School voor Informatietechnologie
Kennistechnologie, Informatica, Wiskunde, ICT

Well-Definedness, Semantic Type-Checking, and Type Inference for Database Query Languages

Proefschrift voorgelegd tot het behalen van de graad van
Doctor in de Wetenschappen, richting Informatica
te verdedigen door

Stijn Vansummeren

Promotor: Prof. dr. J. Van den Bussche

20 mei 2005

D/2005/2451/17

Acknowledgements

This dissertation is dedicated to
Albert and Maria, Guillaume and Anna.
The finest examples a grandson could have.

First and foremost, I am grateful to my advisor Jan Van den Bussche for his continuing enthusiasm and guidance. This dissertation would not have been possible without his knack for pinpointing the core of a problem. I hope one day to be as proficient in defining elegant formalisms as he is.

Second, I am much in debt to Dirk Van Gucht. His knack for asking lots of "small" questions was crucial in gaining an understanding of the well-definedness problem for the nested relational calculus. Chapter 2 discusses our joint work, which has its roots in a very enjoyable visit to Indiana University in Bloomington. I am grateful for the hospitality I received during this visit, and especially thank Dirk, his wife Linda, Alia Alkasimi, Bassem Sayrafi, and George Fletcher.

I also thank the members of our research group, the department, and the administrative staff for creating a stimulating environment. Furthermore, I am grateful to my employer, the Fund for Scientific Research, Flanders.

On a personal note, I am in great debt to my parents and brother for the guidance and support they have given me over the years. They have always inspired me to go the extra mile.

My gratitude also goes out to my many friends (you know who you are). Their very enjoyable company was always offered when I needed it the most.

Finally, I am grateful for the support, patience, and encouragement I have received from my companion in life, Evi. Thank you.

Diepenbeek, March 2005

Contents

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 [42, 49]. 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 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 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. The most prominent examples of the latter are *database query languages* such as SQL [40], OQL [11], and XQuery [6].[1] Expressions in all these languages can be undefined. For example, consider the following OQL expression:

```
select author: element(b.authors), title: b.title
from books b
where b.pub_year > 2000
```

This expression returns, for each book published after the year 2000, the book's author and title. The subexpression `element(b.authors)` checks that the set of `b`'s authors is a singleton, and if so, extracts this single author. If the book is written by more than one author however, the result of the expression is undefined.

As query languages do not have full computational power, Rice's theorem does not apply and it is hence worthwhile to investigate if we can't decide the well-definedness problem for them. If so, then we obtain in essence a type system which is both sound and complete. In this dissertation we therefore study the well-definedness problem for database query languages. We start our study with well-definedness for the Nested Relational Calculus (NRC for short). The NRC is a well-known query language for the complex object data model [1, 9, 60]. It is a conservative extension [59] of the relational algebra [1, 16] (which serves as the data processing core of SQL) and can itself be viewed as a data processing core of OQL. Furthermore, the NRC inspired the design of various semi-structured languages such as UnQL [8], StruQL [21], and Quilt [13], on which XQuery is based. As such, our study of well-definedness for the NRC serves as a good starting point for the study of well-definedness in SQL, OQL, and XQuery.

Certain features of the latter two languages are not captured by the standard set-based NRC however. Indeed, OQL operates on bags and lists in addition to sets, while XQuery operates on lists. Both languages have object identity and the ability to create new objects. We therefore continue our study by identifying broad classes of first-order, object-creating query languages operating on list-based data for which the well-definedness problem is (un)decidable. Specifically, we identify properties of basic operations in such languages which can make the well-definedness problem undecidable and give

[1] XQuery is in fact a full-fledged general-purpose programming language. Most XQuery programs are of the restricted form "for-let-where-return" however, which we regard as the true query language part of XQuery.

corresponding restrictions which are sufficient to ensure decidability. The obtained results can be transfered to a bag-based data model, and are directly applicable to OQL and XQuery.

A problem which is related to well-definedness is the *semantic type-checking problem*: decide, given an input type, an expression and an output type, whether the expression only produces outputs in the output type on inputs in the input type. This problem is useful in a "producer-consumer" setting where a producer generates data, which is processed by a consumer. In order to ensure good operation by the consumer, the producer is expected to only produce data adhering to a certain type. Unfortunately, the semantic type-checking problem is also undecidable for any computationally complete programming language, by Rice's theorem. In practice however, the producer will often consist of a query against a database. It is therefore interesting to see if we can't solve the semantic type-checking problem for the query languages mentioned above. We study this problem for the NRC. For XQuery and other XML-related languages, the problem has already been studied extensively [2, 3, 37, 38, 41, 52].

Our study will show that both well-definedness and semantic type-checking remain undecidable for query languages which are powerful enough to simulate the relational algebra. It follows that a sound and complete type system for SQL, OQL, or XQuery does not exist (although we will identify several useful fragments for which such a system does exist). Query languages that want to verify the absence of certain programming errors statically will hence have to do so by means of a traditional, incomplete type system.

In the second part of this dissertation we therefore study classical type system problems from the theory of programming languages in the context of database query languages. Recently, Van den Bussche and Waller [56] have noted that the operators of the relational algebra are *polymorphic*. For instance, we can take the natural join of any two relations, regardless of their schema. We can take the union of any two relations with the same schema. We can perform a projection $\pi_{A,\ldots,B}$ of any relation having at least the attributes $\{A, \ldots, B\}$. When combining operators into expressions, these typing conditions become more involved. For example, for the expression

$$\pi_A(r \bowtie s) \bowtie ((r \times u) - v)$$

to be well-typed, the attribute A must be an attribute of r or s (or both). But if it is an attribute of r, then it must also be one of v. Moreover, by the subexpression $(r \times u) - v$, the relation schemas of r and s must be disjoint, and their union must be the schema of v.

A natural question thus arises: given a relational algebra expression e, under which database schemas is e well-typed? And what is the result rela-

tion schema of e under these database schemas? In particular, can we give an explicit description of the typically infinite set of these *typings*? This is nothing but the relational algebra version of the classical *type inference* problem. Type inference is an extensively studied topic in the theory of programming languages [43, 49] and is used in industrial-strength functional programming languages such as Standard ML [55] and Haskell [30]. For the relational algebra, this problem was studied by Van den Bussche and Waller [56].

Some expressions, for example $\sigma_A(\pi_B(R))$, are inherently *untypable* (i.e., these expressions do not admit any typing). Checking typability of relational algebra expressions is the analog in the relational algebra of static type-checking in implicitly typed programming languages with polymorphic type systems, such as ML. It is therefore interesting to see what its complexity is. It is known for instance that typability is P-complete for the simply typed lambda calculus [19] and EXPTIME-complete for ML [31, 34]. In contrast, Van den Bussche and Waller have shown that typability for the relational algebra is in NP. The precise complexity remained open, however. In this dissertation, we show that the problem is NP-hard, even in various restricted settings. Finally, we also investigate the type inference and typability problems for the NNRC, a named version of the Nested Relational Calculus.

1.1 Detailed Overview and Related Work

Well-Definedness and Semantic Type-Checking

We start in **Chapter 2** by studying the well-definedness problem for the NRC in the standard, set-based, complex object data model [1, 9, 60]. In particular, 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, like OQL's *element* operator, extracts v from a singleton set $\{v\}$ and is undefined on non-singleton inputs. Alas, this operator causes the well-definedness problem to become undecidable again. The core difficulty here is the fact that $\mathit{extract}(\{e_1, e_2\})$ 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. We show that the equivalence problem for the PENRC is undecidable. Finally, we study the well-definedness problem for the PENRC in the presence of *type tests*. Such tests allow the inspection of the type of a value at runtime, and are present for example in XQuery. Unfortunately, type test also cause the problem to become undecidable again. Fortunately however, well-definedness remains decidable if we only allow a

limited form of type tests, which we call *kind tests*.

Next, we study the semantic type-checking problem for the NRC in the presence of standard complex object types. Similar to our results for well-definedness, we obtain that the problem is undecidable for the NRC in general, but is decidable for the PENRC. The semantic type-checking problem has already been studied extensively in XML-related query languages [2, 3, 37, 38, 41, 52]. In particular, our setting closely resembles that of Alon et al. [2, 3] who, like us, study the problem in the presence of data values. In particular they have shown that (un)decidability depends on the expressiveness of both the query language and the type system. While the query language of Alon et al. can simulate the NRC, one needs a feature called *specialization* in order to encode complex object types in their type system. In the presence of this feature, they have shown semantic type-checking for their type system to be undecidable, even in the positive-existential case. In contrast, semantic type-checking for the PENRC in the complex object type system is decidable.

In **Chapter 3** we 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 [7, 15, 44, 45] and are widely used in general-purpose programming languages manipulating tree-structured data, such as XDuce [26, 27, 28], ℂDuce [23], and XQuery [6, 18]. The bounded-depth restriction is motivated by the fact that most tree-structured data (such as for example found in XML documents [61]) 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 results are directly applicable to XQuery, as we show that XQuery's basic functions and operators [35] 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.

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 items. 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. 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 show that well-definedness for the positive-existential NRC equipped with such a base operation, interpreted in a list-based data model, is decidable. As the list-based PENRC is a fragment of OQL, our study of QL(B) hence also leads to a deeper understanding of well-definedness for OQL.

Type Inference and Typability

In **Chapter 4** we study the complexity of deciding typability in the relational algebra. We obtain that the problem is NP-complete in general. In particular, we show that the problem becomes NP-hard due to (1) the cartesian product operator; (2) the selection operator on arbitrary sets of typed predicates; and (3) the selection operator on "well-behaved" sets of typed predicates together with join and projection or renaming. However, the problem is in P when (1) we only allow union, difference, join, and selection on "well-behaved" sets of typed predicates; or (2) we allow all operators except cartesian product, where the set of selection predicates can mention at most one base type. Most of these results follow from a close connection of the typability problem to non-uniform constraint satisfaction.

In **Chapter 5** we study type inference and typability for the NNRC, a named version of the nested relational calculus that is equipped with a static type system. Specifically, we propose an explicit description of the set

of all possible typings of an NNRC expression e by means of a conjunctive logical formula ϕ_e, which is interpreted in the universe of all possible types. The formula ϕ_e is efficiently computable from e. We proceed to show that the satisfiability of such conjunctive formulas belongs to NP. Consequently, typability for the NNRC is also in NP. Since the NNRC is an extension of the relational algebra, for which typability is already NP-complete, this thus shows that typability for the NNRC is not more difficult than for the special case of the relational algebra.

In the theory of programming languages one also finds type inference and type-checking algorithms for languages with sets and records, often in the presence of even more powerful features such as higher order functions [10, 47, 50, 53, 54, 58]. Indeed, the polymorphic type system of the NNRC can be encoded in the very general type inference framework of HM(X) [53, 54]. To our knowledge, however, we are the first to study the complexity of the typability problem for the specific type systems of the relational algebra and the NNRC. Furthermore, a second goal of our work was to provide an elementary, self-contained presentation of polymorphic type inference for the NNRC, accessible for researchers in database query languages who may not be familiar with type theory.

Part I

Well-Definedness and Semantic Type-Checking

2

Well-Definedness and Semantic Type-Checking for the Nested Relational Calculus

In this chapter we study the well-definedness and semantic type-checking problems for the nested relational calculus. We start in Section 2.1 by introducing the nested relational calculus data model and query language. Next, we introduce the well-definedness problem in Section 2.2, where we also show that this problem is undecidable for the NRC in general, but becomes decidable for the positive-existential fragment of the NRC. We study the well-definedness problem for this fragment in the presence of singleton coercion in Section 2.3 and in the presence of type tests in Section 2.4. Finally, we study the semantic type-checking problem in Section 2.5.

2.1 Nested Relational Calculus

Data Model We assume to be given a recursively enumerable set $\mathcal{A} = \{a, b, \dots\}$ of *atoms*, which in practice will contain the usual data values such as integers, strings, and so on. A *complex object value* is either an atom, a pair of complex object values, or a finite set of complex object values. For ex-

ample, $\{a, (b, c), (a, \{a, b\})\}$ is a complex object value. For convenience we will abbreviate "complex object value" by "value" in this chapter. Furthermore, we will range over complex object values by u, v, and w and over finite sets of complex object values by U, V, and W.

Syntax We also assume given a set $\mathcal{X} = \{x, y, \dots\}$ of *variables*. The *Nested Relational Calculus* (NRC) is the set of all expressions generated by the following grammar:

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

Here, e ranges over NRC 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.

Semantics A *complex object context* σ is a function from a finite set of variables $dom(\sigma)$ to complex object values. If $dom(\sigma)$ is a superset of $FV(e)$, then we say that σ is a *complex object context on* e. We denote by $x\colon v, \sigma$ the complex object context σ' with domain $dom(\sigma) \cup \{x\}$ such that $\sigma'(x) = v$ and $\sigma'(y) = \sigma(y)$ for $y \neq x$. We will abbreviate "complex object context" by "context" in this chapter.

The semantics of NRC expressions is described by means of the *evaluation relation*, as defined in Figure 2.1. Here, we write $\sigma \models e \Rightarrow v$ to denote the fact that e evaluates to value v on 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 take the union of sets, flatten a set of sets, iterate over sets, test equality on atoms, and test emptiness of sets. 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, $\sigma' \models e \Rightarrow v$.

Variables

$$\frac{}{\sigma \models x \Rightarrow \sigma(x)}$$

Pair operations

$$\frac{\sigma \models e_1 \Rightarrow v_1 \quad \sigma \models e_2 \Rightarrow v_2}{\sigma \models (e_1, e_2) \Rightarrow (v_1, v_2)} \qquad \frac{\sigma \models e \Rightarrow (v_1, v_2)}{\sigma \models \pi_1(e) \Rightarrow v_1} \qquad \frac{\sigma \models e \Rightarrow (v_1, v_2)}{\sigma \models \pi_2(e) \Rightarrow v_2}$$

Set operations

$$\frac{}{\sigma \models \emptyset \Rightarrow \emptyset} \qquad \frac{\sigma \models e \Rightarrow v}{\sigma \models \{e\} \Rightarrow \{v\}} \qquad \frac{\sigma \models e_1 \Rightarrow V_1 \quad \sigma \models e_2 \Rightarrow V_2}{\sigma \models e_1 \cup e_2 \Rightarrow V_1 \cup V_2}$$

$$\frac{\sigma \models e \Rightarrow \{V_1, \ldots, V_n\}}{\sigma \models \bigcup e \Rightarrow \bigcup \{V_1, \ldots, V_n\}} \qquad \frac{\sigma \models e_1 \Rightarrow V \quad \forall v \in V : (x\colon v, \sigma) \models e_2 \Rightarrow w_v}{\sigma \models \{e_2 \mid x \in e_1\} \Rightarrow \{w_v \mid v \in V\}}$$

Conditional tests

$$\frac{\sigma \models e_1 \Rightarrow a \quad \sigma \models e_2 \Rightarrow b \quad \sigma \models e_3 \Rightarrow v \quad a = b}{\sigma \models e_1 = e_2 \ ?\ e_3 \ :\ e_4 \Rightarrow v} \qquad \frac{\sigma \models e_1 \Rightarrow a \quad \sigma \models e_2 \Rightarrow b \quad \sigma \models e_4 \Rightarrow v \quad a \neq b}{\sigma \models e_1 = e_2 \ ?\ e_3 \ :\ e_4 \Rightarrow v}$$

$$\frac{\sigma \models e_1 \Rightarrow V \quad \sigma \models e_2 \Rightarrow v \quad V = \emptyset}{\sigma \models e_1 = \emptyset \ ?\ e_2 \ :\ e_3 \Rightarrow v} \qquad \frac{\sigma \models e_1 \Rightarrow V \quad \sigma \models e_3 \Rightarrow v \quad V \neq \emptyset}{\sigma \models e_1 = \emptyset \ ?\ e_2 \ :\ e_3 \Rightarrow v}$$

Figure 2.1: The evaluation relation for NRC expressions.

Types The free variables of an expression are usually meant to hold only values of a specific form, which can be specified by means of a type assignment. A *complex object type* is a term generated by the following grammar:

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

A complex object type τ *denotes* a set of complex object values $[\![\tau]\!]$:

- $[\![\mathbf{Atom}]\!] := \mathcal{A}$,
- $[\![\mathbf{Pair}(\tau_1, \tau_2)]\!] := [\![\tau_1]\!] \times [\![\tau_2]\!]$,
- $[\![\mathbf{SetOf}(\tau)]\!]$ is the set of all finite sets over $[\![\tau]\!]$, and,
- $[\![\tau_1 \cup \tau_2]\!] := [\![\tau_1]\!] \cup [\![\tau_2]\!]$.

We will abuse notation and identify τ with $[\![\tau]\!]$. A *complex object type assignment* Γ is a function from a finite set of variables $dom(\Gamma)$ to types. A complex object type assignment denotes the set of complex object 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 complex object type assignment with its denotation. Finally, if $dom(\Gamma)$ is a superset of $FV(e)$, then we say that Γ is a complex object type assignment on e. We will abbreviate "complex object type" and "complex object type assignment" by "type" respectively "type assignment" in this chapter.

Example 2.1. Let *friends* and *John* be two variables. Suppose that the value of *friends* is a set of friends, as a set of pairs of atoms. Suppose also that the value of *John* is a name (an atom). The following expression computes the set of all of *John*'s friends:

$$\bigcup\{\pi_1(x) = \mathit{John} \;?\; \{\pi_2(x)\} \;:\; \emptyset \mid x \in \mathit{friends}\}.$$

The set of intended context inputs to this expression is described by the type assignment Γ on e for which $\Gamma(\mathit{friends}) = \mathbf{SetOf}(\mathbf{Pair}(\mathbf{Atom}, \mathbf{Atom}))$ and $\Gamma(\mathit{John}) = \mathbf{Atom}$. □

2.2 Well-Definedness

As we have already noted in the previous section, $e(\sigma)$ is not necessarily defined (i.e., e does not necessarily evaluate to a value on σ). This leads us to the following central notion:

Definition 2.2. Let e be an NRC expression and let Γ be a 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 NRC* consists of checking, given e and Γ, whether e is well-defined under Γ.

Since an actual implementation of the NRC will produce a runtime error on those contexts σ for which $e(\sigma)$ is undefined, it is worthwhile to ask whether we can let a computer solve the well-definedness problem. Unfortunately, we cannot:

Theorem 2.3. *The well-definedness problem for the NRC is undecidable.*

Proof. It is well-known that the (finite) satisfiability problem for the relational algebra is undecidable [1]. That problem consists of checking, given a relational algebra expression ϕ over a relational schema S, whether ϕ returns a non-empty result on some database instance over S. It is easy to see that a database instance can be encoded as a context. For example, consider the database instance D where relation names r and s are assigned the following respective relations:

A	B	C
a_1	b_1	c_1
a_2	b_1	c_2

C	D
c_1	a_1
c_2	a_2

Clearly, D can then be encoded as the context σ where

$$\sigma(r) = \{(a_1, (b_1, c_1)), (a_2, (b_1, c_2))\}$$
$$\sigma(s) = \{(c_1, a_1), (c_2, a_2)\}.$$

It is well-known [9, 60] that for every ϕ and S there exists and expression e and a type assignment Γ such that

1. e is well-defined under Γ,
2. the contexts in Γ are exactly the encodings of database instances over S, and
3. if D is a database instance over S and σ is an encoding of D, then $e(\sigma)$ is an encoding of $\phi(D)$.

The fact that e is well-defined under Γ stems from the fact that relational algebra expressions are always well-defined with regard to their database schema. It is clear that ϕ is satisfiable if, and only if, e is satisfiable on a context in Γ. Since the expression $\{\pi_1(\emptyset) \mid x \in e\}$ is not well-defined under Γ if, and only if e is satisfiable, we have a reduction from satisfiability to well-definedness. Hence, well-definedness is undecidable. □

Note that satisfiability for the *positive-existential* 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 satisfiable.[1] In order to obtain a fragment of the NRC for which well-definedness is decidable, it is hence worthwhile to investigate which features of the NRC allow the simulation of a difference operation. Assume that R and S are sets of atoms. The following expression then computes the difference of R and S:

$$\bigcup\left\{\bigcup\{x = y\ ?\ \{x\}\ :\ \emptyset \mid y \in S\} = \emptyset\ ?\ \{x\}\ :\ \emptyset \;\middle|\; x \in R\right\}.$$

The inner comprehension returns $\{x\}$ if $x \in S$ and returns $\emptyset$ otherwise. The outer conditional test compares this result with $\emptyset$ to filter out those x in S. Since the ability to test set-emptiness is hence too powerful a feature with regard to well-definedness checking, we will restrict ourselves in what follows to expressions in which the emptiness test does not occur.

