \mid \exists m \in rng(s) : E(m, n)\},$$
and that if $i < j$, then $s'(i) \ll s'(j)$. Note that there are no repeated nodes in s', since $\ll$ is a strict order. XQuery's other axes (i.e., *parent*, *descendant*, *following-sibling*, ...) can similarly be viewed as unary base operations.

- XQuery's atomization function *data* can be modeled as a unary base operation that relates $(\Sigma; s)$ to $(\Sigma; s')$ where s' has the same width as s, and $s'(j)$ is the coercion of $s(j)$ to an atom. That is, $s'(j) = s(j)$ when $s(j)$ is an atom, $s'(j) = \lambda(s(j))$ if $s(j)$ is a text node, and $s'(j) = fold(r)$ if $s(j)$ is an element node. Here, r is the unique list containing all text node descendants of $s(j)$ in document order and *fold* is an abstract function mapping lists of text nodes to atoms. In XQuery, *fold* returns the string concatenation of the text nodes' labels. Figure 2 illustrates the behavior of *data* under this interpretation of *fold*.

- The atomic value comparison *eq* is a binary base operation that relates $(\Sigma; s, s')$ to $(\Sigma; \langle\rangle)$ if s or s' is the empty list, and relates $(\Sigma; \langle a\rangle, \langle b\rangle)$ to $(\Sigma; \langle a = b\rangle)$. We will use a C-style notation for comparisons: $a = b$ evaluates to `true` when a equals b, and evaluates to `false` otherwise. We note that in XQuery, *eq* will actually first atomize its arguments using the *data* function described earlier, and then compare the obtained lists according to our semantics. We show how this behavior can be simulated in our query language QL(B) in Example 2.

- XQuery's node comparisons *is* and $\ll$ are binary base operations that relate $(\Sigma; s, s')$ to $(\Sigma; \langle\rangle)$ if s or s' is the empty list, and relate $(\Sigma; \langle n\rangle, \langle m\rangle)$ to $(\Sigma; \langle n = m\rangle)$ respectively $(\Sigma; \langle n \ll m\rangle)$.

- XQuery's kind tests *is-element* and *is-text* are unary base operations that relate $(\Sigma; \langle n\rangle)$ to $(\Sigma; \langle n \in \mathcal{N}^e\rangle)$ respectively $(\Sigma; \langle n \in \mathcal{N}^t\rangle)$.[4] We will also consider a kind test *is-atom* which relates $(\Sigma; \langle i\rangle)$ with i an item to $(\Sigma; \langle i \in \mathcal{A}\rangle)$.

[4]Kind tests are part of XPath expressions in XQuery, and are written as for example `$x/self::element()` or `$x/self::text()`.

- XQuery's *node-name* function is a unary base operation that relates $(\Sigma; \langle\rangle)$ to $(\Sigma; \langle\rangle)$, relates $(\Sigma; \langle n\rangle)$ to $(\Sigma; \langle\lambda(n)\rangle)$ when $n \in \mathcal{N}^e$, and relates $(\Sigma; \langle n\rangle)$ to $(\Sigma; \langle\rangle)$ when $n \in \mathcal{N}^t$. We will also consider a base operation *content* which behaves like *node-name*, but then on text nodes.

- The element node constructor *element* is the most involved operation in XQuery. In order to simplify our proofs later on, we here present a simplified version of this constructor as a base operation. By composition with other base operations we are capable of simulating the XQuery version in our query language QL(B), as we show in Example 3.

  The element node constructor *element* is a binary base operation that relates $(\Sigma; \langle a\rangle, \langle n_1, \ldots, n_k\rangle)$ to $(\Sigma \circ \Theta, \langle m\rangle)$ where Θ is a tree, disjoint with Σ whose root element node m is labeled by a such that

  - m has exactly k children, and
  - if m_j is the j-th child of m (in sibling order), then $\Sigma|_{n_j} \equiv_{node} \Theta|_{m_j}$.

  Note that there can be duplicates in $n_1, \ldots, n_k$. Figure 2 illustrates the behavior of *element*.

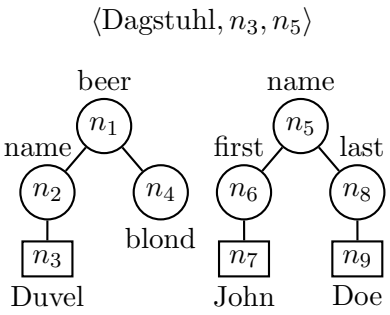
- XQuery's text node constructor *text* is a unary base operation that relates $(\Sigma; \langle a\rangle)$ to $(\Sigma \circ \Theta, \langle m\rangle)$ where Θ is a tree, disjoint with Σ, whose root text node m is labeled by a.

- After constructing a new element node, XQuery merges adjacent text node children into a single text node whose label is the concatenation of the labels of the original text nodes. This behavior can be modeled as a unary base operation *merge-text* that relates $(\Sigma; \langle n\rangle)$ to $(\Sigma \circ \Theta; \langle m\rangle)$ where Θ is a tree with root node m, disjoint with Σ, which is isomorphic to $\Sigma|_n$ after we merge all adjacent text nodes in $\Sigma|_n$ into a single text node by means of the abstract *fold* function introduced above for the atomization function *data*. Figure 2 illustrates the behavior of *merge-text*.

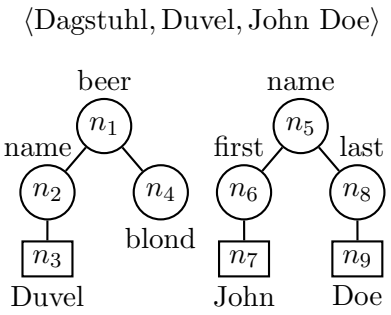
- A final example of a unary base operation is XQuery's emptiness test function *empty* that relates $(\Sigma; s)$ to $(\Sigma; \langle\texttt{true}\rangle)$ when $s = \langle\rangle$, and relates $(\Sigma; s)$ to $(\Sigma; \langle\texttt{false}\rangle)$ otherwise.

3.2 Expressions

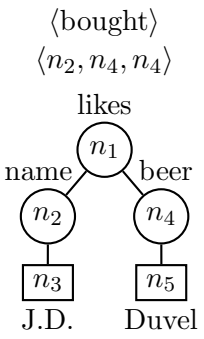
We create a query language QL(B) out of a finite set of base operations B by adding variables, constants, and basic control-flow as follows. For each base



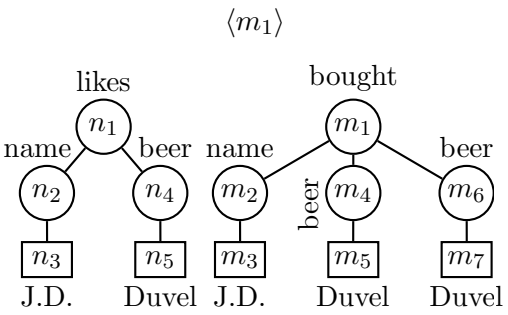
(a) Value u



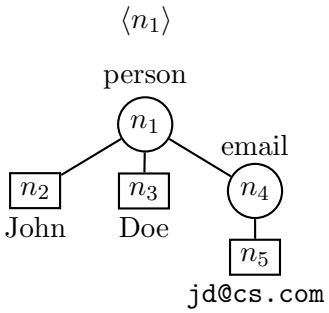
(b) Value in $data(u)$



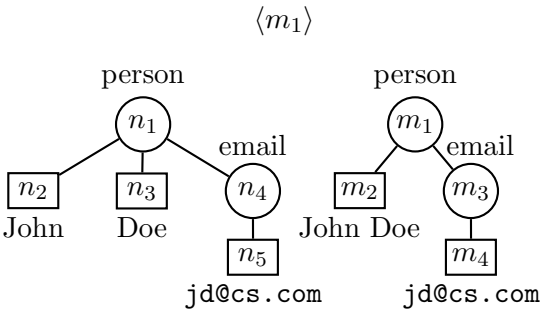
(c) Value-tuple v



(d) Value in $element(v)$



(e) Value w



(f) Value in $merge\text{-}text(w)$

Figure 2: Illustration of the base operations *data*, *element*, and *merge-text*.

operation R we assume to be given a *base expression* f: a unique syntactical entity which denotes R. For ease of notation we will often not distinguish between a base operation and its associated base expression. As such we will write for example $v \in f(w)$ to denote $v \in R(w)$.

The syntax of QL(B) is defined by the following grammar:[5]

$$\begin{array}{rcl} e & ::= & x \mid a \mid () \mid f(e_1, \ldots, e_p) \\ & \mid & \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \\ & \mid & \texttt{let } x := e_1 \texttt{ return } e_2 \\ & \mid & \texttt{for } x \texttt{ in } e_1 \texttt{ return } e_2 \end{array}$$

Here, e ranges over *expressions*, x ranges over variables, a ranges over atoms, f ranges over base expressions in B, and p is the arity of f. We view expressions as abstract syntax trees and omit parentheses. The set $FV(e)$ of *free variables* of an expression e is defined as usual. That is, the free variables of x is $\{x\}$, the free variables of a and $()$ is the empty set, the free variables of $f(e_1, \ldots, e_p)$ and $\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$ is the union of the free variables of their immediate subexpressions, and the free variables of $\texttt{let } x := e_1 \texttt{ return } e_2$ and $\texttt{for } x \texttt{ in } e_1 \texttt{ return } e_2$ is

$$FV(e_1) \cup (FV(e_2) \setminus \{x\}).$$

3.3 Semantics

The input to an expression e is described by a *context* $(\Sigma; \sigma)$ *on* e, consisting of a store Σ and a function σ from a finite superset $\{x, \ldots, y\}$ of the free variables of e to lists of items such that $(\Sigma; \sigma(x), \ldots, \sigma(y))$ is a value-tuple. A function from a finite set of variables to lists of items is called an *environment*. We write $x\colon s, \sigma$ for the environment σ' with domain $dom(\sigma) \cup \{x\}$ such that $\sigma'(x) = s$ and $\sigma'(y) = \sigma(y)$ for $y \neq x$.

The semantics of an expression e is described by means of the *evaluation relation*, as defined in Figure 3. Here, we write $(\Sigma; \sigma) \models e \Rightarrow (\Sigma'; s)$ to denote the fact that e evaluates to value $(\Sigma'; s)$ on context $(\Sigma; \sigma)$ on e. We note that the disjointness requirements in the rules for base operation invocation and for loop ensure that different invocations of a subexpression add different nodes to the input store. We will write $e(\Sigma; \sigma)$ for the set of all values to which e can evaluate on context $(\Sigma; \sigma)$. It is easy to see that the semantics of an expression only depends on its free variables: if two environments σ and σ' are equal on $FV(e)$, then $(\Sigma; \sigma) \models e \Rightarrow (\Sigma'; s)$ if, and only if, $(\Sigma; \sigma') \models e \Rightarrow (\Sigma'; s)$.

[5] One may wonder why we consider let-expressions of the form $\texttt{let } x := e_1 \texttt{ return } e_2$. As will become clear from the semantics as defined in Section 3.3, such expressions are not redundant. This is due to the fact that base operations can create new nodes. Hence, $\texttt{let } x := e_1 \texttt{ return } e_2$ is not necessarily equivalent to the expression we obtain by replacing every free occurrence of x in e_2 by e_1.

$$\frac{\sigma(x) = s}{(\Sigma;\sigma) \models x \Rightarrow (\Sigma; s)} \qquad \frac{}{(\Sigma;\sigma) \models a \Rightarrow (\Sigma; \langle a \rangle)} \qquad \frac{}{(\Sigma;\sigma) \models () \Rightarrow (\Sigma; \langle\rangle)}$$

$$\frac{\begin{array}{c}(\Sigma;\sigma) \models e_1 \Rightarrow (\Sigma_1; \langle \texttt{true} \rangle) \\ (\Sigma;\sigma) \models e_2 \Rightarrow (\Sigma_2; s_2)\end{array}}{(\Sigma;\sigma) \models \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow (\Sigma_2; s_2)}$$

$$\frac{\begin{array}{c}(\Sigma;\sigma) \models e_1 \Rightarrow (\Sigma_1; \langle \texttt{false} \rangle) \\ (\Sigma;\sigma) \models e_3 \Rightarrow (\Sigma_3; s_3)\end{array}}{(\Sigma;\sigma) \models \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow (\Sigma_3; s_3)}$$

$$\frac{\begin{array}{c}(\Sigma;\sigma) \models e_1 \Rightarrow (\Sigma_1; s_1) \\ (\Sigma_1; x\colon s_1, \sigma) \models e_2 \Rightarrow (\Sigma_2; s_2)\end{array}}{(\Sigma;\sigma) \models \texttt{let}\ x := e_1\ \texttt{return}\ e_2 \Rightarrow (\Sigma_2; s_2)}$$

$$\frac{\begin{array}{c}(\Sigma;\sigma) \models e_j \Rightarrow (\Sigma \circ \Sigma_j; s_j) \quad j \in [1, p] \\ \Sigma_j \text{ is disjoint with } \Sigma_{j'} \text{ when } j \neq j' \\ (\Sigma'; s') \in f(\Sigma \circ \Sigma_1 \circ \cdots \circ \Sigma_p; s_1, \ldots, s_p)\end{array}}{(\Sigma, \sigma) \models f(e_1, \ldots, e_p) \Rightarrow (\Sigma', s')}$$

$$\frac{\begin{array}{c}(\Sigma;\sigma) \models e_1 \Rightarrow (\Sigma_0; s) \quad (\Sigma_0; x\colon \langle s(j) \rangle, \sigma) \models e_2 \Rightarrow (\Sigma_0 \circ \Sigma_j; s_j) \quad j \in [1, |s|] \\ \Sigma_j \text{ is disjoint with } \Sigma_{j'} \text{ when } j \neq j'\end{array}}{(\Sigma;\sigma) \models \texttt{for}\ x\ \texttt{in}\ e_1\ \texttt{return}\ e_2 \Rightarrow (\Sigma_0 \circ \cdots \circ \Sigma_{|s|}; s_1 \circ \cdots \circ s_{|s|})}$$

Figure 3: The evaluation relation.

Example 1. XPath expressions like `$bib/child::book` can be simulated in QL(*children*, *is-element*, *node-name*, *eq*) as follows:

```
for b in children(bib) return
  if is-element(b) then
    if eq(node-name(b), 'book') then b else ()
  else ()
```

Example 2. In Section 3.1 we noted that XQuery's value comparison first atomizes its arguments and then compares them using the semantics of our base operation *eq*. Since QL(B) allows composition of base operations, we can simulate this behavior. For example, `$x eq 'ACM'` can be simulated by *eq*(*data*(x), `'ACM'`). Note that there is actually no need to apply *data* to the constant `'ACM'`, as this returns `'ACM'` itself. □

Example 3. In Section 3.1 we also noted that XQuery's element constructor is more complex than our base operation *element*. Indeed, the XQuery expression `element{$x}{$y}` will create a new tree whose root node is labeled by x (after atomization) and whose children are copies of the items in y where new text nodes are created for the atoms, and where adjacent text nodes are merged. Using the base operations *element*, *text*, *is-atom*, *data*, and *merge-text* we can simulate this behavior as follows:

```
let z:= for u in y return
        if is-atom(u) then text(u) else u
return merge-text(element(data(x), z))
```
□

Example 4. XQuery's quantified expressions can be simulated using the emptiness test. For example,

```
some $x in data($pubs) satisfies $x eq 'ACM'
```

can be expressed in QL(*eq*, *data*, *empty*) as follows:

```
let z :=
  for x in data(pubs) return
    if eq(x, 'ACM') then x else ()
return
  if empty(z) then false else true
```

It immediately follows that XQuery's generalized comparisons (such as for example `$pubs = "ACM"`) can also be simulated using the emptiness test, as such comparisons are just syntactic sugar for quantified expressions like the one shown above. □

Example 5. We can simulate the XQuery expression

```
for $b in $bib/book
where $b/publisher eq 'ACM'
return element{$b/author}{$b/title}
```

in QL(*children*, *data*, *eq*, *is-element*, *node-name*, *element*, *merge-text*) as follows. For the sake of brevity we will not expand XPath expressions such as `$bib/book`, as we have already shown how to simulate these in Example 1.

```
for b in bib/book return
  if eq(data(b/publisher), 'ACM') then
    merge-text(element(data(b/author), b/title))
  else ()
```

Here we omit the creation of new text nodes for atoms returned by `b/title`, as XPath expressions always return nodes, never atoms. □

When we fix some order on the set of all variables, a context $(\Sigma; \sigma)$ with $dom(\sigma) = \{x, \ldots, y\}$ is fully determined by the value-tuple

$$(\Sigma; \sigma(x), \ldots, \sigma(y)).$$

Since the semantics of an expression only depends on its free variables, every expression e hence defines a relation on $\mathcal{V}_p \times \mathcal{V}$, where p is the number of free variables in e.

In the remainder of this section we will prove the following proposition.

Proposition 6. *Every expression e in QL(B) defines a base operation.*

The store-increasing and node-generic properties follow by an easy induction on e. For the reachable-only property we first state the following lemma.

Lemma 7. *If R is a reachable-only relation relating value-tuples to values and*

$$(\Theta_1 \circ \cdots \circ \Theta_k \circ \Sigma'; s') \in R(\Theta_1 \circ \cdots \circ \Theta_k; \vec{s}),$$

then s' only contains a node in Θ_j if $\vec{s}$ contains a node in Θ_j, for every $j \in [1, k]$.

Proof. Let $j \in [1, k]$, let $\Pi_1 = \Theta_1 \circ \cdots \circ \Theta_{j-1}$, and let $\Pi_2 = \Theta_{j+1} \circ \cdots \circ \Theta_k$. Suppose, for the purpose of contradiction, that s' contains a node n in Θ_j, but $\vec{s}$ does not. Since $\Theta_1, \ldots, \Theta_k$, and Σ' all have pairwise disjoint sets of nodes, n cannot be a node of $\Pi_1 \circ \Pi_2 \circ \Sigma'$. Hence, $(\Pi_1 \circ \Pi_2 \circ \Sigma'; s')$ is not a value. Since $(\Pi_1 \circ \Theta_j \circ \Pi_2 \circ \Sigma'; s') \in R(\Pi_1 \circ \Theta_j \circ \Pi_2; \vec{s})$ and since R is reachable-only, we also should have

$$(\Pi_1 \circ \Pi_2 \circ \Sigma'; s') \in R(\Pi_1 \circ \Pi_2; \vec{s}).$$

This is a contradiction, since R relates value-tuples to values. □

Proposition 8. *Every expression in QL(B) defines a reachable-only relation.*

Proof. Let e be an expression in QL(B) and let σ be an environment on e. Let $\Theta_1, \ldots, \Theta_k$ and $\Theta'_1, \ldots, \Theta'_k$ be trees such that $\Theta_1, \ldots, \Theta_k$ are all pairwise disjoint, $\Theta'_1, \ldots, \Theta'_k$ are all pairwise disjoint, and $\Theta_j = \Theta'_j$ if a node of Θ_j is mentioned in σ. Let $\Pi = \Theta_1 \circ \cdots \circ \Theta_k$, $\Pi' = \Theta'_1 \circ \cdots \circ \Theta'_k$, and let Σ be a store disjoint with $\Theta_1, \ldots, \Theta_k, \Theta'_1, \ldots, \Theta'_k$. We prove by induction on e that $(\Pi \circ \Sigma; s) \in e(\Pi; \sigma)$ if, and only if, $(\Pi' \circ \Sigma; s) \in e(\Pi'; \sigma)$. In every step we only show the "only if" direction, the "if" direction is similar.

- The cases where $e = x$, $e = a$, or $e = ()$ are trivial.
- If $e = \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$, then $(\Pi \circ \Sigma; s) \in e(\Pi; \sigma)$ only if there exists $(\Pi \circ \Sigma'; \langle b \rangle) \in e_1(\Pi; \sigma)$ with $b = \texttt{true}$ or $b = \texttt{false}$ such that $(\Pi \circ \Sigma; s) \in e_2(\Pi \circ \Sigma; s)$ if $b = \texttt{true}$ and $(\Pi \circ \Sigma; s) \in e_3(\Pi \circ \Sigma; s)$ if $b = \texttt{false}$. The result then readily follows by the induction hypothesis:

  $$\begin{aligned}
  &(\Pi \circ \Sigma; s) \in e(\Pi; \sigma) \\
  &\quad\Rightarrow\ (\Pi \circ \Sigma'; \langle b \rangle) \in e_1(\Pi; \sigma) \text{ and } (\Pi \circ \Sigma; s) \in e_2(\Pi; \sigma) \cup e_3(\Pi; \sigma) \\
  &\quad\Rightarrow\ (\Pi' \circ \Sigma'; \langle b \rangle) \in e_1(\Pi'; \sigma) \text{ and } (\Pi' \circ \Sigma; s) \in e_2(\Pi'; \sigma) \cup e_3(\Pi'; \sigma) \\
  &\quad\Rightarrow\ (\Pi' \circ \Sigma; s) \in e(\Pi'; \sigma)
  \end{aligned}$$

- If $e = \texttt{let}\ x := e_1\ \texttt{return}\ e_2$, then $(\Pi \circ \Sigma; s) \in e(\Pi; \sigma)$ only if, there exists $(\Pi \circ \Sigma_1; s_1) \in e_1(\Pi; \sigma)$ such that $(\Pi \circ \Sigma; s) \in e_2(\Pi \circ \Sigma_1; x\colon s_1, \sigma)$. Since e_1 defines a reachable-only relation by the induction hypothesis, it follows from Lemma 7 that s_1 contains a node in Θ_j only if σ contains a node in Θ_j, for every $j \in [1, k]$. Hence, $\Theta'_j = \Theta_j$ when the environment $(x\colon s_1, \sigma)$ contains a node in Θ_j. The result then readily follows by the induction hypothesis:

  $$\begin{aligned}
  &(\Pi \circ \Sigma; s) \in e(\Pi; \sigma) \\
  &\quad\Rightarrow\ (\Pi \circ \Sigma_1; s_1) \in e_1(\Pi; \sigma) \text{ and } (\Pi \circ \Sigma; s) \in e_2(\Pi \circ \Sigma_1; x\colon s_1, \sigma) \\
  &\quad\Rightarrow\ (\Pi' \circ \Sigma_1; s_1) \in e_1(\Pi'; \sigma) \text{ and } (\Pi' \circ \Sigma; s) \in e_2(\Pi' \circ \Sigma_1; x\colon s_1, \sigma) \\
  &\quad\Rightarrow\ (\Pi' \circ \Sigma; s) \in e(\Pi'; \sigma)
  \end{aligned}$$

- If $e = f(e_1, \ldots, e_p)$, then $(\Pi \circ \Sigma; s) \in e(\Pi; \sigma)$ only if for every $i \in [1, p]$ there exists $(\Pi \circ \Sigma_i; s_i) \in e_i(\Pi; \sigma)$ such that the Σ_i are all pairwise disjoint and

  $$(\Pi \circ \Sigma; s) \in f(\Pi \circ \Sigma_1 \circ \cdots \circ \Sigma_p; s_1, \ldots, s_p).$$

  By Lemma 7 it follows that s_i contains a node in Θ_j only if σ contains a node in Θ_j, for every $i \in [1, p]$ and every $j \in [1, k]$. Hence, $\Theta_j = \Theta'_j$

when the list-tuple $s_1, \ldots, s_p$ contains a node in Θ_j. Let us abbreviate $\Sigma_1 \circ \cdots \circ \Sigma_p$ by Σ' and let us write $\vec{s}$ for $s_1, \ldots, s_p$. The result then readily follows by the induction hypothesis and the fact that f itself is reachable-only:

$$\begin{array}{ll} & (\Pi \circ \Sigma; s) \in e(\Pi; \sigma) \\ \Rightarrow & (\Pi \circ \Sigma_i; s_i) \in e_i(\Pi; \sigma) \text{ for every } i \in [1, p] \\ & \quad \text{and } (\Pi \circ \Sigma; s) \in f(\Pi \circ \Sigma'; \vec{s}) \\ \Rightarrow & (\Pi' \circ \Sigma_i; s_i) \in e_i(\Pi'; \sigma) \text{ for every } i \in [1, p] \\ & \quad \text{and } (\Pi' \circ \Sigma; s) \in f(\Pi' \circ \Sigma'; \vec{s}) \\ \Rightarrow & (\Pi' \circ \Sigma; s) \in e(\Pi'; \sigma) \end{array}$$

- If $e = \texttt{for}\ x\ \texttt{in}\ e_1\ \texttt{return}\ e_2$, then $(\Pi \circ \Sigma; s) \in e(\Pi; \sigma)$ only if there exists a value $(\Pi \circ \Sigma_0; s_0) \in e_1(\Pi; \sigma)$ such that for every $i \in [1, |s_0|]$ there exists $(\Pi \circ \Sigma_0 \circ \Sigma_i; s_i) \in e_2(\Pi \circ \Sigma_0; x\colon \langle s_0(i) \rangle, \sigma)$, with the Σ_i's pairwise disjoint, $\Sigma = \Sigma_0 \circ \cdots \circ \Sigma_{|s_0|}$, and $s = s_1 \circ \cdots \circ s_{|s_0|}$. Since e_1 defines a reachable-only relation by the induction hypothesis, it follows from Lemma 7 that s_0 contains a node in Θ_j only if σ contains a node in Θ_j, for every $j \in [1, k]$. Hence, $\Theta_j = \Theta'_j$ when the environment $(x\colon \langle s_0(i) \rangle, \sigma)$ contains a node in Θ_j, for every $j \in [1, k]$ and every $i \in [1, |s_0|]$. The result then readily follows by the induction hypothesis:

$$\begin{array}{ll} & (\Pi \circ \Sigma; s) \in e(\Pi; \sigma) \\ \Rightarrow & (\Pi \circ \Sigma_0; s_0) \in e_1(\Pi; \sigma) \\ & \quad \text{and } \forall i \in [1, |s_0|] : (\Pi \circ \Sigma_0 \circ \Sigma_i; s_i) \in e_1(\Pi; x\colon \langle s_0(i) \rangle, \sigma) \\ \Rightarrow & (\Pi' \circ \Sigma_0; s_0) \in e_1(\Pi'; \sigma) \\ & \quad \text{and } \forall i \in [1, |s_0|] : (\Pi' \circ \Sigma_0 \circ \Sigma_i; s_i) \in e_1(\Pi'; x\colon \langle s_0(i) \rangle, \sigma) \\ \Rightarrow & (\Pi' \circ \Sigma; s) \in e(\Pi'; \sigma) \end{array}$$

□

In order to prove that every expression e defines a semi-function, we first state the following lemmas.