2.2.1 Positive-Existential Nested Relational Calculus

The *Positive-Existential Nested Relational Calculus* is the NRC without emptiness test expression. Before investigating the well-definedness problem in the context of the PENRC, we should verify that we cannot simulate the relational algebra difference operator by the remaining expressions. Otherwise, we will still be able to simulate the full relational algebra, and the well-definedness problem will remain undecidable. We therefore introduce the *containment* relation $\sqsubseteq$ on values as follows [5]:

$$\frac{}{a \sqsubseteq a} \qquad \frac{v \sqsubseteq v' \qquad w \sqsubseteq w'}{(v,w) \sqsubseteq (v',w')} \qquad \frac{\text{for all } v_i \text{ there exists } w_j \text{ such that } v_i \sqsubseteq w_j}{\{v_1,\ldots,v_n\} \sqsubseteq \{w_1,\ldots,w_m\}}$$

This relation is extended component-wise to contexts: if σ and σ' are two contexts with the same domain, then $\sigma \sqsubseteq \sigma'$ if $\sigma(x) \sqsubseteq \sigma'(x)$ for every $x \in dom(\sigma)$.

Lemma 2.4 (Monotonicity). *Let e be a PENRC expression and let σ and σ' be contexts on e such that $\sigma \sqsubseteq \sigma'$. If $e(\sigma)$ and $e(\sigma')$ are defined, then $e(\sigma) \sqsubseteq e(\sigma')$. If $e(\sigma)$ is undefined, then so is $e(\sigma')$.*

The proof is by a straightforward induction on e. It is easy to see that the difference operator is a non-monotone operation. Indeed, let $R = \{a\}$

[1] 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.

and $S = \emptyset$, then $R - S = \{a\}$. However, if we extend S to $S' = \{a\}$ then $R - S' = \emptyset$, which does not contain $\{a\}$ although $R \sqsubseteq R$ and $S \sqsubseteq S'$. It follows that difference is not expressible in the PENRC.

We will show that the well-definedness problem for the PENRC is decidable. The key to this decidability is that the PENRC has a *small model property for undefinedness*. Let us introduce this property by an example.

Example 2.5. Let e be the expression

$$e = \{\{z = y \ ? \ \pi_1(z) \ : \ y \mid y \in x\} \mid x \in R\},$$

and let the type assignment Γ on e be defined by:

$$\Gamma(R) = \mathbf{SetOf}(\mathbf{SetOf}(\mathbf{Atom}))$$
$$\Gamma(z) = \mathbf{Atom}.$$

Let the context $\sigma \in \Gamma$ be defined by $\sigma(R) = \{\{a, b\}, \{c\}, \{d, a, b\}\}$ and $\sigma(z) = d$. Since there is a set in $\sigma(R)$ which contains $\sigma(z)$, we will need to evaluate π_1 on $\sigma(z)$ at some point, which is undefined (as $\sigma(z)$ is an atom). Hence, $e(\sigma)$ is undefined. Note that we do not need all elements in $\sigma(R)$ to reach the state where $e(\sigma)$ becomes undefined. Indeed, $e(\sigma')$ is also undefined if $\sigma'(R) = \{\{d\}\}$ and $\sigma'(z) = d$. Note that every set occurring in σ' has cardinality at most one and that $\sigma' \in \Gamma$. □

We will show in Section 2.2.2 that we can generalize this example as follows. Here, we say that a value v has *width at most* k if every set occurring in v has cardinality at most k. Likewise, a context σ has *width at most* k if $\sigma(x)$ has width at most k, for every $x \in dom(\sigma)$.

Proposition 2.6 (Small model for undefinedness). *Let e be a PENRC expression and let Γ be a type assignment on e such that e is not well-defined under Γ. Then there exists a natural number k, computable from e, and a context $\sigma' \in \Gamma$ of width at most k such that $e(\sigma')$ is also undefined.*

Before showing how this property allows us to solve the well-definedness problem, a definition is in order.

Genericity Let ρ be a permutation of $\mathcal{A}$. We extend ρ to values in the canonical way:

$$\rho((v, w)) := (\rho(v), \rho(w))$$
$$\rho(V) := \{\rho(v) \mid v \in V\}$$

We also extend ρ component-wise to contexts: $\rho(\sigma)(x) := \rho(\sigma(x))$. Two contexts σ and σ' are *isomorphic* if there exists a permutation ρ such that $\rho(\sigma) = \sigma'$. It is easy to see that PENRC expressions cannot distinguish between isomorphic inputs:

Lemma 2.7 (Genericity). *Let e be a PENRC expression, let σ be a context on e, and let ρ be a permutation of $\mathcal{A}$. If $e(\sigma)$ is defined, then so is $e(\rho(\sigma))$ and $e(\rho(\sigma)) = \rho(e(\sigma))$. If $e(\sigma)$ is undefined, then so is $e(\rho(\sigma))$.*

Theorem 2.8. *The well-definedness problem for the PENRC is decidable.*

Proof. Suppose that expression e is not well-defined under type assignment Γ on e. By Proposition 2.6 there exists a natural number k, computable from e, and some context $\sigma \in \Gamma$ of width at most k such that $e(\sigma)$ is undefined.

Let us denote the maximum number of atoms a value in type τ of width at most k can mention by $\mathit{rank}(\tau, k)$. Then clearly,

$$\begin{aligned} \mathit{rank}(\mathbf{Atom}, k) &= 1 \\ \mathit{rank}(\mathbf{Pair}(\tau_1, \tau_2), k) &= \mathit{rank}(\tau_1, k) + \mathit{rank}(\tau_2, k) \\ \mathit{rank}(\mathbf{SetOf}(\tau'), k) &= k \times \mathit{rank}(\tau', k) \\ \mathit{rank}(\tau_1 \cup \tau_2, k) &= \max\{\mathit{rank}(\tau_1, k), \mathit{rank}(\tau_2, k)\} \end{aligned}$$

Consequently, the maximum number of atoms mentioned in σ is bounded by

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

Note that l is computable from Γ and e. Now fix some l-element subset A of $\mathcal{A}$. Since the number of different atoms mentioned in σ is at most l there surely exists a permutation ρ of $\mathcal{A}$ such that $\rho(\sigma)$ mentions only atoms in A. By genericity, $e(\rho(\sigma))$ is also undefined.

Hence, in order to check if e is well-defined under Γ, it suffices to check whether $e(\gamma)$ is defined for all contexts $\gamma \in \Gamma$ which mention only atoms in A. It is easy to see that there are only a finite number of such γ, from which the result follows. □

2.2.2 Small Model Properties

In this section we prove the small model property for undefinedness (Proposition 2.6): if there is an input on which an expression e is undefined, then there is also a "small" input on which it is undefined. We first note:

Lemma 2.9 (Type preservation). *Let τ be a type. If $w \in \tau$ and $v \sqsubseteq w$, then also $v \in \tau$.*

The proof is by a straightforward induction on τ. In order to prove Proposition 2.6 it hence suffices to show that, given an expression e and a context σ for which $e(\sigma)$ is undefined, we can deduce $\sigma' \sqsubseteq \sigma$ whose width depends only on e such that $e(\sigma')$ is also undefined. We will prove this property by induction on e by "tracing" the reason why $e(\sigma)$ is undefined through σ (from the bottom up). In order to do so we will need a *small model property for definedness*, as we outline in the following example.

Example 2.10. Let $e = \{\pi_1(x) \mid x \in e_1\}$ and suppose that σ is a context on e for which $e_1(\sigma) = \{a, (a,b), (c,d), (a,d)\}$. Since at some point we will evaluate $\pi_1(x)$ on $(x\colon a, \sigma)$ (which is undefined), it follows that $e(\sigma)$ is also undefined. Clearly, the undefinedness of $\pi_1(x)(x\colon a, \sigma)$ is solely due to the fact that x is bound to the atom a. As we are searching for a "small" context $\sigma' \sqsubseteq \sigma$ on which the whole expression e is undefined, we want to make sure that at some point we still evaluate $\pi_1(x)$ under a context in which x is bound to a. That is, we will want to construct σ' in such a way that $\{a\} \sqsubseteq e_1(\sigma')$. If, for example, $e_1 = R \cup S$ with $\sigma(R) = \{a, (a,b)\}$ and $\sigma(S) = \{(a,b), (c,d), (a,d)\}$, then we could take $\sigma'(R) = \{a\}$ and $\sigma'(S) = \emptyset$. □

As this example illustrates, we will want to show that, given an expression e_1, a context σ for which $e_1(\sigma)$ is defined, and a value $u \sqsubseteq e_1(\sigma)$, we can "trace" the reason that $e_1(\sigma)$ contains u through σ. In particular we want to show that we can always deduce a context $\sigma' \sqsubseteq \sigma$ whose width depends only on e_1 and u such that $u \sqsubseteq e_1(\sigma')$. This is our small model property for definedness, which we prove below. First however, some additional definitions are in order.

Union of Values We start by defining the union operation $\sqcup$ on values of the same kind:

$$a \sqcup a := a \qquad (u_1, u_2) \sqcup (v_1, v_2) := (u_1 \sqcup v_1, u_2 \sqcup v_2) \qquad U \sqcup V := U \cup V$$

On all other arguments, $\sqcup$ is undefined. It is easy to see that $u \sqcup v$ (if it exists) is a least upper bound (according to $\sqsubseteq$) of values u and v:

Lemma 2.11. *If $u \sqsubseteq w$ and $v \sqsubseteq w$, then $u \sqcup v$ exists, $u \sqsubseteq u \sqcup v$, $v \sqsubseteq u \sqcup v$, and $(u \sqcup v) \sqsubseteq w$.*

Note that, if u has width at most k and v has width at most l, then $u \sqcup v$ (if it exists) has width at most $k+l$. The value union is extended component-wise to contexts: if σ and σ' are contexts with the same domain, then $\sigma \sqcup \sigma'$ is the context with $(\sigma \sqcup \sigma')(x) = \sigma(x) \sqcup \sigma'(x)$ for every $x \in \mathit{dom}(\sigma)$. It follows from Lemma 2.11 that, if $\sigma \sqsubseteq \gamma$ and $\sigma' \sqsubseteq \gamma$, then $\sigma \sqcup \sigma'$ exists, $\sigma \sqsubseteq \sigma \sqcup \sigma'$, $\sigma' \sqsubseteq \sigma \sqcup \sigma'$, and $\sigma \sqcup \sigma' \sqsubseteq \gamma$. Moreover, if σ has width at most k and σ' has width at most l, then $\sigma \sqcup \sigma'$ (if it exists) has width at most $k + l$.

Minimization Next, we introduce the minimization operation *minimize* on values:

$$\begin{aligned} \mathit{minimize}(a) &:= a \\ \mathit{minimize}(u_1, u_2) &:= (\mathit{minimize}(u_1), \mathit{minimize}(u_2)) \\ \mathit{minimize}(V) &:= \emptyset \end{aligned}$$

It is clear that $\mathit{minimize}(v) \sqsubseteq v$ and that $\mathit{minimize}(v)$ has width zero. As before, we extend the minimization operation component-wise to contexts: $\mathit{minimize}(\sigma)(x) := \mathit{minimize}(\sigma(x))$.

Convention: In what follows we will write $\mathcal{V}_k$ for the set of all values of width at most k and $\mathcal{C}_k$ for the set of all contexts of width at most k.

Proposition 2.12 (Small model property for definedness). *For every PENRC expression e there exists a computable function c_e mapping natural numbers to natural numbers such that for every natural number k, every context σ on e for which $e(\sigma)$ is defined, and every $u \sqsubseteq e(\sigma)$ of width at most k, there exists a context $\sigma' \sqsubseteq \sigma$ of width at most $c_e(k)$ such that $u \sqsubseteq e(\sigma')$. Moreover, an arithmetic expression defining c_e is effectively computable from e.*

Proof. Let e be a PENRC expression. Define the function c_e inductively as follows.

$$\begin{aligned} c_x(k) &:= k \\ c_{(e_1,e_2)}(k) &:= c_{e_1}(k) + c_{e_2}(k) \\ c_{\pi_1(e')}(k) &:= c_{e'}(k) \\ c_{\pi_2(e')}(k) &:= c_{e'}(k) \\ c_{\emptyset}(k) &:= 0 \\ c_{\{e'\}}(k) &:= k \times c_{e'}(k) \\ c_{e_1 \cup e_2}(k) &:= c_{e_1}(k) + c_{e_2}(k) \\ c_{\bigcup e'}(k) &:= c_{e'}(k) \\ c_{\{e_2 \mid x \in e_1\}}(k) &:= c_{e_1}(\max\{k, c_{e_2}(k)\}) + k \times c_{e_2}(k) \\ c_{e_1 = e_2 \ ?\ e_3\ :\ e_4}(k) &:= \max\{c_{e_3}(k), c_{e_4}(k)\} \end{aligned}$$

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 function mapping natural numbers to natural numbers. Let k be a natural number, let σ be a context on e for which $e(\sigma)$ is defined, and let $u \sqsubseteq e(\sigma)$ be a value of width at most k. Define the predicate $P(u, e, \sigma, k)$ as follows:

$$P(u, e, \sigma, k) := \{\sigma' \mid \sigma' \in \mathcal{C}_{c_e(k)}, \sigma' \sqsubseteq \sigma, \text{ and } u \sqsubseteq e(\sigma')\}.$$

We will prove by induction on e that $P(u, e, \sigma, k)$ is non-empty, from which the proposition follows. Note that, since $e(\sigma)$ is defined, $e(\delta)$ is also defined for every $\delta \sqsubseteq \sigma$ by monotonicity. We also remind the reader that if $\sigma_1 \sqsubseteq \sigma$ has width at most k and $\sigma_2 \sqsubseteq \sigma$ has width at most l, then $\sigma_1 \sqcup \sigma_2$ exists and has width at most $k + l$. Furthermore, $\sigma_1 \sqsubseteq \sigma_1 \sqcup \sigma_2$, $\sigma_2 \sqsubseteq \sigma_1 \sqcup \sigma_2$, and $\sigma_1 \sqcup \sigma_2 \sqsubseteq \sigma$ by Lemma 2.11. We will use these facts silently throughout the induction.

- If $e = x$, then we define σ' by

$$\sigma'(y) = \begin{cases} u & \text{if } y = x \\ \mathit{minimize}(\sigma(y)) & \text{otherwise} \end{cases}$$

- If $e = \emptyset$, then we take $\sigma' = \mathit{minimize}(\sigma)$.

- If $e = (e_1, e_2)$, then $e(\sigma)$ is a pair. Hence, $u = (u_1, u_2)$ for some $u_1, u_2 \in \mathcal{V}_k$. By the induction hypothesis there exist $\sigma_1 \in P(u_1, e_1, \sigma, k)$ and $\sigma_2 \in P(u_2, e_2, \sigma, k)$. Then $\sigma_1 \sqcup \sigma_2 \in \mathcal{C}_{c_{e_1}(k)+c_{e_2}(k)} = \mathcal{C}_{c_e(k)}$. Moreover, by monotonicity:

$$(u_1, u_2) \sqsubseteq (e_1(\sigma_1), e_2(\sigma_2)) \sqsubseteq (e_1(\sigma_1 \sqcup \sigma_2), e_2(\sigma_1 \sqcup \sigma_2)) = e(\sigma_1 \sqcup \sigma_2)$$

  Hence, $\sigma_1 \sqcup \sigma_2 \in P(u, e, \sigma, k)$.

- If $e = e_1 \cup e_2$, then $e(\sigma)$ is a set. Since $u \sqsubseteq e(\sigma)$ there exists, for every $v \in u$, a $w_v \in e(\sigma)$ such that $v \sqsubseteq w_v$. Define,

$$u_1 := \{v \in u \mid w_v \in e_1(\sigma)\}$$
$$u_2 := \{v \in u \mid w_v \in e_2(\sigma)\}$$

  Then $u = u_1 \cup u_2$, $u_1 \sqsubseteq e_1(\sigma)$, and $u_2 \sqsubseteq e_2(\sigma)$. Moreover, $u_1, u_2 \in \mathcal{V}_k$. The result then follows from the induction hypothesis by a reasoning similar to the previous case.

- If $e = \pi_1(e')$, then $e'(\sigma)$ is a pair (v, w). Let $u' = (u, \mathit{minimize}(w))$. Then $u' \sqsubseteq (v, w)$ since $u \sqsubseteq v$ and $\mathit{minimize}(w) \sqsubseteq w$. Moreover, $u' \in \mathcal{V}_k$ since $\mathit{minimize}(w) \in \mathcal{V}_0$. Hence there exists $\sigma' \in P(u', e', \sigma, k)$ by the induction hypothesis. Hence,

$$u = \pi_1(u') \sqsubseteq \pi_1(e'(\sigma')) = e(\sigma').$$

  Since also $\mathcal{C}_{c_{e'}(k)} = \mathcal{C}_{c_e(k)}$, we have $\sigma' \in P(u, e, \sigma, k)$. The case where $e = \pi_2(e')$ is similar.

- If $e = \{e'\}$, then we discern two cases. If $u = \emptyset$, then it is clear that $u \sqsubseteq e(\sigma')$ for any $\sigma' \sqsubseteq \sigma$. Hence, it suffices to take $\sigma' = \mathit{minimize}(\sigma)$, which is in $\mathcal{C}_0 \subseteq \mathcal{C}_{c_e(k)}$. Otherwise, u contains at least one and at most k elements. For each $v \in u$ we have that v is of width at most k and that $v \sqsubseteq e'(\sigma)$. Hence, there exists $\sigma_v \in P(v, e', \sigma, k)$ for every $v \in u$ by the induction hypothesis. Let $\sigma' = \bigsqcup_{v \in u} \sigma_v$. Then $\sigma_v \sqsubseteq \sigma'$ for every $v \in u$, and $\sigma' \sqsubseteq \sigma$. By monotonicity we then have $v \sqsubseteq e'(\sigma_v) \sqsubseteq e'(\sigma')$, and hence $u \sqsubseteq \{e'(\sigma')\} = e(\sigma')$. Moreover, $\sigma' \in \mathcal{C}_{k \times c_{e'}(k)} = \mathcal{C}_{c_e(k)}$. Hence, $\sigma' \in P(u, e, \sigma, k)$.

- If $e = \bigcup e'$, then $e'(\sigma)$ is a set of sets. Since $u \sqsubseteq e(\sigma)$ there exists, for every $v \in u$, a $w_v \in e(\sigma)$ such that $v \sqsubseteq w_v$. Let $e'(\sigma) = \{V_1, \ldots, V_n\}$. Note that $e(\sigma) = V_1 \cup \cdots \cup V_n$. Define, for each $i \in [1, n]$,

$$U_i := \{v \in u \mid w_v \in V_i \setminus \bigcup_{j<i} V_j\}.$$

  Note that since u has width at most k, the cardinality of each of the U_i's is at most k and at most k of the U_i's are non-empty. Furthermore, $U_i \sqsubseteq V_i$. Let u' be the set of all non-empty U_i's. Then $u' \sqsubseteq e'(\sigma)$ and $u' \in \mathcal{V}_k$. The result then follows from the induction hypothesis.

- If $e = e_1 = e_2 \ ? \ e_3 \ : \ e_4$, then $e_1(\sigma), e_2(\sigma) \in \mathcal{A}$. Suppose $e_1(\sigma) = e_2(\sigma)$, then $u \sqsubseteq e_3(\sigma)$. By the induction hypothesis there exists $\sigma' \in P(u, e_3, \sigma, k)$. Then $e_1(\sigma') = e_1(\sigma) = e_2(\sigma) = e_2(\sigma')$ by monotonicity and hence $e(\sigma') = e_3(\sigma')$. We then have by the induction hypothesis that $u \sqsubseteq e_3(\sigma') = e(\sigma')$. Since also $\sigma' \in \mathcal{C}_{c_{e_3}(k)} \subseteq \mathcal{C}_{c_e(k)}$, we have $\sigma' \in P(u, e, \sigma, k)$. The case where $e_1(\sigma) \neq e_2(\sigma)$ is similar.

- If $e = \{e_2 \mid x \in e_1\}$, then $e(\sigma)$ is a set. Since $u \sqsubseteq e(\sigma)$ there exists, for every $v \in u$, a value $w_v \in e(\sigma)$ such that $v \sqsubseteq w_v$. Since $e(\sigma)$ is obtained by a comprehension over $e_1(\sigma)$, there also must exist, for every $v \in u$, a value $z_v \in e_1(\sigma)$ such that $w_v = e_2(x\colon z_v, \sigma)$. Hence there exists, for every $v \in u$, a context $x\colon z'_v, \sigma'_v \in P(v, e_2, (x\colon z_v, \sigma), k)$ by the induction hypothesis. Let $u' = \{z'_v \mid v \in u\}$. Then u' contains at most k elements of $\mathcal{V}_{c_{e_2}(k)}$. Hence, $u' \in \mathcal{V}_m$ with $m = \max\{k, c_{e_2}(k)\}$. Moreover, $x\colon z'_v, \sigma' \sqsubseteq x\colon z_v, \sigma$ by the induction hypothesis, so $z'_v \sqsubseteq z_v$, and hence $u' \sqsubseteq e_1(\sigma)$.

  By applying the induction hypothesis again, there exists $\sigma_1 \in P(u', e_1, \sigma, m)$. Let $\sigma' = \sigma_1 \sqcup \bigsqcup_{v \in u} \sigma'_v$. Note that $\sigma_1 \sqsubseteq \sigma'$ and $\sigma'_v \sqsubseteq \sigma'$, for every $v \in u$. Furthermore, $\sigma' \sqsubseteq \sigma$ and the width of σ' is bounded by:

$$c_{e_1}(\max\{k, c_{e_2}(k)\}) + k \times c_{e_2}(k) = c(e, k).$$

Now $u' \sqsubseteq e_1(\sigma_1) \sqsubseteq e_1(\sigma')$ by monotonicity. Hence, for every z'_v there exists some $z''_v \in e_1(\sigma')$ with $z'_v \sqsubseteq z''_v$. Then $x\colon z'_v, \sigma'_v \sqsubseteq x\colon z''_v, \sigma'$. Hence, by monotonicity:

$$v \sqsubseteq e_2(x\colon z'_v, \sigma'_v) \sqsubseteq e_2(x\colon z''_v, \sigma').$$

Since this holds for every $v \in u$, we have $u \sqsubseteq e(\sigma')$. □

Lemma 2.13. *For every PENRC expression e there exists a natural number k_e, computable from e, such that for every context σ on e for which $e(\sigma)$ is undefined, there exists $\sigma' \sqsubseteq \sigma$ of width at most k_e such that $e(\sigma')$ is also undefined.*

Proof. By Proposition 2.12 there exists, for every expression e, a computable function c_e such that for every natural number k, every context σ on e for which $e(\sigma)$ is defined, and every $u \sqsubseteq e(\sigma)$ of width at most k, there exists a context $\sigma' \sqsubseteq \sigma$ of width at most $c_e(k)$ such that $u \sqsubseteq e(\sigma')$. Let, for every expression e, the natural number k_e be inductively defined as follows:

$$\begin{aligned}
k_x &:= 0 \\
k_{(e_1,e_2)} &:= \max\{k_{e_1}, k_{e_2}\} \\
k_{\pi_1(e')} &:= k_{e'} \\
k_{\pi_2(e')} &:= k_{e'} \\
k_{\emptyset} &:= 0 \\
k_{\{e'\}} &:= k_{e'} \\
k_{e_1 \cup e_2} &:= \max\{k_{e_1}, k_{e_2}\} \\
k_{\bigcup e'} &:= \max\{k_{e'}, c_e(1)\} \\
k_{\{e_2 | x \in e_1\}} &:= \max\{k_{e_1}, c_{e_1}(\max\{1, k_{e_2}\}) + k_{e_2}\} \\
k_{e_1 = e_2 \ ?\ e_3\ :\ e_4}(k) &:= \max\{k_{e_1}, k_{e_2}, k_{e_3}, k_{e_4}\}
\end{aligned}$$

Since an arithmetic expression defining $c_{e'}$ is computable from e' by Proposition 2.12, it follows that k_e is effectively computable from e. Let e be a PENRC expression and let σ be a context on e for which $e(\sigma)$ is undefined. We prove by induction on e that there exists $\sigma' \sqsubseteq \sigma$ of width at most k_e such that $e(\sigma')$ is also undefined.

- If $e = x$ or $e = \emptyset$, then there is nothing to prove, since $e(\sigma)$ is always defined.
- If $e = (e_1, e_2)$, then either $e_1(\sigma)$ or $e_2(\sigma)$ is undefined. The result then follows by the induction hypothesis. The case where $e = \{e'\}$ is similar.

- If $e = \pi_1(e')$, then either $e'(\sigma)$ is undefined, in which case the result follows from the induction hypothesis, or $e'(\sigma)$ is not a pair. In that case, let $\sigma' = \mathit{minimize}(\sigma)$. By monotonicity $e'(\sigma')$ cannot be a pair and hence $e(\sigma')$ is also undefined. Moreover, $\sigma' \in \mathcal{C}_0 \subseteq \mathcal{C}_{k_e}$.

- If $e = e_1 \cup e_2$, then either $e_1(\sigma)$ is undefined, $e_2(\sigma)$ is undefined, $e_1(\sigma)$ is not a set, or $e_2(\sigma)$ is not a set. In the first two cases the result follows from the induction hypothesis. In the third case, let $\sigma' = \mathit{minimize}(\sigma)$. By monotonicity $e_1(\sigma)$ cannot be a set and hence $e(\sigma')$ is undefined. Moreover, $\sigma' \in \mathcal{C}_0 \subseteq \mathcal{C}_{k_e}$. The last case is similar.

- If $e = \bigcup e'$, then either $e'(\sigma)$ is undefined, in which case the result follows from the induction hypothesis, or $e'(\sigma)$ is not a set of sets. In that case we discern two possibilities. If $e'(\sigma)$ is not a set, then let $\sigma' = \mathit{minimize}(\sigma)$. By monotonicity, $e'(\sigma')$ cannot be a set and hence $e(\sigma)$ is undefined. Moreover, $\sigma' \in \mathcal{C}_0 \subseteq \mathcal{C}_{k_e}$. If $e'(\sigma)$ is a set, but not a set of sets, then there exist some $u \in e'(\sigma)$ that is not a set. Then $\{\mathit{minimize}(u)\} \in \mathcal{V}_1$ and $\{\mathit{minimize}(u)\} \sqsubseteq e'(\sigma)$. By Proposition 2.12 there exists $\sigma'' \in \mathcal{C}_{c_{e'}(1)} \subseteq \mathcal{C}_{k_e}$ with $\sigma'' \sqsubseteq \sigma$ such that $\{\mathit{minimize}(u)\} \sqsubseteq e'(\sigma'')$. Hence, $e'(\sigma'')$ is not a set of sets and $e(\sigma'')$ is also undefined.

- If $e = e_1 = e_2 \ ?\ e_3 \ :\ e_4$, then we discern the following possibilities.

  1. If $e_1(\sigma)$ or $e_2(\sigma)$ is undefined, then the result follows from the induction hypothesis.
  2. If $e_1(\sigma)$ and $e_2(\sigma)$ are defined, but $e_1(\sigma)$ is not an atom, then let $\sigma' = \mathit{minimize}(\sigma)$. By monotonicity, $e_1(\sigma')$ cannot be an atom and hence $e(\sigma')$ is undefined. Moreover, $\sigma' \in \mathcal{C}_0 \subseteq \mathcal{C}_{k_e}$. The case where $e_1(\sigma)$ and $e_2(\sigma)$ are defined, but $e_2(\sigma)$ is not an atom is similar.
  3. If $e_1(\sigma)$ and $e_2(\sigma)$ are defined, $e_1(\sigma)$ and $e_2(\sigma)$ are atoms, and $e_1(\sigma) = e_2(\sigma)$, then $e_3(\sigma)$ must be undefined. By the induction hypothesis, there exists $\sigma' \in \mathcal{C}_{k_{e_3}} \subseteq \mathcal{C}_{k_e}$ with $\sigma' \sqsubseteq \sigma$ such that $e_3(\sigma')$ is also undefined. By monotonicity $e_1(\sigma') = e_2(\sigma')$, and hence $e(\sigma')$ is undefined. If $e_1(\sigma) \neq e_2(\sigma)$ the reasoning is similar.

- If $e = \{e_2 \mid x \in e_1\}$, then we discern the following possibilities.

  1. If $e_1(\sigma)$ is undefined, then the result follows from the induction hypothesis.
  2. If $e_1(\sigma)$ is defined, but is not a set, then let $\sigma' = \mathit{minimize}(\sigma)$. By monotonicity, $e_1(\sigma')$ cannot be a set and hence $e(\sigma')$ is undefined. Moreover, $\sigma' \in \mathcal{C}_0 \subseteq \mathcal{C}_{k_e}$.

3. Otherwise, $e_1(\sigma)$ is defined and a set, but there is some $v \in e_1(\sigma)$ such that $e_2(x\colon v, \sigma)$ is undefined. By the induction hypothesis there then exists $x\colon u, \sigma_2 \in \mathcal{C}_{k_{e_2}}$ with $x\colon u, \sigma_2 \sqsubseteq x\colon v, \sigma$ such that $e_2(x\colon u, \sigma_2)$ is undefined. Then $\{u\} \in \mathcal{V}_{\max\{1,k_{e_2}\}}$ and $\{u\} \sqsubseteq e_1(\sigma)$. By Proposition 2.12 there exists $\sigma_1 \in \mathcal{C}_{c_{e_1}(\max\{1,k_{e_2}\})}$ with $\sigma_1 \sqsubseteq \sigma$ such that $\{u\} \sqsubseteq e_1(\sigma_1)$. Since both $\sigma_1 \sqsubseteq \sigma$ and $\sigma_2 \sqsubseteq \sigma$, $\sigma_1 \sqcup \sigma_2$ is defined by Lemma 2.11. Let $\sigma' = \sigma_1 \sqcup \sigma_2$. Note that $\sigma_1 \sqsubseteq \sigma'$ and $\sigma_2 \sqsubseteq \sigma'$ by Lemma 2.11. By monotonicity $\{u\} \sqsubseteq e_1(\sigma')$. Hence, there exists some $u' \in e_1(\sigma')$ such that $u \sqsubseteq u'$. Then clearly

$$x\colon u, \sigma_2 \sqsubseteq x\colon u', \sigma',$$

and hence $e_2(x\colon u', \sigma')$ is also undefined by monotonicity. Hence, $e(\sigma')$ is undefined. Moreover, $\sigma' \in \mathcal{C}_{c_{e_1}(\max\{1,k_{e_2}\})+k_{e_2}} \subseteq \mathcal{C}_{k_e}$. □

Proposition 2.6 now follows by Lemma 2.13 and Lemma 2.9. Indeed, let e be a PENRC expression, let Γ be a type assignment on e, and let $\sigma \in \Gamma$ such that $e(\sigma)$ is undefined. By Lemma 2.13 there a natural number k_e, computable from e alone, and $\sigma' \sqsubseteq \sigma$ of width at most k_e such that $e(\sigma')$ is also undefined. Since $\sigma \in \Gamma$, it follows that σ' is also in Γ by Lemma 2.9. Hence the proposition.

2.3 The Impact of Singleton Coercion

The expressions of the NRC are designed around the guiding principle that every value constructor should have a corresponding "destructor" [60]. As such, the pair constructor (e_1, e_2) has the projection operations π_1 and π_2 as destructors, and the set union $e_1 \cup e_2$ has set comprehension as a "destructor". The singleton set constructor has no corresponding destructor in the standard NRC, however. In this section we study the well-definedness problem for the PENRC in the presence of such a destructor.[2]

Formally, we denote by PENRC(*extract*) the version of the PENRC to which we add *extract* as an expression:

$$e ::= \cdots \mid \mathit{extract}(e).$$

The semantics of *extract* is defined as follows:

$$\frac{\sigma \models e \Rightarrow \{v\}}{\sigma \models \mathit{extract}(e) \Rightarrow v}$$

[2] We note that OQL, the object-oriented cousin of SQL, also has such a destructor, written *element*(*e*).

That is, *extract* coerces a singleton $\{v\}$ into the value v it contains and is undefined on other inputs.

Although *extract* appears quite harmless at first sight, it invalidates one of the fundamental monotonicity properties we use to prove our small model property for undefinedness. Indeed, it is no longer true that if $e(\sigma)$ is undefined and $\sigma \sqsubseteq \sigma'$, then $e(\sigma')$ is also undefined. For example, take $e = \mathit{extract}(x)$, $\sigma(x) = \{\{a\}, \{a, b\}\}$, and $\sigma'(x) = \{\{a, b\}\}$). It is clear that $e(\sigma)$ is undefined, but $e(\sigma')$ is not. Note however that $\{\{a\}, \{a, b\}\} \sqsubseteq \{\{a, b\}\}$ since both $\{a\}$ and $\{a, b\}$ are contained in $\{a, b\}$.

One could hope to find another containment relation under which we regain our monotonicity property and can redo the proof in the previous section. Unfortunately however, such a containment relation does not exist. Indeed, we will show that the well-definedness problem for PENRC(*extract*) is undecidable. To see why, the following definition is in order.

Definition 2.14. Let e_1 and e_2 be two expressions with the same set of free variables, such that e_1 and e_2 are well-defined under type assignment Γ. We say that e_1 and e_2 are *equivalent under* Γ when $e_1(\sigma) = e_2(\sigma)$ for every $\sigma \in \Gamma$. The *equivalence problem* consists of checking, given such e_1, e_2, and Γ, whether e_1 and e_2 are equivalent under Γ.

Note that the well-definedness problem for PENRC(*extract*) is at least as difficult as the equivalence problem for the PENRC. Indeed, e_1 is equivalent to e_2 under Γ if, and only if, $\mathit{extract}(\{e_1\} \cup \{e_2\})$ is well-defined under Γ (as e_1 and e_2 are already well-defined under Γ). Hence, the undecidability of well-definedness for PENRC(*extract*) follows from the following theorem.

Theorem 2.15. *The equivalence problem for PENRC is undecidable.*[3]

Proof. In order to focus on the crux of the proof, we will assume without loss of generality that the PENRC is equipped with tuples of arbitrary (but fixed) arity. This feature can clearly be encoded using pairs. For example, we could encode $t = (a_1, a_2, a_3)$ by $t' = (a_1, (a_2, a_3))$. We also assume that we have projection functions for such tuples. For example, $\pi_3(t)$ can be simulated by $\pi_2(\pi_2(t'))$. As an extension of this, if $I = i_1, \ldots, i_n$ is a sequence of positive integers, then we write $\Pi_I(t)$ for $(\pi_{i_1}(t), \ldots, \pi_{i_n}(t))$. Furthermore, we will use the polyadic type constructor $\mathbf{Tuple}(\tau_1, \ldots, \tau_n)$ which denotes the set of all tuples t of arity n such that $\pi_i(t) \in \tau_i$ for all $i \in [1, n]$. This type constructor can be simulated using the pair type constructor. For example, $\mathbf{Tuple}(\mathbf{Atom}, \mathbf{Atom}, \mathbf{Atom})$ can be simulated by

[3] We note that, in contrast, the containment problem for the PENRC (with regard to $\sqsubseteq$, not ordinary set-containment) in the absence of union is decidable [33].

$\mathbf{Pair}(\mathbf{Atom}, \mathbf{Pair}(\mathbf{Atom}, \mathbf{Atom}))$. Finally, we will allow conditional tests to compare entire tuples of atomic values with the same arity. Again, this can be simulated using only tests on atomic values.

The proof is by a reduction from the implication problem of functional and inclusion dependencies over a single relation symbol, which is known to be undecidable [1, 14]. This problem is defined as follows. Let x be a variable, let n be a natural number, and let Γ be the type assignment with domain $\{x\}$ such that

$$\Gamma(x) = \mathbf{SetOf}(\mathbf{Tuple}(\underbrace{\mathbf{Atom}, \mathbf{Atom}, \ldots, \mathbf{Atom}}_{n \text{ times}})).$$

A *functional dependency* is a rule of the form $X \to Y$ where X and Y are sequences over $[1, n]$. We say that a context $\sigma \in \Gamma$ satisfies $X \to Y$, denoted by $\sigma \models X \to Y$, if for all tuples $t_1, t_2 \in \sigma(x)$, if $\pi_X(t_1) = \pi_X(t_2)$ then also $\pi_Y(t_1) = \pi_Y(t_2)$. An *inclusion dependency* is a rule of the form $X \subseteq Y$ where X and Y are sequences over $[1, n]$ of the same length. We say that $\sigma \in \Gamma$ satisfies $X \subseteq Y$, denoted by $\sigma \models X \subseteq Y$ if

$$\{\pi_X(t) \mid t \in \sigma(x)\} \subseteq \{\pi_Y(t) \mid t \in \sigma(x)\}.$$

Let Σ be a finite set of functional and inclusion dependencies. We say that $\sigma \in \Gamma$ satisfies Σ, denoted by $\sigma \models \Sigma$, if σ satisfies every dependency in Σ. Let ρ be an additional target functional dependency. We say that Σ *implies* ρ if every context σ which satisfies Σ also satisfies ρ. The *implication problem for functional and inclusion dependencies* consists of checking, given n, Σ, and ρ, whether Σ implies ρ. It is well-known that this problem is undecidable [1, 14].

We reduce the implication problem to the equivalence problem by constructing two expressions which are equivalent under Γ if, and only if, Σ implies ρ. For every functional dependency $X \to Y \in \Sigma \cup \{\rho\}$ we define the expression $e_{X \to Y}$ as follows:

$$\bigcup \left\{ \bigcup \{\Pi_X(t_1) = \Pi_X(t_2) \wedge \Pi_Y(t_1) \neq \Pi_Y(t_2) \ ?\ \{x\}\ :\ \emptyset \mid t_2 \in x\} \;\middle|\; t_1 \in x \right\}.$$

On input $\sigma \in \Gamma$ this expression returns $\emptyset$ if $\sigma \models X \to Y$ and $\{\sigma(x)\}$ otherwise. For every inclusion dependency $X \subseteq Y \in \Sigma$ we define the expression $e_{X \subseteq Y}$ as follows:

$$\left\{ \bigcup \{\Pi_X(t_1) = \Pi_Y(t_2) \ ?\ \{x\}\ :\ \emptyset \mid t_2 \in x\} \mid\ t_1 \in x \right\} \cup \left\{ \{x\} \right\}.$$

On input $\sigma \in \Gamma$ this expression returns $\{\{\sigma(x)\}\}$ if $\sigma \models X \subseteq Y$ and $\{\{\sigma(x)\}, \emptyset\}$ otherwise.

Let $\phi_1, \ldots, \phi_k$ be the functional dependencies in Σ and let $\psi_1, \ldots, \psi_l$ be the inclusion dependencies in Σ. By construction, Σ implies ρ if, and only if, for every $\sigma \in \Gamma$, whenever all the $e_{\phi_i}(\sigma) = \emptyset$ and all the $e_{\psi_j}(\sigma) = \{\{\sigma(x)\}\}$, then $e_\rho(\sigma) = \emptyset$. Then let f_0 be the expression

$$f_0 := (e_{\phi_1}, \ldots, e_{\phi_k}, e_{\psi_1}, \ldots, e_{\psi_l}, e_\rho).$$

Furthermore, let $f_1, \ldots, f_p$ be all the expressions of the form

$$(r_1, \ldots, r_k, s_1, \ldots, s_l, t),$$

where the r_i are either $\emptyset$ or $\{x\}$, the s_j are either $\{\{x\}\}$ or $\{\{x\}\} \cup \{\emptyset\}$, and t is either $\emptyset$ or $\{x\}$ such that, if the r_i are all of the form $\emptyset$ and the s_j are all of the form $\{\{x\}\}$, then t is $\emptyset$. Then Σ implies ρ if, and only if, for every $\sigma \in \Gamma$ there exists $j \in [1, p]$ such that $f_0(\sigma) = f_j(\sigma)$. Hence Σ implies ρ if, and only if,

$$(\{f_0\} \cup \{f_1\} \cup \cdots \cup \{f_p\})(\sigma) = (\{f_1\} \cup \cdots \cup \{f_p\})(\sigma),$$

for every $\sigma \in \Gamma$. □

Corollary 2.16. *The well-definedness problem for PENRC(extract) is undecidable.*

Interestingly enough, the well-definedness problem for PENRC(*extract*) evaluated under a list-based instead of a set-based semantics *is* decidable, as we will show in Section 3.7.2.