Lemma 9. *Let $\rho_1, \ldots, \rho_k$ be node-renamings and let $N_1, \ldots, N_k$ be pairwise disjoint finite sets of nodes such that $\rho_j(N_j) \cap \rho_{j'}(N_{j'}) = \emptyset$ when $j \neq j'$. There exists a node-renaming ρ such that $\rho|_{N_j} = \rho_j|_{N_j}$ for all $j \in [1, k]$.*

Proof. Let $X = N_1 \cup \cdots \cup N_k$ and let $Y = \rho_1(N_1) \cup \cdots \cup \rho_k(N_k)$. Let γ be the function on X which equals ρ_j on N_j for every $j \in [1, k]$. Note that, since the N_j are pairwise disjoint, γ is indeed a function. Further note that γ is injective since every ρ_j is injective and since $\rho_j(N_j) \cap \rho_{j'}(N_{j'}) = \emptyset$ when $j \neq j'$. Hence, γ is a bijection from X to Y. Hence, $|X| = |Y|$,

and consequently, $|Y - X| = |X - Y|$. We can therefore extend γ to a permutation γ' of $X \cup Y$ by picking for each $n \in Y - X$ a unique element $\gamma'(n)$ in $X - Y$. Let π be a permutation of $\mathcal{N} - (X \cup Y)$. Then $\pi \cup \gamma'$ is a permutation of $\mathcal{N}$ which equals ρ_j on N_j for every $j \in [1, k]$. Hence, the node-renaming ρ which equals $\pi \cup \gamma'$ on $\mathcal{N}$ also has this property. □

Lemma 10. *Let $\Sigma_1 \circ \Sigma_2$ and $\Sigma_3 \circ \Sigma_4$ be two stores such that Σ_1 is node-isomorphic to Σ_3 and let ρ be a node-renaming such that $\rho(\Sigma_1 \circ \Sigma_2) = (\Sigma_3 \circ \Sigma_4)$. Then $\rho(\Sigma_1) = \Sigma_3$ and $\rho(\Sigma_2) = \Sigma_4$.*

Proof. It is easy to see that node-renamings commute with concatenation. Hence, $\rho(\Sigma_1) \circ \rho(\Sigma_2) = \rho(\Sigma_1 \circ \Sigma_2) = \Sigma_3 \circ \Sigma_4$, which implies that the first j trees of $\rho(\Sigma_1) \circ \rho(\Sigma_2)$ equal the first j trees of $\Sigma_3 \circ \Sigma_4$. Since Σ_1 is node-isomorphic to Σ_3 it follows that they consists of exactly the same number of trees. Hence, $\rho(\Sigma_1) = \Sigma_3$ and thus $\rho(\Sigma_2) = \Sigma_4$. □

Lemma 11. *Let $(\Sigma \circ \Sigma_1; \vec{s_1}), \ldots, (\Sigma \circ \Sigma_k; \vec{s_k}), (\Sigma' \circ \Sigma'_1; \vec{s'_1}), \ldots, (\Sigma' \circ \Sigma'_k; \vec{s'_k})$ be value-tuples such that Σ, Σ_1, …, Σ_k are all pairwise disjoint, Σ', Σ'_1, …, Σ'_k are all pairwise disjoint, Σ is node-isomorphic to Σ', and such that $(\Sigma \circ \Sigma_j; \vec{s_j})$ is node-isomorphic to $(\Sigma' \circ \Sigma'_j; \vec{s'_j})$, for every $j \in [1, k]$. Then*

$$(\Sigma \circ \bigcirc_{j=1}^{k} \Sigma_j; \vec{s_1}, \ldots, \vec{s_k}) \equiv_{node} (\Sigma' \circ \bigcirc_{j=1}^{k} \Sigma'_j; \vec{s'_1}, \ldots, \vec{s'_k}).$$

Proof. Since $(\Sigma \circ \Sigma_j; \vec{s_j})$ and $(\Sigma' \circ \Sigma'_j; \vec{s'_j})$ are node-isomorphic value-tuples for every $j \in [1, k]$, there exist node-renamings ρ_j such that

$$\rho_j(\Sigma \circ \Sigma_j; \vec{s_j}) = (\Sigma' \circ \Sigma'_j; \vec{s'_j}).$$

Since Σ is node-isomorphic to Σ', it follows from Lemma 10 that $\rho_j(\Sigma) = \Sigma'$ and that $\rho_j(\Sigma_j) = \Sigma'_j$. Let N be the set of nodes in Σ. Then in particular we have that $\rho_j|_N = \rho_{j'}|_N$ for every j and j' in $[1, k]$ (as the nodes in a store are ordered). Let $\pi = \rho_1|_N$. Since $\Sigma, \Sigma_1, \ldots, \Sigma_k$ are all pairwise disjoint and since $\Sigma', \Sigma'_1, \ldots, \Sigma'_k$ are also all pairwise-disjoint it follows from Lemma 9 that there exists a node-renaming ρ such that ρ equals π on nodes in Σ and equals ρ_j on nodes in Σ_j, for every $j \in [1, k]$. Hence,

$$\begin{aligned} \rho(\Sigma \circ \bigcirc_{j=1}^{k} \Sigma_j; \vec{s_1}, \ldots, \vec{s_k}) &= (\rho(\Sigma) \circ \bigcirc_{i=1}^{k} \rho_j(\Sigma_j); \rho(\vec{s_1}), \ldots, \rho(\vec{s_k})) \\ &= (\Sigma' \circ \bigcirc_{j=1}^{k} \Sigma'_j; \vec{s'_1}, \ldots, \vec{s'_k}), \end{aligned}$$

as desired. □

Corollary 12. *If $R \subseteq \mathcal{V}_p \times \mathcal{V}$ is a store-increasing semi-function and $(\Sigma_1; s_1)$ and $(\Sigma_2; s_2)$ are two values in $R(\Sigma; \vec{s})$, then $(\Sigma_1; s_1, \vec{s})$ is node-isomorphic to $(\Sigma_2; s_2, \vec{s})$.*

Proof. Since R is store-increasing, there exist stores Σ_1' and Σ_2' such that $\Sigma_1 = \Sigma \circ \Sigma_1'$ and $\Sigma_2 = \Sigma \circ \Sigma_2'$. Since R is a semi-function, $(\Sigma \circ \Sigma_1'; s_1)$ and $(\Sigma \circ \Sigma_2'; s_2)$ are node-isomorphic. Furthermore, Σ is certainly node-isomorphic to itself. We then apply Lemma 11 on $(\Sigma; \vec{s})$, $(\Sigma \circ \Sigma_1'; s_1)$ and $(\Sigma; \vec{s})$, $(\Sigma \circ \Sigma_2'; s_2)$, from which the result follows. □

In a similar way we obtain:

Corollary 13. *If $R \subseteq \mathcal{V}_p \times \mathcal{V}$ is a store-increasing semi-function and $(\Sigma_1; s_1)$ and $(\Sigma_2; s_2)$ are two values in $R(\Sigma; \vec{s})$, then $|s_1| = |s_2|$ and $(\Sigma_1; \langle s_1(j) \rangle, \vec{s})$ is node-isomorphic to $(\Sigma_2; \langle s_2(j) \rangle, \vec{s})$, for every $j \in [1, |s_1|]$.*

Proposition 14. *The relation defined by an expression in $QL(B)$ is a semi-function.*

Proof. The proof goes by induction on e.

- The cases where $e = x$, $e = a$, or $e = ()$ are trivial.
- If $e = \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$, then it follows from the induction hypothesis that e_1, e_2 and e_3 are all semi-functions. Therefore, if $(\Sigma_1; \langle \texttt{true} \rangle) \in e_1(\Sigma; \sigma)$, then *all* values in $e_1(\Sigma; \sigma)$ are of the form $(\Sigma_1'; \langle \texttt{true} \rangle)$, and if $(\Sigma_1; \langle \texttt{false} \rangle) \in e_1(\Sigma; \sigma)$, then *all* values in $e_1(\Sigma; \sigma)$ are of the form $(\Sigma_1'; \langle \texttt{false} \rangle)$. Consequently, if $e_1(\Sigma; \sigma)$ is defined, then either $e(\Sigma; \sigma) = e_2(\Sigma; \sigma)$ or $e(\Sigma; \sigma) = e_3(\Sigma; \sigma)$. Since e_2 and e_3 are semi-functions it follows that e is also a semi-function.
- If $e = \texttt{let}\ x := e_1\ \texttt{return}\ e_2$, then it follows from the induction hypothesis that e_1 and e_2 are semi-functions. Suppose that v and w are two values in $e(\Sigma; \sigma)$. Then there exists $(\Sigma_1; s_1) \in e_1(\Sigma; \sigma)$ such that $v \in e_2(\Sigma_1; x\colon s_1, \sigma)$ and there exists $(\Sigma_1'; s_1') \in e_1(\Sigma; \sigma)$ such that $w \in e_2(\Sigma_1'; x\colon s_1', \sigma)$. Since e_1 is a store-increasing semi-function it follows from Corollary 12 that there exists a node-renaming ρ such that $\rho(\Sigma_1; x\colon s_1, \sigma) = (\Sigma_1'; x\colon s_1', \sigma)$. Since e_1 is node-generic, we have
$$\rho(v) \in e_2(\rho(\Sigma_1; x\colon s_1, \sigma)) = e_2(\Sigma_1'; x\colon s_1', \sigma).$$
Since we also have $w \in e_2(\Sigma_1'; x\colon s_1', \sigma)$ and since e_2 is a semi-function there exists a node-renaming π such that $\pi(\rho(v)) = w$, from which the result follows.
- If $e = f(e_1, \ldots, e_p)$, then it follows from the induction hypothesis that $e_1, \ldots, e_p$ are all semi-functions. Suppose that v and w are two values in $e(\Sigma; \sigma)$. Then there exist $(\Sigma \circ \Sigma_j; s_j)$ and $(\Sigma \circ \Sigma_j'; s_j')$ in $e_j(\Sigma; \sigma)$ for every $j \in [1, p]$ such that the Σ_j are all pairwise disjoint, the Σ_j' are all pairwise disjoint, and
$$v \in f(\Sigma \circ \textstyle\bigcirc_{j=1}^{p} \Sigma_j; s_1, \ldots, s_p)$$
$$w \in f(\Sigma \circ \textstyle\bigcirc_{j=1}^{p} \Sigma_j'; s_1', \ldots, s_p').$$

Since every e_j is a semi-function we know that $(\Sigma \circ \Sigma_j; s_j)$ is node-isomorphic to $(\Sigma \circ \Sigma'_j; s_j)$, for every $j \in [1, p]$. By Lemma 11 it follows that there exists a node-renaming ρ such that

$$\rho(\Sigma \circ \bigcirc_{j=1}^{k} \Sigma_j; s_1, \ldots, s_k) = (\Sigma \circ \bigcirc_{j=1}^{k} \Sigma'_j; s'_1, \ldots, s'_k).$$

Since f is node-generic we hence have

$$\rho(v) \in f(\rho(\Sigma \circ \bigcirc_{j=1}^{k} \Sigma_j; s_1, \ldots, s_k)) = f(\Sigma \circ \bigcirc_{j=1}^{k} \Sigma'_j; s'_1, \ldots, s'_k).$$

Since we also have $w \in f(\Sigma \circ \bigcirc_{j=1}^{k} \Sigma'_j; s'_1, \ldots, s'_k)$ and since f is a semi-function, there exists a node-renaming π such that $\pi(\rho(v)) = w$, from which the result follows.

- If $e = \texttt{for}\ x\ \texttt{in}\ e_1\ \texttt{return}\ e_2$, then it follows from the induction hypothesis that e_1 and e_2 are semi-functions. Suppose that v and w are two values in $e(\Sigma; \sigma)$. Then there exists $(\Sigma_0; s) \in e_1(\Sigma; \sigma)$ and values $(\Sigma_0 \circ \Sigma_j; s_j) \in e_2(\Sigma_0; x\colon \langle s(j) \rangle, \sigma)$ for every $j \in [1, |s|]$ such that the Σ_j are all pairwise disjoint and

$$v = (\Sigma_0 \circ \bigcirc_{j=1}^{|s|} \Sigma_j; \bigcirc_{j=1}^{|s|} s_j).$$

Moreover, there exists $(\Sigma'_0; s') \in e_1(\Sigma; \sigma)$ and values $(\Sigma'_0 \circ \Sigma'_j; s'_j) \in e_2(\Sigma'_0; x\colon \langle s'(j) \rangle, \sigma)$ for every $j \in [1, |s'|]$ such that the Σ'_j are all pairwise disjoint and

$$w = (\Sigma'_0 \circ \bigcirc_{j=1}^{|s'|} \Sigma'_j; \bigcirc_{j=1}^{|s'|} s'_j).$$

Since e_1 is a store-increasing semi-function, it follows from Corollary 13 that $|s| = |s'|$ and that there exists a node-renaming ρ_j for every $j \in [1, |s|]$ such that

$$\rho_j(\Sigma_0; x\colon \langle s(j) \rangle, \sigma) = (\Sigma'_0; x\colon \langle s'(j) \rangle, \sigma). \tag{1}$$

Since e_2 is node-generic it follows that

$$\rho_j(\Sigma_0 \circ \Sigma_j; s_j) \in e_2(\rho_j(\Sigma_0; x\colon \langle s(j) \rangle, \sigma)) = e_2(\Sigma'_0; x\colon \langle s'(j) \rangle, \sigma).$$

Since also $(\Sigma'_0 \circ \Sigma'_j; s'_j) \in e_2(\Sigma'_0; x\colon \langle s'(j) \rangle, \sigma)$ and since e_2 is a semi-function there exists, for every $j \in [1, |s|]$, a node-renaming π_j such that $\pi_j(\rho_j(\Sigma_0 \circ \Sigma_j; s_j)) = (\Sigma'_0 \circ \Sigma'_j; s'_j)$. Hence, $(\Sigma_0 \circ \Sigma_j; s_j)$ is node-isomorphic to $(\Sigma'_0 \circ \Sigma'_j; s'_j)$ for every $j \in [1, |s|]$. As in addition, (1) implies that Σ_0 is node-isomorphic to Σ'_0, it follows from Lemma 11 that

$$(\Sigma_0 \circ \bigcirc_{j=1}^{|s|} \Sigma_j; s_1, \ldots, s_{|s|}) \equiv_{node} (\Sigma'_0 \circ \bigcirc_{j=1}^{|s|} \Sigma'_j; s'_1, \ldots, s'_{|s|}).$$

It is now easy to see that this implies

$$(\Sigma_0 \circ \bigcirc_{j=1}^{|s|} \Sigma_j; \bigcirc_{j=1}^{|s|} s_j) \equiv_{node} (\Sigma'_0 \circ \bigcirc_{j=1}^{|s|} \Sigma'_j; \bigcirc_{j=1}^{|s|} s'_j),$$

from which the result follows. □

Using the fact that every $e \in \mathrm{QL}(B)$ is a node-generic, store-increasing semi-function, an easy induction shows that e is also computable. Hence, e defines a base operation.

4 Well-definedness

The evaluation of an expression e on an input $(\Sigma; \sigma)$ may be *undefined*, i.e., potentially $e(\Sigma; \sigma) = \emptyset$. For example, the expression

$$\texttt{if } \mathit{eq}(\mathit{publisher}, \texttt{'ACM'}) \texttt{ then } \mathit{element}(\mathit{authors}, \mathit{title}) \texttt{ else } ()$$

returns the empty set when

1. *publisher* is the empty list (as the subexpression $\mathit{eq}(\mathit{publisher}, \texttt{'ACM'})$ then returns the empty list, on which the conditional test is undefined);
2. *publisher* is not a singleton atom and not the empty list (as the subexpression $\mathit{eq}(\mathit{publisher}, \texttt{'ACM'})$ is then undefined); or
3. *publisher* is the singleton atom $\langle \texttt{ACM} \rangle$ and *authors* is not a singleton atom (as the element constructor is then evaluated on a value-tuple of the form $(\Sigma; s, s')$ with s not a singleton atom, which is undefined).

Since the fact that e is undefined on $(\Sigma; \sigma)$ models the situation where an actual implementation would produce a runtime error, it is a natural question to ask whether we can decide, given an expression e and a (possibly infinite) set S of contexts on e, whether e is defined on every context in S. The answer to this problem clearly depends on both the set B of base operations and the class of context sets used as inputs. For the purpose of this paper we will focus on the class of context sets specified by bounded-depth regular expression types. Regular expression types are widely used in general-purpose programming languages manipulating tree-structured data, such as XDuce [18, 19, 20], ℂDuce [15], and XQuery [5, 13]. The bounded-depth restriction is motivated by the fact that most tree-structured data (such as for example found in XML documents [39]) in practice has nesting depth at most five or six, and that unbounded-depth nesting is hence often not needed.

Formally, a *type* is a term generated by the following grammar:

$$\begin{array}{rcl} \tau & ::= & \mathbf{atom} \mid \mathbf{text} \mid \mathbf{element}(a, \tau) \\ & \mid & \mathbf{empty} \mid \tau + \tau \mid \tau \circ \tau \mid \tau^* \end{array}$$

A type denotes a set of values, as defined in Figure 4. Here we denote trees by Θ. For ease of notation we will not distinguish between a type and the set of values it denotes. A *type assignment* Γ on an expression e is a function

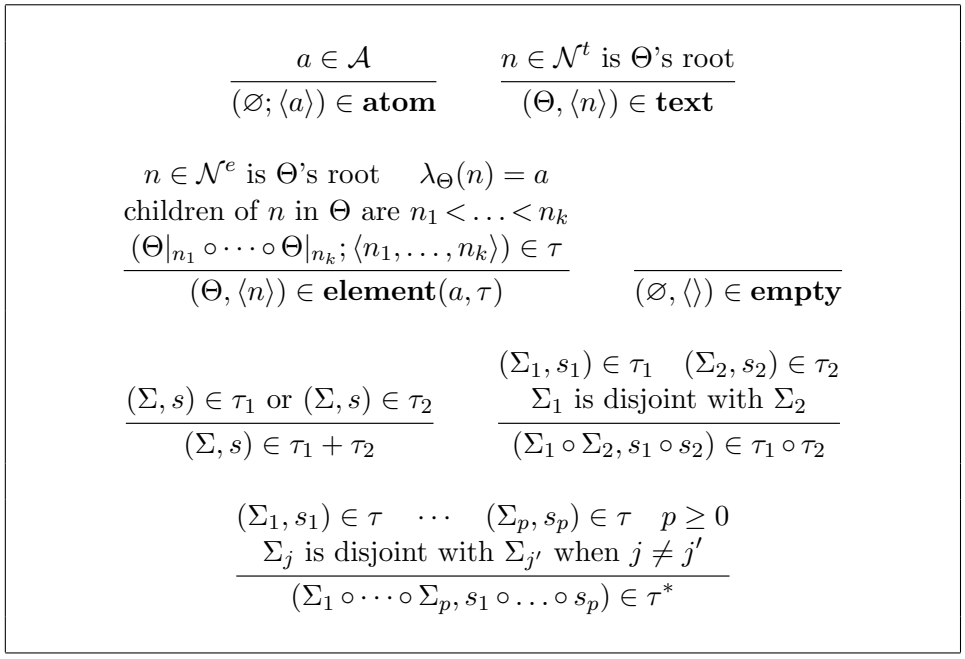


Figure 4: The denotation of types.

from a finite superset $\{x, \ldots, y\}$ of the free variables of e to types. A type assignment denotes the set of contexts

$$\{(\Sigma_x \circ \cdots \circ \Sigma_y; \sigma) \mid (\Sigma_z; \sigma(z)) \in \Gamma(z) \text{ for all } z \in \{x, \ldots, y\}\}.$$

Again we will not distinguish between a type assignment and the set of contexts it denotes.

Definition 15. Let B be a finite set of base operations. We say that $e \in \mathrm{QL}(B)$ is *well-defined* under a type assignment Γ on e if $e(\Sigma; \sigma) \neq \emptyset$ for every context $(\Sigma; \sigma) \in \Gamma$. The *well-definedness problem for QL*(B) consists of checking, for a given expression $e \in \mathrm{QL}(B)$ and a given type assignment Γ on e, whether e is well-defined under Γ.

It is not obvious that the well-definedness problem is decidable. Indeed, we will next identify several properties of B which can make the problem undecidable.

5 Satisfiability and the restriction to monotone base operations

Definition 16. Let B be a finite set of base operations. Let e be an expression in $\mathrm{QL}(B)$ and let Γ be a type assignment under which e is well-defined.

We say that e *is satisfiable under* Γ if there exists a context $(\Sigma; \sigma) \in \Gamma$ such that s is non-empty for every value $(\Sigma'; s) \in e(\Sigma; \sigma)$. The *satisfiability problem* for QL(B) consists of checking, given e and Γ, whether e is satisfiable under Γ.

Since every expression defines a semi-function by Proposition 6, s is non-empty for some $(\Sigma'; s) \in e(\Sigma; \sigma)$ if, and only if, *all* values in $e(\Sigma; \sigma)$ have a non-empty list. Hence, e is satisfiable under Γ if, and only if, there exists a context $(\Sigma; \sigma) \in \Gamma$ and a value $(\Sigma'; s)$ in $e(\Sigma; \sigma)$ such that s is non-empty.

The satisfiability problem is reducible to the well-definedness problem. Indeed, let e be an expression in QL(B) and let Γ be a type assignment under which e is well-defined. It is easy to see that e is satisfiable under Γ if, and only if, the expression

$$\texttt{for } x \texttt{ in } e \texttt{ return } (\texttt{if } () \texttt{ then } () \texttt{ else } ())$$

is not well-defined under Γ (as the subexpression `if () then () else ()` is always undefined). We have hence shown:

Proposition 17. *If the satisfiability problem for QL(B) is undecidable, then the well-definedness problem for QL(B) is also undecidable.*

The converse is not true however. Indeed, in Section 7.1 we will give a set of base operations B for which the well-definedness of QL(B) is undecidable, but the satisfiability problem of QL(B) is nevertheless decidable.

Unsurprisingly, there are QL(B) for which satisfiability is undecidable, as exemplified by the following proposition.

Proposition 18. *If B includes the base operations concat, children, eq, node-name, content, element, and empty, then QL(B) can simulate the relational algebra. Concretely, for every relational algebra expression ϕ over database schema S there exists an expression $e_\phi \in$ QL(B) and a type assignment Γ such that*

- *every database over S can be encoded as a context in Γ,*
- *e_ϕ is well-defined under Γ, and,*
- *e_ϕ evaluated on an encoding of database D equals an encoding of $\phi(D)$.*

Consequently, satisfiability for QL(B) is undecidable, as it is already undecidable for the relational algebra.

Proof. We use a well-known, straightforward, one-to-many encoding of relations as values. For example, the relation R in Figure 5(a) can be encoded as the value $(\Sigma; s)$ in Figure 5(b). Specifically, we encode each tuple t in R as a tree in Σ. The root node n of this tree is labeled by some arbitrarily fixed atom T. Furthermore, n has one element child node m_A for every attribute

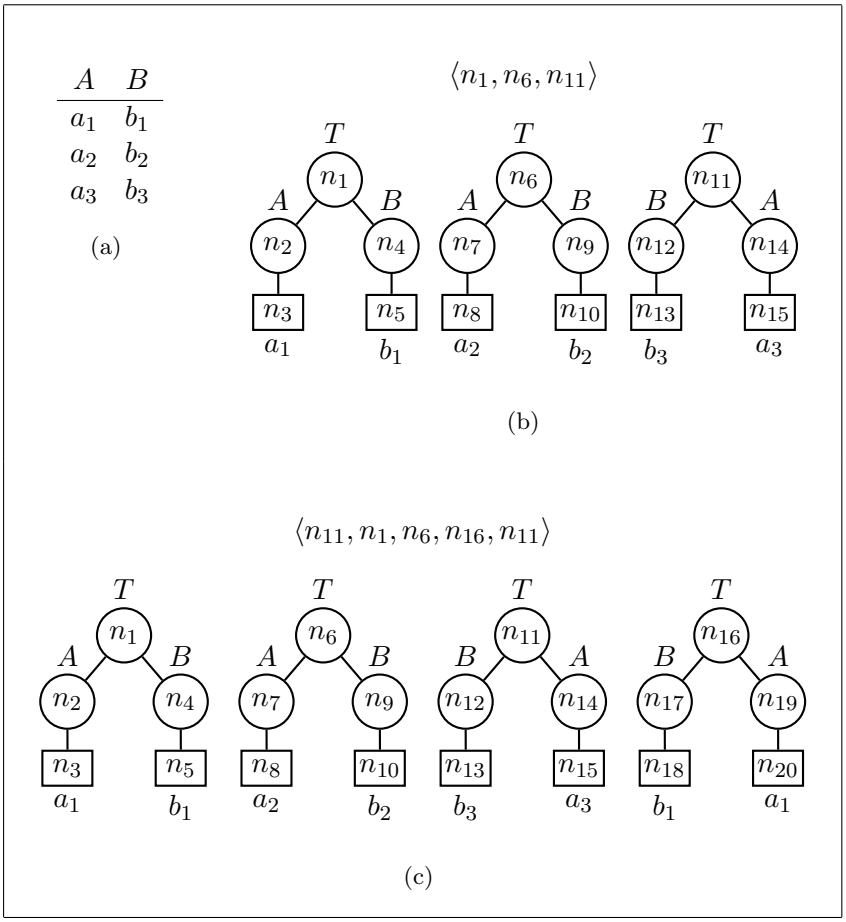


Figure 5: Encoding relations as values.

name A in the schema of R, and this child is labeled by A. The node m_A itself has exactly one child, which is a text node labeled by the value of t on A. The whole relation is then encoded by the value $(\Sigma; s)$ such that

1. for each tuple $t \in R$ the root node of a tree encoding t is mentioned in s, and

2. each node mentioned in s is the root node of an encoding of a tuple in R.

The order in which these root nodes are mentioned in s does not matter and there can be multiple nodes whose trees encode the same tuple. As such, the value in Figure 5(c) is also a valid encoding of the relation in Figure 5(a).

Let S be a database schema. A database D over S can then be encoded as a context $(\Sigma; \sigma)$ such that $(\Sigma; \sigma(r))$ is an encoding of the relation assigned to relation name r by D, for every relation name r in S. Let Γ be the type

assignment on the relation names in S such that

$$\Gamma(r) := \mathbf{element}(T, \mathbf{element}(A_1, \mathbf{text}) \circ \cdots \circ \mathbf{element}(A_k, \mathbf{text}))^*$$

where $\{A_1, \ldots, A_k\}$ is the relation schema assigned to r by S. It is easy to see that for every database D over S there exists a context in Γ which encodes it and that every context in Γ encodes a database over S.

We will now show how to construct, for every relational algebra expression ϕ over S, an expression $e_\phi \in \mathrm{QL}(B)$ such that $e_\phi(\Sigma; \sigma)$ is an encoding of $\phi(D)$ whenever $(\Sigma; \sigma)$ is an encoding of a database D over S. Note that in particular, $e_\phi(\Sigma; \sigma)$ is hence well-defined on Γ. In order to simplify presentation, we will allow to bind multiple variables in one for loop. We will also allow boolean combinations in the condition of an if test. Both features can clearly be simulated in $\mathrm{QL}(B)$. The construction is by induction on ϕ:

- If ϕ is the relation name r, then $e_\phi = r$.
- If $\phi = \sigma_{A_1=A_2}(\psi)$, then e_ϕ is defined as follows:

```
for t in e_ψ return
  for x_1, x_2 in children(t) return
    if eq(node-name(x_1), A_1)
      and eq(node-name(x_2), A_2)
      and eq(content(children(x_1)), content(children(x_2)))
    then t else ()
```

- If $\phi = \pi_{A_1,\ldots,A_k}(\psi)$, then e_ϕ is defined as follows:

```
for t in e_ψ return
  element(T,
    for x in children(t) return
      if eq(node-name(x_1), A_1)
        or ...
        or eq(node-name(x_k), A_k)
      then x else ()
  )
```

- If $\phi = \rho_{B \leftarrow A}(\psi)$, then e_ϕ is defined as follows:

```
for t in e_ψ return
  element(T,
    for x in children(t) return
      if eq(node-name(x), A)
      then element(B, children(x)) else x
  )
```

- If $\phi = \psi_1 \times \psi_2$, then e_ϕ is defined as follows:

```
for t1 in e_ψ1 return
  for t2 in e_ψ2 return
    element(T, concat(children(t1), children(t2)))
```

- If $\phi = \psi_1 \cup \psi_2$, then $e_\phi = concat(e_{\psi_1}, e_{\psi_2})$.

- If $\phi = \psi_1 - \psi_2$, then we note that ψ_1 and ψ_2 have the same output schema $\{A_1, \ldots, A_k\}$. We define e_ϕ as follows:

```
for t1 in e_ψ1 return
  let z:= for t2 in e_ψ2 return
          if same-tuple(t1,t2) then t2 else ()
  return
    if empty(z) then t1 else ()
```

  Here, *same-tuple*(t_1,t_2) is an abbreviation for the following expression, which returns `true` if t_1 and t_2 encode the same tuple over $\{A_1, \ldots, A_k\}$, and `false` otherwise.

```
let z :=
  for x1,...,xk in children(t1) return
  for y1,...,yk in children(t2) return
    if eq(node-name(x1), A1)
      and ...
      and eq(node-name(xk), Ak)
      and eq(node-name(y1), A1)
      and ...
      and eq(node-name(yk), Ak)
      and eq(content(children(x1)), content(children(y1)))
      and ...
      and eq(content(children(xk)), content(children(yk)))
    then t1 else ()
return
  if empty(z) then false else true
```

  In e_ϕ we hence compute, for each node t_1 returned by e_{ψ_1}, the nodes returned by e_{ψ_2} which encode the same tuple as t_1. If there are no such encodings, then t_1 is returned (as it's encoding is hence not in the result of ψ_2), otherwise it is filtered out.

It is easy to verify that e_ϕ indeed returns an encoding of $\phi(D)$ when evaluated on an encoding of D. In particular, e_ϕ is hence defined on such encodings, as desired. □

Corollary 19. *If B includes the base operations concat, children, eq, node-name, concat, element, and empty, then the well-definedness problem for $QL(B)$ is undecidable.*

We note that the fact that XQuery's atomic value comparison and element constructor are more complex than the *eq* and *element* base operations we use above has no effect on the undecidability of the well-definedness problem. Indeed, it is easily verified that the simulation in the proof of Proposition 18 still works if we replace *eq* and *element* by their XQuery counterparts (whose semantics was given in Examples 2 and 3).