2.4 The Impact of Type Tests

Modern programming languages have *type test* expressions which allow the inspection of the type of a value at runtime. The manner in which the value is to be processed can depend on the outcome of such an inspection. For example, the expression

$$x \in \mathbf{Pair}(\mathbf{Atom}, \mathbf{Atom}) \ ? \ \{\pi_1(x)\} \ : \ \emptyset$$

computes $\{\pi_1(x)\}$ if x is a pair of atoms and $\emptyset$ otherwise. In this section we study the well-definedness problem for PENRC(*type*), the PENRC extended with such a type test expression:

$$e ::= \cdots \mid e \in \tau \ ? \ e \ : \ e.$$

Here, τ ranges over types. The semantics of a type test is the obvious one:

$$\frac{\sigma \models e_1 \Rightarrow v_1 \quad v_1 \in \tau \quad \sigma \models e_2 \Rightarrow v}{\sigma \models e_1 \in \tau \ ? \ e_2 \ : \ e_3 \Rightarrow v} \qquad \frac{\sigma \models e_1 \Rightarrow v_1 \quad v_1 \notin \tau \quad \sigma \models e_3 \Rightarrow v}{\sigma \models e_1 \in \tau \ ? \ e_2 \ : \ e_3 \Rightarrow v}$$

Proposition 2.17. *The NRC is semantically contained in PENRC(type); in other words, type tests can be used to simulate emptiness tests.*

Indeed, the emptiness test $e_1 = \emptyset \ ?\ e_2\ :\ e_3$ can be expressed as follows:

$$\{(x,x) \mid x \in e_1\} \in \mathbf{SetOf}(\mathbf{Atom})\ ?\ e_2\ :\ e_3.$$

If e_1 returns the empty set, then the comprehension $\{(x,x) \mid x \in e_1\}$ also returns the empty set (which is a set of atoms) and we evaluate e_2. Otherwise, the comprehension returns a set of pairs (which is not a set of atoms) and we evaluate e_3.

It follows from Theorem 2.3 that well-definedness for PENRC(*type*) is undecidable.

Corollary 2.18. *The well-definedness problem for PENRC(type) is undecidable.*

Type tests are hence too powerful a feature with regard to well-definedness checking. Still, when dealing with heterogeneous collections a limited form of type tests is desirable. We clarify this claim by an example.

Example 2.19. Let $e = \{\pi_1(x) \mid x \in R\}$. This expression is well-defined under the type assignment Γ with $\Gamma(R) = \mathbf{SetOf}(\mathbf{Pair}(\mathbf{Atom}, \mathbf{Atom}))$, but is undefined under the type assignment Γ' with

$$\Gamma'(R) = \mathbf{SetOf}(\mathbf{Pair}(\mathbf{Atom}, \mathbf{Atom}) \cup \mathbf{Atom}).$$

Indeed, every comprehension processes the set over which it iterates in a uniform manner. Hence, although a set value can in principle be heterogeneous, such values cannot be processed in a well-defined manner. When we can check at runtime whether or not x contains a pair however, then e can be rewritten as follows:

$$e' = \bigcup\{x \in \mathbf{Pair}\ ?\ \{\pi_1(x)\}\ :\ \emptyset \mid x \in R\}.$$

It is clear that e' computes the same result as e on contexts Γ and that e' is well-defined under Γ'. Therefore, when we wish to query heterogeneous sets, we need to be able to distinguish the various forms of the elements of the sets at runtime. □

As a limited form of type tests, we propose the following. A *kind* is a term generated by the following grammar:

$$\kappa \ ::= \ \mathbf{Atom} \mid \mathbf{Pair} \mid \mathbf{Set}$$

Here, κ ranges over kinds. A kind denotes a set of values, which is the set of all atoms, the set of all pairs of values, and the set of all finite sets of values,

respectively. We will not distinguish between a kind and its denotation. We extend the PENRC with the ability to test the kind of a value at runtime:

$$e := \cdots \mid e \in \kappa \ ?\ e\ :\ e$$

Here, κ ranges over kinds. We denote the obtained language by PENRC(*kind*). The semantics of kind tests is the obvious one:

$$\frac{\sigma \models e_1 \Rightarrow v_1 \quad v_1 \in \kappa \quad \sigma \models e_2 \Rightarrow v}{\sigma \models e_1 \in \kappa\ ?\ e_2\ :\ e_3 \Rightarrow v} \qquad \frac{\sigma \models e_1 \Rightarrow v_1 \quad v_1 \notin \kappa \quad \sigma \models e_3 \Rightarrow v}{\sigma \models e_1 \in \kappa\ ?\ e_2\ :\ e_3 \Rightarrow v}$$

Lemma 2.20. *Let κ be a kind and let v and w be values such that $v \sqsubseteq w$. Then $v \in \kappa$ if, and only if, $w \in \kappa$.*

The proof is by an easy case analysis on κ. As a consequence, it is easy to see that the PENRC(*kind*) is also monotone (in the sense of Lemma 2.4). We can therefore extend the proofs of Proposition 2.12 and Lemma 2.13 to show that the PENRC(*kind*) also has the small model properties for definedness and undefinedness.

Proposition 2.21. *For every PENRC(kind) expression e there exists a computable function c_e mapping natural numbers to natural numbers such that for every natural number k, every context σ on e for which $e(\sigma)$ is defined, and every $u \sqsubseteq e(\sigma)$ of width at most k, there exists a context $\sigma' \sqsubseteq \sigma$ of width at most $c_e(k)$ such that $u \sqsubseteq e(\sigma')$. Moreover, an arithmetic expression defining c_e is effectively computable from e.*

Proof. Let e be a PENRC(*kind*) expression. Add the following induction step to the definition of the function c_e in the proof of Proposition 2.12:

$$c_{e_1 \in \kappa\ ?\ e_2\ :\ e_3}(k) := \max\{c_{e_2}(k), c_{e_3}(k)\}.$$

It is clear that an arithmetic expression defining c_e remains computable from e and that c_e is a computable function mapping natural numbers to natural numbers. Let k be a natural number, let σ be a context on e for which $e(\sigma)$ is defined, and let $u \sqsubseteq e(\sigma)$ be a value of width at most k. We prove by induction on e that there exists $\sigma' \sqsubseteq \sigma$ of width at most $c_e(k)$ such that $u \sqsubseteq e(\sigma')$. We only treat the case where $e = e_1 \in \kappa\ ?\ e_2\ :\ e_3$, as the other cases are the same as in the proof of Proposition 2.12.

So, let $e = e_1 \in \kappa\ ?\ e_2\ :\ e_3$. We discern two cases. If $e_1(\sigma) \in \kappa$ then $u \sqsubseteq e_2(\sigma)$. By the induction hypothesis there exist $\sigma' \sqsubseteq \sigma$ of width at most $c_{e_2}(k)$ such that $u \sqsubseteq e_2(\sigma')$. By monotonicity, $e_1(\sigma') \sqsubseteq e_1(\sigma)$. Hence, $e_1(\sigma') \in \kappa$ by Lemma 2.20. Then $e(\sigma') = e_2(\sigma')$, and hence $u \sqsubseteq e_2(\sigma') = e(\sigma')$. Since $\sigma' \in \mathcal{C}_{c_{e_2}(k)} \subseteq \mathcal{C}_{c_e(k)}$, the result follows. The case where $e_1(\sigma) \notin \kappa$ is similar. □

Lemma 2.22. *For every PENRC(kind) expression e there exists a natural number k_e, computable from e, such that for every context σ on e for which $e(\sigma)$ is undefined, there exists $\sigma' \sqsubseteq \sigma$ of width at most k_e such that $e(\sigma')$ is also undefined.*

Proof. Let e be a PENRC(*kind*) expression. Add the following induction step to the definition of the natural number k_e in the proof of Lemma 2.13:

$$k_{e_1 \in \kappa \ ?\ e_2\ :\ e_3} := \max\{k_{e_1}, k_{e_2}, k_{e_3}\}.$$

It is clear that k_e remains computable from e. Let σ be a context on e for which $e(\sigma)$ is undefined. We prove by induction on e that there exists $\sigma' \sqsubseteq \sigma$ of width at most k_e such that $e(\sigma')$ is also undefined. We only treat the case where $e = e_1 \in \kappa\ ?\ e_2\ :\ e_3$, as the other cases are the same as in the proof of Lemma 2.13.

So, let $e = e_1 \in \kappa\ ?\ e_2\ :\ e_3$. If $e_1(\sigma)$ is undefined, then the result follows from the induction hypothesis. If $e_1(\sigma)$ is defined and $e_1(\sigma) \in \kappa$, then $e_2(\sigma)$ must be undefined. By the induction hypothesis we have $\sigma' \in \mathcal{C}_{k_{e_2}} \subseteq \mathcal{C}_{k_e}$ with $\sigma' \sqsubseteq \sigma$ such that $e_2(\sigma')$ is still undefined. By monotonicity, $e_1(\sigma') \sqsubseteq e_1(\sigma)$, and hence $e_1(\sigma') \in \kappa$ by Lemma 2.20. Hence, $e(\sigma')$ is also undefined. If $e_1(\sigma)$ is defined and $e_1(\sigma) \notin \kappa$, then the reasoning is similar. □

As a corollary to this lemma and Lemma 2.9, the small model property for undefinedness continues to hold in the presence of kind tests. It readily follows (cf. the proof of Theorem 2.8):

Theorem 2.23. *The well-definedness problem for PENRC(kind) is decidable.*

2.5 Semantic Type-Checking

A problem that is reminiscent of the well-definedness problem is the semantic type-checking problem: given an expression e, a type assignment Γ under which e is well-defined, and an output type τ, check that $e(\sigma) \in \tau$ for every $\sigma \in \Gamma$. If so, then we say that e *has output type* τ *under* Γ.

It is easily seen that the satisfiability problem for the NRC reduces to the semantic type-checking problem for the NRC. Indeed, the NRC expression

$$e = \emptyset\ ?\ \{x\}\ :\ (x, x)$$

has output type $\mathbf{SetOf}(\Gamma(x))$ under type assignment Γ if, and only if, e is unsatisfiable. As a consequence, the semantic type-checking problem for the NRC is undecidable.

Proposition 2.24. *The semantic type-checking problem for the NRC is undecidable.*

On the positive side, the semantic type-checking problem for the PENRC with kind tests is decidable, as we will show below. We first note:

Lemma 2.25. *If τ is a type and $v \notin \tau$, then there exists a natural number k, computable from τ, and a value $u \sqsubseteq v$ of width at most k such that $u \notin \tau$.*

Proof. Let us define the complexity $c(\tau)$ of a type τ as follows.

$$\begin{aligned} c(\mathbf{Atom}) &:= 0 \\ c(\mathbf{Pair}(\tau_1, \tau_2)) &:= \max(c(\tau_1), c(\tau_2)) \\ c(\mathbf{SetOf}(\tau')) &:= \max(1, c(\tau')) \\ c(\tau_1 \cup \tau_2) &:= c(\tau_1) + c(\tau_2) \end{aligned}$$

Let τ be a type and let $v \notin \tau$. We show that there exists a value $u \in \mathcal{V}_{c(\tau)}$ with $u \sqsubseteq v$ such that $u \notin \tau$ by induction on τ.

- If $\tau = \mathbf{Atom}$, then take $u = \mathit{minimize}(v)$.
- If $\tau = \mathbf{Pair}(\tau_1, \tau_2)$, then either
  1. v is not a pair, in which case we take $u = \mathit{minimize}(v)$; or
  2. $v = (v_1, v_2)$ with $v_1 \notin \tau_1$ or $v_2 \notin \tau_2$. The result then follows from the induction hypothesis.
- If $\tau = \mathbf{SetOf}(\tau')$, then either
  1. v is not a set, in which case we take $u = \mathit{minimize}(v)$, or
  2. there exists some $v' \in v$ such that $v' \notin \tau'$. By the induction hypothesis there exists $u' \in V_{c(\tau')}$ such that $u' \sqsubseteq v'$ and $u' \notin \tau'$. Then $\{u'\} \sqsubseteq v$ and $\{u'\} \notin \tau$.
- Finally, if $\tau = \tau_1 \cup \tau_2$, then $v \notin \tau_1$ and $v \notin \tau_2$. By the induction hypothesis there exist $u_1 \in c(\tau_1)$ and $u_2 \in c(\tau_2)$ with $u_1 \sqsubseteq v$ and $u_2 \sqsubseteq v$ such that $u_1 \notin \tau_1$ and $u_2 \notin \tau_2$. Take $u = u_1 \sqcup u_2$ and suppose that $u \in \tau$. Then either $u \in \tau_1$ or $u \in \tau_2$. If $u \in \tau_1$, then also $u_1 \sqsubseteq u$ would have to be in τ_1 by Lemma 2.9, which is a contradiction. If $u \in \tau_2$, then also $u_2 \sqsubseteq u$ would have to be in τ_2, which is also a contradiction. Hence, $u \notin \tau$. Moreover, $u \in V_{c(\tau_1)+c(\tau_2)} = V_{c(\tau)}$. □

Corollary 2.26 (Small model for semantic type-checking). *Let e be a PENRC(kind)-expression, let Γ be a type assignment under which e is well-defined, and let τ be a type. If e does not have output type τ under Γ, then there exists a natural number k, computable from e and τ, and a context $\sigma' \in \Gamma$ of width at most k such that $e(\sigma') \notin \tau$.*

Proof. Suppose that e does not have output type τ under Γ. Then there exists a context $\sigma \in \Gamma$ such that $e(\sigma) \notin \tau$. There exists a natural number l, computable from τ, and a value $u \sqsubseteq e(\sigma)$ of width at most l such that $u \notin \tau$ by Lemma 2.25. By Proposition 2.21 there exists a computable function c_e, computable from e, and a context $\sigma' \sqsubseteq \sigma$ of width at most $k := c_e(l)$ such that $u \sqsubseteq e(\sigma')$. Since $u \notin \tau$, $e(\sigma')$ is also not in τ by Lemma 2.9. Since $\sigma \in \Gamma$, also $\sigma' \in \Gamma$ by Lemma 2.9. □

It readily follows (cf. the proof of Theorem 2.8):

Proposition 2.27. *The semantic type-checking problem for PENRC(kind) is decidable.*

3

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

In this chapter we 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 [7, 15, 44, 45] and are widely used in general-purpose programming languages manipulating tree-structured data, such as XDuce [26, 27, 28], ℂDuce [23], and XQuery [6, 18]. The bounded-depth restriction is motivated by the fact that most tree-structured data (such as for example found in XML documents [61]) 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.

Our study is motivated by XQuery, the XML query language currently under development by the World Wide Web Consortium [6, 18]. Expressions in XQuery can be undefined. As an example, consider the following variation on one of the XQuery use cases [12]:

```
<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 [12] 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 [35] 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 [18] 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.

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 we have already shown in Section 2.3 that such base operations already cause the well-definedness problem for the positive-existential fragment of the NRC to become undecidable. 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 PENRC(*extract*) interpreted in a list-based data model is also decidable.

Organization This chapter is further organized as follows. We formally introduce the tree-structured, list-based data model in Section 3.1. In Section 3.2 we define the notion of a base operation and show how to extend such base operations to a query language QL(B). In Section 3.3 we introduce the well-definedness problem for QL(B) and introduce bounded-depth regular expression types. In Sections 3.4, 3.5, and 3.6 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 3.7. In that section we also show that the well-definedness problem for PENRC(*extract*) is decidable when we interpret this language in a list-based data model.

3.1 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 3.4), a list-based version of the complex object data model (as we show in Section 3.7.2), and the tree-structured data found in XML documents. For example, Figure 3.1(a) depicts a store where the first tree represents the XML fragment in Figure 3.1(b) and the second tree represents the XML fragment in Figure 3.1(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 expres-

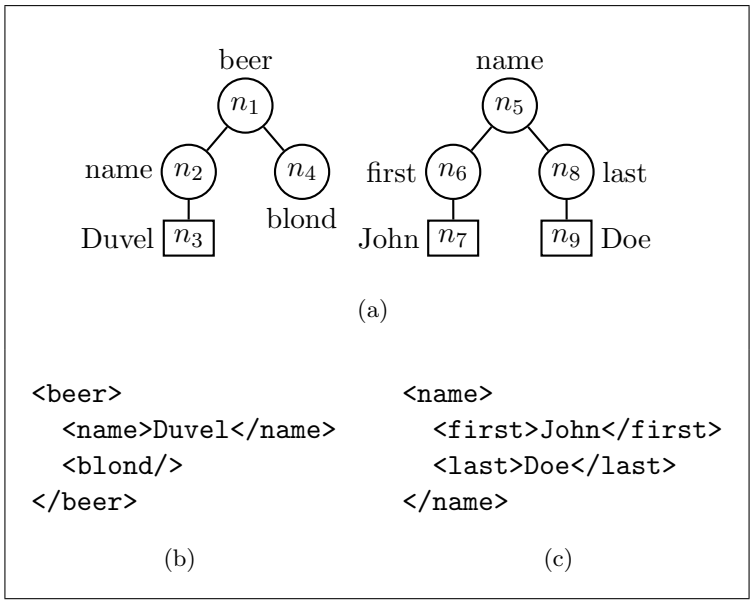


Figure 3.1: A store and the XML fragments it represents.

sions do not operate directly on XML text, but on instances of the XQuery data model [22]. 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, our aim is to study well-definedness, which is more fundamental than static or dynamic type-checking. Furthermore, we will not consider type tests, as we have already shown in Section 2.4 that these quickly turn the well-definedness problem undecidable. Values in our data model therefore correspond to un-

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

validated values in the XQuery data model. All references to the semantics of XQuery should hence also be understood to mean "the semantics of XQuery when the input is unvalidated".

3.1.1 Atoms and Nodes

As in Chapter 2, we assume given a recursively enumerable set $\mathcal{A} = \{a, b, \dots\}$ of *atoms* which we here assume to contain the booleans `true` and `false`. We further assume 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*.

3.1.2 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);
- $\lambda : V \to \mathcal{A}$ 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 3.1(a) depicts the store $(V, E, \lambda, <, \prec)$ where

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

[2] The notion of a store was first developed for the XQuery data model [25, 32].
[3] The label of a text node is called the node's *content* in XQuery terminology.

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 3.1(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 $\mathit{roots}(\Sigma)$ 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 $\varnothing$.

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.

3.1.3 Value-Tuples

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 *tree value* is a value-tuple of arity one. For convenience we will abbreviate "tree value" by "value" in this

chapter. Furthermore, we will 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 $rng(s)$ for the set of items occurring in s.

3.1.4 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 $\rho(v) = v'$.

3.1.5 Conventions

We will further use the following conventions throughout this chapter. We will abbreviate tuples such as $t_1, \ldots, t_p$ by $\vec{t}$ 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 $g(i) = j$.

3.2 Syntax and Semantics

3.2.1 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 a 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 $s'(j) = \mathit{fold}(r)$ 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 3.2 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 3.2.

- 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 $(\Sigma; \langle i \in \mathcal{A}\rangle)$.

- 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 3.3.

  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

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

– 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 3.2 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 3.2 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.

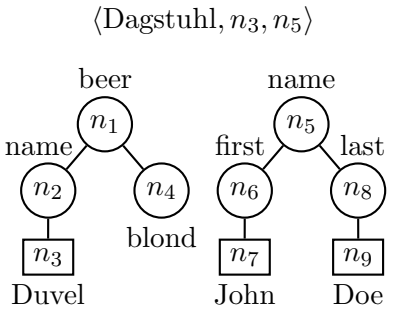
3.2.2 Expressions

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 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 $v \in R(w)$.

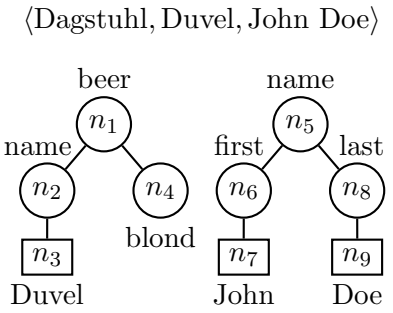
The syntax of QL(B) is defined by the following grammar:[5]

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

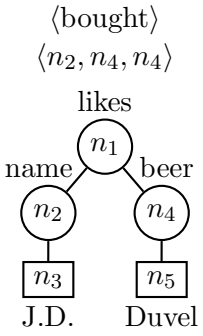
[5] One may wonder why we consider let-expressions of the form `let` $x := e_1$ `return` e_2. As will become clear from the semantics as defined in Section 3.2.3, such expressions are not redundant. This is due to the fact that base operations can create new nodes. Hence, `let` $x := e_1$ `return` e_2 is not necessarily equivalent to the expression we obtain by replacing every free occurrence of x in e_2 by e_1.



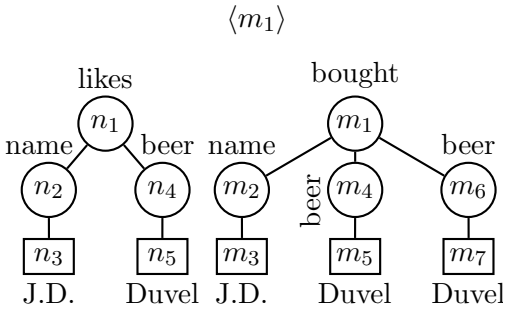
(a) Value u



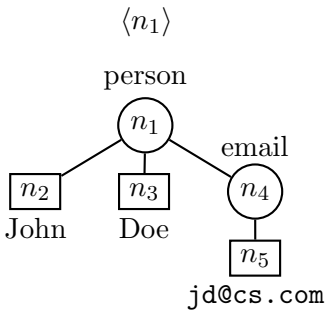
(b) Value in $data(u)$



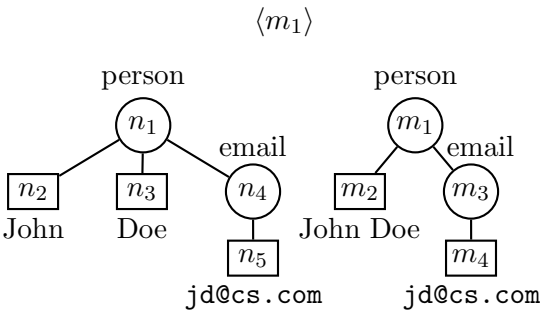
(c) Value-tuple v



(d) Value in $element(v)$



(e) Value w



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

Figure 3.2: Illustration of the base operations *data*, *element*, and *merge-text*.

Here, e ranges over QL(B) *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 `if` e_1 `then` e_2 `else` e_3 is the union of the free variables of their immediate subexpressions, and the free variables of `let` $x :=$ e_1 `return` e_2 and `for` x `in` e_1 `return` e_2 is $FV(e_1) \cup (FV(e_2) \setminus \{x\})$.

3.2.3 Semantics

The input to a QL(B) expression e is described by a *tree 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*. Using a similar notation as in Chapter 2 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 $y \neq x$. For convenience we will abbreviate "tree context" by "context" in this chapter.

The semantics of QL(B) expressions is described by means of the *evaluation relation*, as defined in Figure 3.3. 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, $(\Sigma; \sigma') \models e \Rightarrow (\Sigma'; s)$.

Example 3.1. 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 3.2. In Section 3.2.1 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. □

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

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

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

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

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

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

Figure 3.3: The evaluation relation for QL(B) expressions.

Example 3.3. In Section 3.2.1 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 3.4. 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 `$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 3.5. 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 3.1.

```
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 tree 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 $e \in \mathrm{QL}(B)$ 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 3.6. *Every expression e in $\mathrm{QL}(B)$ 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 3.7. *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 should also 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 3.8. *Every expression in $\mathrm{QL}(B)$ defines a reachable-only relation.*

Proof. Let e be an expression in $\mathrm{QL}(B)$ 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:

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

- 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 3.7 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:

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

- 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 3.7 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:

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

- 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 3.7 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:

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

□

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

Lemma 3.9. *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 3.10. *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. If ρ is 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 3.11. *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* $\Sigma, \Sigma_1, \ldots, \Sigma_k$ *are all pairwise disjoint,* $\Sigma', \Sigma'_1, \ldots, \Sigma'_k$ *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 3.10 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 3.9 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 3.12. *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 Σ'_1 and Σ'_2 such that $\Sigma_1 = \Sigma \circ \Sigma'_1$ and $\Sigma_2 = \Sigma \circ \Sigma'_2$. Since R is also 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. The result then follows by applying Lemma 3.11 on $(\Sigma; \vec{s})$, $(\Sigma \circ \Sigma'_1; s_1)$ and $(\Sigma; \vec{s})$, $(\Sigma \circ \Sigma'_2; s_2)$. □

In a similar way we obtain:

Corollary 3.13. *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 3.14. *The relation defined by an expression in* $\mathrm{QL}(B)$ *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 $e_1(\Sigma;\sigma)$ contains a value of the form $(\Sigma_1; \langle \texttt{true} \rangle)$, then *all* values in $e_1(\Sigma;\sigma)$ have that form. Similarly, if $e_1(\Sigma;\sigma)$ contains a value of the form $(\Sigma_1; \langle \texttt{false} \rangle)$, then *all* values in $e_1(\Sigma;\sigma)$ have that form. 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 3.12 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 3.11 it follows that there exists a node-renaming ρ such that

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

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 3.13 that $|s| = |s'|$ and that there exists a node-renaming ρ_j for every $j \in [1, |s|]$ such that

$$\rho_j(\Sigma_0; x\colon \langle s(j)\rangle, \sigma) = (\Sigma'_0; x\colon \langle s'(j)\rangle, \sigma). \tag{3.1}$$

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, (3.1) implies that Σ_0 is node-isomorphic to Σ'_0, it follows from Lemma 3.11 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.

3.3 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 tree contexts on e, whether e is defined on every tree context in S. The answer to this problem clearly depends on both the set B of base operations and the class of tree context sets used as inputs. Here, we will focus on the class of tree context sets specified by bounded-depth regular expression types. Regular expression types are based on regular tree languages [7, 15, 44, 45] and are widely used in general-purpose programming languages manipulating tree-structured data, such as XDuce [26, 27, 28], ℂDuce [23], and XQuery [6, 18]. The bounded-depth restriction is motivated by the fact that most tree-structured data (such as for example found in XML documents [61]) in practice has nesting depth at most five or six, and that unbounded-depth nesting is hence often not needed.

Formally, a *(bounded-depth) regular expression 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 regular expression type denotes a set of values, as defined in Figure 3.4. Here we denote trees by Θ. For ease of notation we will not distinguish between a regular expression type and the set of values it denotes. A *regular expression type assignment* Γ on an expression e is a function from a finite superset

$$\frac{a \in \mathcal{A}}{(\varnothing; \langle a \rangle) \in \textbf{Atom}} \qquad \frac{n \in \mathcal{N}^t \text{ is } \Theta\text{'s root}}{(\Theta, \langle n \rangle) \in \textbf{Text}}$$

$$\frac{\begin{array}{c} n \in \mathcal{N}^e \text{ is } \Theta\text{'s root} \quad \lambda_\Theta(n) = a \\ \text{children of } n \text{ in } \Theta \text{ are } n_1 < \ldots < n_k \\ (\Theta|_{n_1} \circ \cdots \circ \Theta|_{n_k}; \langle n_1, \ldots, n_k \rangle) \in \tau \end{array}}{(\Theta, \langle n \rangle) \in \textbf{Element}(a, \tau)} \qquad \frac{}{(\varnothing, \langle\rangle) \in \textbf{Empty}}$$

$$\frac{(\Sigma, s) \in \tau_1 \text{ or } (\Sigma, s) \in \tau_2}{(\Sigma, s) \in \tau_1 + \tau_2} \qquad \frac{\begin{array}{c} (\Sigma_1, s_1) \in \tau_1 \quad (\Sigma_2, s_2) \in \tau_2 \\ \Sigma_1 \text{ is disjoint with } \Sigma_2 \end{array}}{(\Sigma_1 \circ \Sigma_2, s_1 \circ s_2) \in \tau_1 \circ \tau_2}$$

$$\frac{\begin{array}{c} (\Sigma_1, s_1) \in \tau \quad \cdots \quad (\Sigma_p, s_p) \in \tau \quad p \geq 0 \\ \Sigma_j \text{ is disjoint with } \Sigma_{j'} \text{ when } j \neq j' \end{array}}{(\Sigma_1 \circ \cdots \circ \Sigma_p, s_1 \circ \ldots \circ s_p) \in \tau^*}$$

Figure 3.4: The denotation of regular expression types.

$\{x, \ldots, y\}$ of the free variables of e to regular expression types. A regular expression 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 regular expression type assignment and the set of contexts it denotes. For convenience we will abbreviate "regular expression type" and "regular expression type assignment" by "type" respectively "type assignment" in this chapter.

Definition 3.15. Let B be a finite set of base operations. We say that $e \in \mathrm{QL}(B)$ is *well-defined* under a regular expression type assignment Γ on e if $e(\Sigma; \sigma) \neq \emptyset$ for every context $(\Sigma; \sigma) \in \Gamma$. The *well-definedness problem for* $\mathrm{QL}(B)$ consists of checking, given e and Γ, whether e is well-defined under Γ.

3.4 Satisfiability and The Restriction to Monotone Base Operations

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.

Definition 3.16. 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 $\mathrm{QL}(B)$ consists of checking, given e and Γ, whether e is satisfiable under Γ.

Since every expression defines a semi-function by Proposition 3.6, 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 $\mathrm{QL}(B)$ 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 3.17. *If the satisfiability problem for* $\mathrm{QL}(B)$ *is undecidable, then the well-definedness problem for* $\mathrm{QL}(B)$ *is also undecidable.*

The converse is not true however. Indeed, in Section 3.6.1 we will give a set of base operations B for which the well-definedness of $\mathrm{QL}(B)$ is undecidable, but the satisfiability problem of $\mathrm{QL}(B)$ is nevertheless decidable.

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

Proposition 3.18. *If B includes the base operations concat, children, eq, node-name, content, element, and empty, then* $\mathrm{QL}(B)$ *can simulate the relational algebra. Concretely, for every relational algebra expression ϕ over database schema S there exists an expression $e_\phi \in \mathrm{QL}(B)$ 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 $\mathrm{QL}(B)$ *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 3.5(a) can be encoded as the value $(\Sigma; s)$ in Figure 3.5(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 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 3.5(c) is also a valid encoding of the relation in Figure 3.5(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 x1, x2 in children(t) return
    if eq(node-name(x1), A1)
```

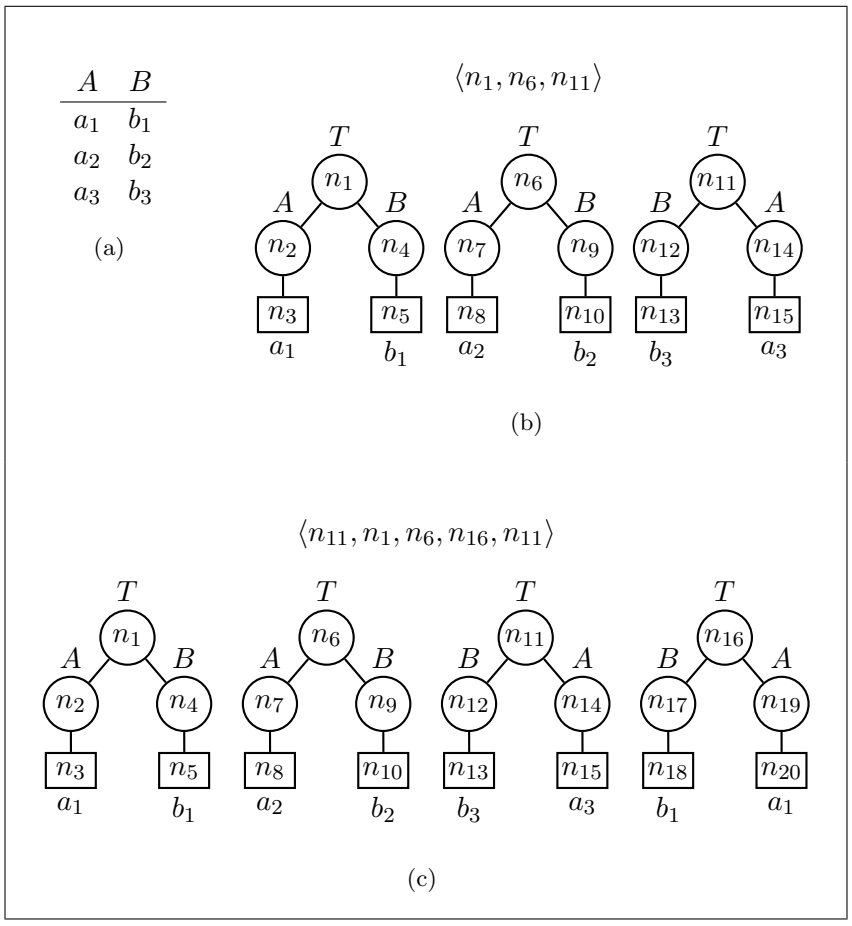


Figure 3.5: Encoding relations as values.

```
      and eq(node-name(x2), A2)
      and eq(content(children(x1)), content(children(x2)))
    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(x1), A1)
        or ...
        or eq(node-name(xk), Ak)
      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 x_1,...,x_k in children(t_1) return
  for y_1,...,y_k in children(t_2) return
    if eq(node-name(x_1), A_1)
      and ...
      and eq(node-name(x_k), A_k)
      and eq(node-name(y_1), A_1)
      and ...
      and eq(node-name(y_k), A_k)
      and eq(content(children(x_1)), content(children(y_1)))
      and ...
      and eq(content(children(x_k)), content(children(y_k)))
    then t_1 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 3.19. *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 3.18 still works if we replace *eq* and *element* by their XQuery counterparts (whose semantics was given in Examples 3.2 and 3.3).

3.4.1 Monotone Base Operations

Corollary 3.19 is the QL(B) equivalent of Theorem 2.3 in the NRC. We obtained a fragment of the NRC for which well-definedness is decidable by

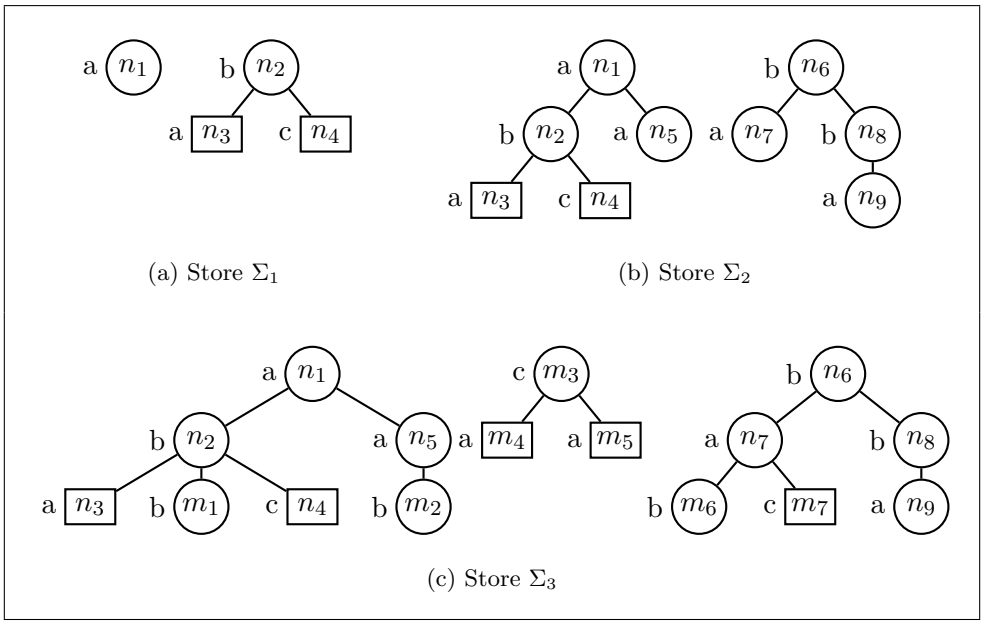


Figure 3.6: Containment of stores. Store Σ_1 is not contained in Σ_2 or in Σ_3. Store Σ_2 is contained in Σ_3.

restricting ourselves to the positive-existential expressions, which are monotone by Lemma 2.4. In the hope of finding QL(B) for which well-definedness is decidable, it may hence be 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 value containment to (ordered) tree 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 3.6. 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* [8]: 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 $s(j) = s'(h(j))$ 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: $(\Sigma; s_1, \ldots, s_p)$ is contained in $(\Sigma'; s'_1, \ldots, s'_p)$ 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 3.20. 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,

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

- 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 $element(w)$. We show that there exists a value in $element(v)$ 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 $(\Sigma \circ \Theta; \langle m'\rangle) \in element(v)$. 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 3.20 we obtain:

Proposition 3.21. *The base operations concat, children, descendant, parent, ancestor, preceding-sibling, following-sibling, eq, is, ≪, 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 3.19 are monotone by Proposition 3.21. Fortunately:

Example 3.22. 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 $(\Sigma; \langle\rangle) \sqsubseteq (\Sigma; \langle a \rangle)$. □

3.4.2 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 $b \neq a$.[6] 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, *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 *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*, *content*, *element*, *text*, *merge-text*) are at least as expressive as QL(*concat*, *children*,

[6] 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.

eq, *node-name*, *element*, *empty*). It follows by Propositions 3.17 and 3.18 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.

3.4.3 Monotone Expressions

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

Lemma 3.23. *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 3.24. *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 3.25. *Let B be a finite set of monotone base operations. Every expression e in $\mathrm{QL}(B)$ defines a monotone relation.*

Proof. Let e be an expression in $\mathrm{QL}(B)$. Let $(\Sigma;\sigma)$ and $(\Sigma';\sigma')$ be two contexts on e such that $e(\Sigma;\sigma) \neq \emptyset$, $e(\Sigma';\sigma') \neq \emptyset$, and $(\Sigma;\sigma) \sqsubseteq (\Sigma';\sigma')$.[7] 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 3.6) and that if an expression is defined on an input, it is also defined on all the node-isomorphic copies of this input (Lemma 3.24).

- 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 exists $(\Sigma'_1; \langle b \rangle) \in e_1(\Sigma';\sigma')$ 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 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 $e_2(\Sigma;\sigma) = e(\Sigma,\sigma)$

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

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 exists $(\Sigma_1';s_1') \in e_1(\Sigma';\sigma')$ 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 3.12 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
$$w \in f(\Sigma' \circ \bigcirc_{j=1}^{p} \Sigma_j';s_1',\ldots,s_p').$$
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
$$f(\Sigma \circ \bigcirc_{j=1}^{p} \Pi_j;t_1,\ldots,t_p) \neq \emptyset.$$
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 3.23 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 3.11 we hence obtain that
$$(\Sigma \circ \bigcirc_{j=1}^{p} \Sigma_j;s_1,\ldots,s_p) \equiv_{node} (\Sigma \circ \bigcirc_{j=1}^{p} \Pi_j;t_1,\ldots,t_p).$$
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
$$(\Sigma \circ \bigcirc_{j=1}^{p} \Sigma_j;s_1,\ldots,s_p) \sqsubseteq (\Sigma' \circ \bigcirc_{j=1}^{p} \Sigma_j';s_1',\ldots,s_p').$$
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 3.13 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 3.23 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 $e(\Sigma; \sigma)$. □

3.5 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

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

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 [39], 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 3.26. *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.*[8]

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 3.27. *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.

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

3.6 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 3.28. 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 3.29. *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 [29]. 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 [29] (p. 317) that it is undecidable to check whether $P_1(x_1, \ldots, x_p) \leq P_2(x_1, \ldots, x_p)$ 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 $concat(x, y)$ on $(\Sigma; \sigma)$ then has width $k + l$. Moreover, the list of the value returned by `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 3.30. *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* $v \in \Gamma$.[9]

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 3.29, it follows that the containment problem is also undecidable. □

3.6.1 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 3.31. *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 3.29, 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 3.7.1:

Proposition 3.32. *The satisfiability problem for* QL(*concat*, *smaller-width*) *is decidable.*

[9] We note that, in contrast, the corresponding problem in a set-based data model is decidable [17].

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 contrast, we have shown in Section 2.2.1 that the PENRC *is* monotone in its undefinedness behavior (Lemma 2.4), and we have used this property to show decidability of well-definedness for the PENRC in Section 2.2.2. In order to obtain QL(B) for which well-definedness is decidable, we could hence restrict ourselves to base operations which are also monotone in their undefinedness behavior. In that case, however, we would disallow base operations such as *is-element*, *is-text*, *is-atom*, *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 $P_j \subseteq \mathit{rng}(h_j)$. 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 3.6(b) respectively Figure 3.6(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 $\{2\} \subseteq \mathit{rng}(h)$.

The following lemma establishes some basic properties of requirements.

Lemma 3.33. *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 3.33 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 3.33 that $|s'_1| \geq 1$ and $|s'_2| \geq 2$. Hence, *eq* is undefined on $(\Sigma'; s'_1, s'_2)$.

Note that reasons are not necessarily unique. For example, $(\emptyset; \{1, 2\}, \{2\})$ is another reason why *eq* is undefined on $(\varnothing; \langle a, c\rangle, \langle a, b, c\rangle)$. Also note that 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 3.34. 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 3.33 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 3.33 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 3.33 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 3.33 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 3.33 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 3.33. 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 3.34 we obtain:

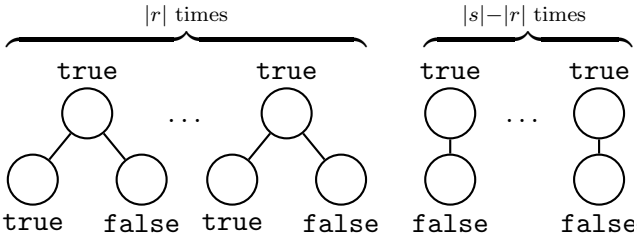
Proposition 3.35. *The base operations concat, children, descendant, parent, ancestor, preceding-sibling, following-sibling, data, eq, is, $\ll$, is-element, is-text, is-atom, node-name, content, element, merge-text, text, empty, $+$, and $\times$ are all locally-undefined.*

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.