5.1 Monotone base operations

Note that satisfiability for the *monotone* fragment of the relational algebra (i.e., the relational algebra without difference) is trivially decidable. Indeed, it is easy to see that every relational algebra expression in this fragment is decidable. [6] In the hope of finding QL(B) for which the well-definedness is decidable, it is therefore worthwhile to restrict ourselves to those base operations which are in a sense also "monotone". For this purpose, we adapt the notion of (unordered) complex object containment [4] to (ordered) values.

Containment of stores Intuitively, a store Σ is contained in a store Σ' if Σ can be obtained by removing nodes from Σ' in such a way that if we remove a node, we also remove all of its descendants. Formally, Σ is contained in Σ' when every component of Σ is a subset of the corresponding component of Σ', i.e., $V_\Sigma \subseteq V_{\Sigma'}$, $E_\Sigma \subseteq E_{\Sigma'}$, $\lambda_\Sigma \subseteq \lambda_{\Sigma'}$, $<_\Sigma \subseteq <_{\Sigma'}$, $\prec_\Sigma \subseteq \prec_{\Sigma'}$, and $\mathit{roots}(\Sigma) \subseteq \mathit{roots}(\Sigma')$.

As an example of store containment, consider the three stores depicted in Figure 6. It is easy to verify that Σ_2 is contained in Σ_3. Store Σ_1 is not contained in Σ_2 however, as n_1 is a root in Σ_1, but not in Σ_2. It can similarly be seen that Σ_1 is also not contained in Σ_3.

We note that store-containment is closely related to the notion of *simulation* [7]: there exists a simulation from Σ to Π which respects document order and relates all roots of Σ to roots of Π if, and only if, there exists a store Σ', node-isomorphic to Π, such that Σ is contained in Σ'.

Containment of lists Intuitively, a list s is contained in a list s' if s can be obtained from s' by deleting items in it. Formally, s is contained in s' if there exists a strictly increasing function $h : [1, |s|] \to [1, |s'|]$ such that

[6] Here we are referring to the standard version of the relational algebra where selection only tests equality between attributes. When other selection predicates are allowed, expressions in the positive-existential fragment of the relational algebra need not be satisfiable.

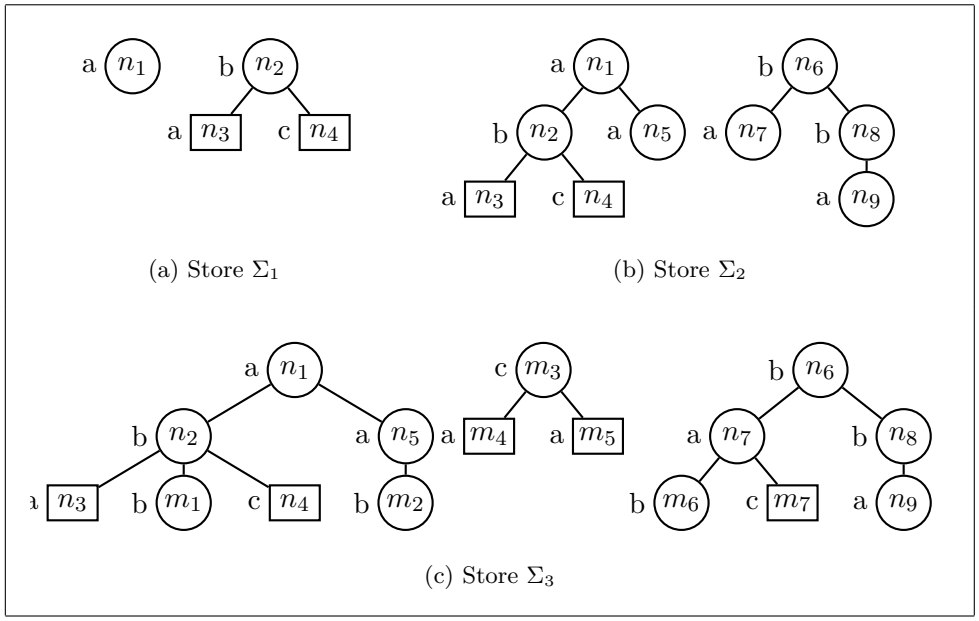


Figure 6: Containment of stores. Store Σ_1 is not contained in Σ_2 or in Σ_3. Store Σ_2 is contained in Σ_3.

$s(j) = s'(h(j))$ for every $j \in [1, |s|]$. Such a function h is called a *witness* of the fact that s is contained in s'.

As an example, consider the lists $s = \langle a, b, c\rangle$ and $s' = \langle a, b, b, a, c\rangle$. Then s is contained s', as witnessed by the function $h : [1, 3] \to [1, 5]$ with $h(1) = 1$, $h(2) = 2$, and $h(3) = 5$. In contrast, when $s = \langle a, a, b, c\rangle$ then s is not contained in s', as we cannot obtain s from s' simply by deleting items in s'.

Containment of value-tuples Finally, containment extends naturally to value-tuples: a value-tuple $(\Sigma; s_1, \ldots, s_p)$ is contained in a value-tuple $(\Sigma'; s'_1, \ldots, s'_p)$ if Σ is contained in Σ' and every s_j is contained in the corresponding s'_j. We are now ready to introduce the notion of a monotone base operation.

Monotonicity A set of value-tuples S is contained in a set of value-tuples S' if for every $v' \in S'$ there exists $v \in S$ such that v is contained in v'. In what follows we will denote the containment relation on stores, lists, value-tuples and sets of value-tuples by $\sqsubseteq$. A relation $R \subseteq \mathcal{V}_p \times \mathcal{V}$ is *monotone* if for all v and w in $\mathcal{V}_p$ with $R(v) \neq \emptyset$, $R(w) \neq \emptyset$, and $v \sqsubseteq w$, we have $R(v) \sqsubseteq R(w)$.

Example 20. Let us give some examples of monotone base operations:

- The concatenation operator *concat* is clearly monotone.

- The children axis is also monotone. Indeed, let $(\Sigma; s) \sqsubseteq (\Sigma'; s')$ and suppose that *children* is defined on $(\Sigma; s)$ and $(\Sigma'; s')$. Since $s \sqsubseteq s'$ we know that every node mentioned in s is also mentioned in s'. Furthermore, since $\Sigma \sqsubseteq \Sigma'$ we know the set of children of a node n in Σ is a subset of the set of children of n in Σ'. Finally, since $\Sigma \sqsubseteq \Sigma'$ we know that if n precedes m in document order in Σ, it also precedes m in document order in Σ'. Hence, if $(\Sigma; t)$ is the result of *children* on (Σ, s) and $(\Sigma'; t')$ is the result of *children* on $(\Sigma'; s')$, then it is easily seen that $t \sqsubseteq t'$. Therefore,

$$\mathit{children}(\Sigma; s) = \{(\Sigma; t)\} \sqsubseteq \{(\Sigma'; t')\} = \mathit{children}(\Sigma'; s').$$

- The atomic value comparison *eq* is another example of a monotone base operation. Indeed, let $(\Sigma; s, s') \sqsubseteq (\Pi; t, t')$ and suppose that *eq* is defined on $(\Sigma; s, s')$ and $(\Pi; t, t')$. There are two possibilities:

  1. If s or s' is the empty list, then *eq* relates $(\Sigma; s, s')$ only to $(\Sigma; \langle\rangle)$. Monotonicity is immediate, since $\langle\rangle$ is contained in every other list.
  2. Otherwise, we know that $s = \langle a \rangle$ and $s' = \langle b \rangle$ with a and b atoms since *eq* is defined on $(\Sigma; s, s')$. Since $s \sqsubseteq t$ and $s' \sqsubseteq t'$, it follows that t and t' cannot be empty. Since *eq* is also defined on $(\Pi; t, t')$, it hence follows that $t = \langle c \rangle$ and $t' = \langle d \rangle$ with c and d atoms. Since $\langle a \rangle \sqsubseteq \langle c \rangle$ and $\langle b \rangle \sqsubseteq \langle d \rangle$, it follows that $a = c$ and $b = d$. Hence,

$$\mathit{eq}(\Sigma; s, s') = \{(\Sigma; \langle a = b \rangle)\} \sqsubseteq \{(\Pi; \langle c = d \rangle\} = \mathit{eq}(\Sigma; t, t').$$

- Finally, the element constructor *element* is also a monotone base operation. Indeed, suppose that $v \sqsubseteq w$ and that *element* is defined on v and w. Then v and w must be of the form:

$$v = (\Sigma; \langle a \rangle, \langle n_1, \ldots, n_k \rangle)$$
$$w = (\Sigma'; \langle a \rangle, \langle n'_1, \ldots, n'_l \rangle).$$

  Let $(\Sigma' \circ \Theta'; \langle m' \rangle)$ be a value in $\mathit{element}(w)$. We show that there exists a value in $\mathit{element}(v)$ which is contained in $(\Sigma' \circ \Theta'; \langle m' \rangle)$. By definition of *element*, we know that the root element node m' of the tree Θ' is labeled by a, that m' has exactly l children, and that if m'_j is the j-th child of m' (in sibling order), then there exists a node-renaming ρ_j such that $\rho_j(\Sigma'|_{n'_j}) = \Theta'|_{m'_j}$. Let h be a witness of $\langle n_1, \ldots, n_k \rangle \sqsubseteq \langle n'_1, \ldots, n'_l \rangle$. Since $\Sigma \sqsubseteq \Sigma'$ it follows that $\Sigma|_{n_j} \sqsubseteq \Sigma'|_{n'_{h(j)}}$ for every $j \in [1, k]$. Hence, we also have

$$\rho_{h(j)}(\Sigma|_{n_j}) \sqsubseteq \rho_{h(j)}(\Sigma'|_{n'_{h(j)}}) = \Theta'|_{m'_{h(j)}}.$$

Then let Θ be the tree whose root element node m' is labeled by a such that m' has k children and that the j-th child of m' (in document order) is $\rho_{h(j)}(\Sigma|_{n_j})$. It is clear that $(\Sigma \circ \Theta; \langle m' \rangle) \in \mathit{element}(v)$. Moreover, it is easy to see that $(\Theta; \langle m' \rangle) \sqsubseteq (\Theta'; \langle m' \rangle)$. It follows that $(\Sigma \circ \Theta; \langle m' \rangle) \sqsubseteq (\Sigma' \circ \Theta'; \langle m' \rangle)$, as desired. □

Using similar reasonings as the ones employed in Example 20 we obtain:

Proposition 21. *The base operations concat, children, descendant, parent, ancestor, preceding-sibling, following-sibling, eq, is, $\ll$, is-element, is-text, is-atom, node-name, content, element, and text are all monotone.*

Note that if our restriction to monotone base operations is to have any chance of leading to QL(B) for which well-definedness is decidable, *empty* must be non-monotone. Indeed, all other base operations mentioned in Corollary 19 are monotone by Proposition 21. Fortunately:

Example 22. The emptiness test is not monotone. Indeed, let Σ be a store. The emptiness test relates $(\Sigma; \langle\rangle)$ only to $(\Sigma; \langle \texttt{true} \rangle)$ and $(\Sigma; \langle a \rangle)$ only to $(\Sigma; \langle \texttt{false} \rangle)$. However, $\langle \texttt{true} \rangle \not\sqsubseteq \langle \texttt{false} \rangle$, although $(\Sigma; \langle\rangle) \sqsubseteq (\Sigma; \langle a \rangle)$. □

5.2 The impact of automatic coercions

We note that the atomization function *data*, depending on the concrete interpretation of the abstract function *fold* which maps lists of text nodes to atoms, is potentially not monotone. Indeed, suppose for example that *fold*, when viewed as a base operation, relates $(\Sigma; \langle n \rangle)$ with n a text node labeled by a to $(\Sigma; \langle a \rangle)$ and relates $(\Sigma; \langle n_1, \ldots, n_k \rangle)$ with $k \geq 2$ and $n_1, \ldots, n_k$ text nodes labeled by a to $(\Sigma; \langle b \rangle)$ for some $b \neq a$.[7] Then clearly *fold* (and hence *data*) is not monotone. Note that with this interpretation of *fold* we can simulate the emptiness test in QL(B). Indeed, $\mathit{empty}(e)$ is simulated by

```
let y := (for x in e return text(a)) return
let z := concat(text(a), y) return
  eq(data(element(c,z)), a)
```

Here, c is an arbitrary atom. In y we first compute a list of text nodes, all labeled by a. Note that y is empty if, and only, if e is empty. Hence, z contains a single text node labeled by a if, and only if, e is empty. The atomization of a newly created element node with z as children hence returns a if, and only if, e is empty.

Note that, since *fold* is also used to merge adjacent text nodes into a single text node in the *merge-text* base operation, it follows that hence

[7] The actual interpretation of *fold* in XQuery (i.e., string concatenation of the text nodes' labels) has this kind of behavior: concatenating a non-empty string k times does not produce the string itself.

merge-text is also not monotone. Also note that with this interpretation of *merge-text* we can again simulate the emptiness test in QL(B). Indeed, *empty*(e) is simulated by

```
let y := (for x in e return text(a)) return
let z := concat(text(a), y) return
let w := merge-text(element(c,z)) return
  eq(content(children(w)), a)
```

Indeed, using a similar reasoning as above, it is easy to see that the single text node child of w is labeled by a if, and only if, e is empty.

As such, we obtain that QL(*concat*, *children*, *eq*, *node-name*, *content*, *element*, *text*, *data*) and QL(*concat*, *children*, *eq*, *node-name*, *element*, *text*, *merge-text*, *content*) are at least as expressive as QL(*concat*, *children*, *eq*, *node-name*, *element*, *empty*). It follows by Propositions 17 and 18 that their well-definedness problem is hence also undecidable. This reasoning clearly illustrates that automatic coercions, such as the ones performed by atomization and text node merging, are not harmless with regard to deciding well-definedness.

5.3 Monotone expressions

In this section we show that if B is a set of monotone base operations, then monotonicity transfers to all expressions in QL(B). Hereto, we first state the following lemma's.

Lemma 23. *Let $R \subseteq \mathcal{V}_p \times \mathcal{V}$ be a monotone base operation and let $(\Sigma; \vec{s})$ and $(\Sigma'; \vec{s'})$ be value-tuples of arity p such that $(\Sigma; \vec{s}) \sqsubseteq (\Sigma'; \vec{s'})$ and $R(\Sigma; \vec{s}) \neq \emptyset$. For every $(\Sigma' \circ \Sigma'_1; s'_1) \in R(\Sigma'; \vec{s'})$ there exists $(\Sigma \circ \Sigma_1; s_1) \sqsubseteq (\Sigma' \circ \Sigma'_1, s'_1)$ in $R(\Sigma; \vec{s})$ such that $\Sigma_1 \sqsubseteq \Sigma'_1$.*

Proof. Every store can be written as a concatenation of trees. Let $\Theta'_1, \ldots, \Theta'_k$ be the non-empty trees such that $\Sigma' = \Theta'_1 \circ \cdots \circ \Theta'_k$. Since $\Sigma \sqsubseteq \Sigma'$, we can write Σ as a concatenation of trees $\Theta_1 \circ \cdots \circ \Theta_k$ such that $\Theta_j \sqsubseteq \Theta'_j$ for every $j \in [1, k]$, where if Σ does not contain any node in Θ'_j, we take Θ_j to be the empty tree. Let $\Delta_1, \ldots, \Delta_k$ be the trees such that, for every $j \in [1, k]$, $\Delta_j = \Theta_j$ if Θ_j is non-empty, and $\Delta_j = \Theta'_j$ otherwise. In particular, $\Delta_j = \Theta_j$ whenever $\vec{s}$ mentions a node in Θ_j. Since R is reachable-only and since $R(\Theta_1 \circ \cdots \circ \Theta_k; \vec{s})$ is defined, it is easy to see that $R(\Delta_1 \circ \cdots \circ \Delta_k; \vec{s})$ is also defined. Moreover, by construction we have $\Delta_j \sqsubseteq \Theta'_j$ for every $j \in [1, k]$. Hence, $(\Delta_1 \circ \cdots \circ \Delta_k; \vec{s}) \sqsubseteq (\Theta'_1 \circ \cdots \circ \Theta'_k; \vec{s'})$. Since R is a monotone base operation, there exists $(\Delta_1 \circ \cdots \circ \Delta_k \circ \Sigma_1; s_1) \in R(\Delta_1 \circ \cdots \circ \Delta_k; \vec{s})$ such that

$$(\Delta_1 \circ \cdots \circ \Delta_k \circ \Sigma_1; s_1) \sqsubseteq (\Theta'_1 \circ \cdots \circ \Theta'_k \circ \Sigma'_1; s'_1).$$

Hence, $\Sigma_1 \sqsubseteq \Theta'_1 \circ \cdots \circ \Theta'_k \circ \Sigma'_1$. Assume, for the purpose of contradiction, that Σ_1 has some node m in common with Θ'_j, for some $j \in [1, k]$. Let

n be the root node of m in Σ_1. In particular there exists a path from n to m in Σ_1. Since $\Sigma_1 \sqsubseteq \Theta'_1 \circ \cdots \circ \Theta'_k \circ \Sigma'_1$, n must also a root node in $\Theta'_1 \circ \cdots \circ \Theta'_k \circ \Sigma'_1$. By definition of $\sqsubseteq$ there must also exist a path from n to m in $\Theta'_1 \circ \cdots \circ \Theta'_k \circ \Sigma'_1$. By definition of concatenation however, there can be no path in $\Theta'_1 \circ \cdots \circ \Theta'_k \circ \Sigma'_1$ connecting a node not in Θ'_j to a node in Θ'_j. Hence, n must be the root node of Θ'_j. As Θ'_j is non-empty, Δ_j is also non-empty by construction. Let n' be the root node of Δ_j. Since $\Delta_j \sqsubseteq \Theta'_j$, n' must also be a root node in Θ'_j. Since trees have at most one root node, $n = n'$. Hence, n is a node in Δ_j. This is a contradiction however, as $(\Delta_1 \circ \cdots \circ \Delta_k \circ \Sigma_1; s_1)$ is a value and Δ_j should hence be disjoint with Σ_1. It follows that Σ_1 is disjoint with Θ'_j for every $j \in [1, k]$ and hence that $\Sigma_1 \sqsubseteq \Sigma'_1$.

Finally, since R is reachable-only, since $\Delta_j = \Theta_j$ whenever $\vec{s}$ mentions a node in Δ_j, and since $(\Delta_1 \circ \cdots \circ \Delta_k \circ \Sigma_1; s_1) \in R(\Delta_1 \circ \cdots \circ \Delta_k; \vec{s})$ we have $(\Theta_1 \circ \cdots \circ \Theta_k \circ \Sigma_1; s_1) \in R(\Theta_1 \circ \cdots \circ \Theta_k; \vec{s})$, as desired. □

Lemma 24. *If $R \subseteq \mathcal{V}_p \times \mathcal{V}$ is a base operation which is defined on v and if w is node-isomorphic to v, then R is also defined on w.*

Proof. Since R is defined on v there exists $u \in R(v)$. Since v is node-isomorphic to w there exists a node-renaming ρ such that $\rho(v) = w$. Since R is node-generic we have $\rho(u) \in R(\rho(v)) = R(w)$, from which the result follows. □

We are now ready to prove:

Proposition 25. *Let B be a finite set of monotone base operations. Every expression e in $QL(B)$ defines a monotone relation.*

Proof. Let e be an expression in QL(B). Let $(\Sigma; \sigma)$ and $(\Sigma'; \sigma')$ be two contexts on e such that $e(\Sigma; \sigma) \neq \emptyset$, $e(\Sigma'; \sigma') \neq \emptyset$, and $(\Sigma; \sigma) \sqsubseteq (\Sigma'; \sigma')$.[8] Let $w \in e(\Sigma'; \sigma')$. We will prove by induction on e that there exists $v \in e(\Sigma; \sigma)$ such that $v \sqsubseteq w$. Throughout the induction we will use the fact that expressions define base operations (Proposition 6) and that if an expression is defined on an input, it is also defined on all the node-isomorphic copies of this input (Lemma 24).

- If $e = x$, $e = a$ or $e = ()$, then the result follows immediately.
- If $e = \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$, then there must exist $(\Sigma'_1; \langle b \rangle) \in e_1(\Sigma'; \sigma')$ with b either `true` or `false` such that $w \in e_2(\Sigma'; \sigma')$ if $b = \texttt{true}$ and $w \in e_3(\Sigma'; \sigma')$ otherwise. Suppose that $b = \texttt{true}$. Since $e(\Sigma; \sigma)$ is defined, $e_1(\Sigma; \sigma)$ must also be defined. There hence exists

[8] Here we extend the containment relation to contexts in the obvious way: if σ and σ' are environments with the same domain $\{x, \ldots, y\}$, then $(\Sigma; \sigma) \sqsubseteq (\Sigma'; \sigma')$ if $(\Sigma; \sigma(x), \ldots, \sigma(y)) \sqsubseteq (\Sigma'; \sigma'(x), \ldots, \sigma'(y))$.

a value $(\Sigma_1; s_1) \sqsubseteq (\Sigma_1'; \langle \texttt{true} \rangle)$ in $e_1(\Sigma; \sigma)$ by the induction hypothesis. Then s_1 is either $\langle\rangle$ or $\langle \texttt{true} \rangle$. Note however that if $s_1 = \langle\rangle$, then all values in $e_1(\Sigma; \sigma)$ are of the form $(\Sigma_1; \langle\rangle)$ since e_1 is a semi-function. Hence, $e(\Sigma; \sigma)$ would be undefined. Therefore, $s_1 = \langle \texttt{true} \rangle$. Since e_1 is a semi-function it follows that all values in $e_1(\Sigma; \sigma)$ are of the form $(\Sigma_1; \langle \texttt{true} \rangle)$. Hence, $e_2(\Sigma; \sigma) = e(\Sigma; \sigma) \neq \emptyset$. Since $w \in e_2(\Sigma'; \sigma')$ there then exists $v \in e_2(\Sigma; \sigma)$ with $v \sqsubseteq w$ by the induction hypothesis. Since $e_2(\Sigma; \sigma) = e(\Sigma, \sigma)$ we hence have $v \in e(\Sigma; \sigma)$ with $v \sqsubseteq w$, as desired. If $b = \texttt{false}$, then the reasoning is similar.

- If $e = \texttt{let}\ x := e_1\ \texttt{return}\ e_2$, then there must exist $(\Sigma_1'; s_1') \in e_1(\Sigma'; \sigma')$ such that $w \in e_2(\Sigma_1'; x\colon s_1', \sigma')$. Moreover, since $e(\Sigma; \sigma)$ is defined there must exist a $(\Pi_1; t_1) \in e_1(\Sigma; \sigma)$ such that $e_2(\Pi_1; x\colon t_1, \sigma)$ is defined. Note that in particular, $e_1(\Sigma; \sigma)$ is defined. Hence there exists a value $(\Sigma_1; s_1) \sqsubseteq (\Sigma_1'; s_1')$ in $e_1(\Sigma; \sigma)$ by the induction hypothesis. Since e_1 is a store-increasing semi-function, it follows from Corollary 12 that $(\Sigma_1; x\colon s_1, \sigma)$ is node-isomorphic to $(\Pi_1; x\colon t_1, \sigma)$. Since e_2 is a node-generic and since $e_2(\Pi_1; x\colon t_1, \sigma)$ is defined, it follows that $e_2(\Sigma_1; x\colon s_1, \sigma)$ is also defined. Since also $(\Sigma_1; x\colon s_1, \sigma) \sqsubseteq (\Sigma_1'; x\colon s_1', \sigma')$, it follows by the induction hypothesis that there exists $v \in e_2(\Sigma_1; x\colon s_1, \sigma)$ with $v \sqsubseteq w$. The result then follows since $v \in e(\Sigma; \sigma)$.

- If $e = f(e_1, \ldots, e_p)$, then there exist $(\Sigma' \circ \Sigma_j'; s_j') \in e_j(\Sigma'; \sigma')$ for every $j \in [1, p]$ such that the Σ_j' are all pairwise disjoint and
$$w \in f(\Sigma' \circ \textstyle\bigcirc_{j=1}^{p} \Sigma_j'; s_1', \ldots, s_p').$$
Moreover, since $e(\Sigma; \sigma)$ is defined there must also exist $(\Sigma \circ \Pi_j; t_j) \in e_j(\Sigma; \sigma)$ for every $j \in [1, p]$ such that the Π_j are pairwise disjoint and
$$f(\Sigma \circ \textstyle\bigcirc_{j=1}^{p} \Pi_j; t_1, \ldots, t_p) \neq \emptyset.$$
Note that in particular, $e_j(\Sigma; \sigma)$ is defined for every $j \in [1, p]$. Since every e_j defines a monotone base operation by the induction hypothesis, it hence follows from Lemma 23 that there exist $(\Sigma \circ \Sigma_j; s_j) \in e_j(\Sigma; s)$ for every $j \in [1, p]$ such that $(\Sigma \circ \Sigma_j; s_j) \sqsubseteq (\Sigma' \circ \Sigma_j'; s_j')$ and $\Sigma_j \sqsubseteq \Sigma_j'$. Since the Σ_j' are all pairwise disjoint and $\Sigma_j \sqsubseteq \Sigma_j'$, it follows that the Σ_j are also pairwise disjoint. Moreover, $(\Sigma \circ \Sigma_j; s_j)$ is node-isomorphic to $(\Sigma \circ \Pi_j; t_j)$ for every $j \in [1, p]$ since every e_j is a semi-function. By Lemma 11 we hence obtain that
$$(\Sigma \circ \textstyle\bigcirc_{j=1}^{p} \Sigma_j; s_1, \ldots, s_p) \equiv_{node} (\Sigma \circ \bigcirc_{j=1}^{p} \Pi_j; t_1, \ldots, t_p).$$
Since f is a node-generic and since $f(\Sigma \circ \bigcirc_{j=1}^{p} \Pi_j; t_1, \ldots, t_p)$ is defined, it follows that $f(\Sigma \circ \bigcirc_{j=1}^{p} \Sigma_j; s_1, \ldots, s_p)$ is also defined. Moreover, since $\Sigma \sqsubseteq \Sigma'$, $\Sigma_j \sqsubseteq \Sigma_j'$ and $s_j \sqsubseteq s_j'$ for every $j \in [1, p]$ we have
$$(\Sigma \circ \textstyle\bigcirc_{j=1}^{p} \Sigma_j; s_1, \ldots, s_p) \sqsubseteq (\Sigma' \circ \bigcirc_{j=1}^{p} \Sigma_j'; s_1', \ldots, s_p').$$

Since f is a monotone base operation, there hence exists

$$v \in f(\Sigma \circ \bigcirc_{j=1}^{p} \Sigma_j; s_1, \ldots, s_p)$$

such that $v \sqsubseteq w$. The result then follows since $v \in e(\Sigma; \sigma)$.

- If $e = \texttt{for}\ x\ \texttt{in}\ e_1\ \texttt{return}\ e_2$, then there exists $(\Sigma'_0; s') \in e_1(\Sigma'; \sigma')$ and values $(\Sigma'_0 \circ \Sigma'_j; s'_j) \in e_2(\Sigma'_0; x\colon \langle s'(j)\rangle, \sigma')$ for every $j \in [1, |s'|]$ such that the Σ'_j are all pairwise disjoint and

$$w = (\Sigma'_0 \circ \bigcirc_{j=1}^{|s'|} \Sigma'_j; \bigcirc_{j=1}^{|s'|} s'_j).$$

Moreover, since $e(\Sigma; \sigma)$ is defined there exists $(\Pi_0; t) \in e_1(\Sigma; \sigma)$ such that $e_2(\Pi_0; x\colon \langle t(j)\rangle, \sigma) \neq \emptyset$ for every $j \in [1, |t|]$. Note that in particular, $e_1(\Sigma; \sigma)$ is defined. Since $(\Sigma; \sigma) \sqsubseteq (\Sigma'; \sigma')$ there hence exists $(\Sigma_0; s) \sqsubseteq (\Sigma'_0; s')$ in $e_1(\Sigma; \sigma)$ by the induction hypothesis. Since e_1 is a store-increasing semi-function, it follows from Corollary 13 that $|s| = |t|$ and that $(\Sigma_0; x\colon \langle s(j)\rangle, \sigma)$ is node-isomorphic to $(\Pi_0; x\colon \langle t(j)\rangle, \sigma)$, for every $j \in [1, |t|]$. Since e_2 is node-generic and since $e_2(\Pi_0; x\colon \langle t(j)\rangle, \sigma)$ is defined for every $j \in [1, |t|]$, it follows that $e_2(\Sigma_0; x\colon \langle s(j)\rangle, \sigma)$ is also defined for every $j \in [1, |s|]$. Let h be a witness of $s \sqsubseteq s'$. It is easy to see that for every $j \in [1, |s|]$ we have

$$(\Sigma_0; x\colon \langle s(j)\rangle, \sigma) \sqsubseteq (\Sigma'_0; x\colon \langle s'(h(j))\rangle, \sigma).$$

Since e_2 is a monotone base operation by the induction hypothesis, it hence follows from Lemma 23 that there exist

$$(\Sigma_0 \circ \Sigma_j; s_j) \in e_2(\Sigma_0; x\colon \langle s(j)\rangle, \sigma)$$

for every $j \in [1, |s|]$ such that $(\Sigma_0 \circ \Sigma_j; s_j) \sqsubseteq (\Sigma'_0 \circ \Sigma'_j; s'_j)$ and $\Sigma_j \sqsubseteq \Sigma'_j$. Since the Σ'_j are all pairwise disjoint and $\Sigma_j \sqsubseteq \Sigma'_j$, it follows that the Σ_j are also pairwise disjoint. It is easy to see that hence

$$(\Sigma_0 \circ \bigcirc_{j=1}^{|s|} \Sigma_j; \bigcirc_{j=1}^{|s|} s_j) \sqsubseteq (\Sigma'_0 \circ \bigcirc_{j=1}^{|s'|} \Sigma'_j; \bigcirc_{j=1}^{|s'|} s'_j).$$

The result then follows since the left-hand side is in $e(\Sigma; \sigma)$. □

6 Interpretation of atoms and the restriction to generic base operations

Another potential source of undecidability is the interpretation of atoms by base operations. Indeed, suppose that B includes base operations $+$ and $\times$ which interpret the atoms as integers and simulate the addition respectively multiplication on them. That is, $+$ and $\times$ relate $(\Sigma; \langle k\rangle, \langle l\rangle)$ to $(\Sigma; \langle k +$

$l\rangle)$ respectively $(\Sigma; \langle k \times l\rangle)$. Note that with this definition, $+$ and $\times$ are monotone. It is easy to see that for every polynomial $P(x_1, \ldots, x_k)$ with integer coefficients there exists an expression e_P with free variables $x_1, \ldots, x_k$ that simulates P. Hence, the expression

$$\texttt{if } eq(e_P, 0) \texttt{ then } (\texttt{if } () \texttt{ then } () \texttt{ else } ()) \texttt{ else } ()$$

is well-defined under the type assignment which maps every x_j to **atom** if, and only if, the Diophantine equation $P(x_1, \ldots, x_k) = 0$ has no integer solution. Since we now have a reduction from Hilbert's undecidable tenth problem [27], well-definedness for QL(B) is undecidable.

Generic base operations We will therefore restrict ourselves to base operations which do not interpret the atoms, except for the booleans `true` and `false`. Formally, we require that all base operations R are *generic*: for every renaming ρ it must hold that

$$w \in R(v) \Leftrightarrow \rho(w) \in R(\rho(v)).$$

It is easy to see that for example *concat*, *children* and *element* are generic base operations. In fact:

Proposition 26. *The base operations concat, children, descendant, parent, ancestor, preceding-sibling, following-sibling, eq, is, $\ll$, is-element, is-text, is-atom, node-name, content, element, text, and empty are all generic.*[9]

Note that hence genericity alone is not powerful enough to prevent the construction of QL(*concat*, *children*, *eq*, *node-name*, *content*, *elem*, *empty*) for which well-definedness is undecidable.

Semi-generic expressions In contrast to monotonicity, genericity does not transfer literally from base operations to expressions. Indeed, it is obvious that expressions can always interpret the constants they mention. An easy induction shows that expressions cannot interpret more than those constants however:

Proposition 27. *If B is a finite set of generic base operations, then for every expression $e \in QL(B)$ and every renaming ρ which is the identity on the atoms mentioned in e it holds that $w \in e(v) \Leftrightarrow \rho(w) \in e(\rho(v))$.*

We say that e is *semi-generic* in this case.

[9] Remember that renamings are the identity on the booleans. This explains why for example *eq* can be generic.

7 Non-local behavior and the restriction to local and locally-undefined base operations

In this section we will show that, even if B is a set of monotone and generic base operations, well-definedness for QL(B) need not be decidable. In order to illustrate this, we first introduce the following problem.

Definition 28. Let e_1 and e_2 be two expressions with the same set of free variables, and let Γ be a type assignment under which e_1 and e_2 are well-defined. We say that *the list-width of e_1 is less than the list-width of e_2 under* Γ, denoted by $|e_1| \leq_\Gamma |e_2|$ if for all $v \in \Gamma$, all $(\Sigma_1; s_1) \in e_1(v)$ and all $(\Sigma_2; s_2) \in e_2(v)$ it holds that $|s_1| \leq |s_2|$. The *list-width problem* consists of deciding, given e_1, e_2, and Γ whether $|e_1| \leq_\Gamma |e_2|$.

Lemma 29. *The list-width problem for QL(concat) is undecidable.*

Proof. Our proof is based on the reduction used to show that containment of unions of conjunctive queries on bags is undecidable [21]. Let $P_1(x_1, \ldots, x_p)$ and $P_2(x_1, \ldots, x_p)$ be two polynomials in p variables, with natural number coefficients and without constant terms. It was shown by Ioannidis and Ramakrishnan [21] (p. 317) that there is no decision procedure checking that $P_1(x_1, \ldots, x_p) \leq P_2(x_1, \ldots, x_p)$ for all natural number assignments to $x_1, \ldots, x_p$.

We will encode natural numbers k as lists of width k. Note that under this encoding we can simulate addition by concatenation and multiplication by the for loop. Indeed, let $(\Sigma; \sigma)$ be a context such that $|\sigma(x)| = k$ and $|\sigma(y)| = l$. The list of the value returned by the expression $concat(x, y)$ on $(\Sigma; \sigma)$ then has width $k + l$. Moreover, the list of the value returned by the expression `for` z `in` x `return` y on $(\Sigma; \sigma)$ has with $k \times l$. Hence, we can construct an expression e_1 with free variables $x_1, \ldots, x_k$ which simulates $P_1(x_1, \ldots, x_p)$ in the sense that

$$(\Sigma_1; s_1) \in e_1(\Sigma; \sigma) \quad \Rightarrow \quad P_1(|\sigma(x_1)|, \ldots, |\sigma(x_p)|) = |s_1|.$$

We can similarly construct an expression e_2 which simulates $P_2(x_1, \ldots, x_p)$. Let Γ be a type assignment on e_1 and e_2 such that $\Gamma(x_j) = \mathbf{atom}^*$ for all x_j. Since *concat* is defined on every input, it is easy to see that e_1 and e_2 are also defined on every input. Hence e_1 and e_2 are well-defined under Γ. Finally, as Γ contains encodings of all possible natural number assignments to $x_1, \ldots, x_p$ it follows that it is undecidable to check whether $|e_1| \leq_\Gamma |e_2|$. □

As a side note, we state the following corollary which is interesting in its own right.

Corollary 30. *The containment problem for QL(concat) is undecidable: it is undecidable to check, given two expressions e_1 and e_2 with the same set of free variables and a type assignment Γ under which e_1 and e_2 are well-defined, whether $e_1(v) \sqsubseteq e_2(v)$, for all $v \in \Gamma$.*[10]

Proof. Let e_1 and e_2 be two expressions in QL(*concat*) with the same set of free variables and let Γ be a type assignment under which e_1 and e_2 are well-defined. Since we cannot create new nodes in QL(*concat*), it follows that if $(\Sigma_1; s_1) \in e_1(\Sigma; \sigma)$ and $(\Sigma_2; s_2) \in e_2(\Sigma; \sigma)$, then $\Sigma_1 = \Sigma = \Sigma_2$. Hence $|e_1| \leq_\Gamma |e_2|$ if, and only if, for all $v \in \Gamma$ we have

$$(\texttt{for } x \texttt{ in } e_1 \texttt{ return } a)(v) \quad \sqsubseteq \quad (\texttt{for } x \texttt{ in } e_2 \texttt{ return } a)(v).$$

As we now have a reduction from the list-width problem for QL(*concat*) which is undecidable by Lemma 29, it follows that the containment problem is also undecidable. □

7.1 Non-local undefinedness behavior

The undefinedness behavior of base operations such as *children*, *eq*, and *element* is quite simple: the input list contains an atom where it should only contain nodes; one of the input lists contains two or more items; or the first input list is not a singleton atom respectively. Base operations with more complex undefinedness behavior are problematic with regard to well-definedness checking, as the following proposition shows.

Proposition 31. *Let smaller-width be the binary base operation which relates $(\Sigma; s, s')$ to $(\Sigma; \langle\rangle)$ when $|s| \leq |s'|$ and which is undefined otherwise. The well-definedness problem for QL(concat, smaller-width) is undecidable.*

Proof. Let e_1 and e_2 be expressions in QL(*concat*) with the same set of free variables and let Γ be a type assignment under which e_1 and e_2 are well-defined. It is easy to see that *smaller-width*(e_1, e_2) is well-defined under Γ if, and only if, $|e_1| \leq_\Gamma |e_2|$. Since we now have a reduction from the list-width problem for QL(*concat*) which is undecidable by Lemma 29, it follows that well-definedness for QL(*concat*, *smaller-width*) is also undecidable. □

Note, however, that *concat* and *smaller-width* are both monotone and generic. Hence, monotonicity and genericity alone do not imply decidability. In fact, we will prove in Section 8.1:

Proposition 32. *The satisfiability problem for QL(concat, smaller-width) is decidable.*

[10] We note that, in contrast, the corresponding problem in a set-based data model is decidable [12].

Hence, decidability of the satisfiability problem is not sufficient to obtain decidability of the well-definedness problem.

The core difficulty with well-definedness in QL(*concat*, *smaller-width*) is that *smaller-width* can switch arbitrarily from defined to undefined and back again when the input "grows" according to the containment relation. Indeed, *smaller-width* is defined on $(\Sigma; \langle\rangle, \langle\rangle)$; undefined on $(\Sigma; \langle a\rangle, \langle\rangle)$; and defined again on $(\Sigma; \langle a\rangle, \langle b\rangle)$. As such, *smaller-width* is non-monotone in its undefinedness behavior: if *smaller-width*(v) is undefined and $v \sqsubseteq w$, then *smaller-width*(w) is not necessarily undefined. In our companion paper [37] we study well-definedness for the Nested Relational Calculus (NRC), a set-based query language which is well-known from the complex object data model [1, 8, 38]. Specifically, we show that the positive-existential fragment of the NRC *is* monotone in its undefinedness behavior, and we use this property to show decidability of well-definedness for that fragment.

In order to obtain QL(B) for which well-definedness is decidable, we could hence restrict ourselves to base operations which are monotone in their undefinedness behavior. In that case, however, we would disallow base operations such as *is-element*, *is-text*, *element*, and *text* which are undefined when their (first) argument is empty, but are defined when this is a singleton. As we would like to obtain a language with these operators for which well-definedness is decidable, we will use another, looser restriction.

Specifically, we note that *smaller-width*'s undefinedness on a certain input depends on the whole input, and not on a local part of it. We will therefore restrict ourselves to base operations which are undefined on an input due to a local reason. We make this notion precise as follows.

Requirements A *requirement* $\mathbf{w}$ is a tuple $(V; P_1, \ldots, P_p)$ where V is a set of nodes and the P_j are sets of non-zero natural numbers. Let $w = (\Sigma; s_1, \ldots, s_p)$ be a value-tuple. We say that $\mathbf{w}$ is a *requirement on* w when V is a subset of the nodes in Σ and P_j is a subset of $[1, |s_j|]$, for every $j \in [1, p]$. A value-tuple $(\Sigma'; s'_1, \ldots, s'_p)$ *satisfies* $\mathbf{w}$ on w if $(\Sigma'; s'_1, \ldots, s'_p) \sqsubseteq w$, V is a subset of the nodes in Σ', and for every $j \in [1, p]$ there exists a witness h_j for $s'_j \sqsubseteq s_j$ such that $P_j \subseteq rng(h_j)$. Note that w itself trivially satisfies $\mathbf{w}$ on w. We will denote the set of all value-tuples which satisfy $\mathbf{w}$ on w by $[\mathbf{w}, w]$.

As an example, let Σ_2 and Σ_3 be the stores depicted in Figure 6(b) respectively Figure 6(c). Then $\mathbf{w} = (\{n_7\}; \{2\})$ is clearly a requirement on $w = (\Sigma_3; \langle n_1, n_4, a, n_4\rangle)$. Furthermore, the value $(\Sigma_2; \langle n_4\rangle)$ satisfies $\mathbf{w}$ on w. Indeed, it is clear that $(\Sigma_2; \langle n_4\rangle) \sqsubseteq (\Sigma_3; \langle n_1, n_4, a, n_4\rangle)$ and that $\{n_7\}$ is a subset of the nodes in Σ_2. Moreover, the function which maps 1 to 2 is a witness of $\langle n_4\rangle \sqsubseteq \langle n_1, n_4, a, n_4\rangle$ whose range obviously includes $\{2\}$. The value $(\varnothing; \langle a, n_4\rangle)$ does not satisfy $\mathbf{w}$ on w however. Indeed, $\{n_7\}$ is not a subset of the nodes in $\varnothing$ and there exists no witness h of $\langle a, n_4\rangle \sqsubseteq \langle n_1, n_4, a, n_4\rangle$ for which $\{2\} \subseteq rng(h)$.

The following lemma establishes some basic properties of requirements.

Lemma 33. *Let* $(V; P_1, \ldots, P_p)$ *be a requirement on* $(\Sigma; s_1, \ldots, s_p)$*, let* $(\Sigma'; s'_1, \ldots, s'_p)$ *be a value-tuple which satisfies this requirement, and let* $j \in [1, p]$*. Then* $|P_j| \leq |s'|$ *and* $\{s_j(i) \mid i \in P_j\} \subseteq rng(s'_j)$*.*

Proof. Trivial. □

Undefinedness reasons Let $R \subseteq \mathcal{V}_p \times \mathcal{V}$ and $w \in \mathcal{V}_p$ such that $R(w) = \emptyset$. A requirement $\mathbf{w}$ on w is a *reason* why $R(w) = \emptyset$ if $R(v) = \emptyset$ for every $v \in [\mathbf{w}, w]$. Intuitively, a reason why $R(w) = \emptyset$ describes a "part" of w which causes R to be undefined on w.

For example, $\mathbf{w} = (\emptyset; \{2\})$ is a reason why *children* is undefined on $w = (\Sigma; \langle n, a, m, a\rangle)$. Indeed, if $(\Sigma'; s') \in [\mathbf{w}, w]$, then it follows by Lemma 33 that $\{a\} \subseteq rng(s')$. Since s' hence mentions an atom, *children* is also undefined on $(\Sigma'; s')$. Likewise, $\mathbf{w}' = (\emptyset; \{1\}, \{1, 2\})$ is a reason why *eq* is undefined on $w' = (\varnothing; \langle a, c\rangle, \langle a, b, c\rangle)$. Indeed, if $(\Sigma'; s'_1, s'_2) \in [\mathbf{w}', w']$, then it follows by Lemma 33 that $|s'_1| \geq 1$ and $|s'_2| \geq 2$. Hence, *eq* is undefined on $(\Sigma'; s'_1, s'_2)$.

Reasons are not necessarily unique. For example, $(\emptyset; \{1, 2\}, \{2\})$ is another reason why *eq* is undefined on $(\varnothing; \langle a, c\rangle, \langle a, b, c\rangle)$. Furthermore, there always exists a reason why R is undefined on $w = (\Sigma; s_1, \ldots, s_p)$. Indeed, it suffices to take $\mathbf{w} = (V; P_1, \ldots, P_p)$ with V the set of nodes in Σ and $P_j = [1, |s_j|]$, for every $j \in [1, p]$.

Locally-undefinedness The *size* of a requirement $\mathbf{w} = (V; P_1, \ldots, P_p)$, denoted by $|\mathbf{w}|$, is the maximum of $|V|$, $|P_1|$, …, $|P_p|$. We say that R is *locally-undefined* if there exists a constant k such that for every v on which R is undefined there exists a reason why $R(v) = \emptyset$ of size at most k. We call k a *witness* of the fact that R is locally-undefined. Intuitively, a locally-undefined base operation cannot base its decision to be undefined on a certain input on the whole input, but only on a small part of it.

Example 34. Let us give some examples of locally-undefined base operations.

- The concatenation operator *concat*, the atomization function *data*, and the emptiness test *empty* are always defined. Hence, they are also locally-undefined.
- The children axis is locally-undefined with witness 1. Indeed, suppose that *children* is undefined on $w = (\Sigma; s)$. Then there exists $j \in [1, s]$ such that $s(j)$ is a atom. Let $\mathbf{w}$ be the requirement $(\emptyset; \{j\})$ on $(\Sigma; s)$. It is clear that $\mathbf{w}$ has size 1. Furthermore, let $(\Sigma'; s') \in [\mathbf{w}, w]$. It follows by Lemma 33 that $s(j) \in rng(s')$. As s' thus mentions an atom, it follows that *children* is undefined on $(\Sigma'; s')$. Hence, $\mathbf{w}$ is a reason why $children(w) = \emptyset$.

- The atomic value comparison *eq* is locally-undefined with witness 2. Indeed, suppose that $eq(w)$ is undefined. We discern two cases.

  1. If $w = (\Sigma; \langle i_1 \rangle, \langle i_2 \rangle)$ with i_1 or i_2 a node, then let $\mathbf{w}$ be the requirement $(\emptyset; \{1\}, \{1\})$ on w. It is clear that $\mathbf{w}$ has size 1. Furthermore let $(\Sigma'; s'_1, s'_2) \in [\mathbf{w}, w]$. It follows by Lemma 33 that $i_1 \in rng(s'_1)$ and $i_2 \in rng(s'_2)$. Since s'_1 or s'_2 are then non-empty and since one of them mentions a node, it follows that *eq* is undefined on $(\Sigma'; s'_1, s'_2)$. Hence, $\mathbf{w}$ is a reason why $eq(w) = \emptyset$.
  2. Otherwise, w must be of the form $(\Sigma; s_1, s_2)$ with s_1 and s_2 non-empty and one of s_1 or s_2 containing two or more items. We assume without loss of generality that $|s_1| \geq 2$, the other case is similar. Let $\mathbf{w}$ be the requirement $(\emptyset; \{1, 2\}, \{1\})$. It is clear that $\mathbf{w}$ has size 2. Furthermore, let $(\Sigma'; s'_1, s'_2) \in [\mathbf{w}, w]$. It follows by Lemma 33 that $|s'_1| \geq 2$ and $|s'_2| \geq 1$. Hence, *eq* is undefined on $(\Sigma'; s'_1, s'_2)$. As such $\mathbf{w}$ is a reason why $eq(w) = \emptyset$.

- The kind test *is-element* is locally-undefined with witness 2. Indeed, suppose that *is-element*(w) is undefined. We discern three cases.

  1. If $w = (\Sigma; \langle\rangle)$, then let $\mathbf{w}$ be the requirement $(\emptyset; \emptyset)$ on w. It is clear that $\mathbf{w}$ has size 0. Let $(\Sigma'; s') \in [\mathbf{w}, w]$. Since in particular $s' \sqsubseteq \langle\rangle$, it follows that *is-element* is undefined on $(\Sigma'; s')$. Hence, $\mathbf{w}$ is a reason why *is-element*$(w) = \emptyset$.
  2. If $w = (\Sigma; \langle a \rangle)$ with a an atom, then let $\mathbf{w}$ be the requirement $(\emptyset; \{1\})$ on w. It is clear that $\mathbf{w}$ has size 1. Furthermore, let $(\Sigma'; s') \in [\mathbf{w}, w]$. It follows by Lemma 33 that $a \in rng(s')$. As s' hence mentions an atom, it follows that *is-element* is undefined on $(\Sigma'; s')$. Hence, $\mathbf{w}$ is a reason why *is-element*$(w) = \emptyset$.
  3. Otherwise, w must be of the form $(\Sigma; s)$ with $|s| \geq 2$. Let $\mathbf{w}$ be the requirement $(\emptyset; \{1, 2\})$ on w. It is clear that $\mathbf{w}$ has size 2. Furthermore, let $(\Sigma'; s') \in [\mathbf{w}, w]$. It follows by Lemma 33 that $|s'| \geq 2$. Hence, *is-element* is undefined on $(\Sigma'; s')$. As such, $\mathbf{w}$ is a reason why $eq(w) = \emptyset$.

- As a final example, let R be the base operation that relates $(\Sigma; s)$ to $(\Sigma; \langle\rangle)$ if s is a sequence of items in which no element node has an a-labeled element child. If some element node in s does have an a-labeled element child, then $R(\Sigma; s)$ is undefined. Note that R is not some artificially contrived example. Indeed, R can be defined by the following expression.

```
for y in x return
  if is-element(y) then
```

```
  for z in children(y) return
    if is-element(z) then
      if eq(node-name(z), a) then
        if () then () else ()
      else ()
    else ()
else ()
```

We claim that R is locally-undefined with witness 1. Indeed, suppose that R is undefined on $w = (\Sigma; s)$. Then there exists a position $j \in [1, |s|]$ such that $s(j)$ is an element node which has an a-labeled element child node n. Let $\mathbf{w}$ be the requirement $(\{n\}, \{j\})$ on w. It is clear that $\mathbf{w}$ has size 1. Furthermore, let $(\Sigma'; s') \in [\mathbf{w}, w]$. Since $\{n\}$ is a subset of the nodes in Σ' and since $\Sigma' \sqsubseteq \Sigma$, it follows that n is an a-labeled element child of $s(j)$ in Σ'. Furthermore, $s(j) \in rng(s')$ by Lemma 33. As s' thus mentions an element node with an a-labeled element child, it follows that R is undefined on $(\Sigma'; s')$. Hence, $\mathbf{w}$ is a reason why $R(\Sigma; s) = \emptyset$. □

Using similar reasonings as the ones employed in Example 34 we obtain:

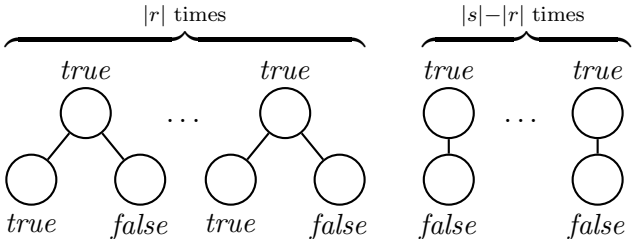
Proposition 35. *The base operations concat, children, descendant, parent, ancestor, preceding-sibling, following-sibling, data, eq, is, ≪, is-element, is-text, node-name, content, element, merge-text, text, empty, +, and × are all locally-undefined.*

Note that hence locally-undefinedness alone is not powerful enough to prevent the construction of QL(*concat*, *children*, *eq*, *node-name*, *content*, *element*, *empty*) and QL($+, \times$), for which well-definedness is undecidable.

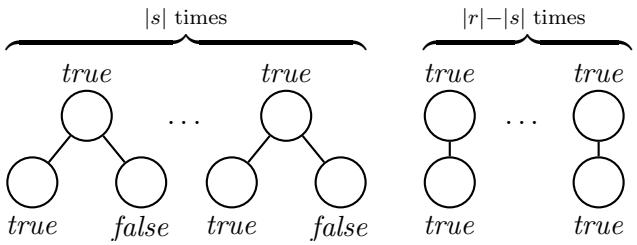
7.2 Non-local behavior

Unfortunately, locally-undefinedness does not transfer from base operations to expressions, a fact which can also cause trouble as we show next.

Let *zip* be a binary base operation which relates $(\Sigma; r, s)$ to $(\Sigma \circ \Pi; t)$ where Π has the form



if $|r| \leq |s|$ and is otherwise of the form



In both cases t is the list of Π's root nodes in document order. Note that Π always has width $\max(|r|, |s|)$. Intuitively, every natural number i between 1 and $\max(|r|, |s|)$ gets represented as a root node, which has a `true`-labeled child if $i \leq |r|$ and has a `false`-labeled child if $i \leq |s|$. Note that *zip* is defined on every input and is hence locally-undefined.

Now let *has-false* be the base operation which relates $(\Sigma; \langle n \rangle)$ to $(\Sigma; \langle\rangle)$ if n has a `false`-labeled child, and which is undefined otherwise. Since the undefinedness behavior of *has-false* is equal to the undefinedness behavior of *is-element*, which we already showed to be locally-undefined in Example 34, it follows that *has-false* is also locally-undefined.

Although *zip* and *has-false* are hence both locally-undefined, there are expressions in QL(*zip*, *has-false*) which are not locally-undefined, as the following lemma shows.

Lemma 36. *The expression*

$$\texttt{for } x \texttt{ in } \mathit{zip}(y, z) \texttt{ return } \mathit{has\text{-}false}(x)$$

is not locally-undefined.

Proof. Let us denote the expression above by e. Suppose, for the purpose of contradiction, that there does exist a natural number k such that for all contexts v on e for which $e(v) = \emptyset$, there exists a reason why $e(v) = \emptyset$ of size at most k. Then let $(\Sigma; y\colon s_1, z\colon s_2)$ be a context on e such that $|s_1| = k+1$ and $|s_2| = k$. Clearly, $e(\Sigma; y\colon s_1, z\colon s_2) = \emptyset$. Hence, there exists a reason $\mathbf{w} := (V; y\colon P_1, z\colon P_2)$ why this is so of size at most k. Since P_1 has at most k elements, there must exist a list $s_1' \sqsubseteq s_1$ of width k for which there exists a witness h of $s_1' \sqsubseteq s_1$ such that $P_1 \subseteq \mathit{rng}(h)$. Then clearly

$$(\Sigma; y\colon s_1', z\colon s_2) \in [\mathbf{w}, (\Sigma; y\colon s_1, z\colon s_2)].$$

Since $|s_1'| = |s_2|$ it follows however that $e(\Sigma; y\colon s_1', z\colon s_2) \neq \emptyset$, which contradicts the fact that $\mathbf{w}$ is a reason why e is undefined on $(\Sigma; y\colon s_1, z\colon s_2)$. □

In fact, expressions as in Lemma 36 are quite problematic with regard to well-definedness checking. Indeed, let e_1 and e_2 be expressions with the same set of free variables and let Γ be a type assignment under which e_1 and e_2 are well-defined. Then $|e_1| \leq_\Gamma |e_2|$ if, and only if,

$$\texttt{for } x \texttt{ in } \mathit{zip}(e_1, e_2) \texttt{ return } \mathit{has\text{-}false}(x)$$

is well-defined under Γ. As we now have a reduction from the list-width problem to well-definedness, it follows from Lemma 29 that

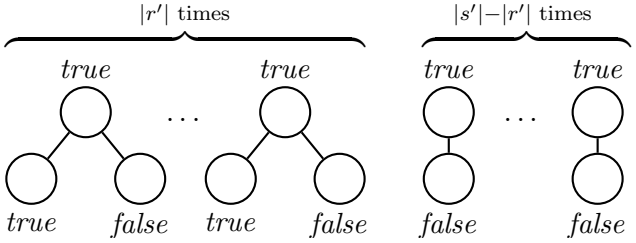
Proposition 37. *Well-definedness for QL(concat, zip, has-false) is undecidable.*

This undecidability is not due to the fact that our set of base operations contains a non-monotone or non-generic base operation. Indeed, we already know that *concat* is monotone and generic from Propositions 21 and 26. Similarly, it is quite easy to see that both *zip* and *has-false* are generic.[11] Since *has-false*$(\Sigma; s)$ when defined always returns $(\Sigma; \langle\rangle)$ it follows that *has-false* is also monotone. Finally, we show that *zip* is also monotone.

Lemma 38. *The base operation zip is monotone.*

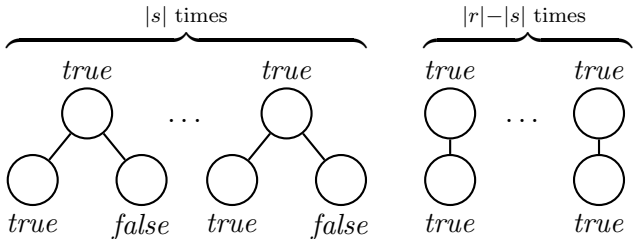
Proof. Suppose that *zip* is defined on $(\Sigma; r, s)$ and $(\Sigma'; r', s')$ and that $(\Sigma; r, s)$ is contained in $(\Sigma'; r', s')$. Let $(\Sigma' \circ \Pi'; t') \in \mathit{zip}(\Sigma'; r', s')$. We will show that we can always find $(\Sigma \circ \Pi; t) \in \mathit{zip}(\Sigma; r, s)$ such that $\Pi \sqsubseteq \Pi'$. Since by definition of *zip* we know that t is the list of all of the root nodes in Π in document order, and that t' is the list of all of the root nodes in Π's in document order, we then also have $t \sqsubseteq t'$. Hence $(\Sigma \circ \Pi; t) \sqsubseteq (\Sigma' \circ \Pi'; t')$, i.e., *zip* is monotone.