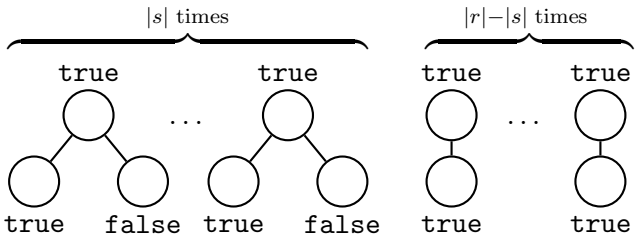
3.6.2 Non-Local Behavior

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 3.34, 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 3.36. *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 $P_1 \subseteq rng(h)$. 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 3.36 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 3.29 that

Proposition 3.37. *Well-definedness for* QL(*concat*, *zip*, *has-false*) *is undecidable.*

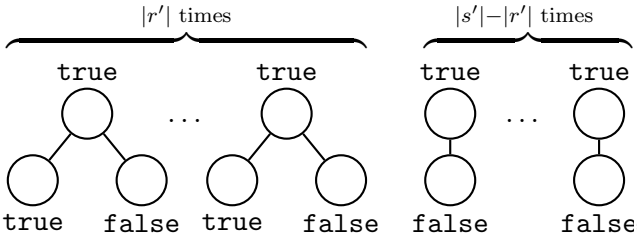
This undecidability is not due to the fact that our set of base operations contains a non-monotone or non-generic base operation. Indeed, we already know that *concat* is monotone and generic from Propositions 3.21 and 3.26. Similarly, it is quite easy to see that both *zip* and *has-false* are generic.[10] 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 3.38. *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.

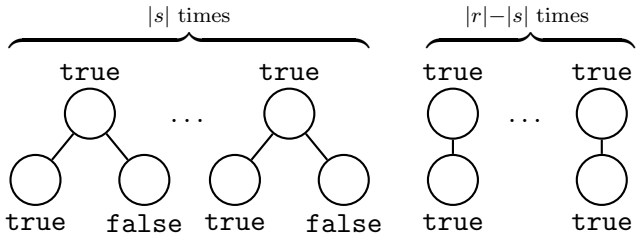
We only treat the case where $|r'| \leq |s'|$, the other case is similar. By definition of *zip* we know that Π' is of the form

[10] 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`.



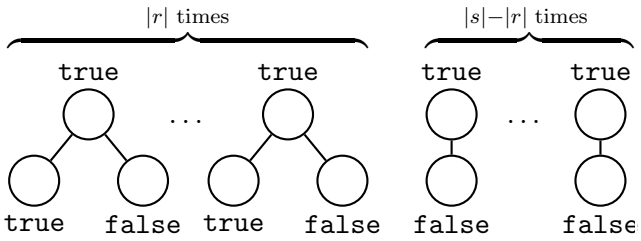
We observe the following cases.

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



  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 3.33; 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 3.39. 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 3.33 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 3.40. Let us give some examples of local base operations.

- The base operation *smaller-width* introduced in Section 3.6.1 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 $\mathit{smaller\text{-}width}(v')$ can only be $\{(\Sigma';\langle\rangle)\}$, we have $\mathit{smaller\text{-}width}(v') \cap [\mathbf{w},w] \neq \emptyset$, 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 $\mathit{children}(v) = \{(\Sigma;t)\}$ 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 $[\mathbf{w},w] \triangleleft \mathit{children}(v)$. Indeed, let $v' = (\Sigma';s') \in [\mathbf{v},v]$ and suppose that $\mathit{children}(v')$ 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 3.33 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. □

Using similar reasonings as the ones employed in Example 3.40 we obtain:

Proposition 3.41. *The base operations concat, children, descendant, parent, ancestor, preceding-sibling, following-sibling, data, eq, is, ≪, is-element, is-text, is-atom, node-name, content, element, merge-text, text, empty, +, ×, 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(+, ×), and QL(*concat*, *smaller-width*), for which well-definedness is undecidable.

3.6.3 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 $\mathrm{QL}(B)$. 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 3.7. 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 $\mathrm{QL}(B)$. We start out by stating the following technical lemmas.

Lemma 3.42. *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 3.43. *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 3.23 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 3.44. *If B is a finite set of monotone and local base operations, then every expression $e \in \mathrm{QL}(B)$ 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 3.6 and 3.25. 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 3.24.

- 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 $|\mathbf{w}|$.[11]

[11] Here we extend the notion of a requirement to contexts in the obvious way: if σ is an

- 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 $\max\{c_{e_2}(|\mathbf{w}|), c_{e_3}(|\mathbf{w}|)\} = c_e(|\mathbf{w}|)$.

- 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

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

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

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

$$\begin{aligned}(\Sigma''_1; x\colon s''_1, \sigma') &\in [(V_1; x\colon P_1, \phi'), (\Sigma_1; x\colon s_1, \sigma)] \\ &\subseteq [(V_1; x\colon P_1, \phi_1), (\Sigma_1; x\colon s_1, \sigma)].\end{aligned} \tag{3.2}$$

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 3.12. Since $e_2(\Sigma'_1; x\colon s'_1, \sigma')$ is defined and since e_2 is node-generic, it follows that

$$e_2(\Sigma''_1; x\colon s''_1, \sigma') \neq \emptyset. \tag{3.3}$$

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 (3.2) and (3.3) 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

$$f(\Sigma' \circ \bigcirc_{j=1}^{p} \Sigma'_j; s'_1, \ldots, s'_p) \neq \emptyset. \tag{3.4}$$

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 3.43 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,

$$(\Sigma' \circ \bigcirc_{j=1}^{p} \Sigma''_j; s''_1, \ldots, s''_p) \in \\ [(V \cup \bigcup_{j=1}^{p} V_j; P_1, \ldots, P_p), (\Sigma \circ \bigcirc_{j=1}^{p} \Sigma_j; s_1, \ldots, s_p)]. \quad (3.5)$$

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 3.11 that

$$(\Sigma' \circ \bigcirc_{j=1}^{p} \Sigma'_j; s'_1, \ldots, s'_p) \equiv_{node} (\Sigma' \circ \bigcirc_{j=1}^{p} \Sigma''_j; s''_1, \ldots, s''_p). \quad (3.6)$$

Since f is node-generic it follows from (3.4) and (3.6) that

$$f(\Sigma' \circ \bigcirc_{j=1}^{p} \Sigma''_j; s''_1, \ldots, s''_p) \neq \emptyset. \quad (3.7)$$

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 (3.5) and (3.7) 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 $e =$ `for` x `in` e_1 `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

$$[(V_0 \cup V_j; P_j), (\Sigma_0 \circ \Sigma_j; s_j)] \triangleleft e_2(\Sigma_0; x\colon \langle s(j)\rangle; \sigma) \tag{3.8}$$

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 3.13 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

$$(\Sigma''_0 \circ \Sigma''_j; s''_j) \in e_2(\Sigma''_0; x\colon \langle s''(h^{-1}(j))\rangle, \sigma') \cap [(V_0 \cup V_j; Q_j), (\Sigma_0 \circ \Sigma_j; s_j)], \quad (3.9)$$

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

$$(\Sigma''_0 \circ \bigcirc_{j=1}^{|s|} \Sigma''_j; \bigcirc_{j=1}^{|s|} s''_j) \in e(v'). \quad (3.10)$$

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 3.42 and (3.10) 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 (3.9). 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

$$(\Sigma_0''; x\colon \langle s''(h^{-1}(j)) \rangle, \sigma') \in [(V_0 \cup W_j, x : Q_j, \phi_j), (\Sigma_0; x\colon \langle s(j) \rangle, \sigma)]\,. \quad (3.11)$$

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 (3.8) that $(V_0 \cup W_j, x\colon Q_j, \phi_j)$ is a reason why

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

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 (3.11) holds, it follows by Lemma 3.43 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 3.23. 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 3.45. *If B is a finite set of monotone, local, and locally undefined base operations, then every expression e in $\mathrm{QL}(B)$ 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 3.44. 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 3.6, 3.25, and 3.44. 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 3.24.

- 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 3.44 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 3.33 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 size 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 3.44 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 3.12 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 3.44 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,

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

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 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 3.43 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

$$f(\Sigma' \circ \bigcirc_{j=1}^{p} \Sigma'_j; s'_1, \ldots, s'_p) = \emptyset. \tag{3.12}$$

Furthermore, since every e_j is a node-generic it follows from Lemma 3.11 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

$$(\Sigma' \circ \bigcirc_{j=1}^{p} \Sigma''_j; s''_1, \ldots, s''_p) \equiv_{node} (\Sigma' \circ \bigcirc_{j=1}^{p} \Sigma'_j; s'_1, \ldots, s'_p). \tag{3.13}$$

Since f is node-generic it follows by (3.12) and (3.13) 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 3.44 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 3.33 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 3.13 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. □

3.7 Decidability Results

The restrictions proposed in Sections 3.4, 3.5, and 3.6 are strong enough to guarantee decidability of well-definedness:

Theorem 3.46. *If B is a finite set of monotone, generic, local, and locally-undefined base operations, then the well-definedness problem for $\mathrm{QL}(B)$ is decidable.*

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

Definition 3.47. 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 3.48. *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 $\tau = \tau_1 \circ \tau_2$. Then w is of the form $(\Sigma_1 \circ \Sigma_2; s_1 \circ s_2)$ with $(\Sigma_1; s_1) \in \tau_1$ and $(\Sigma_2; s_2) \in \tau_2$ since $w \in \tau$. 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

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

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

$$(\Sigma_1'; s_1') \in [(V_1, P_1), (\Sigma_1; s_1)] \cap \tau_1$$
$$(\Sigma_2'; s_2') \in [(V_2, P_2), (\Sigma_2; s_2)] \cap \tau_2$$

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

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

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 3.42 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 3.42 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 3.46. 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 3.45 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 3.48 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 3.27, $e(\rho(v))$ 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 3.2, except *data*, *merge-text*, and *empty* are monotone, generic, locally-undefined and local by Propositions 3.21, 3.26, 3.35, and 3.41, it follows in particular:

Corollary 3.49. *Well-definedness for the XQuery fragment* QL(*concat, children, descendant, parent, ancestor, preceding-sibling, following-sibling, eq, is,* $\ll$*, is-element, is-text, is-atom, node-name, content, element, text*) *is decidable.*

It follows from Proposition 3.17 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].

3.7.1 Satisfiability for QL(concat, smaller-width)

Remember from Section 3.6.1 that the well-definedness problem for QL(*concat*, *smaller-width*) is undecidable. Using Proposition 3.44 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 3.50. *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 3.21 and 3.35, it follows from Proposition 3.44 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 3.48 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 3.33 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 3.27 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 3.6, 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. □

3.7.2 Well-definedness for the Nested Relational Calculus over Lists

In Section 2.3 we have shown that the well-definedness problem for the PENRC in the presence of the singleton coercion operator *extract* is undecidable. The core difficulty there was that *extract* is undefined on non-singleton inputs. As such, $\mathit{extract}(\{e_1, e_2\})$ is defined if, and only if, expressions e_1 and e_2 return the same result on every input. Therefore, in order to solve the well-definedness problem one also needs to solve the equivalence problem, which we have shown 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 3.49 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 Complex Object Data Model A *list-based complex object value* (LB-value for short) is either an atom, a pair of LB-values, or a finite

list of LB-values. Note that, in contrast to the QL data model, lists can hence contain other lists.

List-Based PENRC with Singleton Coercion A *list-based complex object context* (LB-context for short) is a function σ from a finite set of variables $dom(\sigma)$ to LB-values. The list-based semantics of PENRC(*extract*) expressions on LB-contexts is described by means of the evaluation relation defined in Figure 3.7. It is easy to see that the evaluation relation remains functional: an expression evaluates to at most one LB-value on a given LB-context. The evaluation relation also remains partial. Indeed, we can only project on pairs, concatenate lists, flatten lists of lists, iterate over lists, and test equality on atoms. Finally, the list-based semantics of an expression only depends on its free variables: if two LB-contexts σ and σ' on e are equal on $FV(e)$, then $\sigma \models e \Rightarrow v$ if, and only if, $\sigma' \models e \Rightarrow v$.

List-based NRC types A *list-based complex object type* (LB-type for short) is a term generated by the following grammar:

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

An LB-type τ *denotes* a set $[\![\tau]\!]$ of LB-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 $[\![\tau]\!]$. An *LB-type assignment* Γ is a function from a finite set of variables $dom(\Gamma)$ to LB-types. An LB-type assignment denotes the set of LB-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 an LB-type assignment with its denotation.

Well-Definedness

Definition 3.51. Let e be a PENRC(*extract*) expression and let Γ be an LB-type assignment on e. If $e(\sigma)$ is defined for every context $\sigma \in \Gamma$ (according to the semantics in Figure 3.7), then e is *well-defined under* Γ. The *well-definedness problem* for the list-based PENRC(*extract*) consists of checking, given e and Γ, whether e is well-defined under Γ.

Variables

$$\frac{}{\sigma \models x \Rightarrow \sigma(x)}$$

Pair operations

$$\frac{\sigma \models e_1 \Rightarrow v_1 \quad \sigma \models e_2 \Rightarrow v_2}{\sigma \models (e_1, e_2) \Rightarrow (v_1, v_2)} \qquad \frac{\sigma \models e \Rightarrow (v_1, v_2)}{\sigma \models \pi_1(e) \Rightarrow v_1} \qquad \frac{\sigma \models e \Rightarrow (v_1, v_2)}{\sigma \models \pi_2(e) \Rightarrow v_2}$$

List operations

$$\frac{}{\sigma \models \emptyset \Rightarrow \langle\rangle} \qquad \frac{\sigma \models e \Rightarrow v}{\sigma \models \{e\} \Rightarrow \langle v \rangle} \qquad \frac{\sigma \models e_1 \Rightarrow v_1 \quad \sigma \models e_2 \Rightarrow v_2 \quad v_1 \text{ and } v_2 \text{ lists}}{\sigma \models e_1 \cup e_2 \Rightarrow v_1 \circ v_2}$$

$$\frac{\sigma \models e \Rightarrow \langle v_1, \ldots, v_k \rangle \quad v_1, \ldots, v_n \text{ lists}}{\sigma \models \bigcup e \Rightarrow v_1 \circ \cdots \circ v_k} \qquad \frac{\sigma \models e_1 \Rightarrow \langle v_1, \ldots, v_k \rangle \quad (x\colon v_j, \sigma) \models e_2 \Rightarrow w_j \quad j \in [1,k]}{\sigma \models \{e_2 \mid x \in e_1\} \Rightarrow \langle w_1, \ldots, w_k \rangle}$$

Conditional test

$$\frac{\sigma \models e_1 \Rightarrow a \quad \sigma \models e_2 \Rightarrow b \quad \sigma \models e_3 \Rightarrow v \quad a = b}{\sigma \models e_1 = e_2 \ ?\ e_3\ :\ e_4 \Rightarrow v} \qquad \frac{\sigma \models e_1 \Rightarrow a \quad \sigma \models e_2 \Rightarrow b \quad \sigma \models e_4 \Rightarrow v \quad a \neq b}{\sigma \models e_1 = e_2 \ ?\ e_3\ :\ e_4 \Rightarrow v}$$

Figure 3.7: The list-based semantics of PENRC(*extract*) expressions.

Theorem 3.52. *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 3.49. Specifically, let e be a PENRC(*extract*) expression in and let Γ be a LB-type assignment on e. We will show that there exists

1. a one-to-many encoding of LB-values as tree values;
2. a regular expression type assignment $enc(\Gamma)$, computable from Γ, such that every tree context in $enc(\Gamma)$ is an encoding of some LB-context in Γ and every LB-context in Γ has an encoding in $enc(\Gamma)$; and
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 an LB-value. We define the set $enc(v)$ of tree 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 tree 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 tree 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 tree 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 3.8. 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 an LB-type, then we define the regular expression type $enc(\tau)$ which simulates τ as follows by induction on τ.

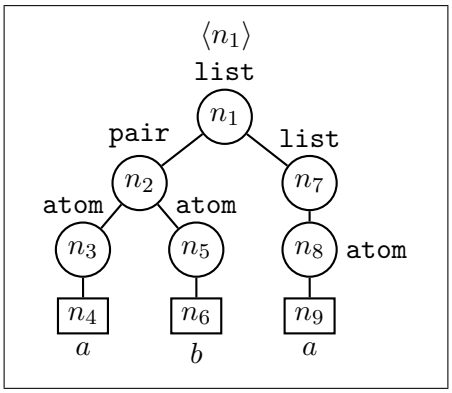


Figure 3.8: One encoding of the LB-value $\langle (a, b), \langle a \rangle \rangle$ in the QL data model.

- 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

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

- 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 $(\Sigma; \sigma') \in enc(\sigma)$. □

Part II

Type Inference and Typability

4

The Complexity of Deciding Typability for the Relational Algebra

As we have seen in Chapters 2 and 3, both the well-definedness problem and the semantic type-checking problem remain undecidable for database query languages which are powerful enough to simulate the relational algebra. It follows that a sound and complete type system for such languages does not exist (although we have identified several useful fragments for which such a type system does exist). For such languages, static verification of the absence of certain programming errors hence has to be done by means of a traditional, incomplete type system.

In the second part of this dissertation we therefore study classical type system problems from the theory of programming languages in the context of database query languages. In this chapter we study the complexity of deciding typability for the relational algebra. Checking typability of relational algebra expressions is the analog in the relational algebra of static type-checking in implicitly typed programming languages with polymorphic type systems, such as ML [55]. It is therefore interesting to see what its complexity is. It is known for instance that typability is P-complete for the simply typed lambda calculus [19] and EXPTIME-complete for ML [31, 34]. In contrast, Van den Bussche and Waller have shown that typability for the relational algebra is in NP [56]. The precise complexity remained open, however. In this chapter

we will show that the problem is NP-complete in general. In particular, we show that the problem becomes NP-hard due to (1) the cartesian product operator; (2) the selection operator on arbitrary sets of typed predicates; and (3) the selection operator on "well-behaved" sets of typed predicates together with join and projection or renaming. However, the problem is in P when (1) we only allow union, difference, join, and selection on "well-behaved" sets of typed predicates; or (2) we allow all operators except cartesian product, where the set of selection predicates can mention at most one base type. Most of these results follow from a close connection of the typability problem to non-uniform constraint satisfaction.

Organization This chapter is further organized as follows. We introduce the relational algebra type system in Section 4.1, including the notions of well-typedness and typability. We then show that the typability problem is NP-complete in its most general setting in Section 4.2. In Section 4.3 we illustrate the close connection between the typability problem and constraint satisfaction problems, which allows us to obtain most of the results mentioned above.

4.1 Preliminaries

We assume given a sufficiently large set $\{A, B, \dots\}$ of *attribute names*. We also assume given a finite, non-empty set of *base types* (such as `int`, `string`, `bool`, ...) and a sufficiently large set of *predicates* (such as $=, \leq, \dots$). Every predicate θ has an *arity* $|\theta|$, which is a natural number, and a *signature* $\varsigma(\theta)$, which is a non-empty, $|\theta|$-ary relation over the set of base types. An example of a signature for the binary predicate *has_length* is $\{(\texttt{string}, \texttt{int})\}$, while an example of a signature for the binary predicate $=$ is

$$\{(\texttt{int}, \texttt{int}), (\texttt{bool}, \texttt{bool}), (\texttt{string}, \texttt{string})\}.$$

Base types will be denoted by β, possibly subscripted. Predicates will be denoted by θ, possibly subscripted. Finite sets of predicates will be denoted by Θ. The set of all base types mentioned in signatures of predicates in Θ will be denoted by $\beta(\Theta)$.