We only treat the case where $|r'| \leq |s'|$, the other case is similar. By definition of *zip* we know that Π' is of the form



We observe the following cases.

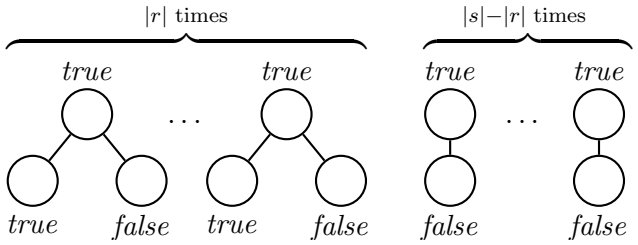
- If $|r| > |s|$, then for every $(\Sigma \circ \Pi; t) \in \mathit{zip}(\Sigma; r, s)$ we know that Π is of the form



[11] Remember that generic base operations can interpret `true` and `false`, which explains why *zip* and *has-false* can be generic. The use of `true` and `false` in these base operations is done solely for simplicity of exposition however. Indeed, *zip* and *has-false* could just as easily have taken extra atoms as input and used those atoms instead of `true` and `false`.

In particular we know that Π consists of $|r|$ trees. Furthermore, $|r| \leq |r'|$ since $r \sqsubseteq r'$. It is then easy to see that there exists at least one $(\Sigma \circ \Pi; t) \in \mathit{zip}(\Sigma; r, s)$ for which Π is contained in the first $|r'|$ trees of Π'. Hence, $\Pi \sqsubseteq \Pi'$, as desired.

- If $|r| \leq |s|$, then for every $(\Sigma \circ \Pi; t) \in \mathit{zip}(\Sigma; r, s)$ we know that Π is of the form



In particular we know that Π consists of $|s|$ trees. Furthermore, $|r| \leq |r'|$ and $|s| \leq |s'|$ since $r \sqsubseteq r'$ and $s \sqsubseteq s'$. If $|s| - |r| \leq |s'| - |r'|$, then it is easy to see that there exists at least one $(\Sigma \circ \Pi; t) \in \mathit{zip}(\Sigma; r, s)$ for which the first $|r|$ trees of Π are contained in the first $|r|$ trees of Π' and the other $|s| - |r|$ trees of Π are contained in the last $|s| - |r|$ trees of Π'. Hence, $\Pi \sqsubseteq \Pi'$, as desired. If on the other hand $|s| - |r| > |s'| - |r'|$, then

$$|r| + (|s| - |r|) - (|s'| - |r'|) = |s| - |s'| + |r'| \leq |r'|$$

Hence there exists at least one $(\Sigma \circ \Pi; t) \in \mathit{zip}(\Sigma; r, s)$ for which the first $|r| + (|s| - |r|) - (|s'| - |r'|)$ trees of Π are contained in the first $|r'|$ trees of Π' and the other $(|s'| - |r'|)$ trees of Π are contained in the last $|s'| - |r'|$ trees of Π'. In both cases we hence have $\Pi \sqsubseteq \Pi'$, as desired. □

Hence our restriction to monotone, generic, and locally-undefined base operations does not prevent the definition of QL(*concat*, *zip*, *has-false*) for which well-definedness is undecidable. The core difficulty here is that *zip* is non-local in the sense that the presence of a tree without `false`-labeled child in the output depends on the whole input, and not on a local part of it. We will therefore restrict ourselves to base operations where every part of the output depends only on a local part of the input. We make this notion precise as follows.

Parts Let $R \subseteq \mathcal{V}_p \times \mathcal{V}$ be a base operation and let v be an input on which R is defined. If $w \in R(v)$ and if $\mathbf{w}$ is a requirement on w, then we say that the set $[\mathbf{w}, w]$ is a *part* of $R(v)$, denoted by $[\mathbf{w}, w] \triangleleft R(v)$.

The set $[\mathbf{w}, w]$ intuitively describes a property of values. For example, consider a value $w = (\Sigma; s)$ where s mentions a node n in Σ without a-labeled child but with a b-labeled child m. Let $\mathbf{w} = (\{m\}, \{j\})$ be the requirement on w where $s(j) = n$. The set $[\mathbf{w}, w]$ then contains all values $(\Sigma'; s') \sqsubseteq w$ such that s' mentions n; n does not have an a-labeled child in Σ'; and m is a b-labeled child of n in Σ'. Indeed, $n \in \mathit{rng}(s)$ by Lemma 33; n does not have an a-labeled child in Σ' since $\Sigma' \sqsubseteq \Sigma$; and m is a b-labeled child of n in Σ' since $\Sigma' \sqsubseteq \Sigma$ and since $\{m\}$ is a subset of the nodes in Σ'. The fact that $[\mathbf{w}, w]$ is a part of $R(v)$ hence registers the fact every value in $R(v)$ has a node in its list without an a-labeled child, but with a b-labeled child (as R is a semi-function).

Output reasons A requirement $\mathbf{v}$ on v is a *reason* why $[\mathbf{w}, w] \triangleleft R(v)$ if for every $v' \in [\mathbf{v}, v]$ on which R is defined, $[\mathbf{w}, w] \cap R(v') \neq \emptyset$.

Intuitively, the fact that $[\mathbf{w}, w] \cap R(v') \neq \emptyset$ implies that the values in $R(v')$ also satisfy the property described by $[\mathbf{w}, w]$. For our earlier example, this implies that the values in $R(v')$ mention a node in their list without an a-labeled child, but with a b-labeled child. Since this is true for every v' which satisfies $\mathbf{v}$ on v, we can say that $\mathbf{v}$ is a "reason" why $R(v)$ has the property described by $[\mathbf{w}, w]$.

Example 39. The requirement $(\emptyset; \{1\})$ is a reason why

$$[(\emptyset; \emptyset), (\varnothing; \langle \texttt{false} \rangle)] \triangleleft \mathit{empty}(\varnothing; \langle a, n \rangle).$$

Indeed, let $(\Sigma'; s') \in [(\emptyset; \{1\}), (\varnothing; \langle a, n \rangle)]$. By Lemma 33 it follows that $|s'| \geq 1$. Hence, s' is non-empty, and thus $\mathit{empty}(\Sigma'; s') = \{(\Sigma'; \langle \texttt{false} \rangle)\}$. It is easy to see that hence $[(\emptyset; \emptyset), (\varnothing; \langle \texttt{false} \rangle)] \cap \mathit{empty}(\Sigma'; s') \neq \emptyset$. □

Locality We say that R is *local* if there exists a computable increasing function c mapping natural numbers to natural numbers such that for every input v and every part $[\mathbf{w}, w]$ of $R(v)$ there exists a reason $\mathbf{v}$ why $[\mathbf{w}, w] \triangleleft R(v)$ of size at most $c(|\mathbf{w}|)$. We call c a *witness* of the fact that R is local.

Example 40. Let us give some examples of local base operations.

- The base operation *smaller-width* introduced in Section 7.1 is local as witnessed by the identity function. Indeed, suppose that $[\mathbf{w}, w]$ is part of $R(v)$, for some $v = (\Sigma; s_1, s_2)$. Since $\mathit{smaller\text{-}width}(v) = \{(\Sigma; \langle\rangle)\}$, it follows that $w = (\Sigma; \langle\rangle)$. Let $(V; P) = \mathbf{w}$. Since $P \subseteq [1, |\langle\rangle|] = \emptyset$, it follows that P is the empty set. It is clear that $\mathbf{v} = (V; \emptyset, \emptyset)$ is a requirement on v of size $|V| \leq |\mathbf{w}|$. We claim that $\mathbf{v}$ is a reason why $[\mathbf{w}, w] \triangleleft \mathit{smaller\text{-}width}(v)$. Indeed, let $v' = (\Sigma'; s'_1, s'_2) \in [\mathbf{v}, v]$ and suppose that $\mathit{smaller\text{-}width}(v')$ is defined. Since $v' \in [\mathbf{v}, v]$, $\Sigma' \sqsubseteq \Sigma$

and V is a subset of the nodes in Σ'. Hence, $(\Sigma'; \langle\rangle) \in [\mathbf{w}, w]$. Since *smaller-width*$(v')$ can only be $\{(\Sigma'; \langle\rangle)\}$, we have *smaller-width*$(v') \cap [\mathbf{w}, w] \neq \emptyset$, as desired.

- The children axis is local as witnessed by the function which maps k to $2k$. Indeed, suppose that $[\mathbf{w}, w]$ is part of $R(v)$, for some $v = (\Sigma; s)$. Since *children*$(v) = \{(\Sigma; t)\}$ with t containing the children of nodes in s in document order, this implies that $w = (\Sigma; t)$. Let $\mathbf{w} = (V; P)$. Let, for each $j \in P$, i_j be the position in $[1, |s|]$ such that $s(i_j)$ is the parent of $t(j)$. Let $P' = \{i_j \mid j \in P\}$ and $V' = V \cup \{t(j) \mid j \in P\}$. Clearly, $\mathbf{v} = (V'; P')$ is a requirement on v of size

$$\max\{|V'|, |P'|\} \leq \max\{|V| + |P|, |P|\} \leq 2|\mathbf{w}|.$$

  We claim that $\mathbf{v}$ is a reason why $[\mathbf{w}, w] \triangleleft$ *children*(v). Indeed, let $v' = (\Sigma'; s') \in [\mathbf{v}, v]$ and suppose that *children*(v') is defined. It follows in particular that $V' \supseteq V$ is a subset of the nodes in Σ' and that $(\Sigma'; s') \sqsubseteq (\Sigma; s)$. Let $(\Sigma'; t')$ be the value related to v' by *children* (where t' hence contains the children of nodes in s' in document order). Since every child of a node m in Σ' is also a child of m in Σ (as $\Sigma' \sqsubseteq \Sigma$) and since $rng(s') \subseteq rng(s)$ (as $s' \sqsubseteq s$), it follows that $rng(t') \subseteq rng(t)$. Moreover, $\{t(j) \mid j \in P\} \subseteq rng(t')$ since $\{s(i_j) \mid i_j \in P'\} \subseteq rng(s')$ by Lemma 33 and since $V' \supseteq \{t(j) \mid j \in P\}$ is a subset of the nodes in Σ'. It is not difficult to see that, since the nodes mentioned in t' and t occur in document order and since this document order is maintained by the fact that $\Sigma' \sqsubseteq \Sigma$, there hence exists a witness h of $t' \sqsubseteq t$ for which $P \subseteq rng(h)$. Hence, $(\Sigma'; t') \in [\mathbf{w}, w]$. Since *children*$(v') = \{(\Sigma'; t')\}$, we have *children*$(v') \cap [\mathbf{w}, w] \neq \emptyset$, as desired.

- The element constructor *element* is also a local base operation as witnessed by the identity function. Indeed, suppose that $[\mathbf{w}, w]$ is part of *element*(v). We know that v and w are of the form

$$v = (\Sigma; \langle a\rangle, \langle n_1, \ldots, n_k\rangle)$$
$$w = (\Sigma \circ \Theta; \langle m\rangle).$$

  Here, Θ is a tree, disjoint with Σ, whose root element node m is labeled by a such that

  - m has exactly k children $m_1, \ldots, m_k$ with $m_1 <_\Theta \ldots <_\Theta m_k$, and
  - for every $j \in [1, k]$ there exists a node-renaming ρ_j such that $\rho_j(\Sigma|_{n_j}) = \Theta|_{m_j}$.

  Let $(V; P) = \mathbf{w}$. We partition V into V_Σ and V_Θ such that V_Σ is a subset of the nodes in Σ and V_Θ is a subset of the nodes in Θ. Then let, for every $j \in [1, k]$, V_j be the maximal subset of V_Θ which is also

a subset of $\Theta|_{m_j}$. Let $W_j = \rho_j^{-1}(V_j)$ for every $j \in [1,k]$. Since ρ_j is a bijection, it is clear that $|W_j| = |V_j|$. Hence

$$|\bigcup_{j=1}^{k} W_j| = |\bigcup_{j=1}^{k} V_j| \leq |V_\Theta|.$$

Let Q be the set of all $j \in [1,k]$ for which $V_j \neq \emptyset$. It is clear that $|Q| \leq |V_\Theta|$. Then let $\mathbf{v}$ be the requirement on v defined by

$$\mathbf{v} := (V_\Sigma \cup \bigcup_{j=1}^{k} W_j; \emptyset, Q).$$

It is clear that the size of $\mathbf{v}$ is given by

$$\max\left\{|V_\Sigma \cup \bigcup_{j=1}^{k} W_j|, |Q|\right\} \leq \max\{|V_\Sigma| + |V_\Theta|, |V_\Theta|\} \leq |\mathbf{w}|.$$

We claim that $\mathbf{v}$ is a reason why $[\mathbf{w}, w] \triangleleft \mathit{element}(v)$. Indeed, let $v' \in [\mathbf{v}, v]$ and suppose that $\mathit{element}(v')$ is defined. In particular we know that $v' \sqsubseteq v$. Since $\mathit{element}(v')$ is defined, we hence know that v' is of the form

$$v' = (\Sigma'; \langle a \rangle, \langle n'_1, \ldots, n'_l \rangle).$$

Furthermore, since $v' \in [\mathbf{v}, v]$, there exists a witness h of

$$\langle n'_1, \ldots, n'_l \rangle \sqsubseteq \langle n_1, \ldots, n_k \rangle$$

such that $Q \subseteq \mathit{rng}(h)$. Note that, by definition of $\sqsubseteq$, we have $n'_i = n_{h(i)}$ for every $i \in [1,l]$. Then let Θ' be the tree with the a-labeled root node m in which m has $m_{h(1)}, \ldots, m_{h(l)}$ as children with

$$m_{h(1)} <_{\Theta'} \ldots <_{\Theta'} m_{h(l)},$$

such that for every $i \in [1,l]$:

$$\rho_{h(i)}\left(\Sigma'|_{n'_i}\right) = \Theta'|_{m_{h(i)}}.$$

It is easy to see that $\Theta' \sqsubseteq \Theta$ and hence $(\Sigma' \circ \Theta'; \langle m \rangle) \sqsubseteq (\Sigma \circ \Theta; \langle m \rangle)$. Furthermore, since for every $j \in [1,k]$ for which $V_j \neq \emptyset$ there exists $i \in [1,l]$ such that $h(i) = j$ and since W_j is a subset of the nodes in Σ', it follows by construction that V_j is a subset of the nodes in $\Theta'|_{m_j}$, for every $j \in [1,k]$. Hence, V_Θ is a subset of the nodes in Θ. Since V_Σ is also subset of the nodes in Σ', it follows that hence $V_\Sigma \cup V_\Theta = V$ is a subset of the nodes in $\Sigma' \circ \Theta'$. Furthermore, the identity function is certainly a witness of $\langle m \rangle \sqsubseteq \langle m \rangle$ whose range contains P (as $P \subseteq \{1\}$). Hence, $(\Sigma' \circ \Theta'; \langle m \rangle) \in [\mathbf{w}, w]$. Finally, it is easy to see that $(\Sigma' \circ \Theta'; \langle m \rangle) \in \mathit{element}(v')$. Hence, $\mathit{element}(v') \cap [\mathbf{w}, w] \neq \emptyset$, as desired. $\square$

Using similar reasonings as the ones employed in Example 40 we obtain:

Proposition 41. *The base operations concat, children, descendant, parent, ancestor, preceding-sibling, following-sibling, data, eq, is, $\ll$, is-element, is-text, is-atom, node-name, content, element, merge-text, text, empty, $+$, $\times$, and smaller-width are all local.*

Note that hence locality alone is not powerful enough to prevent the construction of QL(*concat*, *children*, *eq*, *node-name*, *content*, *element*, *empty*), QL($+, \times$), and QL(*concat*, *smaller-width*), for which well-definedness is undecidable.

7.3 Local and locally-undefined expressions

In this section we show that if B is a finite set of monotone, local, and locally-undefined base operations, then locally-undefinedness transfers to expressions in QL(B). Moreover, a witness of the fact that $e \in \mathrm{QL}(B)$ is locally-undefined can be computed from e. This property lies at the heart of our decidability result in Section 8. Before we are able to prove it we first show that if B is a finite set of monotone and local base operations, then locality transfers to expressions in QL(B). We start out by stating the following technical lemmas.

Lemma 42. *Let $(V; P)$ be a requirement on $(\Sigma_0 \circ \bigcirc_{j=1}^{p} \Sigma_j; \bigcirc_{j=1}^{p} s_j)$. Let $V_0, \ldots, V_p$ be the partition of V such that V_0 is a subset of the nodes in Σ_0 and V_j is a subset of the nodes in Σ_j, for every $j \in [1, p]$. Let for each $j \in [1, p]$, P_j be the subset of P defined by*

$$P_j := \left\{ k - \sum_{i=1}^{j-1} |s_i| \quad \middle| \quad k \in P \text{ and } \sum_{i=1}^{j-1} |s_i| < k \leq \sum_{i=1}^{j} |s_i| \right\}.$$

Let, for every $j \in [1, p]$, $(\Sigma'_0 \circ \Sigma'_j; s'_j)$ be a value in $[(V_0 \cup V_j; P_j), (\Sigma_0 \circ \Sigma_j; s_j)]$ such that $\Sigma'_0 \sqsubseteq \Sigma_0$ and $\Sigma'_j \sqsubseteq \Sigma_j$. Then

$$(\Sigma'_0 \circ \bigcirc_{j=1}^{p} \Sigma'_j; \bigcirc_{j=1}^{p} s'_j) \in [(V; P), (\Sigma_0 \circ \bigcirc_{j=1}^{p} \Sigma_j; \bigcirc_{j=1}^{p} s_j)].$$

Proof. It immediately follows that

$$(\Sigma'_0 \circ \bigcirc_{j=1}^{p} \Sigma'_j; \bigcirc_{j=1}^{p} s'_j) \sqsubseteq (\Sigma_0 \circ \bigcirc_{j=1}^{p} \Sigma_j; \bigcirc_{j=1}^{p} s_j).$$

Since $\Sigma'_0 \circ \Sigma'_j$ contains all nodes in $V_0 \cup V_j$ for every $j \in [1, p]$ it follows that $\Sigma'_0 \circ \bigcirc_{j=1}^{p} \Sigma'_j$ contains all nodes in

$$V_0 \cup \bigcup_{j=1}^{p} V_j = V.$$

Furthermore, for every $j \in [1,p]$ there exists a witness h_j of $s'_j \sqsubseteq s_j$ such that $P_j \subseteq rng(h_j)$. Then let h be the function mapping $[1, |\bigcirc_{j=1}^{p} s'_j|]$ to $[1, |\bigcirc_{j=1}^{p} s_j|]$ defined by

$$h(q) := h_j\left(q - \sum_{i=1}^{j-1} |s'_i|\right) + \sum_{i=1}^{j-1} |s_i| \quad \text{when } \sum_{i=1}^{j-1} |s'_i| < q \leq \sum_{i=1}^{j} |s'_i|.$$

It is easy to see that h is a witness of $\bigcirc_{j=1}^{p} s'_j \sqsubseteq \bigcirc_{j=1}^{p} s_j$. It remains to show that $P \subseteq rng(h)$. Let $k \in P$. Since $P \subseteq [1, |\bigcirc_{j=1}^{p} s_j|]$, there exists $j \in [1,p]$ such that

$$\sum_{i=1}^{j-1} |s_i| < k \leq \sum_{i=1}^{j} |s_i|.$$

It follows that hence

$$k - \sum_{i=1}^{j-1} |s_i| \in P_j.$$

Since $P_j \subseteq rng(h_j)$ there exists $l \in [1, |s'_j|]$ such that

$$h_j(l) = k - \sum_{i=1}^{j-1} |s_i|.$$

Then obviously

$$\sum_{i=1}^{j-1} |s'_i| < l + \sum_{i=1}^{j-1} |s'_i| \leq \sum_{i=1}^{j} |s'_i|,$$

and hence

$$h(l + \sum_{i=1}^{j-1} |s'_i|) = h_j(l) + \sum_{i=1}^{j-1} |s_i| = k.$$

Hence, $k \in rng(h)$, as desired. □

Lemma 43. *Let R be a base operation, let $(\Sigma \circ \Sigma_1; s_1) \in R(\Sigma; \vec{s})$, and let $(V, \vec{P})$ be a reason why $[(W,Q), (\Sigma \circ \Sigma_1; s_1)] \triangleleft R(\Sigma; \vec{s})$ such that V contains all nodes of W in Σ. For every $(\Sigma'; \vec{s'}) \in [(V, \vec{P}), (\Sigma; \vec{s})]$ on which R is defined there exists*

$$(\Sigma' \circ \Sigma'_1; s'_1) \in R(\Sigma'; \vec{s'}) \cap [(W,Q), (\Sigma \circ \Sigma_1; s_1)]$$

with $\Sigma'_1 \sqsubseteq \Sigma_1$.

Proof. Every store can be written as a concatenation of trees. Let $\Theta_1, \ldots, \Theta_k$ be the non-empty trees such that $\Sigma = \Theta_1 \circ \cdots \circ \Theta_k$. Since $\Sigma' \sqsubseteq \Sigma$, we can write Σ' as a concatenation of trees $\Theta'_1 \circ \cdots \circ \Theta'_k$ such that $\Theta'_j \sqsubseteq \Theta_j$ for every

$j \in [1,k]$, where if Σ' does not contain any node in Θ_j, we take Θ'_j to be the empty tree. Let $\Delta_1, \ldots, \Delta_k$ be the trees such that, for every $j \in [1,k]$, $\Delta_j = \Theta'_j$ if Θ'_j is non-empty, and $\Delta_j = \Theta_j$ otherwise. In particular, $\Delta_j = \Theta'_j$ whenever $\vec{s'}$ mentions a node in Θ_j. Since R is reachable-only and since $R(\Theta'_1 \circ \cdots \circ \Theta'_k; \vec{s'})$ is defined, it is easy to see that $R(\Delta_1 \circ \cdots \circ \Delta_k; \vec{s'})$ is also defined. Moreover, by construction we have $\Delta_j \sqsubseteq \Theta_j$ for every $j \in [1,k]$. Hence,

$$(\Delta_1 \circ \cdots \circ \Delta_k; \vec{s'}) \in [(V, \vec{P}), (\Sigma; \vec{s})].$$

Since $(V, \vec{P})$ is a reason why $[(W,Q), (\Sigma \circ \Sigma_1; s_1)] \triangleleft R(\Sigma; \vec{s})$, there exists

$$(\Delta_1 \circ \cdots \circ \Delta_k \circ \Sigma'_1; s'_1) \in R(\Delta_1 \circ \cdots \circ \Delta_k; \vec{s'}) \cap [(W,Q), (\Sigma \circ \Sigma_1; s_1)].$$

Hence, $\Sigma'_1 \sqsubseteq \Sigma \circ \Sigma_1$. Using a similar reasoning as in the proof of Lemma 23 it now follows that $\Sigma'_1 \sqsubseteq \Sigma_1$. Then, since R is reachable-only, since $\Delta_j = \Theta'_j$ whenever $\vec{s'}$ mentions a node in Δ_j, and since $(\Delta_1 \circ \cdots \circ \Delta_k \circ \Sigma'_1; s'_1) \in R(\Delta_1 \circ \cdots \circ \Delta_k; \vec{s'})$ we have $(\Sigma' \circ \Sigma'_1; s'_1) \in R(\Sigma'; \vec{s'})$. Moreover, since V contains all nodes of W in Σ, it is easy to see that W is a subset of the nodes in $\Sigma' \circ \Sigma'_1$. Hence, $(\Sigma' \circ \Sigma'_1; s'_1) \in [(W,Q), (\Sigma \circ \Sigma_1; s_1)]$, as desired. □

Proposition 44. *If B is a finite set of monotone and local base operations, then every expression $e \in QL(B)$ is also local. Moreover, an arithmetic expression defining a witness of this locality can effectively be computed from e.*

Proof. Let c_f be a witness of the fact that base operation $f \in B$ is local. For every $e \in \mathrm{QL}(B)$ we then define the function c_e inductively as follows:

$$\begin{aligned}
c_x(k) &:= k \\
c_a(k) &:= k \\
c_{()}(k) &:= k \\
c_{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3}(k) &:= \max\{c_{e_2}(k), c_{e_3}(k)\} \\
c_{\texttt{let } x:=e_1 \texttt{ return } e_2}(k) &:= c_{e_1}(c_{e_2}(k)) + c_{e_2}(k) \\
c_{\texttt{for } x \texttt{ in } e_1 \texttt{ return } e_2}(k) &:= c_{e_1}(\max\{k + 2kc_{e_2}(k), 2k\}) + 2kc_{e_2}(k) \\
c_{f(e_1,\ldots,e_p)}(k) &:= c_f(k) + c_{e_1}(c_f(k)) + \cdots + c_{e_p}(c_f(k))
\end{aligned}$$

It is clear from this inductive definition that an arithmetic expression defining c_e can effectively be computed from e. It is also clear that c_e is a computable, increasing function mapping natural numbers to natural numbers. Let v be a context of e and let $[\mathbf{w}, w]$ be a part of $e(v)$. We will prove by induction on e that there exists a reason why $[\mathbf{w}, w'] \triangleleft e(v)$ of size at most $c_e(|\mathbf{w}|)$. During our induction we will often use the fact that every expression $e' \in \mathrm{QL}(B)$ defines a monotone base operation by Propositions 6 and 25. We will also use the fact that if $e' \in \mathrm{QL}(B)$ is defined on input v', then e' is also defined on every input node-isomorphic to v' by Lemma 24.

- If $e = x$, then let $(\Sigma; \sigma) = v$. Since $e(v) = \{(\Sigma; \sigma(x))\}$ and since $[\mathbf{w}, w] \triangleleft e(v)$, it follows that $w = (\Sigma; \sigma(x))$. Let $(V; P) = \mathbf{w}$ and let $\mathbf{v}$ be the requirement $(V; \phi)$ on v such that ϕ is defined by

$$\phi(y) := \begin{cases} P & \text{if } y = x \\ \emptyset & \text{otherwise.} \end{cases}$$

  It is easy to see that $\mathbf{v}$ is a reason why $[\mathbf{w}, w] \triangleleft e(v)$ of size $|\mathbf{w}|$.[12]

- If $e = a$, then let $(\Sigma; \sigma) = v$. Since $e(v) = \{(\Sigma; \langle a \rangle)\}$ and since $[\mathbf{w}, w] \triangleleft e(v)$, it follows that $w = (\Sigma; \langle a \rangle)$. Let $(V; P) = \mathbf{w}$ and let $\mathbf{v}$ be the requirement $(V; \phi)$ on v such that ϕ is defined by

$$\phi(x) := \emptyset \quad \text{for all } x.$$

  It is easy to see that $\mathbf{v}$ is a reason why $[\mathbf{w}, w] \triangleleft e(v)$ of size $|\mathbf{w}|$.

- The case where $e = ()$ is similar.

- If $e = \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$, then there exists $(\Sigma_1; \langle b \rangle) \in e_1(v)$ with b a boolean such that $[\mathbf{w}, w] \triangleleft e_2(v)$ if $b = \texttt{true}$ and $[\mathbf{w}, w] \triangleleft e_3(v)$ otherwise. Suppose that $b = \texttt{true}$. Then there exists a reason $\mathbf{v}$ why $[\mathbf{w}, w] \triangleleft e_2(v)$ of size at most $c_{e_2}(|\mathbf{w}|)$ by the induction hypothesis. We claim that $\mathbf{v}$ is also a reason why $[\mathbf{w}, w] \triangleleft e(v)$. Indeed, suppose that $v' \in [\mathbf{v}, v]$ and that $e(v')$ is defined. In particular $e_1(v')$ must then also be defined. Since e_1 is monotone, there exists $(\Sigma_1'; s') \in e_1(v')$ with $(\Sigma_1'; s') \sqsubseteq (\Sigma_1; \langle \texttt{true} \rangle)$. It follows that $s' = \langle\rangle$ or $s' = \langle \texttt{true} \rangle$. Suppose that $s' = \langle\rangle$. Then we know that the list of all values in $e_1(v')$ is empty, since e_1 is a semi-function. Hence, $e(v')$ would be undefined, a contradiction. Hence, s' must be $\langle \texttt{true} \rangle$. Since e_1 is a semi-function, it follows that the list of every value in $e_1(v')$ is $\langle \texttt{true} \rangle$. Hence, $e(v') = e_2(v')$ and thus

$$e(v') \cap [\mathbf{w}, w] = e_2(v') \cap [\mathbf{w}, w] \neq \emptyset.$$

  In a similar way we can show that if $b = \texttt{false}$, then the reason $\mathbf{v}$ why $[\mathbf{w}, w] \triangleleft e_3(v)$ of size at most $c_{e_3}(|\mathbf{w}|)$ as given by the induction hypothesis is also a reason why $[\mathbf{w}, w] \triangleleft e(v)$. Hence, we can always find a reason why $[\mathbf{w}, w] \triangleleft e(v)$ of size at most $\max\{c_{e_2}(|\mathbf{w}|), c_{e_3}(|\mathbf{w}|)\} = c_e(|\mathbf{w}|)$.

[12] Here we extend the notion of a requirement to contexts in the obvious way: if σ is an environment with domain $\{x, \ldots, y\}$ and ϕ is function from $\{x, \ldots, y\}$ to the positive natural numbers, then $(V; \phi)$ is a requirement on context $(\Sigma; \sigma)$ if $(V; \phi(x), \ldots, \phi(y))$ is a requirement on $(\Sigma; \sigma(x), \ldots, \sigma(y))$. We say that $(\Sigma'; \sigma')$ satisfies $(V; \phi)$ on $(\Sigma; \sigma)$ if $(\Sigma'; \sigma'(x), \ldots, \sigma'(y))$ satisfies $(V; \phi(x), \ldots, \phi(y))$ on $(\Sigma; \sigma(x), \ldots, \sigma(y))$.

- If $e = \texttt{let}\ x := e_1\ \texttt{return}\ e_2$, then let $(\Sigma; \sigma) = v$. Since $[\mathbf{w}, w] \triangleleft e(v)$ there exists $(\Sigma_1; s_1) \in e_1(v)$ such that $[\mathbf{w}, w] \triangleleft e_2(\Sigma_1; x\colon s_1, \sigma)$. By the induction hypothesis there exists a reason $(V_1; x\colon P_1, \phi_1)$ why this is so of size at most $c_{e_2}(|\mathbf{w}|)$. We have in particular that $(V_1; P_1)$ is a requirement on $(\Sigma_1; s_1)$. By the induction hypothesis there hence exists a reason $(V; \phi)$ why $[(V_1; P_1), (\Sigma_1; s_1)] \triangleleft e_1(v)$ of size at most $c_{e_1}(c_{e_2}(|\mathbf{w}|))$. Let ϕ' be the function with domain $dom(\sigma)$ defined by

$$\phi'(y) := \phi(y) \cup \phi_1(y),$$

and let $\mathbf{v} = (V; \phi')$. It is easy to see that $\mathbf{v}$ is a requirement on v. Moreover,

$$\begin{aligned} |\mathbf{v}| &= \max\left\{|V|, |\phi'(x)| \;\middle|\; x \in dom(\sigma)\right\} \\ &\leq \max\{c_{e_1}(c_{e_2}(|\mathbf{w}|)), c_{e_1}(c_{e_2}(|\mathbf{w}|)) + c_{e_2}(|\mathbf{w}|)\} \\ &= c_{e_1}(c_{e_2}(|\mathbf{w}|)) + c_{e_2}(|\mathbf{w}|) \\ &= c_e(|\mathbf{w}|). \end{aligned}$$

We claim that $\mathbf{w}$ is a reason why $[\mathbf{w}, w] \triangleleft e(v)$. Indeed, let $v' = (\Sigma'; \sigma') \in [\mathbf{v}, v]$ and suppose that $e(v')$ is defined. There hence exists $(\Sigma_1'; s_1') \in e_1(v')$ such that $e_2(\Sigma_1'; x\colon s_1', \sigma')$ is defined. Since $e_1(v')$ is hence defined; since $(V; \phi)$ is a reason why

$$[(V_1; P_1), (\Sigma_1; s_1)] \triangleleft e_1(\Sigma; \sigma);$$

and since $v' \in [(V; \phi'), (\Sigma; \sigma)] \subseteq [(V; \phi), (\Sigma; \sigma)]$, it follows that

$$e_1(v') \cap [(V_1; P_1), (\Sigma_1; s_1)] \neq \emptyset.$$

Let $(\Sigma_1''; s_1'')$ be a value in this non-empty intersection. Then clearly

$$\begin{aligned} (\Sigma_1''; x\colon s_1'', \sigma') &\in [(V_1; x\colon P_1, \phi'), (\Sigma_1; x\colon s_1, \sigma)] \\ &\subseteq [(V_1; x\colon P_1, \phi_1), (\Sigma_1; x\colon s_1, \sigma)]. \end{aligned} \tag{2}$$

Furthermore, since e_1 is a base operation it follows that

$$(\Sigma_1''; x\colon s_1'', \sigma') \equiv_{node} (\Sigma_1'; x\colon s_1', \sigma')$$

by Corollary 12. Since $e_2(\Sigma_1'; x\colon s_1', \sigma')$ is defined and since e_2 is node-generic, it follows that

$$e_2(\Sigma_1''; x\colon s_1'', \sigma') \neq \emptyset. \tag{3}$$

Since $(V_1; x\colon P_1, \phi_1)$ is a reason why $[\mathbf{w}, w] \triangleleft e_2(\Sigma_1; x\colon s_1, \sigma)$ it follows from (2) and (3) that $e_2(\Sigma_1''; x\colon s_1'', \sigma') \cap [\mathbf{w}, w] \neq \emptyset$. Hence, $e(v') \cap [\mathbf{w}, w] \neq \emptyset$, as desired.

- If $e = f(e_1, \ldots, e_p)$, then let $(\Sigma; \sigma) = v$. Since $[\mathbf{w}, w] \triangleleft e(v)$ there exists $(\Sigma \circ \Sigma_j; s_j) \in e_j(v)$ for every $j \in [1, p]$ such that the Σ_j are all pairwise disjoint and
$$[\mathbf{w}, w] \triangleleft f(\Sigma \circ \bigcirc_{j=1}^{p} \Sigma_j; s_1, \ldots, s_p).$$
Since f is a local base operation with witness c_f there hence exists a reason $(V \cup \bigcup_{j=1}^{p} V_j; P_1, \ldots, P_p)$ why this is so of size at most $c_f(|\mathbf{w}|)$. Here, V is a subset of the nodes in Σ and V_j is a subset of the nodes in Σ_j, for every $j \in [1, p]$. We have in particular that $(V \cup V_j; P_j)$ is a requirement on $(\Sigma \circ \Sigma_j; s_j)$. By the induction hypothesis there hence exists for every $j \in [1, p]$ a reason $(W_j; \phi_j)$ why
$$[(V \cup V_j; P_j), (\Sigma \circ \Sigma_j; s_j)] \triangleleft e_j(v)$$
of size at most $c_{e_j}(c_f(|\mathbf{w}|))$. Let $\mathbf{v}$ be the requirement $(V \cup \bigcup_{j=1}^{p} W_j; \phi)$ on v such that ϕ is the function with domain $dom(\sigma)$ defined by
$$\phi(y) := \bigcup_{j=1}^{p} \phi_j(y).$$
Clearly,
$$\begin{aligned} |\mathbf{v}| &\leq \max \left\{ |V| + \sum_{j=1}^{p} |W_j|, \sum_{j=1}^{p} |\phi_j(x)| \;\middle|\; x \in dom(\sigma) \right\} \\ &\leq c_f(|\mathbf{w}|) + \sum_{j=1}^{p} c_{e_j}(c_f(|\mathbf{w}|)) = c_e(|\mathbf{w}|). \end{aligned}$$
Moreover, since $[\mathbf{v}, v] \subseteq [(W_j; \phi_j), v]$ for every $j \in [1, p]$, $\mathbf{v}$ is a reason why
$$[(V \cup V_j; P_j), (\Sigma \circ \Sigma_j; s_j)] \triangleleft e_j(v).$$
We claim that $\mathbf{v}$ is also a reason why $[\mathbf{w}, w] \triangleleft e(v)$. Indeed, let $v' = (\Sigma'; \sigma') \in [\mathbf{v}, v]$ and suppose that $e(v')$ is defined. There hence exist $(\Sigma' \circ \Sigma'_j; s'_j) \in e_j(v')$ for every $j \in [1, p]$ such that the Σ'_j are all pairwise disjoint and
$$f(\Sigma' \circ \bigcirc_{j=1}^{p} \Sigma'_j; s'_1, \ldots, s'_p) \neq \emptyset. \tag{4}$$
Since $e_j(v')$ is hence defined; since e_j is a base operation; since $\mathbf{v}$ is a reason why
$$[(V \cup V_j; P_j), (\Sigma \circ \Sigma_j; s_j)] \triangleleft e_j(v);$$
and since $\mathbf{v}$ contains all nodes of $V \cup V_j$ in Σ, it follows from Lemma 43 that for every $j \in [1, p]$ there exists
$$(\Sigma' \circ \Sigma''_j; s''_j) \in e_j(v') \cap [(V \cup V_j; P_j), (\Sigma \circ \Sigma_j; s_j)],$$

with $\Sigma''_j \sqsubseteq \Sigma_j$. Since in particular $V \cup V_j$ is a subset of the nodes in $\Sigma' \circ \Sigma''_j$, it follows that $V \cup \bigcup_{j=1}^{p} V_j$ is a subset of the nodes in $\Sigma' \circ \bigcirc_{j=1}^{p} \Sigma''_j$. Hence,

$$(\Sigma' \circ \bigcirc_{j=1}^{p} \Sigma''_j; s''_1, \ldots, s''_p) \in \\ [(V \cup \bigcup_{j=1}^{p} V_j; P_1, \ldots, P_p), (\Sigma \circ \bigcirc_{j=1}^{p} \Sigma_j; s_1, \ldots, s_p)]. \quad (5)$$

Since every e_j is a semi-function, it follows that $(\Sigma' \circ \Sigma'_j; s'_j)$ is node-isomorphic to $(\Sigma' \circ \Sigma''_j; s''_j)$. Since $\Sigma''_j \sqsubseteq \Sigma_j$ and since the Σ_j are all pairwise disjoint, it follows that the Σ''_j are also pairwise disjoint. Since the Σ'_j are also pairwise disjoint, it follows by Lemma 11 that

$$(\Sigma' \circ \bigcirc_{j=1}^{p} \Sigma'_j; s'_1, \ldots, s'_p) \equiv_{node} (\Sigma' \circ \bigcirc_{j=1}^{p} \Sigma''_j; s''_1, \ldots, s''_p). \quad (6)$$

Since f is node-generic it follows from (4) and (6) that

$$f(\Sigma' \circ \bigcirc_{j=1}^{p} \Sigma''_j; s''_1, \ldots, s''_p) \neq \emptyset. \quad (7)$$

Furthermore, since $(V \cup \bigcup_{j=1}^{p} V_j; P_1, \ldots, P_p)$ is a reason why

$$[\mathbf{w}, w] \triangleleft f(\Sigma \circ \bigcirc_{j=1}^{p} \Sigma_j; s_1, \ldots, s_p),$$

it follows from (5) and (7) that

$$f(\Sigma' \circ \bigcirc_{j=1}^{p} \Sigma''_j; s''_1, \ldots, s''_p) \cap [\mathbf{w}, w] \neq \emptyset.$$

Hence, $e(v') \cap [\mathbf{w}, w] \neq \emptyset$, as desired.

- If $e = \texttt{for}\ \ x\ \ \texttt{in}\ \ e_1\ \ \texttt{return}\ \ e_2$ then let $(\Sigma; \sigma) = v$. Since $[\mathbf{w}, w] \triangleleft e(v)$ there exists $(\Sigma_0; s) \in e_1(v)$ and values

$$(\Sigma_0 \circ \Sigma_j; s_j) \in e_2(\Sigma_0; x\colon \langle s(j) \rangle, \sigma)$$

for every $j \in [1, |s|]$ such that the Σ_j are all pairwise disjoint and

$$w = (\Sigma_0 \circ \bigcirc_{j=1}^{|s|} \Sigma_j; \bigcirc_{j=1}^{|s|} s_j).$$

Let $\mathbf{w} = (V; P)$. Note that in particular V is a subset of the nodes in

$$\Sigma_0 \circ \bigcirc_{j=1}^{|s|} \Sigma_j.$$

Hence, we can partition V into $V_0, \ldots, V_{|s|}$ such that V_0 is contained in the nodes of Σ_0 and V_j is contained in the nodes of Σ_j, for every $j \in [1, |s|]$. Let, for every $j \in [1, |s|]$, P_j be the subset of P defined by

$$P_j := \left\{ k - \sum_{i=1}^{j-1} |s_i| \quad \middle| \quad k \in P \text{ and } \sum_{i=1}^{j-1} |s_i| < k \leq \sum_{i=1}^{j} |s_i| \right\}.$$

We have in particular that $(V_0 \cup V_j; P_j)$ is a requirement on $(\Sigma_0 \circ \Sigma_j; s_j)$. By the induction hypothesis there hence exists, for every $j \in [1, |s|]$, a reason $(W_j; x\colon Q_j, \phi_j)$ why

$$[(V_0 \cup V_j; P_j), (\Sigma_0 \circ \Sigma_j; s_j)] \triangleleft e_2(\Sigma_0; x\colon \langle s(j)\rangle; \sigma) \tag{8}$$

of size at most

$$c_{e_2}(|(V_0 \cup V_j; Q_j)|) \leq c_{e_2}(|\mathbf{w}|).$$

Let J be the set of j in $[1, |s|]$ for which V_j or P_j is non-empty. Note that there can be at most $|\mathbf{w}|$ of the V_j non-empty and that there can be at most $|\mathbf{w}|$ of the P_j non-empty. Hence, J contains at most $2|\mathbf{w}|$ elements. Hence, the requirement $(V_0 \cup \bigcup_{j\in J} W_j; J)$ on $(\Sigma_0; s)$ has size at most

$$\begin{aligned}\max\{|V_0 \cup \bigcup_{j\in J} W_j|, |J|\} &\leq \max\{|V_0| + \sum_{j\in J} |W_j|, |J|\} \\ &\leq \max\{|\mathbf{w}| + \sum_{j\in J} c_{e_2}(|\mathbf{w}|), |J|\} \\ &\leq \max\{|\mathbf{w}| + 2|\mathbf{w}|c_{e_2}(|\mathbf{w}|), 2|\mathbf{w}|\}.\end{aligned}$$

Hence, by the induction hypothesis there exists a reason $(W; \phi)$ why

$$[(V_0 \cup \bigcup_{j\in J} W_j; J), (\Sigma_0; s)] \triangleleft e_1(v)$$

of size at most

$$c_{e_1}\left(|(V_0 \cup \bigcup_{j\in J} W_j; J)|\right) \leq c_{e_1}(\max\{|\mathbf{w}| + 2|\mathbf{w}|c_{e_2}(|\mathbf{w}|), 2|\mathbf{w}|\}).$$

Let ϕ' be the function with domain $dom(\sigma)$ defined by

$$\phi'(y) := \phi(y) \cup \bigcup_{j\in J} \phi_j(y),$$

and let $\mathbf{v} = (W; \phi')$. It is easy to see that $\mathbf{v}$ is a requirement on v of size at most

$$\begin{aligned}&c_{e_1}\left(\max\{|\mathbf{w}| + 2|\mathbf{w}|c_{e_2}(|\mathbf{w}|), 2|\mathbf{w}|\}\right) + \sum_{j\in J} c_{e_2}(|\mathbf{w}|) \\ &\quad\leq c_{e_1}(\max\{|\mathbf{w}| + 2|\mathbf{w}|c_{e_2}(|\mathbf{w}|), 2|\mathbf{w}|\}) + 2|\mathbf{w}|c_{e_2}(|\mathbf{w}|) = c_e(|\mathbf{w}|).\end{aligned}$$

We claim that $\mathbf{v}$ is a reason why $[\mathbf{w}, w] \triangleleft e(v)$. Indeed, let $v' = (\Sigma'; \sigma') \in [\mathbf{v}, v]$ such that $e(v')$ is defined. In particular there must hence exist $(\Sigma'_0; s') \in e_1(v')$ such that $e_2(\Sigma'_0; x\colon \langle s'(j)\rangle, \sigma')$ is defined

for every $j \in [1, |s'|]$. Since $e_1(\Sigma'; \sigma')$ is hence defined; since (W, ϕ) is a reason why

$$[(V_0 \cup \bigcup_{j \in J} W_j; J), (\Sigma_0; s)] \triangleleft e_1(\Sigma; \sigma);$$

and since

$$(\Sigma'; \sigma') \in [(W; \phi'), (\Sigma; \sigma)] \subseteq [(W; \phi), (\Sigma; \sigma)],$$

it follows that

$$e_1(\Sigma'; \sigma') \cap [(V_0 \cup \bigcup_{j \in J} W_j; J), (\Sigma_0; s)] \neq \emptyset.$$

Let $(\Sigma_0''; s'')$ be a value in this non-empty intersection. Since e_1 is a store-increasing semi-function, it follows from Corollary 13 that $|s'| = |s''|$ and that

$$(\Sigma_0'; x\colon \langle s'(j)\rangle, \sigma') \equiv_{node} (\Sigma_0''; x\colon \langle s''(j)\rangle, \sigma')$$

for every $j \in [1, |s'|]$. Since e_2 is node-generic and since e_2 is defined on $(\Sigma_0'; x\colon \langle s'(j)\rangle, \sigma')$ for every $j \in [1, |s'|]$, it follows that e_2 is also defined on $(\Sigma_0''; x\colon \langle s''(j)\rangle, \sigma')$ for every $j \in [1, |s''|]$. Since

$$(\Sigma_0''; s'') \in [(V_0 \cup \bigcup_{j \in J} W_j; J), (\Sigma_0; s)],$$

there exists a witness h of $s'' \sqsubseteq s$ such that $J \subseteq rng(h)$. We will prove below that for every $j \in rng(h)$ there exists

$$(\Sigma_0'' \circ \Sigma_j''; s_j'') \in e_2(\Sigma_0''; x\colon \langle s''(h^{-1}(j))\rangle, \sigma') \cap [(V_0 \cup V_j; Q_j), (\Sigma_0 \circ \Sigma_j; s_j)], \quad (9)$$

such that $\Sigma_j'' \sqsubseteq \Sigma_j$. Note that $h^{-1}(j)$ is uniquely determined as h is strictly increasing and hence injective. Let $\Sigma_j'' = \emptyset$ and $s_j'' = \langle\rangle$ for every $j \in [1, |s|] \setminus rng(h)$. Then

$$(\Sigma_0'' \circ \bigcirc_{i=1}^{|s''|} \Sigma_{h(i)}''; \bigcirc_{i=1}^{|s''|} s_{h(i)}'') = (\Sigma_0'' \circ \bigcirc_{j=1}^{|s|} \Sigma_j''; \bigcirc_{j=1}^{|s|} s_j'').$$

Since the left-hand side is in $e(v')$, it follows that

$$(\Sigma_0'' \circ \bigcirc_{j=1}^{|s|} \Sigma_j''; \bigcirc_{j=1}^{|s|} s_j'') \in e(v'). \quad (10)$$

Note that, if $j \in [1, |s|] \setminus rng(h)$, then also $j \notin J$ as $rng(h) \supseteq J$. Hence, for such j we know that V_j and P_j are empty. Then

$$(\Sigma_0'' \circ \Sigma_j''; s_j'') \in [(V_0 \cup V_j; P_j), (\Sigma_0 \circ \Sigma_j; s_j)]$$

and $\Sigma''_j \sqsubseteq \Sigma_j$ for every $j \in [1, |s|]$. Note that, since $\Sigma''_j \sqsubseteq \Sigma_j$ and since the Σ_j are all pairwise disjoint, it follows that the Σ''_j are also pairwise disjoint. By Lemma 42 and (10) it then follows that

$$(\Sigma''_0 \circ \bigcirc_{j=1}^{|s|} \Sigma''_j; \bigcirc_{j=1}^{|s|} s''_j) \in [(V; P), (\Sigma_0 \circ \bigcirc_{j=1}^{|s|} \Sigma_j; \bigcirc_{j=1}^{|s|} s_j)] \cap e(v').$$

Hence, $e(v') \cap [\mathbf{w}, w] \neq \emptyset$, as desired.

It remains to show (9). Let $j \in rng(h)$. Since h is a witness of $s'' \sqsubseteq s$ we have $s''(h^{-1}(j)) = s(j)$. (Remember that $h^{-1}(j)$ is uniquely determined as h is strictly increasing and hence injective.) We discern two possibilities.

- Case $j \in J$. Since we have chosen $(\Sigma''_0; s'')$ such that

  $$(\Sigma''_0; s'') \in [(V_0 \cup \bigcup_{j \in J} W_j; J), (\Sigma_0; s)],$$

  W_j is a subset of the nodes in Σ''_0. Moreover, since Q_j is a subset of $[1, |\langle s(j)\rangle|] = \{1\}$ it is clear that the identity function is a witness of $\langle s''(h^{-1}(j))\rangle \sqsubseteq \langle s(j)\rangle$ whose range includes Q_j. Since also $\phi_j(y) \subseteq \phi'(y)$ for every $y \in dom(\phi')$ by construction, it follows that

  $$(\Sigma''_0; x\colon \langle s''(h^{-1}(j))\rangle, \sigma') \in [(V_0 \cup W_j, x : Q_j, \phi_j), (\Sigma_0; x\colon \langle s(j)\rangle, \sigma)]\,. \quad (11)$$

  Since $[(V_0 \cup W_j, x\colon Q_j, \phi_j), (\Sigma_0; x\colon \langle s(j)\rangle, \sigma)]$ is clearly a subset of $[(W_j, x\colon Q_j, \phi_j), (\Sigma_0; x\colon \langle s(j)\rangle, \sigma)]$, it follows by (8) that $(V_0 \cup W_j, x\colon Q_j, \phi_j)$ is a reason why

  $$[(V_0 \cup V_j; Q_j), (\Sigma_0 \circ \Sigma_j; s_j)] \lhd e_2(\Sigma_0; x\colon \langle s(j)\rangle, \sigma).$$

  Then, since $V_0 \cup W_j$ contains the nodes of $V_0 \cup V_j$ in Σ_0; since e_2 is defined on $(\Sigma''_0; x\colon \langle s''(h^{-1}(j))\rangle, \sigma')$; and since (11) holds, it follows by Lemma 43 that there exists $(\Sigma''_0 \circ \Sigma''_j; s''_j)$ in

  $$e_2(\Sigma''_0; x\colon \langle s''(h^{-1}(j)\rangle, \sigma') \cap [(V_0 \cup V_j; Q_j), (\Sigma_0 \circ \Sigma_j; s_j)]$$

  with $\Sigma''_j \sqsubseteq \Sigma_j$, as desired.
- Case $j \notin J$. Note that

  $$(\Sigma''_0; x\colon \langle s''(h^{-1}(j))\rangle, \sigma') \sqsubseteq (\Sigma_0; x\colon \langle s(j)\rangle, \sigma).$$

  Since e_2 is monotone; since

  $$(\Sigma_0 \circ \Sigma_j; s_j) \in e_2(\Sigma_0; x\colon \langle s(j)\rangle, \sigma);$$

and since e_2 is defined on $(\Sigma_0''; x\colon \langle s''(h^{-1}(j))\rangle, \sigma')$, there exists $(\Sigma_0'' \circ \Sigma_j''; s_j'') \sqsubseteq (\Sigma_0 \circ \Sigma_j; s_j)$ in $e_2(\Sigma_0''; x\colon \langle s''(h^{-1}(j))\rangle, \sigma')$ such that $\Sigma_j'' \sqsubseteq \Sigma_j$ by Lemma 23. Since

$$(\Sigma_0''; s'') \in [(V_0 \cup \bigcup_{j \in J} W_j; J), (\Sigma_0; s)],$$

we know that V_0 is a subset of the nodes in Σ_0''. Since $j \notin J$, we have by construction that both V_j and Q_j are empty. Hence

$$(\Sigma_0'' \circ \Sigma_j''; s_j'') \in [(V_0 \cup V_j; Q_j), (\Sigma_0 \circ \Sigma_j; s_j)]\,,$$

as desired. □

We are now ready to prove the main proposition of this section.

Proposition 45. *If B is a finite set of monotone, local, and locally undefined base operations, then every expression e in QL(B) is also locally-undefined. Moreover, a witness k_e for this locally-undefinedness can effectively be computed from e.*

Proof. Let k_f be a witness for the locally-undefinedness of base operation $f \in B$. Let $e \in \mathrm{QL}(B)$. We then define the natural number k_e inductively as follows:

$$\begin{aligned}
k_x = k_a = k_{()} &:= 0 \\
k_{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3} &:= \max\{k_{e_1}, k_{e_2}, k_{e_3}, c_{e_1}(2)\} \\
k_{\texttt{let } x:=e_1 \texttt{ return } e_2} &:= \max\{k_{e_1}, c_{e_1}(k_{e_2}) + k_{e_2}\} \\
k_{f(e_1,\ldots,e_p)} &:= \max\{k_{e_1}, \ldots, k_{e_p}, k_f + c_{e_1}(k_f) + \cdots + c_{e_p}(k_f)\} \\
k_{\texttt{for } x \texttt{ in } e_1 \texttt{ return } e_2} &:= \max\{k_{e_1}, c_{e_1}(k_{e_2} + 1)\}
\end{aligned}$$

Here, $c_{e'}$ denotes a witness for the locality of $e' \in \mathrm{QL}(B)$, which exists by Proposition 44. Since by the same proposition an arithmetic expression defining $c_{e'}$ is moreover computable from e', it follows that k_e is effectively computable from e. Let v be a context such that $e(v)$ is undefined. We will prove by induction on e that there exists a reason $\mathbf{v}$ why $e(v) = \emptyset$ of size at most k_e. During our induction we will often use the fact that every expression $e' \in \mathrm{QL}(B)$ defines a monotone, local base operation by Propositions 6, 25, and 44. We will also use that fact that if $e' \in \mathrm{QL}(B)$ is undefined on input v', then e' is also undefined on every input node-isomorphic to v' by Lemma 24.

- If $e = x$, $e = a$ or $e = ()$, then there is nothing to prove since $e(v)$ is always defined.
- If $e = \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$, then we make a case distinction.

- Case $e_1(v) = \emptyset$. By the induction hypothesis there then exists a reason $\mathbf{v}$ why $e_1(v) = \emptyset$ of size at most k_{e_1}. We claim that $\mathbf{v}$ is also a reason why $e(v) = \emptyset$. Indeed, let $v' \in [\mathbf{v}, v]$. Since $e_1(v')$ is then undefined, $e(v')$ is also undefined, as desired.
- Case $e_1(v) \neq \emptyset$ and there exists $(\Sigma_1; s_1) \in e_1(v)$ with $s_1 = \langle\rangle$ or $s_1 = \langle a \rangle$ with a not a boolean. Since e_1 is a semi-function, it follows that every value in $e_1(v)$ is of this form. Then take $\mathbf{v} = (\emptyset; \emptyset)$. Obviously, $\mathbf{v}$ is a requirement on v of size zero. We claim that $\mathbf{v}$ is a reason why $e(v) = \emptyset$. Indeed, let $v' \in [\mathbf{v}, v]$. If $e_1(v')$ is undefined then $e(v')$ is also undefined, in which case we are done. Hence, suppose that $e_1(v')$ is defined. Since e_1 is a monotone base operation; since $(\Sigma_1; s_1) \in e_1(v)$; since $v' \sqsubseteq v$; and since $e_1(v') \neq \emptyset$, there exists $(\Sigma_1'; s_1') \sqsubseteq (\Sigma_1; s_1)$ in $e_1(v')$. Since $s_1' \sqsubseteq s_1$ it follows that either $s_1' = \langle\rangle$ or $s_1' = \langle a \rangle$. Since e_1 is a semi-function, it follows that all values in $e_1(v')$ are of this form. Hence, $e(v') = \emptyset$, as desired.
- Case $e_1(v) \neq \emptyset$ and there exists $(\Sigma_1; \langle \texttt{true} \rangle) \in e_1(v)$. Since e_1 is a semi-function, it follows that every value in $e_1(v)$ is of this form. Hence $e_2(v) = e(v) = \emptyset$. By the induction hypothesis there then exists a reason $\mathbf{v}$ why $e_2(v) = \emptyset$ of size at most k_{e_2}. We claim that $\mathbf{v}$ is also a reason why $e(v) = \emptyset$. Indeed, let $v' \in [\mathbf{v}, v]$. If $e_1(v')$ is undefined then $e(v')$ is also undefined, in which case we are done. Hence suppose that $e_1(v')$ is defined. Since e_1 is a monotone base operation; since $(\Sigma_1; \langle \texttt{true} \rangle) \in e_1(v)$; since $v' \sqsubseteq v$; and since $e_1(v') \neq \emptyset$, there exists $(\Sigma_1'; s_1') \sqsubseteq (\Sigma_1; \langle \texttt{true} \rangle)$ in $e_1(v')$. In particular we have $s_1' \sqsubseteq \langle \texttt{true} \rangle$. If $s_1' = \langle\rangle$, then every value in $e_1(v')$ is of the form $(\Sigma_1''; \langle\rangle)$ since e_1 is a semi-function. Hence, $e(v')$ is undefined in that case. If on the other hand $s_1' = \langle \texttt{true} \rangle$, then every value in $e_1(v')$ is of the form $(\Sigma_1''; \langle \texttt{true} \rangle)$ since e_1 is a semi-function. Hence, $e(v') = e_2(v')$. Since $\mathbf{v}$ is a reason why $e_2(v) = \emptyset$ and since $v' \in [\mathbf{v}, v]$, it follows that $e(v') = e_2(v') = \emptyset$, as desired.
- Case $e_1(v) \neq \emptyset$ and there exists $(\Sigma_1; \langle \texttt{false} \rangle) \in e_1(v)$. Since e_1 is a semi-function, it follows that every value in $e_1(v)$ is of this form. Hence, $e_3(v) = e(v) = \emptyset$. By the induction hypothesis there then exists a reason $\mathbf{v}$ why $e_2(v) = \emptyset$ of size at most k_{e_3}. By a reasoning similar to the previous case it can be seen that $\mathbf{v}$ is also a reason why $e(v) = \emptyset$.
- Case $e_1(v) \neq \emptyset$ and there exists $(\Sigma_1; s_1) \in e_1(v)$ with $|s_1| \geq 2$. It is easy to see that $(\emptyset; \{1, 2\})$ is a requirement on $(\Sigma_1; s_1)$ of size two. By Proposition 44 there exists a reason $\mathbf{v}$ why

$$[(\emptyset; \{1, 2\}), (\Sigma_1; s_1)] \triangleleft e_1(v)$$

of size at most $c_{e_1}(2)$. We claim that $\mathbf{v}$ is a also reason why $e(v) = \emptyset$. Indeed, let $v' \in [\mathbf{v}, v]$. If $e_1(v')$ is undefined then $e(v')$ is also undefined, in which case we are done. Hence, suppose that $e_1(v')$ is defined. Then $e_1(v') \cap [(\emptyset; \{1,2\}), (\Sigma_1; s_1)] \neq \emptyset$. Let $(\Sigma_1'; s_1')$ be a value in this non-empty intersection. It follows by Lemma 33 that $|s_1'| \geq 2$. Since e_1 is a semi-function, it follows that all values in $e_1(v')$ are of this form. Hence, $e(v') = \emptyset$, as desired.