The relational algebra with selection predicates in Θ, denoted by $\mathcal{R}^{\Theta}$, is the set of all expressions generated by the following grammar:

$$\begin{array}{rcl} e & ::= & x \\ & | & (e \cup e) \mid (e - e) \mid (e \bowtie e) \mid (e \times e) \\ & | & \sigma_{\theta(A_1,\dots,A_n)}(e) \mid \pi_{A_1,\dots,A_n}(e) \mid \rho_{A/B}(e) \end{array}$$

Here e ranges over relational algebra expressions, x ranges over variables, θ ranges over selection predicates in Θ, n is the arity of θ, and A, B, and A_i are attribute names. If V is a set of operators, then $\mathcal{R}_V^\Theta$ denotes the subset of expressions in $\mathcal{R}^\Theta$ using only operators in V. The semantics of relational algebra is the well-known one [1, 16] and will actually not concern us in the present chapter. The set of all variables occurring in expression e is denoted by $FV(e)$ and the set of all attributes occurring in e is denoted by $\mathit{Specattrs}(e)$.

A *relational type* τ is a function from a finite set of attribute names $dom(\tau)$ to the set of base types. Two relational types are *compatible*, denoted by $\tau_1 \sim \tau_2$, if $\tau_1(A) = \tau_2(A)$ for every A in $dom(\tau_1) \cap dom(\tau_2)$. Clearly, the union of two compatible relational types is defined, and is again a relational type. If τ is a relational type, then $\rho_{A/B}(\tau)$ is the relational type with domain $dom(\tau) \setminus \{A\} \cup \{B\}$ such that $\rho_{A/B}(\tau)(B) = \tau(A)$ and $\rho_{A/B}(\tau)(C) = \tau(C)$ for every $C \in dom(\tau) \setminus \{A, B\}$. A *relational type assignment* Γ is a function from a finite set of variables $dom(\Gamma)$ to relational types. If $dom(\Gamma)$ is a superset of $FV(e)$ then we say that Γ is a relational type assignment on e. For convenience we will abbreviate "relational type" and "relational type assignment" by "type" respectively "type assignment" in this chapter.

The *typing relation* for the relational algebra is defined in Figure 4.1. Here we write $\Gamma \vdash e : \tau$ to indicate that expression $e \in \mathcal{R}^\Theta$ has type τ under type assignment Γ on e. Note that e has at most one type under Γ, which can easily be derived from Γ by applying the rules in an order determined by the syntax of expression e. If $A \in dom(\tau)$, then we say that A is *present* in e under Γ. We say that A is absent in e under Γ otherwise. If $\Gamma \vdash e : \tau$, then we call (Γ, τ) a *typing* of e.

Some expressions, such as for example $\pi_A(\rho_{A/B}(x))$ and $\sigma_{A=5}(x) \bowtie (x \times \pi_A(y))$, do not have any typing. We will refer to such expressions as *untypable*.

Definition 4.1. Let e be an expression in $\mathcal{R}^\Theta$ and let Γ be a relational type assignment on e. If there exists a τ such that $\Gamma \vdash e : \tau$, then we say that e is *well-typed* under Γ. Expression e is called *typable* if there exists a relational type assignment Γ on e such that e is well-typed under Γ.

Let V be a subset of the relational algebra operators. We denote the set of all typable expressions in $\mathcal{R}_V^\Theta$ by $\mathcal{T}(\mathcal{R}_V^\Theta)$. Deciding membership of $\mathcal{T}(\mathcal{R}_V^\Theta)$ is called the *typability problem for* $\mathcal{R}_V^\Theta$.

4.2 Deciding Typability

Van den Bussche and Waller noted that the typability problem for the relational algebra can be solved in non-deterministic polynomial time [56]. We

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash r : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \cup e_2 : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 - e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \sim \tau_2}{\Gamma \vdash e_1 \bowtie e_2 : \tau_1 \cup \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad dom(\tau_1) \cap dom(\tau_2) = \emptyset}{\Gamma \vdash e_1 \times e_2 : \tau_1 \cup \tau_2}$$

$$\frac{\Gamma \vdash e : \tau \quad A_1, \ldots, A_n \in dom(\tau) \quad (\tau(A_1), \ldots, \tau(A_n)) \in \varsigma(\theta)}{\Gamma \vdash \sigma_{\theta(A_1,\ldots,A_n)}(e) : \tau} \qquad \frac{\Gamma \vdash e : \tau \quad A_1, \ldots, A_n \in dom(\tau)}{\Gamma \vdash \pi_{A_1,\ldots,A_n}(e) : \{A_1, \ldots, A_n\}}$$

$$\frac{\Gamma \vdash e : \tau \quad A \in dom(\tau) \quad B \notin dom(\tau)}{\Gamma \vdash \rho_{A/B}(e) : \rho_{A/B}(\tau)}$$

Figure 4.1: The typing relation for the relational algebra.

reiterate their result here for completeness' sake. If τ is a type and S is a set of attribute names, then we write $\tau|_S$ for the type defined by $\tau|_S(A) := \tau(A)$ for every A in $dom(\tau) \cap S$. If Γ is a type assignment, then we write $\Gamma|_S$ for the type assignment defined by $\Gamma|_S(r) := \Gamma(r)|_S$.

Lemma 4.2 (Van den Bussche and Waller). *If (Γ, τ) is a typing of e and $Specattrs(e) \subseteq S$, then $(\Gamma|_S, \tau|_S)$ is also a typing of e.*

The proof is straightforward. As a consequence, in order to decide whether there exists a type assignment under which e is well-typed, it suffices to consider type assignments Γ with the property that

$$dom(\Gamma(x)) \subseteq Specattrs(e),$$

for every $x \in FV(e)$. It follows immediately that typability is in NP. This upper bound is tight, as the following theorem shows.

Theorem 4.3. *$\mathcal{T}(\mathcal{R}^\Theta)$ is* NP*-complete for any predicate-set Θ.*

Proof. We give a LOGSPACE reduction from POSITIVE ONE-IN-THREE 3SAT, which is known to be NP-hard [24]. The POSITIVE ONE-IN-THREE 3SAT problem consists of deciding for a given 3CNF formula with only positive clauses of the form $(x \vee y \vee z)$, whether there exists a truth assignment that makes exactly one literal per clause true.

Let $\phi = (x_1 \vee y_1 \vee z_1) \wedge \cdots \wedge (x_n \vee y_n \vee z_n)$ be a 3CNF formula where every clause is positive. Let X be the set of all variables occurring in ϕ. We construct the expression e_ϕ with $FV(e_\phi) = X$ such that ϕ is one-in-three satisfiable if, and only if, e_ϕ is typable:

$$e_\phi := \bigcup_{i=1}^{n} \pi_A(x_i \times y_i \times z_i).$$

It is clear that e_ϕ can be constructed from ϕ in logarithmic space.

Suppose ϕ is one-in-three satisfiable. Then there exists a truth assignment w on X such that for every i exactly one of $w(x_i)$, $w(y_i)$, and $w(z_i)$ is true. To show that e_ϕ is typable, we construct the type assignment Γ on e_ϕ as follows. Let τ be a type with domain $\{A\}$. We define $\Gamma(x) := \tau$ if $w(x)$ is true and $\Gamma(x) := \emptyset$ otherwise, for every $x \in X$. Since exactly one of $w(x_i)$, $w(y_i)$, and $w(z_i)$ is true for every i, we have by construction that exactly one of $\Gamma(x_i)$, $\Gamma(y_i)$, and $\Gamma(z_i)$ is τ, the others being $\emptyset$. Since the domains of $\Gamma(x_i)$, $\Gamma(y_i)$, and $\Gamma(z_i)$ are then disjoint and since $\Gamma(x_i) \cup \Gamma(y_i) \cup \Gamma(z_i) = \tau$, the expressions $x_i \times y_i \times z_i$ have type τ under Γ. Then every $\pi_A(x_i \times y_i \times z_i)$ also has type τ under Γ. Therefore every operand of the union operator has type τ, and thus e_ϕ is well-typed under Γ.

Conversely, suppose e_ϕ is typable. Then there exists a type assignment Γ on e_ϕ such that every subexpression of e (including e) is well-typed under Γ. To show that ϕ is satisfiable, we construct the truth assignment w on X such that $w(u)$ is true if, and only if, $A \in dom(\Gamma(u))$. Since every $\pi_A(x_i \times y_i \times z_i)$ is well-typed under Γ, the type of $x_i \times y_i \times z_i$ must be defined on A. Because of the typing rule for $\times$, this means that exactly one of $\Gamma(x_i), \Gamma(y_i)$, or $\Gamma(z_i)$ is defined on A. Hence, exactly one of $w(x_i), w(y_i)$, or $w(z_i)$ is true for every i, and ϕ is one-in-three satisfiable. □

The following natural question now arises: for which operators of the relational algebra can typability be decided in polynomial time? We note that expressions in $\mathcal{R}^{\Theta}_{\cup,-,\bowtie,\times}$ are always well-typed under the type assignment which maps every variable to the empty type. Hence $\mathcal{T}(\mathcal{R}^{\Theta}_{\cup,-,\bowtie,\times})$ is trivially in P. Adding π, ρ, or σ to the set of operators, however, immediately makes the problem NP-complete. This is clear for π from the reduction above. Also, the reduction still works if we define e_ϕ as

$$e_\phi := \overset{n}{\underset{i=1}{\times}} \rho_{A/B_i}(x_i \times y_i \times z_i).$$

Here the B_i are auxiliary attribute names used to make sure that the various operands of $\times$ have a disjoint domain. Finally, if $\theta \in \Theta$, then we can define

e_ϕ as:

$$e_\phi := \bigcup_{i=1}^{n} \sigma_{\theta(A_1,\ldots,A_k)}(x_i \times y_i \times z_i).$$

Indeed, if ϕ is one-in-three satisfiable, then we can show typability of e_ϕ simply by taking τ in the reasoning above to be a type for which

$$(\tau(A_1), \ldots, \tau(A_k)) \in \varsigma(\theta).$$

Conversely, if e_ϕ is typable, we can show one-in-three satisfiability of ϕ by taking w in the reasoning above such that $w(u)$ is true if, and only if, $A_1 \in dom(\Gamma(u))$.

Hence, deciding typability for restrictions of the relational algebra containing $\{-, \times, \pi\}, \{-, \times, \sigma\}$, $\{\cup, \times, \sigma\}$, or $\{\times, \rho\}$ as a subset of operators remains NP-complete for any (non-empty) predicate-set Θ.

4.3 Typability and Constraint Satisfaction

The results of Section 4.2 seem to imply that the cartesian product operator is the main reason why the typability problem for the relational algebra is NP-hard. Consider expressions of the following form however:

$$\sigma_{\theta_1(A_1,\ldots,A_k)} \cdots \sigma_{\theta_n(B_1,\ldots,B_l)}(x).$$

In order to decide typability of such expressions, we need to make sure that there are no base-type clashes between the various uses of an attribute. It is not hard to see that this is another potential source of intractability.

We will formalize this intuition by showing that typability in $\mathcal{R}_\sigma^\Theta$ is a disguised form of the *non-uniform constraint satisfaction problem*, which is known to be NP-complete in general.

A *relational structure* is a tuple $(C, R_1, \ldots, R_n)$ where C is a finite set and $R_1, \ldots, R_n$ are relations over C. Let $\mathbf{A} = (C, R_1, \ldots, R_n)$ and $\mathbf{B} = (D, S_1, \ldots, S_n)$ be two relational structures where the arity of R_i equals the arity of S_i for every $i \in [1, n]$. A *homomorphism* from $\mathbf{A}$ to $\mathbf{B}$ is a function h from C to D such that for every $i \in [1, n]$:

$$(c_1, \ldots, c_{k_i}) \in R_i \Rightarrow (h(c_1), \ldots, h(c_{k_i})) \in S_i.$$

Here, k_i denotes the arity of R_i and S_i.

The *constraint satisfaction problem* consists of deciding, given relational structures $\mathbf{A}$ and $\mathbf{B}$ whether there is a homomorphism from $\mathbf{A}$ to $\mathbf{B}$. This problem is NP-complete in general, since it is clearly in NP and it contains

NP-hard problems as special cases. For example, 3-COLORABILITY is equivalent to the problem of deciding whether there is a homomorphism from a given graph $\mathbf{H}$ to the complete graph with 3 nodes

$$\mathbf{K_3} = (\{r, g, b\}, \{(r, b), (b, r), (r, g), (g, r), (b, g), (g, b)\}).$$

The constraint satisfaction problem for which $\mathbf{B}$ is fixed is called *the non-uniform constraint satisfaction problem* (non-uniform CSP for short). Let us define $\mathcal{H}(\mathbf{B})$ as the set of all structures for which there exists a homomorphism to $\mathbf{B}$. It is well-known that there exist structures $\mathbf{B}$ such that $\mathcal{H}(\mathbf{B})$ is NP-complete ($\mathbf{K_3}$ being an example).

We will now relate the typability problem to non-uniform CSP. Let $\Theta = \{\theta_1, \ldots, \theta_n\}$ be a set of predicates. We define the *structure* of an expression $e \in \mathcal{R}^{\Theta}$, denoted by *Struc*$(e)$, as the relational structure

$$(\mathit{Specattrs}(e), \theta_1(e), \ldots, \theta_n(e)),$$

where

$$\theta_i(e) = \{(A_1, \ldots, A_{|\theta_i|}) \mid \sigma_{\theta_i(A_1,\ldots,A_{|\theta_i|})}(e') \text{ is a subexpression of } e\}.$$

Likewise, we define the *structure of* Θ, denoted by *Struc*(Θ), as the relational structure $(\beta(\Theta), \varsigma(\theta_1), \ldots, \varsigma(\theta_n))$. Here, $\beta(\Theta)$ denotes the set of all base types mentioned in the signatures of predicates in Θ.

Lemma 4.4. *If Θ is a finite set of predicates and $e \in \mathcal{R}^{\Theta}_{\cup,-,\bowtie,\sigma}$, then e is typable if, and only if, there is a homomorphism from Struc(e) to Struc(Θ).*

Proof. Intuitively, we need to make sure that there are no base-type clashes between the various uses of an attribute in e in order to decide typability of e. This is exactly what the existence of a homomorphism from *Struc*(e) to *Struc*(Θ) indicates.

Suppose that there is a homomorphism h from *Struc*(e) to *Struc*(Θ). Let Γ be the type assignment defined by $\Gamma(x) = h$ for every $x \in FV(e)$. In order to show that e is typable, it suffices that to show that every subexpression e' of e has type h under Γ. We do so by induction on e'.

- Clearly, $\Gamma \vdash r : h$.
- If $e' = e_1 \cup e_2$ or $e' = e_1 - e_2$, then $\Gamma \vdash e_1 : h$ and $\Gamma \vdash e_2 : h$ by the induction hypothesis. Hence, all the premises of the type rule for union, respectively difference, are met and $\Gamma \vdash e' : h$ holds.
- If $e' = e_1 \bowtie e_2$, then $\Gamma \vdash e_1 : h$ and $\Gamma \vdash e_2 : h$ by the induction hypothesis. Clearly, $h \sim h$ and $h = h \cup h$. Hence, all the premises of the type rule for join are met and $\Gamma \vdash e' : h$ holds.

- If $e' = \sigma_{\theta_i(A_1,\ldots,A_k)}(e'')$, then we have by the induction hypothesis that $\Gamma \vdash e'' : h$. By definition, $(A_1, \ldots, A_k) \in \theta_i(e)$. Since $\{A_1, \ldots, A_k\} \subseteq \mathit{Specattrs}(e)$ and since h is a homomorphism from $\mathit{Struc}(e)$ to $\mathit{Struc}(\Theta)$ we have $(h(A_1), \ldots, h(A_k)) \in \varsigma(\theta_i)$. Hence, all the premises of the type rule for selection are met and $\Gamma \vdash e' : h$ holds.

Conversely, suppose that there exists a type assignment Γ under which e is well-typed. Let us write τ_e for the type of e under Γ, and let us write $H(e)$ for the set of homomorphisms from $\mathit{Struc}(e)$ to $\mathit{Struc}(\Theta)$. We prove by induction on e that $\tau_e|_{\mathit{Specattrs}(e)} \in H(e)$.

- This is clear if $e = r$.

- If $e = e_1 \cup e_2$, then $\tau_e = \tau_{e_1} = \tau_{e_2}$ by the type rule for union. Then $\tau_e|_{\mathit{Specattrs}(e_1)} \in H(e_1)$ and $\tau_e|_{\mathit{Specattrs}(e_2)} \in H(e_2)$ by the induction hypothesis. Since $\mathit{Struc}(e)$ is the component-wise union of $\mathit{Struc}(e_1)$ and $\mathit{Struc}(e_2)$, it follows that $\tau_e|_{\mathit{Specattrs}(e)} \in H(e)$. If $e = e_1 - e_2$ we make an analogous reasoning.

- If $e = e_1 \bowtie e_2$, then $\tau_{e_1} \sim \tau_{e_2}$ and $\tau_e = \tau_{e_1} \cup \tau_{e_2}$ by the type rule for join. Moreover, $\tau_{e_1}|_{\mathit{Specattrs}(e_1)} \in H(e_1)$ and $\tau_{e_2}|_{\mathit{Specattrs}(e_2)} \in H(e_2)$ by the induction hypothesis. Since $\tau_{e_1}|_{\mathit{Specattrs}(e_1)} \subseteq \tau_e|_{\mathit{Specattrs}(e)}$ and $\tau_{e_2}|_{\mathit{Specattrs}(e_2)} \subseteq \tau_e|_{\mathit{Specattrs}(e)}$, and since $\mathit{Struc}(e)$ is the component-wise union of $\mathit{Struc}(e_1)$ and $\mathit{Struc}(e_2)$, it follows that $\tau_e|_{\mathit{Specattrs}(e)} \in H(e)$.

- If $e = \sigma_{\theta_i(A_1,\ldots,A_k)}(e')$, then $\tau_{e'}|_{\mathit{Specattrs}(e')} \in H(e')$ by the induction hypothesis. Furthermore, $(\tau_{e'}(A_1), \ldots, \tau_{e'}(A_k)) \in \varsigma(\theta_i)$ and $\tau_e = \tau_{e'}$ by the type rule for σ. For every $i \in [1, n]$ we have

$$\theta_j(e) = \begin{cases} \theta_i(e') \cup \{(A_1, \ldots, A_k)\} & \text{if } i = j \\ \theta_j(e') & \text{otherwise.} \end{cases}$$

  Hence, $\tau_e|_{\mathit{Specattrs}(e)} \in H(e)$.

□

Using this lemma, we may conclude that typability is as least as difficult as non-uniform CSP.

Theorem 4.5. *If Θ is a finite, non-empty set of predicates and V is a subset of the relational algebra operators containing σ, then $\mathcal{H}(\mathit{Struc}(\Theta))$ is* LOGSPACE *reducible to $\mathcal{T}(\mathcal{R}_V^\Theta)$.*

Proof. We show that for every relational structure $\mathbf{A}$ we can create an expression $e \in \mathcal{R}^{\Theta}_{\sigma}$ (in logarithmic space) such that there is a homomorphism from $\mathbf{A}$ to $\mathit{Struc}(\Theta)$ if, and only if, there is a homomorphism from $\mathit{Struc}(e)$ to $\mathit{Struc}(\Theta)$. The result then follows by Lemma 4.4.

Let $\Theta = \{\theta_1, \ldots, \theta_n\}$ and let $\mathbf{A} = (C, R_1, \ldots, R_n)$ be a relational structure where the arity of R_i equals $|\theta_i|$. Let $\mathit{adom}(\mathbf{A})$ denote the *active domain* of $\mathbf{A}$, i.e., the set of all elements in C actually mentioned in one of the R_i. It is easy to see that there is a homomorphism from $\mathbf{A}$ to $\mathit{Struc}(\Theta)$ if, and only if, there is a homomorphism from $(\mathit{adom}(\mathbf{A}), R_1, \ldots, R_n)$ to $\mathit{Struc}(\Theta)$.

We now create e such that $\mathit{Struc}(e) = (\mathit{adom}(\mathbf{A}), R_1, \ldots, R_n)$. This is true whenever e is of the form $\sigma \ldots \sigma(r)$ such that for every R_i and every tuple $(c_1, \ldots, c_k)$ in R_i there is a subexpression of the form $\sigma_{\theta_i(c_1,\ldots,c_k)}(e')$ in e. Here we view $c_1, \ldots, c_k$ as attribute names. It is easy to see that we can create such an e in logarithmic space: we simply iterate over the tuples in $\mathbf{A}$, and in each iteration add an extra selection operator of the correct form to the expression built so far. □

Corollary 4.6. *Let V be a subset of the relational algebra operators containing σ. Then $\mathcal{T}(\mathcal{R}^{\Theta}_{V})$ is* NP*-complete if $\mathcal{H}(\mathit{Struc}(\Theta))$ is.*

As we have noted before, there are structures $\mathbf{B}$ for which $\mathcal{H}(\mathbf{B})$ is NP-complete. For every such structure $\mathbf{B} = (C, S_1, \ldots, S_n)$ we can create a set of predicates Θ such that $\mathit{Struc}(\theta) = \mathbf{B}$. Indeed, we simply take Θ to contain predicates $\theta_1, \ldots, \theta_n$ such that $|\theta_i|$ equals the arity of S_i and such that $\varsigma(\theta_i) = S_i$. Hence, there are predicate sets Θ for which $\mathcal{T}(\mathcal{R}^{\Theta}_{V})$ is NP-complete.