Hence, there always exists a reason $\mathbf{v}$ why $e(v) = \emptyset$ of size at most $\max\{k_{e_1}, k_{e_2}, k_{e_3}, c_{e_1}(2)\}$, as desired.

- If $e = \texttt{let}\ \ x := e_1\ \ \texttt{return}\ \ e_2$, then we make a case distinction.
  - Case $e_1(v) = \emptyset$. By the induction hypothesis there exists a reason $\mathbf{v}$ why $e_1(v) = \emptyset$ of width at most k_{e_1}. It is easily seen that $\mathbf{v}$ is also a reason why $e(v) = \emptyset$.
  - Case $e_1(v) \neq \emptyset$. Let $(\Sigma; \sigma) = v$ and let $(\Sigma_1; s_1) \in e_1(v)$. Since $e(v)$ is undefined it follows that $e_2(\Sigma_1; x\colon s_1, \sigma)$ is undefined. By the induction hypothesis there hence exists a reason $(V_1; x\colon P_1, \phi_1)$ why this is so of size at most k_{e_2}. In particular we have that $(V_1; P_1)$ is a requirement on $(\Sigma_1; s_1)$ of size at most k_{e_2}. By Proposition 44 there exists a reason $(V; \phi)$ why

    $$[(V_1; P_1), (\Sigma_1; s_1)] \triangleleft e_1(v)$$

    of size at most $c_{e_1}(k_{e_2})$. Let ϕ' be the function with domain $dom(\sigma)$ defined by

    $$\phi'(y) := \phi(y) \cup \phi_1(y),$$

    and let $\mathbf{v} = (V; \phi')$. It is easy to see that $\mathbf{v}$ is a requirement on v. Moreover,

    $$\begin{aligned} |\mathbf{v}| &= \max\left\{|V|, |\phi'(x)| \;\middle|\; x \in dom(\sigma)\right\} \\ &\leq \max\{c_{e_1}(k_{e_2}), c_{e_1}(k_{e_2}) + k_{e_2}\} \\ &= c_{e_1}(k_{e_2}) + k_{e_2}. \end{aligned}$$

    We claim that $\mathbf{v}$ is a reason why $e(v) = \emptyset$. Indeed, let $v' = (\Sigma'; \sigma') \in [\mathbf{v}, v]$. If $e_1(v')$ is undefined then $e(v')$ is also undefined, in which case we are done. Hence, suppose that $e_1(v')$ is defined. Since $(V; \phi)$ is a reason why

    $$[(V_1; P_1), (\Sigma_1; s_1)] \triangleleft e_1(v)$$

    and since $v' \in [(V; \phi'), v] \subseteq [(V; \phi), v]$, it follows that

    $$e_1(v') \cap [(V_1; P_1), (\Sigma_1; s_1)] \neq \emptyset.$$

Let $(\Sigma_1'; s_1')$ be a value in this non-empty intersection. Then clearly

$$\begin{aligned}(\Sigma_1'; x\colon s_1', \sigma') &\in [(V_1; x\colon P_1, \phi'), (\Sigma_1; x\colon s_1, \sigma)] \\ &\subseteq [(V_1; x\colon P_1, \phi_1), (\Sigma_1; x\colon s_1, \sigma)].\end{aligned}$$

Since $(V_1; x\colon P_1, \phi_1)$ is a reason why $e_2(\Sigma_1; x\colon s_1, \sigma) = \emptyset$, it follows that also $e_2(\Sigma_1'; x\colon s_1', \sigma') = \emptyset$. Furthermore, since e_1 is a base operation it follows by Corollary 12 that for every other value $(\Sigma_1''; s_1'')$ in $e_1(\Sigma'; \sigma')$ we have

$$(\Sigma_1''; x\colon s_1'', \sigma') \equiv_{node} (\Sigma_1'; x\colon s_1', \sigma').$$

Since e_2 is node-generic and since $e_2(\Sigma_1'; x\colon s_1', \sigma')$ is undefined, it follows that $e_2(\Sigma_1''; x\colon s_1'', \sigma')$ is also undefined for every other value $(\Sigma_1''; s_1'')$ in $e_1(\Sigma'; \sigma')$. Hence, $e(v') = \emptyset$, as desired.

Hence, there always exists a reason $\mathbf{v}$ why $e(v) = \emptyset$ of size at most $\max\{k_{e_1}, c_{e_1}(k_{e_2}) + k_{e_2}\}$, as desired.

- If $e = f(e_1, \ldots, e_p)$, then we make a case distinction.
  - Case $e_j(\Sigma; \sigma) = \emptyset$ for some $j \in [1, p]$. By the induction hypothesis there exists a reason $\mathbf{v}$ why this is so of width at most k_{e_j}. It is easily seen that $\mathbf{v}$ is also a reason why $e(v) = \emptyset$.
  - Case $e_j(\Sigma; \sigma) \neq \emptyset$ for all $j \in [1, p]$. Let $(\Sigma; \sigma) = v$. Since every e_j is node-generic and store-increasing there certainly exist $(\Sigma \circ \Sigma_j; s_j) \in e_j(v)$ for every $j \in [1, p]$ such that the Σ_j are pairwise disjoint. Since $e(v)$ is undefined it follows that

    $$f(\Sigma \circ \bigcirc_{j=1}^{p} \Sigma_j; s_1, \ldots, s_p) = \emptyset.$$

    Since f is a locally-undefined base operation with witness k_f there hence exists a reason $(V \cup \bigcup_{j=1}^{p} V_j; P_1, \ldots, P_p)$ why this is so of size at most k_f. Here, V is a subset of the nodes in Σ and V_j is a subset of the nodes in Σ_j, for every $j \in [1, p]$. We have in particular that $(V \cup V_j; P_j)$ is a requirement on $(\Sigma \circ \Sigma_j; s_j)$ of size at most k_f. By Proposition 44 there hence exists for every $j \in [1, p]$ a reason $(W_j; \phi_j)$ why

    $$[(V \cup V_j; P_j), (\Sigma \circ \Sigma_j; s_j)] \triangleleft e_j(v)$$

    of size at most $c_{e_j}(k_f)$. Let $\mathbf{v}$ be the requirement $(V \cup \bigcup_{j=1}^{p} W_j; \phi)$ on v such that the function ϕ with domain $dom(\sigma)$ is defined by

    $$\phi(x) := \bigcup_{j=1}^{p} \phi_j(x) \quad \text{for all } x.$$

Clearly,

$$|\mathbf{v}| \leq \max\left\{ |V| + \sum_{j=1}^{p} |W_j|, \sum_{j=1}^{p} |\phi_j(x)| \;\middle|\; x \in dom(\sigma) \right\}$$
$$\leq k_f + \sum_{j=1}^{p} c_{e_j}(k_f).$$

Moreover, since $[\mathbf{v}, v] \subseteq [(W_j; \phi_j), v]$ for every $j \in [1, p]$, $\mathbf{v}$ is a reason why

$$[(V \cup V_j; P_j), (\Sigma \circ \Sigma_j; s_j)] \triangleleft e_j(v).$$

We claim that $\mathbf{v}$ is a also reason why $e(v) = \emptyset$. Indeed, let $v' = (\Sigma'; \sigma') \in [\mathbf{v}, v]$. If $e_j(v')$ is undefined for some $j \in [1, p]$ then $e(v')$ is also undefined, in which case we are done. Hence, suppose that $e_j(v')$ is defined for all $j \in [1, p]$. Since e_j is a base operation; since $\mathbf{v}$ is a reason why

$$[(V \cup V_j; P_j), (\Sigma \circ \Sigma_j; s_j)] \triangleleft e_j(v);$$

and since $\mathbf{v}$ contains all nodes of $V \cup V_j$ in Σ, it follows from Lemma 43 that for every $j \in [1, p]$ there exists

$$(\Sigma' \circ \Sigma'_j; s'_j) \in e_j(v') \cap [(V \cup V_j; P_j), (\Sigma \circ \Sigma_j; s_j)],$$

with $\Sigma'_j \sqsubseteq \Sigma_j$. Since $\Sigma'_j \sqsubseteq \Sigma_j$ and since the Σ_j are all pairwise disjoint, it follows that the Σ'_j are also all pairwise disjoint. Since $V \cup V_j$ is a subset of the nodes in $\Sigma' \circ \Sigma'_j$, it follows that $V \cup \bigcup_{j=1}^{p} V_j$ is a subset of the nodes in $\Sigma' \circ \bigcirc_{j=1}^{p} \Sigma'_j$. Hence,

$$(\Sigma' \circ \bigcirc_{j=1}^{p} \Sigma'_j; s'_1, \ldots, s'_p) \in$$
$$[(V \cup \bigcup_{j=1}^{p} V_j; P_1, \ldots, P_p), (\Sigma \circ \bigcirc_{j=1}^{p} \Sigma_j; s_1, \ldots, s_p)].$$

Since $(V \cup \bigcup_{j=1}^{p} V_j; P_1, \ldots, P_p)$ is a reason why

$$f(\Sigma \circ \bigcirc_{j=1}^{p} \Sigma_j; s_1, \ldots, s_p) = \emptyset,$$

it follows that hence

$$f(\Sigma' \circ \bigcirc_{j=1}^{p} \Sigma'_j; s'_1, \ldots, s'_p) = \emptyset. \tag{12}$$

Furthermore, since every e_j is a node-generic it follows from Lemma 11 that for every $j \in [1, p]$ and every $(\Sigma' \circ \Sigma''_j; s''_j)$ in $e_j(v')$ for which the Σ''_j are disjoint we have

$$(\Sigma' \circ \bigcirc_{j=1}^{p} \Sigma''_j; s''_1, \ldots, s''_p) \equiv_{node} (\Sigma' \circ \bigcirc_{j=1}^{p} \Sigma'_j; s'_1, \ldots, s'_p). \tag{13}$$

Since f is node-generic it follows by (12) and (13) that

$$f(\Sigma' \circ \bigcirc_{j=1}^{p} \Sigma''_j; s''_1, \ldots, s''_p) = \emptyset$$

for all $(\Sigma' \circ \Sigma''_j; s''_j)$ in $e_j(v')$ for which the Σ''_j are disjoint. Hence, $e(v') = \emptyset$, as desired.

Hence, there always exists a reason $\mathbf{v}$ why $e(v) = \emptyset$ of size at most $\max\{k_{e_1}, \ldots, k_{e_p}, c_{e_1}(k_f) + \cdots + c_{e_p}(k_f)\}$, as desired.

- If $e = \texttt{for}\ \ x\ \ \texttt{in}\ \ e_1\ \ \texttt{return}\ \ e_2$ then we make a case distinction.
  - Case $e_1(v) = \emptyset$. By the induction hypothesis there exists a reason $\mathbf{v}$ why this is so of width at most k_{e_1}. It is easily seen that $\mathbf{v}$ is also a reason why $e(v) = \emptyset$.
  - Case $e_1(v) \neq \emptyset$. Let $(\Sigma; \sigma) = v$ and let $(\Sigma_1; s) \in e_1(v)$. Since $e(v)$ is undefined it follows that there exists $j \in [1, |s|]$ such that $e_2(\Sigma_1; x\colon \langle s(j)\rangle, \sigma)$ is undefined. By the induction hypothesis there hence exists a reason $(V_1; x\colon P_1, \phi_1)$ why this is so of size at most k_{e_2}. In particular, $(V_1; \{j\})$ is a requirement on $(\Sigma_1; s)$ of size at most $\max\{k_{e_2}, 1\}$. By Proposition 44 there hence exists a reason $(V; \phi)$ why

    $$[(V_1; \{j\}), (\Sigma_1; s)] \triangleleft e_1(v)$$

    of size at most $c_{e_1}(\max\{k_{e_2}, 1\})$. Let ϕ' be the function with domain $dom(\sigma)$ defined by

    $$\phi'(y) := \phi(y) \cup \phi_1(y),$$

    and let $\mathbf{v} = (V; \phi')$. It is easy to see that $\mathbf{v}$ is a requirement on v. Moreover,

    $$\begin{aligned} |\mathbf{v}| &= \max\left\{|V|, |\phi'(x)| \;\middle|\; x \in dom(\sigma)\right\} \\ &\leq \max\{c_{e_1}(\max\{k_{e_2}, 1\}), c_{e_1}(\max\{k_{e_2}, 1\}) + k_{e_2}\} \\ &= c_{e_1}(\max\{k_{e_2}, 1\}) + k_{e_2}. \end{aligned}$$

    We claim that $\mathbf{v}$ is a reason why $e(v) = \emptyset$. Indeed, let $v' = (\Sigma'; \sigma') \in [\mathbf{v}, v]$. If $e_1(v')$ is undefined then $e(v')$ is also undefined, in which case we are done. Hence suppose that $e_1(v')$ is defined. Since $(V; \phi)$ is a reason why

    $$[(V_1; \{j\}), (\Sigma_1; s)] \triangleleft e_1(v)$$

    and since $v' \in [(V; \phi'), v] \subseteq [(V; \phi), v]$ it follows that

    $$e_1(v') \cap [(V_1; \{j\}), (\Sigma_1; s)] \neq \emptyset.$$

Let $(\Sigma_1'; s')$ be a value in this non-empty intersection. It follows by Lemma 33 that $s(j) \in rng(s')$. There hence exists $i \in [1, |s'|]$ such that $s'(i) = s(j)$. Since hence $\langle s'(i) \rangle = \langle s(j) \rangle$ and since $P_1 \subseteq [1, |\langle s'(j) \rangle|] = \{1\}$, it follows that the identity function is a witness of $\langle s'(i) \rangle \sqsubseteq \langle s(j) \rangle$ whose range includes P_1. Then clearly

$$\begin{aligned}(\Sigma_1'; x\colon \langle s'(i) \rangle, \sigma') &\in [(V_1; x\colon P_1, \phi'), (\Sigma_1; x\colon \langle s(j) \rangle, \sigma)] \\ &\subseteq [(V_1; x\colon P_1, \phi_1), (\Sigma_1; x\colon s, \sigma)].\end{aligned}$$

Since $(V_1; x\colon P_1, \phi_1)$ is a reason why $e_2(\Sigma_1; x\colon \langle s(j) \rangle, \sigma) = \emptyset$, it follows that also $e_2(\Sigma_1'; x\colon \langle s'(i) \rangle, \sigma') = \emptyset$. Furthermore, since e_1 is a node-generic it follows by Corollary 13 that for every other value $(\Sigma_1''; s'')$ in $e_1(\Sigma'; \sigma')$ we have that $|s''| = |s'|$ and

$$(\Sigma_1''; x\colon \langle s''(i) \rangle, \sigma') \equiv_{node} (\Sigma_1'; x\colon \langle s'(i) \rangle, \sigma').$$

Since e_2 is node-generic and since $e_2(\Sigma_1'; x\colon \langle s'(i) \rangle, \sigma')$ is undefined, it follows that $e_2(\Sigma_1''; x\colon \langle s''(i) \rangle, \sigma')$ is also be undefined for every other value $(\Sigma_1''; s'')$ in $e_1(\Sigma'; \sigma')$. Hence $e(v') = \emptyset$, as desired.

Hence there always exists a reason $\mathbf{v}$ why $e(v) = \emptyset$ of size at most $\max\{k_{e_1}, c_{e_1}(\max\{k_{e_2}, 1\}) + k_{e_2}\}$, as desired. □

8 Decidability results

The restrictions proposed in Sections 5, 6, and 7 are strong enough to guarantee decidability of well-definedness:

Theorem 46. *If B is a finite set of monotone, generic, local, and locally-undefined base operations, then the well-definedness problem for $QL(B)$ is decidable.*

In order to prove this theorem, we first introduce the following notions.

Definition 47. The *size* $|\Sigma|$ of a store Σ is the number of nodes in Σ. The *size* $|(\Sigma; s_1, \ldots, s_p)|$ of a value-tuple $(\Sigma; s_1, \ldots, s_p)$ is the sum

$$|\Sigma| + |s_1| + \cdots + |s_p|.$$

Lemma 48. *For every type τ there exists a computable function c_τ mapping natural numbers to natural numbers such that for every $w \in \tau$ and every requirement $\mathbf{w}$ on w there exists $v \in [\mathbf{w}, w] \cap \tau$ of size at most $c_\tau(|\mathbf{w}|)$. Moreover, an arithmetic expression defining c_τ is effectively computable from τ.*

Proof. Let $c_\tau(k)$ be defined by induction on τ as follows:

$$\begin{aligned}
c_{\mathbf{atom}}(k) &:= 1 \\
c_{\mathbf{text}}(k) &:= 1 \\
c_{\mathbf{element}}(a, \tau')(k) &:= 1 + c_{\tau'}(k) \\
c_{\mathbf{empty}}(k) &:= 0 \\
c_{\tau_1 + \tau_2}(k) &:= \max\{c_{\tau_1}(k), c_{\tau_2}(k)\} \\
c_{\tau_1 \circ \tau_2}(k) &:= c_{\tau_1}(k) + c_{\tau_2}(k) \\
c_{\tau'*}(k) &:= 2k c_{\tau'}(k)
\end{aligned}$$

It is clear from this inductive definition that an arithmetic expression defining c_τ can effectively be computed from τ. It is also clear that c_τ is a computable function mapping natural numbers to natural numbers. Let $w \in \tau$ and let $\mathbf{w} = (V; P)$ be a restriction on w. We prove that there exists $v \in [\mathbf{w}, w] \cap \tau$ of size at most $c_\tau(k)$ by induction on τ:

- Case $\tau = \mathbf{atom}$. Since $w \in \tau$ we know that $w = (\varnothing; \langle a \rangle)$. Then clearly $w \in [\mathbf{w}, w]$. The result then follows since $|w| = 1$.
- The cases where $\tau = \mathbf{text}$ or $\tau = \mathbf{empty}$ are similar.
- Case $\tau = \mathbf{element}(a, \tau')$. Since $w \in \tau$, we know that w is of the form $(\Theta; \langle n \rangle)$ with Θ a tree such that n is the root element node of Θ which is labeled by a. Furthermore, if $n_1, \ldots, n_p$ are the children of n in Θ in document order, we have $(\Theta|_{n_1} \circ \cdots \circ \Theta|_{n_p}; \langle n_1, \ldots, n_p \rangle) \in \tau'$. Let $V' = V \setminus \{n\}$. It is clear that then $(V'; \emptyset)$ is a requirement on $(\Theta|_{n_1} \circ \cdots \circ \Theta|_{n_p}; \langle n_1, \ldots, n_p \rangle)$ of size at most $|\mathbf{w}|$. By the induction hypothesis there exists

  $$(\Sigma; \langle n'_1, \ldots, n'_{p'} \rangle) \in [(V'; \emptyset), (\Theta|_{n_1} \circ \cdots \circ \Theta|_{n_p}; \langle n_1, \ldots, n_p \rangle)] \cap \tau',$$

  of size at most $c_{\tau'}(|\mathbf{w}|)$. Since $n'_1, \ldots, n'_{p'}$ are hence all nodes and since $(\Sigma; \langle n'_1, \ldots, n'_{p'} \rangle) \in \tau'$, it is easy to see by another induction on τ' that Σ is of the form $\Theta'_1 \circ \ldots \circ \Theta'_{p'}$ such that Θ'_j is a tree with root node n'_j for every $j \in [1, p']$. Since $\Theta'_1 \circ \ldots \circ \Theta'_{p'} \sqsubseteq \Theta|_{n_1} \circ \cdots \circ \Theta|_{n_p}$, and since n is not a node in $\Theta|_{n_1} \circ \cdots \circ \Theta|_{n_p}$, n cannot be a node in $\Theta'_1 \circ \ldots \circ \Theta'_{p'}$. Then let Θ' be the tree with a-labeled root node n in which n has children $n'_1, \ldots, n'_{p'}$ such that $n'_1 <_{\Theta'} \ldots <_{\Theta'} n'_{p'}$ and $\Theta'|_{n'_j} = \Theta'_j$ for every $j \in [1, p']$. Now define $v := (\Theta'; \langle n \rangle)$. Then

  $$|v| = |\Theta'_1 \circ \cdots \circ \Theta'_{p'}| + 1 \leq c_{\tau'}(|\mathbf{w}|) + 1 = c_\tau(|\mathbf{w}|).$$

  We claim that $v \in [\mathbf{w}, w] \cap \tau$. Indeed, it is easy to see that $(\Theta'; \langle n \rangle) \in \tau$. Furthermore, since $\Theta'_1 \circ \ldots \circ \Theta'_{p'}$ contains all nodes in V', it follows that Θ' contains all nodes in V. Since $P \subseteq [1, |\langle n \rangle|] = \{1\}$, it is easy

to see that the identity function is a witness of $\langle n \rangle \sqsubseteq \langle n \rangle$ whose range certainly contains P. It is also easy to see that $\Theta' \sqsubseteq \Theta$, and hence $v \in [\mathbf{w}, w] \cap \tau$, as desired.

- If $\tau = \tau_1 + \tau_2$, then $w \in \tau_1$ or $w \in \tau_2$. In both cases the result follows immediately from the induction hypothesis.

- Case $\tau = \tau_1 \circ \tau_2$. Since $w \in \tau$ we know that w is of the form $(\Sigma_1 \circ \Sigma_2; s_1 \circ s_2)$ with $(\Sigma_1; s_1) \in \tau_1$ and $(\Sigma_2; s_2) \in \tau_2$. Let V_1, V_2 be the partition of V such that V_1 is a subset of the nodes in Σ_1 and V_2 is a subset of the nodes in Σ_2. Let P_1 and P_2 be the subsets of P defined by

$$\begin{aligned} P_1 &= \{k \mid k \in P \text{ and } 1 < k \leq |s_1|\} \\ P_2 &= \{k - |s_1| \mid k \in P \text{ and } |s_1| < k \leq |s|\}. \end{aligned}$$

  It is clear that $(V_1; P_1)$ and $(V_2; P_2)$ are requirements on $(\Sigma_1; s_1)$ respectively $(\Sigma_2; s_2)$ of size at most $|\mathbf{w}|$. By the induction hypothesis there hence exist

$$\begin{aligned} (\Sigma_1'; s_1') &\in [(V_1, P_1), (\Sigma_1; s_1)] \cap \tau_1 \\ (\Sigma_2'; s_2') &\in [(V_2, P_2), (\Sigma_2; s_2)] \cap \tau_2 \end{aligned}$$

  of size at most $c_{\tau_1}(|\mathbf{w}|)$ respectively $c_{\tau_2}(|\mathbf{w}|)$. Since $\Sigma_1' \sqsubseteq \Sigma_1$, since $\Sigma_2' \sqsubseteq \Sigma_2$, and since Σ_1 is disjoint with Σ_2, it follows that Σ_1' is disjoint with Σ_2'. Let $v = (\Sigma_1' \circ \Sigma_2'; s_1' \circ s_2')$. Then,

$$\begin{aligned} |v| = |\Sigma_1' \circ \Sigma_2'| + |s_1' \circ s_2'| &= |\Sigma_1'| + |\Sigma_2'| + |s_1'| + |s_2'| \\ &\leq c_{\tau_1}|\mathbf{w}| + c_{\tau_2}|\mathbf{w}| = c_\tau(|\mathbf{w}|). \end{aligned}$$

  We claim that $v \in [\mathbf{w}, w] \cap \tau$. Indeed, it is easy to see that $v \in \tau$. Moreover, since $\Sigma_1' \sqsubseteq \Sigma_1$ and $\Sigma_2' \sqsubseteq \Sigma_2$ it follows by Lemma 42 that $v \in [(V; P), (\Sigma_1 \circ \Sigma_2; s_1 \circ s_2)]$.

- Case $\tau = \tau'^*$. Since $w \in \tau$ we know that w is of the form

$$(\bigcirc_{j=1}^{p} \Sigma_j; \bigcirc_{j=1}^{p} s_j)$$

  for some $p \geq 0$ such that $(\Sigma_j; s_j) \in \tau$ for every $j \in [1, p]$. Let, $V_1, \ldots, V_p$ be the partition of V such that V_j is a subset of the nodes in Σ_j for every $j \in [1, p]$. Let for each $j \in [1, p]$, P_j be the subset of P defined by

$$P_j := \left\{ k - \sum_{i=1}^{j-1} |s_i| \;\middle|\; k \in P \text{ and } \sum_{i=1}^{j-1} |s_i| < k \leq \sum_{i=1}^{j} |s_i| \right\}.$$

It is clear that $(V_j; P_j)$ is a requirement on $(\Sigma_j; s_j)$ of size at most $|\mathbf{w}|$ for every $j \in [1, p]$. Let J be the set of j in $[1, p]$ for which $V_j \neq \emptyset$ or $P_j \neq \emptyset$. By the induction hypothesis there exists, for every $j \in J$, a value

$$(\Sigma'_j; s'_j) \in [(V_j; P_j), (\Sigma_j; s_j)] \cap \tau'$$

of size at most $c_{\tau'}(|\mathbf{w}|)$. Then let $v = (\bigcirc_{j \in J} \Sigma'_j; \bigcirc_{j \in J} s'_j)$. Note that there can be at most $|\mathbf{w}|$ of the V_j non-empty and that there can be at most $|\mathbf{w}|$ of the P_j non-empty. Hence, J contains at most $2|\mathbf{w}|$ elements. Hence,

$$|v| = \sum_{j \in J} |\Sigma'_j| + \sum_{j \in J} |s'_j| = \sum_{j \in J} (|\Sigma'_j| + |s'_j|) \leq \sum_{j \in J} c_{\tau'}(|\mathbf{w}|)$$
$$\leq 2|\mathbf{w}| c_{\tau'}(|\mathbf{w}|) = c_\tau(|\mathbf{w}|).$$

We claim that $v \in [\mathbf{w}, w] \cap \tau$. Indeed, it is easy to see that $v \in \tau$. Furthermore, let $\Sigma'_j = \varnothing$ and $s'_j = \langle\rangle$ for every $j \in [1, p] \setminus J$. Since $V_j = \emptyset$ and $P_j = \emptyset$ for $j \notin J$, we have in particular that for such j:

$$(\Sigma'_j; s'_j) \in [(V_j; P_j), (\Sigma_j; s_j)].$$

Since by construction we then have $\Sigma'_j \sqsubseteq \Sigma_j$ for every $j \in [1, p]$, it follows by Lemma 42 that

$$(\bigcirc_{j=1}^{p} \Sigma'_j; \bigcirc_{j=1}^{p} s'_j) \in [(V; P), (\bigcirc_{j=1}^{p} \Sigma_j; \bigcirc_{j=1}^{p} s_j)].$$

Hence, $v \in [\mathbf{w}, w] \cap \tau$, as desired. □

We are now ready for:

Proof of Theorem 46. Suppose that $e \in \mathrm{QL}(B)$ is not well-defined under type assignment Γ on e. Then there exists some context $w \in \Gamma$ such that $e(w) = \emptyset$. By Proposition 45 there exists a natural number k, computable from e, and a requirement $\mathbf{w}$ on w of size at most k such that $e(v) = \emptyset$ for all $v \in [\mathbf{w}, w]$. By Lemma 48 there exists $v \in [\mathbf{w}, w] \cap \Gamma$ of size at most

$$l := \sum_{x \in dom(\Gamma)} c_{\Gamma(x)}(k).$$

Here, $c_{\Gamma(x)}$ is a function for which a defining arithmetic expression is computable from $\Gamma(x)$, for every $x \in dom(\Gamma)$. Hence, l is computable from e and Γ. It is easy to see that v contains at most l nodes and that v can mention at most l different atoms. Let N be a set of nodes consisting of l element nodes and l text nodes. Let A be a set of atoms containing all constants mentioned in e and l other atoms. Then surely there exists a renaming ρ which is the identity on constants in e such that $\rho(v)$ contains only nodes

in N and mentions only atoms in A. By Proposition 27, $e(\rho(v))$ is also undefined.

Hence, in order to check if e is well-defined under Γ, it suffices to enumerate all contexts $v' \in \Gamma$ of size at most l with nodes in N and atoms in A, and check whether $e(v')$ is defined. There are only a finite number of such v', from which the result follows. □

Since all base operations mentioned in Section 3, except *data*, *merge-text*, and *empty* are monotone, generic, locally-undefined and local by Propositions 21, 26, 35, and 41, it follows in particular:

Corollary 49. *Well-definedness for the XQuery fragment QL(concat, children, descendant, parent, ancestor, preceding-sibling, following-sibling, eq, is, ≪, is-element, is-text, is-atom, node-name, content, element, text) is decidable.*

It follows from Proposition 17 that satisfiability for this fragment is also decidable. In contrast, the *semantic type-checking problem* (i.e., is, for every input in a given input type, the output of a given expression always in a given output type) for this fragment is known to be undecidable [3].

8.1 Satisfiability for QL(concat, smaller-width)

Remember from Section 7.1 that well-definedness for QL(*concat*, *smaller-width*) is undecidable. Using Proposition 44 we are able to show, however, that the satisfiability problem for QL(*concat*, *smaller-width*) is decidable. Hence decidability of the satisfiability problem does not imply decidability of the well-definedness problem.

Proposition 50. *The satisfiability problem for QL(concat, smaller-width) is decidable.*

Proof. Let e be an expression in QL(B) and let Γ be a type assignment on e such that e is well-defined under Γ. Suppose that e is satisfiable under Γ. Then let w be a context in Γ and let $(\Sigma; s)$ be a value in $e(w)$ such that s is non-empty. Let $\mathbf{w} = (\emptyset; \{1\})$. It is clear that $\mathbf{w}$ is a requirement on $(\Sigma; s)$ of size one. Since *concat* and *smaller-width* are monotone and local by Propositions 21 and 35, it follows from Proposition 44 that e is local and that an arithmetic expression defining a witness c of this locality can effectively be computed from e. In particular there hence exists a reason $\mathbf{w}$ why $[(\emptyset; \{1\}), (\Sigma; s)] \triangleleft e(w)$ of size at most $c(1)$. By Lemma 48 there then exists $v \in [\mathbf{w}, w] \cap \Gamma$ of size at most

$$l := \sum_{x \in dom(\Gamma)} c_{\Gamma(x)}(c(1)).$$

Here, $c_{\Gamma(x)}$ is a function for which a defining arithmetic expression is computable from $\Gamma(x)$, for every $x \in dom(\Gamma)$. Hence l is computable from e and Γ. Since e is well-defined under Γ, it follows that e is defined on v. Furthermore, since $v \in [\mathbf{w}, w]$ and since $\mathbf{w}$ is reason why $[(\emptyset; \{1\}), (\Sigma; s)] \triangleleft e(w)$, it follows that $e(v) \cap [(\emptyset; \{1\}), (\Sigma; s)] \neq \emptyset$. Let $(\Sigma'; s')$ be a value in this non-empty intersection. Then it follows from Lemma 33 that $|s'| \geq 1$. Hence s' is non-empty.

It is easy to see that v contains at most l nodes and that v can mention at most l different atoms. Let N be a set of nodes consisting of l element nodes and l text nodes. Let A be a set of atoms containing all constants mentioned in e and l other atoms. Then surely there exists a renaming ρ which is the identity on constants in e such that $\rho(v)$ contains only nodes in N and mentions only atoms in A. It is easy to see that *concat* and *smaller-width* are both generic. By Proposition 27 it hence follows that $(\rho(\Sigma'); \rho(s')) \in e(\rho(v))$. Since renamings do not alter the width of a list, it follows that $\rho(s')$ is non-empty. Furthermore, since e is a semi-function by Proposition 6, it follows that every value in $e(\rho(v))$ has a non-empty list.

Hence, in order to check if e is satisfiable under Γ it suffices to enumerate all contexts $v' \in \Gamma$ of size at most l with nodes in N and atoms in A, and check whether some $e(v')$ contains a value with a non-empty list. There are only a finite number of such v', from which the result follows. □

8.2 Well-definedness for the Nested Relational Calculus over lists

In a companion paper [37] we study the well-definedness problem for the Nested Relational Calculus (NRC), a well-known query language for the complex object data model [1, 8, 38]. Specifically, we study the well-definedness problem for the NRC in the standard, set-based, complex object data model. We obtain that the problem is undecidable for the NRC in general, but is decidable for the positive-existential fragment of the NRC (PENRC for short). Next, we study well-definedness for the PENRC in the presence of the singleton coercion operator *extract*. This operator extracts v from a singleton set $\{v\}$ and is undefined on non-singleton inputs. Our study revealed that this operator causes the well-definedness problem to become undecidable again. The core difficulty here is the fact that $extract(\{e_1, e_2\})$ is defined if, and only if, expressions e_1 and e_2 return the same result on every input. As such, in order to solve the well-definedness problem one also needs to solve the equivalence problem, which we show to be undecidable for the PENRC.

Note that, in contrast, the presence of an operator which becomes undefined due to a non-singleton input does not necessarily cause the well-definedness problem for QL(B) to become undecidable. For example, the operation *is-element* mentioned in Corollary 49 is only defined on singleton

inputs. Decidability in the presence of this operation is due to the fact that the difficulty with sets, where $\{e_1, e_2\}$ is a singleton if, and only if, e_1 and e_2 are equivalent, no longer holds for lists. Indeed, in this section we will show that well-definedness for the PENRC with *extract* interpreted in a list-based data model *is* decidable.

List-based NRC data model A (list-based) *NRC-value* is either a basic atom, a pair of values, or a finite list of values. Note that, in contrast to the QL data model, lists can hence contain other lists.

List-based PENRC with singleton coercion The *Positive-Existential Nested Relational Calculus with singleton coercion* (PENRC(*extract*) for short) is the set of all expressions generated by the following grammar:

$$\begin{array}{rcl} e & ::= & x \\ & | & (e, e) \mid \pi_1(e) \mid \pi_2(e) \\ & | & \emptyset \mid \{e\} \mid e \cup e \mid \bigcup e \mid \{e \mid x \in e\} \mid \mathit{extract}(e) \\ & | & e = e \ ?\ e \ :\ e \end{array}$$

Here, e ranges over expressions and x ranges over variables. We view expressions as abstract syntax trees and omit parentheses. The set $FV(e)$ of *free variables* of an expression e is defined as usual. That is, $FV(x) := \{x\}$, $FV(\emptyset) := \emptyset$, $FV(\{e_2 \mid x \in e_1\}) := FV(e_1) \cup (FV(e_2) \setminus \{x\})$, and $FV(e)$ is the union of the free variables of e's immediate subexpressions otherwise.

A *list-based NRC-context* is a function σ from a finite set of variables $dom(\sigma)$ to list-based NRC-values. If $dom(\sigma)$ is a superset of $FV(e)$, then we say that σ is a *context on* e. The semantics of PENRC(*extract*) expressions on lists is described by means of the *evaluation relation*, as defined in Figure 7. Here, we write $\sigma \models e \Rightarrow v$ to denote the fact that e evaluates to value v on list-based context σ on e. It is easy to see that the evaluation relation is functional: an expression evaluates to at most one value on a given context. The evaluation relation is not total however. For example, if $\sigma(x)$ is an atom then $\pi_1(x)$ does not evaluate to any value on σ, since π_1 is only defined on pairs. Likewise, we can only concatenate lists, flatten lists of lists, iterate over lists, and test equality on atoms. An expression e can hence be viewed as a partial function from contexts on e to values. We will write $e(\sigma)$ for the unique value v for which $\sigma \models e \Rightarrow v$. If no such value exists, then we say that $e(\sigma)$ is *undefined*.

We note that the semantics of an expression only depends on its free variables: if two contexts σ and σ' on e are equal on $FV(e)$, then $\sigma \models e \Rightarrow v$ if, and only if, $\sigma' \models e \Rightarrow v$.

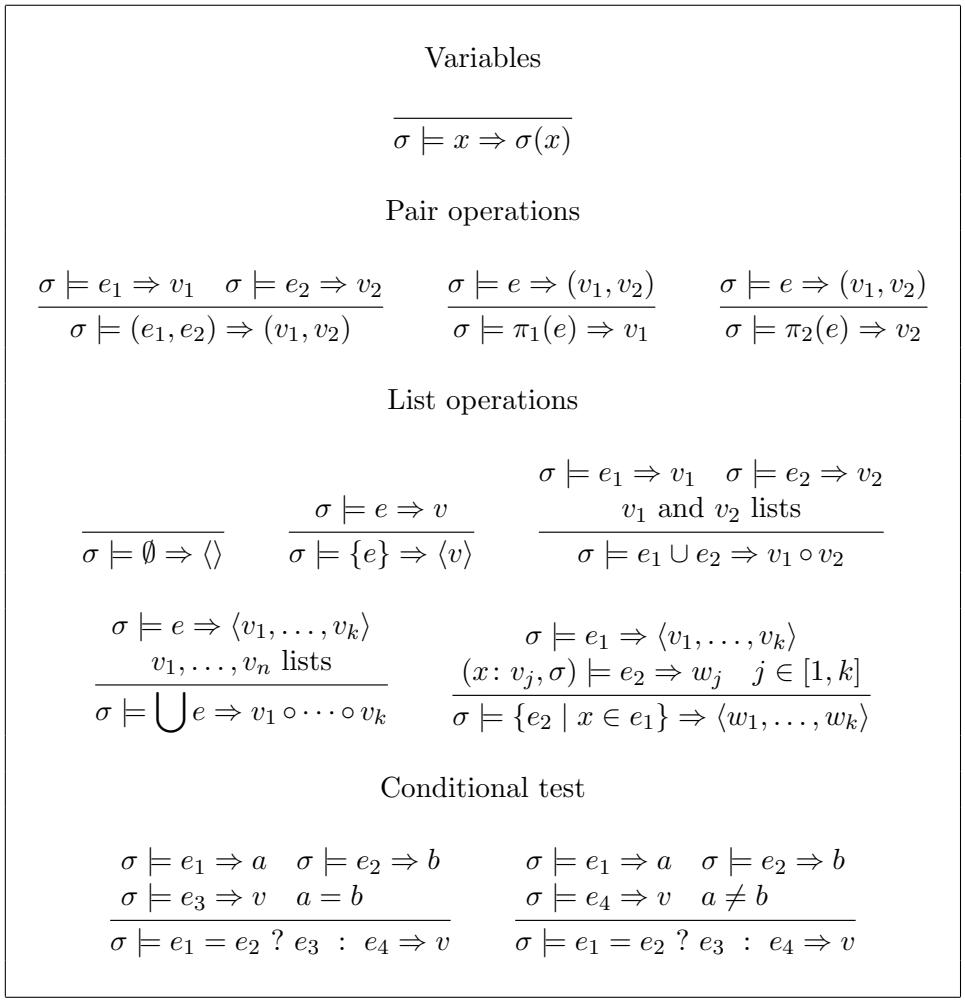


Figure 7: The semantics of PENRC(*extract*) expressions in the list-based NRC data model.

List-based NRC types A *list-based NRC-type* is a term generated by the following grammar:

$$\tau \quad ::= \quad \mathbf{atom} \mid \mathbf{Pair}(\tau, \tau) \mid \mathbf{ListOf}(\tau) \mid \tau \cup \tau.$$

A list-based NRC-type τ *denotes* a set $[\![\tau]\!]$ of list-based NRC-values :

- $[\![\mathbf{atom}]\!] := \mathcal{A}$;
- $[\![\mathbf{Pair}(\tau_1, \tau_2)]\!] := [\![\tau_1]\!] \times [\![\tau_2]\!]$;
- $[\![\mathbf{ListOf}(\tau)]\!]$ denotes the set of all finite lists over $[\![\tau]\!]$; and
- $[\![\tau_1 \cup \tau_2]\!] := [\![\tau_1]\!] \cup [\![\tau_2]\!]$.

We will abuse notation and identify τ with $[\![\tau]\!]$. A *list-based NRC-type assignment* Γ is a function from a finite set of variables $dom(\Gamma)$ to list-based NRC-types. A type assignment denotes the set of contexts σ for which $dom(\sigma) = dom(\Gamma)$ and $\sigma(x) \in \Gamma(x)$, for every $x \in dom(\sigma)$. Again, we will abuse notation and identify a type assignment with its denotation. Finally, if $dom(\Gamma)$ is a superset of $FV(e)$, then we say that Γ is a type assignment on e.

Well-definedness We have already noted that $e(\sigma)$ is not necessarily defined (i.e., e does not necessarily evaluate to a value on σ). This leads us to the following notion:

Definition 51. Let e be a PENRC(*extract*) expression and let Γ be a list-based NRC-type assignment on e. If $e(\sigma)$ is defined for every context $\sigma \in \Gamma$, then e is *well-defined under* Γ. The *well-definedness problem* for the list-based PENRC(*extract*) consists of checking, given an expression e and a list-based NRC-type assignment Γ on e, whether e is well-defined under Γ.

Theorem 52. *Well-definedness for the list-based PENRC(extract) is decidable.*

Proof. We give a reduction to the well-definedness problem for QL(*concat*, *children*, *eq*, $\ll$, *node-name*, *content*, *element*) which is decidable by Corollary 49. Specifically, let e be an expression in PENRC(*extract*) and let Γ be a list-based NRC-type assignment on e. We will show that there exists

1. a one-to-many encoding of list-based NRC-values as values in the XQuery data model;
2. a type assignment $enc(\Gamma)$, computable from Γ, such that every context in $enc(\Gamma)$ is an encoding of some context in Γ and every context in Γ has an encoding in $enc(\Gamma)$; and

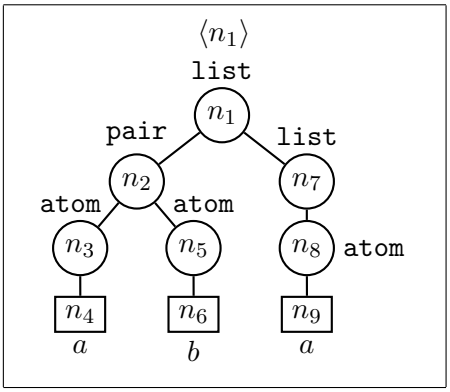


Figure 8: One encoding of the list-based NRC-value $\langle (a, b), \langle a \rangle \rangle$ in the QL data model.

3. an expression $enc(e)$ in QL(B), computable from e, such that e is defined on an input if, and only if, $enc(e)$ is defined on every encoding of this input.

Note that hence e is well-defined under Γ if, and only if, $enc(e)$ is well-defined under $enc(\Gamma)$. Since $enc(e)$ and $enc(\Gamma)$ can moreover be computed from e respectively Γ, we hence have a reduction to well-definedness in QL(*concat*, *children*, *eq*, $\ll$, *node-name*, *element*), as desired.

Let v be a list-based NRC-value. We define the set $enc(v)$ of QL-values which encode v by induction on v as follows. Here, we assume without loss of generality that the special labels `atom`, `pair`, and `list` are atoms.

- If $v = a$, then $enc(v)$ is the set of all values $(\Sigma; \langle n \rangle)$ where n is an element node labeled by `atom` which has exactly one leaf child text node n', which is labeled by a.
- If $v = (v_1, v_2)$, then $enc(v)$ is the set of all values $(\Sigma; \langle n \rangle)$ where n is an element node labeled by `pair` which has exactly two children n_1 and n_2 such that $n_1 < n_2$, $(\Sigma; \langle n_1 \rangle) \in enc(v_1)$, and $(\Sigma; \langle n_2 \rangle) \in enc(v_2)$.
- If $v = \langle v_1, \ldots, v_k \rangle$, then $enc(v)$ is the set of all values $(\Sigma; \langle n \rangle)$ where n is an element node labeled by `list` which has exactly k children $n_1, \ldots, n_k$ such that $n_1 < \ldots < n_k$ and $(\Sigma; \langle n_j \rangle) \in enc(v_j)$ for all $j \in [1, k]$.

For example, a value in $enc(\langle (a, b), \langle a \rangle \rangle)$ is shown in Figure 8. The set $enc(\sigma)$ of QL-contexts which encode a list-based NRC-context σ is then defined as

$$enc(\sigma) := \{(\Sigma; \sigma') \mid (\Sigma; \sigma'(x)) \in enc(\sigma(x)) \text{ for all } x \in dom(\sigma)\}.$$

Next, we define $enc(\Gamma)$. If τ is a list-based NRC-type, then we define the QL-type $enc(\tau)$ which simulates τ as follows by induction on τ.

- If $\tau = \mathbf{atom}$, then $enc(\tau)$ is $\mathbf{element}(\texttt{atom}, \mathbf{text})$.
- If $\tau = \mathbf{Pair}(\tau_1, \tau_2)$, then $enc(\tau)$ is $\mathbf{element}(\texttt{pair}, enc(\tau_1) \circ enc(\tau_2))$.
- If $\tau = \mathbf{ListOf}(\tau')$, then $enc(\tau)$ is $\mathbf{element}(\texttt{list}, enc(\tau')^*)$.
- If $\tau = \tau_1 \cup \tau_2$, then $enc(\tau)$ is $enc(\tau_1) + enc(\tau_2)$.

The type assignment $enc(\Gamma)$ is then defined by

$$enc(\Gamma)(x) := enc(\Gamma(x))$$

for all $x \in dom(\Gamma)$. It is easy to see that every context in $enc(\Gamma)$ is an encoding of some context in Γ, and that every context in Γ has an encoding in $enc(\Gamma)$.

Finally, we construct $enc(e)$ by induction on e. In order to simplify presentation, we will allow to bind multiple variables in one for loop, and we will also allow boolean combinations in the condition of an if test. Both features can clearly be simulated in QL(B).

- If $e = x$, then $enc(e) = x$.
- If $e = (e_1, e_2)$, then $enc(e)$ is defined as

  $element(\texttt{pair, } concat(enc(e_1), \ enc(e_2)))$

- If $e = \pi_1(e')$, then $enc(e)$ is defined as

```
let x := enc(e') return
if eq(node-name(x),pair) then
  for y,z in children(x) return
    if ≪(y, z) then y else ()
else if () then () else ()
```

- If $e = \pi_2(e')$, then $enc(e)$ is defined as

```
let x := enc(e') return
if eq(node-name(x),pair) then
  for y,z in children(x) return
    if ≪(z, y) then y else ()
else if () then () else ()
```

- If $e = \emptyset$, then $enc(e)$ is defined as $element(\texttt{list}, ())$.
- If $e = \{e'\}$, then $enc(e)$ is defined as $element(\texttt{list}, enc(e'))$.
- If $e = \bigcup e'$, then $enc(e)$ is defined as

```
let x := enc(e′) return
element(list,
  if eq(node-name(x),list) then
    for y in children(x) return
      if eq(node-name(y),list) then
        children(y)
      else if () then () else ()
  else if () then () else ()
)
```

- If $e = \{e_2 \mid x \in e_1\}$, then $enc(e)$ is defined as

```
let y := enc(e1) return
if eq(node-name(y),list) then
  for x in children(y) return enc(e2)
else if () then () else ()
```

  Here we assume without loss of generality that y is not free in e_2.

- If $e = extract(e')$, then $enc(e)$ is defined as

```
let x := enc(e′) return
if eq(node-name(x),list) then
  let y := is-element(children(x)) return
    children(x)
else if () then () else ()
```

- If $e = e_1 = e_2 \ ?\ e_3\ :\ e_4$, then $enc(e)$ is defined as

```
let x1 := enc(e1) return
let x2 := enc(e2) return
if eq(node-name(x1),atom) and eq(node-name(x2),atom) then
  if eq(content(children(x1)),content(children(x2)))
  then enc(e3) else enc(e4)
else if () then () else ()
```

  Here we assume without loss of generality that x_1 and x_2 are not free in e_3 or e_4.

A straightforward induction on e no shows that

1. if $e(\sigma) = v$, then $enc(e)(\Sigma; \sigma') \subseteq enc(v)$ for every $(\Sigma; \sigma') \in enc(\sigma)$; and that
2. $e(\sigma)$ is defined if, and only if, $enc(e)(\Sigma; \sigma)$ is defined for every $(\Sigma; \sigma') \in enc(\sigma)$. □

9 Conclusion

We have shown that well-definedness for QL(B) is decidable if B contains only monotone, generic, local, and locally-undefined base operations. Violation of any one of these conditions allows the definition of a language for which well-definedness is undecidable.

Although our results have been developed for a list-based data model, we mention that our results also hold in a bag-based data model (where we hence disregard order). Indeed, note that our undecidability proofs are never based on the fact that values are ordered. Moreover, if we adapt the notion of a monotone, generic, local and locally-undefined base operation to a bag-based data model, then it is not hard to see that we can obtain equivalent versions of Propositions 25, 27, 44, and 45. Hence, the well-definedness problem remains decidable in this case.

It is clear that the number of possible counter-examples we need to check according to the decision procedure outlined in the proof of Theorem 46 can grow huge very fast. Hence the obvious question for future work: what is the computational complexity of well-definedness?

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations Of Databases*. Addison-Wesley, 1995.

[2] Noga Alon, Tova Milo, Frank Neven, Dan Suciu, and Victor Vianu. Typechecking XML views of relational databases. *ACM Transactions on Computational Logic*, 4(3):315–354, 2003.

[3] Noga Alon, Tova Milo, Frank Neven, Dan Suciu, and Victor Vianu. XML with data values: typechecking revisited. *Journal of Computer and System Sciences*, 66(4):688–727, 2003.

[4] Francois Bancilhon and Setrag Khoshafian. A calculus for complex objects. In *Proceedings of the fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 53–60. ACM Press, 1986.

[5] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. *XQuery 1.0: An XML Query Language*. W3C Working Draft, February 2005.

[6] Anne Brüggemann-Klein, Makoto Murata, and Derick Wood. Regular tree and regular hedge languages over unranked alphabets. Unpublished manuscript, version 1, 2001.

[7] Peter Buneman, Mary F. Fernandez, and Dan Suciu. UnQL: a query language and algebra for semistructured data based on structural recursion. *VLDB Journal*, 9(1):76–110, 2000.

[8] Peter Buneman, Shamim A. Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, 1995.

[9] Don Chamberlin, Peter Fankhauser, Daniela Florescu, Massimo Marchiori, and Jonathan Robie. *XML Query Use Cases*. W3C Working Draft, November 2003.

[10] Dario Colazzo, Giorgio Ghelli, Paolo Manghi, and Carlo Sartiani. Types for path correctness of XML queries. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming (ICFP 2004)*, pages 126–137, 2004.

[11] Hubert Comon, Max Dauchet, Remi Gilleron, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications, 2002. `http://www.grappa.univ-lille3.fr/tata/`.

[12] Xin Dong, Alon Y. Halevy, and Igor Tatarinov. Containment of nested XML queries. In *Proceedings of the 30th VLDB Conference*, pages 132–143. Morgan Kaufmann, 2004.

[13] Denise Draper, Peter Fankhauser, Mary F. Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. *XQuery 1.0 and XPath 2.0 Formal Semantics*. W3C Working Draft, February 2005.

[14] Mary F. Fernández, Ashok Malhotra, Jonathan Marsh, Marton Nagy, and Norman Walsh. *XQuery 1.0 and XPath 2.0 Data Model*. W3C Working Draft, February 2005.

[15] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. ℂDuce: an XML-centric general-purpose language. In *Proceedings of the eighth ACM SIGPLAN International Conference on Functional Programming*, pages 51–63. ACM Press, 2003.

[16] Jan Hidders. Satisfiability of XPath expressions. In *9th International Workshop on Database Programming Languages, DBPL 2003, Revised Papers*, volume 2921 of *Lecture Notes in Computer Science*, pages 21–36. Springer, 2004.

[17] Jan Hidders, Jan Paredaens, Roel Vercammen, and Serge Demeyer. A light but formal introduction to XQuery. In *Proceedings of the Second International XML Database Symposium (XSym 2004)*, volume 3186 of *Lecture Notes in Computer Science*, pages 5–20. Springer-Verlag, 2004.

[18] Haruo Hosoya. *Regular Expression Types for XML.* PhD thesis, University of Tokyo, 2000.

[19] Haruo Hosoya and Benjamin C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology (TOIT)*, 3(2):117–148, 2003.

[20] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems*, 27(1):46–90, 2005.

[21] Yannis E. Ioannidis and Raghu Ramakrishnan. Containment of conjunctive queries: Beyond relations as sets. *ACM Transactions on Database Systems*, 20(3):288–324, September 1995.

[22] Howard Katz, editor. *XQuery from the Experts.* Addison-Wesley, 2003.

[23] Laks V. S. Lakshmanan, Ganesh Ramesh, Hui Wang, and Zheng (Jessica) Zhao. On testing satisfiability of tree pattern queries. In *Proceedings of the 30th VLDB Conference*, pages 120–131. Morgan Kaufmann, 2004.

[24] Ashok Malhotra, Jim Melton, and Norman Walsh. *XQuery 1.0 and XPath 2.0 Functions and Operators.* W3C Working Draft, February 2005.

[25] Wim Martens and Frank Neven. Typechecking top-down uniform unranked tree transducers. In *Database Theory - ICDT 2003*, volume 2572 of *Lecture Notes in Computer Science*, pages 64–78. Springer-Verlag, 2003.

[26] Wim Martens and Frank Neven. Frontiers of tractability for typechecking simple XML transformations. In *Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 23–34. ACM Press, 2004.

[27] Yuri Matiyasevich. *Hilbert's 10th Problem.* MIT Press, 1993.

[28] Gerome Miklau and Dan Suciu. Containment and equivalence for a fragment of XPath. *Journal of the ACM*, 51(1):2–45, 2004.

[29] Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for XML transformers. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 11–22. ACM Press, 2000.

[30] John C. Mitchell. *Concepts in programming languages.* Cambridge University Press, 2003.

[31] Frank Neven. Automata, logic, and XML. In *Computer Science Logic - CSL 2002*, volume 2471 of *Lecture Notes in Computer Science*, pages 2–26. Springer, 2002.

[32] Frank Neven. Automata theory for XML researchers. *ACM SIGMOD Record*, 31(3):39–46, 2002.

[33] Frank Neven and Thomas Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. In *Database Theory - ICDT 2003*, volume 2572 of *Lecture Notes in Computer Science*, pages 315–329. Springer, 2003.

[34] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[35] Dan Suciu. Typechecking for semistructured data. In *Database Programming Languages, 8th International Workshop, DBPL 2001, Revised Papers*, volume 2397 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, 2001.

[36] Wolfgang Thomas. *Languages, Automata, and Logic*, volume 3 of *Handbook of Formal Languages*, chapter 7, pages 389–456. Springer, 1997.

[37] Jan Van den Bussche, Dirk Van Gucht, and Stijn Vansummeren. Well-definedness and semantic type-checking in the nested relational calculus.

[38] Limsoon Wong. *Querying nested collections*. PhD thesis, University of Pennsylvania, 1994.

[39] Franois Yergeau, Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. *Extensible Markup Language (XML) 1.0 (Third Edition)*. W3C Recommendation, February 2004.