On the positive side, the following corollary to Lemma 4.4 tells us that the complexity of $\mathcal{T}(\mathcal{R}^{\Theta}_{\cup,-,\bowtie,\sigma})$ is in P when $\mathcal{H}(\mathit{Struc}(\Theta))$ is in P.

Corollary 4.7. *If Θ is a finite, non-empty set of predicates, then $\mathcal{T}(\mathcal{R}^{\Theta}_{\cup,-,\bowtie,\sigma})$ is* LOGSPACE *reducible to $\mathcal{H}(\mathit{Struc}(\Theta))$.*

This result cannot be generalized further to include π or ρ. To see why, let us fix a set of unary predicates $\Omega = \{\theta_1, \theta_2\}$ where $\varsigma(\theta_1) = \{0\}$ and $\varsigma(\theta_2) = \{1\}$. Here, 0 and 1 are base types. Note that such predicates will occur in practice. For instance, we can interpret θ_1 by "equals 5" with 0 being the base type `int` and θ_2 by "equals Mary" with 1 being the base type `string`.

Theorem 4.8. *With Ω the set of predicates described above, $\mathcal{H}(\mathit{Struc}(\Omega))$ is in* P*, but $\mathcal{T}(\mathcal{R}^{\Omega}_{\bowtie,\sigma,\pi})$ and $\mathcal{T}(\mathcal{R}^{\Omega}_{\bowtie,\sigma,\rho})$ are* NP*-complete.*

Proof. Obviously, $\mathbf{A} = (C, R_1, R_2) \in \mathcal{H}(\mathit{Struc}(\Omega))$ if, and only if, $R_1 \cap R_2 = \emptyset$, which can be checked in polynomial time.

By Lemma 4.2 we only need to show NP-hardness of $\mathcal{T}(\mathcal{R}^{\Omega}_{\cup,-,\bowtie,\sigma,\pi,\rho})$, for which we modify a reduction invented by Ohori and Buneman [46]. The

reduction is from MONOTONE 3SAT [24]: decide whether there is a satisfying truth assignment for a given 3CNF boolean formula ϕ whose clauses are either all variables (called a positive clause) or all negated variables (called a negative clause).

Let $\phi = (a_1^1 \vee a_2^1 \vee a_3^1) \wedge \cdots \wedge (a_1^n \vee a_2^n \vee a_3^n)$ be such a formula. Here, every a_j^i is hence a proposition variable x, or a negated proposition variable $\overline{x}$. We assume without loss of generality that $a_1^1, \ldots, a_3^n$ are members of our fixed set of variables $\mathcal{X}$.

We will create an expression e_ϕ such that e_ϕ is typable if, and only if, ϕ is satisfiable. Intuitively, we encode truth assignments w on the set X of all variables in ϕ by type assignments Γ where $A \in dom(\Gamma(x))$ if, and only, if $w(x)$ is true and $A \in dom(\Gamma(\overline{x}))$ if, and only if, $w(x)$ is false.

Let us first define, for every proposition variable x in ϕ, the expression:

$$e_x := \pi_B \sigma_{\theta_1(A)}(x \bowtie x_1) \bowtie \pi_B \sigma_{\theta_2(A)}(\overline{x} \bowtie x_2) \bowtie \pi_B \pi_{A,B}(x \bowtie \overline{x}).$$

Intuitively, e_x is used to verify that every type assignment under which e_ϕ is well-typed is indeed an encoding of a truth assignment. The whole expression is now defined by:

$$e_\phi := \bowtie_{x \in X} e_x \bowtie \bowtie_{i=1}^{n} \pi_B \pi_{A,B}(a_1^i \bowtie a_2^i \bowtie a_3^i).$$

It is clear that e_ϕ can be constructed from ϕ in logarithmic space.

Suppose that ϕ is satisfiable. Then there exists a satisfying truth assignment w on the variables of ϕ. To show that e_ϕ is typable, we construct the type assignment Γ on e as follows. Let τ be a type which is undefined on all attributes except B. Let τ_1 and τ_2 be the types with domain $\{A\}$ such that $\tau_1(A) = 0$ and $\tau_2(A) = 1$. If $w(x)$ is true, then we define

$$\begin{aligned} \Gamma(x) &:= \tau \cup \tau_1 & \qquad \Gamma(\overline{x}) &:= \emptyset \\ \Gamma(x_1) &:= \emptyset & \qquad \Gamma(x_2) &:= \tau \cup \tau_2. \end{aligned}$$

Otherwise, we define

$$\begin{aligned} \Gamma(x) &:= \emptyset & \qquad \Gamma(\overline{x}) &:= \tau \cup \tau_2 \\ \Gamma(x_1) &:= \tau \cup \tau_1 & \qquad \Gamma(x_2) &:= \emptyset. \end{aligned}$$

The reader is asked to verify that every e_x has output type τ under Γ. Since every clause in ϕ consists entirely of un-negated variables or entirely of negated variables, and since by construction $\Gamma(x) \sim \Gamma(y)$ and $\Gamma(\overline{x}) \sim \Gamma(\overline{y})$ for every variable x and y, the subexpressions $(a_1^i \bowtie a_2^i \bowtie a_3^i)$ are well-typed under Γ. Moreover, since $w(a_1^i \vee a_2^i \vee a_3^i)$ is true, at least one of $\Gamma(a_1^i)$, $\Gamma(a_2^i)$ and $\Gamma(a_3^i)$ is

defined on A and B. Since $\Gamma(a_1^i) \cup \Gamma(a_2^i) \cup \Gamma(a_3^i)$ is the type of $(a_1^i \bowtie a_2^i \bowtie a_3^i)$ under Γ, we know that $\pi_B \pi_{A,B}(a_1^i \bowtie a_2^i \bowtie a_3^i)$ also has type τ under Γ. Then e_ϕ is well-typed under Γ since it is a join of subexpressions of type τ and since τ is certainly compatible with itself.

Conversely, suppose e_ϕ is typable. Then there exists a type assignment Γ on e such that every subexpression of e is well-typed under Γ. In particular, e_x is well-typed under Γ for every variable x. Then Γ encodes a truth assignment. Indeed, the subexpression $\pi_B \pi_{A,B}(x \bowtie \overline{x})$ of e_x requires that $A \in \mathit{dom}(\Gamma(x))$ or $A \in \mathit{dom}(\Gamma(\overline{x}))$. However, if $A \in \mathit{dom}(\Gamma(x))$, then subexpression $\sigma_{\theta_1(A)}(x \bowtie x_1)$ of e_x requires $\Gamma(x)(A) = 0$, while subexpression $\sigma_{\theta_2(A)}(\overline{x} \bowtie x_2)$ requires $\Gamma(\overline{x})(A) = 1$ when $A \in \mathit{dom}(\Gamma(\overline{x}))$. Since subexpression $x \bowtie \overline{x}$ of e_x requires that $\Gamma(x)$ and $\Gamma(\overline{x})$ are compatible, we have $A \in \mathit{dom}(\Gamma(x))$ if, and only if, $A \notin \mathit{dom}(\Gamma(\overline{x}))$. Let w be the truth assignment such that $w(x)$ is true if, and only if, $A \in \mathit{dom}(\Gamma(x))$. Let $i \in [1, n]$. Since $\pi_{A,B}(a_1^i \bowtie a_2^i \bowtie a_3^i)$ is well-typed under Γ, the type of $a_1^i \bowtie a_2^i \bowtie a_3^i$ must be defined on A. Hence, $\Gamma(a_j^i)$ is defined on A for some $j \in [1, 3]$. We discern two cases. Either $a_j^i = x$ for some variable x, meaning that the i-th clause in ϕ is positive. Then $w(a_1^i \vee a_2^i \vee a_3^i)$ is true since $w(a_j^i)$ is true. Otherwise, $a_j^i = \overline{x}$ for some variable x and the i-th clause of ϕ is negative. Then $\Gamma(x)$ cannot be defined on A since $\Gamma(\overline{x})$ is defined on A. Hence, $w(x)$ is false which means $w(\overline{x})$ is true and thus $w(a_1^i \vee a_2^i \vee a_3^i)$ is true. Hence, ϕ is satisfiable.

To show NP-hardness of $\mathcal{T}(\mathcal{R}^{\Omega}_{\bowtie,\sigma,\rho})$ a similar reduction can be made: we define

$$e_x := \rho_{A/C_x} \sigma_{\theta_1(A)}(x \bowtie x_1) \bowtie \rho_{A/D_x} \sigma_{\theta_2(A)}(\overline{x} \bowtie x_2) \bowtie \rho_{A/E_x}(x \bowtie \overline{x})$$
$$e_\phi := \bigbowtie_{x \in X} e_x \bowtie \bigbowtie_{i=1}^{n} \rho_{A/F_i}(a_1^i \bowtie a_2^i \bowtie a_3^i).$$

Here the auxiliary attributes C_x, D_x, E_x, and F_i are used to prevent base-type clashes between the various subexpressions. □

As a consequence, $\mathcal{T}(\mathcal{R}^{\Omega}_{V})$ is NP-complete whenever V includes $\{\bowtie, \sigma, \pi\}$ or $\{\bowtie, \sigma, \rho\}$ as a subset of operators.

The predicate set Ω depends heavily on the presence of more than one base type. What is the complexity of deciding typability when we have only one base type? As we will show, this can be done in polynomial time. This implies that we can at least efficiently check expressions for mistakes that require an attribute to be present and absent at the same time, as for example in $\rho_{A/B}(\pi_B(x))$.

Theorem 4.9. *Let Θ be a finite set of predicates over at most one base type, so $|\beta(\Theta)| = 1$. Then $\mathcal{T}(\mathcal{R}^{\Theta}_{\cup,-,\bowtie,\sigma,\pi,\rho})$ is in* P.

To prove this theorem, we will show that we can always reformulate the typability problem as a non-uniform CSP which is solvable in polynomial time. Since $|\beta(\Theta)| = 1$, there can be no base-type clashes in an expression e, and hence we only need to verify that attributes are used consistently, i.e. that an attribute is not required to be present and absent at the same time.

We will record the requirements the type system makes on the presence or absence of attributes in an relational structure as follows. Let $e' \preceq e$ denote the fact that e' is a subexpression of e. To each expression $e \in \mathcal{R}^{\Theta}_{\cup,-,\bowtie,\sigma,\pi,\rho}$ we then associate the relational structure

$$\mathbf{A_e} = (C_e, D_e, U_e, E_e, J_e)$$

where

- C_e is a set of *proposition variables* of the form $A^{e'}$ where e' is a subexpression of e and A is an attribute occurring in e;
- D_e is the set of proposition variables $A^{e'}$ for which the type system requires that A is present in the output type of subexpression e' under any type assignment Γ which makes e well-typed:

$$\begin{array}{rcl} D_e & = & \{A^{e'} \mid \sigma_{\theta(B_1,\ldots,A,\ldots,B_n)}(e') \preceq e\} \\ & \cup & \{A^{e'} \mid \pi_{B_1,\ldots,A,\ldots,B_n}(e') \preceq e\} \\ & \cup & \{A^{e'}, B^{\rho_{A/B}(e')} \mid \rho_{A/B}(e') \preceq e\}; \end{array}$$

- U_e is the set of proposition variables $A^{e'}$ for which the type system requires that A is absent in the output type of e' under any type assignment Γ which makes e well-typed:

$$\begin{array}{rcl} U_e & = & \{A^{\pi_{B_1,\ldots,B_n}(e')} \mid \pi_{B_1,\ldots,B_n}(e') \preceq e, A \notin \{B_1,\ldots,B_n\}\} \\ & \cup & \{A^{\rho_{A/B}(e')}, B^{e'} \mid \rho_{A/B}(e') \preceq e\}; \end{array}$$

- E_e is the set of pairs of proposition variables $(A^{e'}, B^{e''})$ for which the type system requires that A is present in the output type of e' under a type assignment which makes e well-defined, if, and only if, B is present in the output type of e'' under this assignment:

$$\begin{array}{rcl} E_e & = & \{(A^{e_1}, A^{e_2}), (A^{e_1 \cup e_2}, A^{e_1}) \mid e_1 \cup e_2 \preceq e, A \in \mathit{Specattrs}(e)\} \\ & \cup & \{(A^{e_1}, A^{e_2}), (A^{e_1 - e_2}, A^{e_1}) \mid e_1 - e_2 \preceq e, A \in \mathit{Specattrs}(e)\} \\ & \cup & \{(A^{\sigma_{\theta(A_1,\ldots,A_n)}(e')}, A^{e'}) \mid \sigma_{\theta(A_1,\ldots,A_n)}(e') \preceq e, A \in \mathit{Specattrs}(e)\}; \end{array}$$

- J_e captures the relations between the attributes imposed by the type rule for join:

$$J_e = \{(A^{e_1 \bowtie e_2}, A^{e_1}, A^{e_2}) \mid e_1 \bowtie e_2 \preceq e, A \in \mathit{Specattrs}(e)\}.$$

It is clear that e is typable if, and only if, the requirements made by the type system can be met, i.e., if there is a homomorphism from $\mathbf{A_e}$ to the structure $\mathbf{B} = (\{0,1\}, D, U, E, J)$ where

- D is used to verify that the attributes mentioned in D_e are actually present: $D = \{1\}$;
- U is used to verify that the attributes mentioned in U_e are actually absent: $U = \{0\}$;
- E is used to verify that for the pairs of proposition variables $(A^{e'}, B^{e''})$ mentioned in E_e, A is present in the output type of e' if, and only if, B is present in the output type of e': $E = \{(0,0),(1,1)\}$; and
- J is used to verify that for the triples $(A^{e_1 \bowtie e_2}, A^{e_1}, A^{e_2})$ in J_e, the presence of A in the output type of $e_1 \bowtie e_2$ follows the type rule for join: $J = \{(1,1,1),(1,1,0),(1,0,1),(0,0,0)\}$.

Such a relational structure, where the domain contains only two elements, is called a *boolean structure*.

Lemma 4.10. *Let Θ be a finite set of predicates. If $|\beta(\Theta)| = 1$ and $e \in \mathcal{R}^{\Theta}_{\cup,-,\bowtie,\sigma,\pi,\rho}$, then e is typable if, and only if, there is a homomorphism from $\mathbf{A_e}$ to $\mathbf{B}$.*

Proof. Suppose that e is well-typed under type assignment Γ. Then every subexpression e' of e is well-typed under Γ. Let $\tau_{e'}$ be the type of e' under Γ. Define the function h from C_e to $\{0,1\}$ such that $h(A^{e'}) = 1$ if, and only if, $A \in dom(\tau_{e'})$. It is easy to show that h is a homomorphism from $\mathbf{A_e}$ to $\mathbf{B}$. For example, if $(A^{e_1 \cup e_2}, A^{e_1}) \in E_e$, then $e_1 \cup e_2 \preceq e$ by construction. By the type rule for union, we know that $\tau_{e_1} = \tau_{e_1 \cup e_2}$. Hence $h(A^{e_1}) = h(A^{e_1 \cup e_2})$ and thus $(h(A^{e_1 \cup e_2}), h(A^{e_1})) \in E$. Similar reasonings can be made for the other cases.

Conversely, let h be a homomorphism from $\mathbf{A_e}$ to $\mathbf{B}$. Let b be the single base type in $\beta(\Theta)$. For every subexpression e' of e we define $\tau_{e'}$ such that $\tau_{e'}(A) = b$ if $h(A^{e'}) = 1$, and $\tau_{e'}$ is undefined on A otherwise. Let Γ be the type assignment such that $\Gamma(r) = \tau_r$ for every $r \in FV(e)$. It is easy to show by induction on e' that $\Gamma \vdash e' : \tau_{e'}$. For example, if $e' = \sigma_{\theta(A_1,\ldots,A_n)}(e'')$, then $\Gamma \vdash e'' : \tau_{e''}$ by the induction hypothesis. By construction, $A_i^{e''} \in D_e$ for $1 \leq i \leq n$. Hence, $h(A_i^{e''}) \in D = \{1\}$ and thus $A \in dom(\tau_{e''})$. Moreover, $(\tau_{e''}(A_1), \ldots, \tau_{e''}(A_n)) = (b, \ldots, b)$. Since $|\beta(\Theta)| = 1$, $\varsigma(\theta) = \{(b, \ldots, b)\}$, and hence $(\tau_{e''}(A_1), \ldots, \tau_{e''}(A_n)) \in \varsigma(\theta)$. By the type rule for selection, $\Gamma \vdash e' : \tau_{e''}$. Since $(A^{e'}, A^{e''}) \in E_e$, we know that $(h(A^{e'}), h(A^{e''})) \in E$. Hence, $h(A^{e'}) =$

$h(A^{e''})$, $\tau_{e'} = \tau_{e''}$, and $\Gamma \vdash e' : \tau_{e'}$. Similar reasonings can be made for the other cases. □

We recall the following important theorem from constraint satisfaction theory, due to Shaefer [51]:

Theorem 4.11 (Shaefer's Dichotomy Theorem).

- *If* $\mathbf{B}$ *is a Boolean structure, then* $\mathcal{H}(\mathbf{B})$ *is in* P *or it is* NP*-complete.*
- *In particular, if every relation in a Boolean structure* $\mathbf{B}$ *is closed under the function* $g(x, y) = x \vee y$*, then* $\mathcal{H}(\mathbf{B})$ *is in* P*.*

An n-ary relation R is *closed* under the function $g(x, y)$ if for any two tuples $(a_1, \ldots, a_n)$ and $(b_1, \ldots, b_n)$ in R the tuple $(g(a_1, b_1), \ldots, g(a_n, b_n))$ is also in R. It is easy to see that every relation in $\mathbf{B}$ is closed under g. Theorem 4.9 then follows from this theorem and Lemma 4.10.

As a corollary to Theorem 4.9, $\mathcal{T}(\mathcal{R}^{\Theta'}_{\cup,-,\bowtie,\pi,\rho})$ is in P for any predicate set Θ', since $\mathcal{R}^{\Theta'}_{\cup,-,\bowtie,\pi,\rho} \subseteq \mathcal{R}^{\Theta}_{\cup,-,\bowtie,\sigma,\pi,\rho}$ for all predicate sets Θ (and hence in particular for those with $|\beta(\Theta)| = 1$).

5

Polymorphic Type Inference for the Named Nested Relational Calculus

In this chapter we study the type inference and typability problems for the named version of the nested relational calculus (NNRC for short).

The NNRC comes equipped with a natural static type system to detect programming errors. In this type system, the basic operators of the NNRC are *polymorphic*. For example, we can inspect the A attribute of any record, as long as it has an attribute A. We can take the cartesian product of any two records whose attribute sets are disjoint. We can take the union of any two sets of the same type. Similar typing conditions can be formulated for the other operators of the NNRC. When combining operators into expressions, these typing conditions become more evolved. For example, for the expression

$$\{(x \times y).A \mid x \in R\}$$

to be well-typed, R must have a set type containing the type of x; x and y must have record types whose attribute sets are disjoint; and one of these attribute sets must contain A.

A natural question thus arises: given an NNRC expression e, under which assignments of free variables in e to types is e well-typed? And what is the resulting output type of e under these assignments? In particular, can we give an explicit description of the typically infinite collection of these *typings*?

This is nothing but the NNRC version of the classical *type inference* problem. Type inference is an extensively studied topic in the theory of programming languages [42, 49], and is used in industrial-strength functional programming languages such as Standard ML [55] and Haskell [30].

In this chapter, we propose an explicit description of the set of all possible typings of an NNRC expression e by means of a conjunctive logical formula ϕ_e, which is interpreted in the universe of all possible types. The formula ϕ_e is efficiently computable from e. We proceed to show that the satisfiability problem of such conjunctive formulas belongs to NP. Consequently, typability for the NNRC is also in NP. Since the NNRC is an extension of the relational algebra, for which typability is already NP-complete, this thus shows that typability for the NNRC is not more difficult than for the special case of the relational algebra.

Organization This chapter is further organized as follows. We introduce the named nested relational calculus and its static type system in Section 5.1. In Section 5.2 we show that the set of all typings of an expression can be described by a logical formula. Finally, we show in Section 5.3 that satisfiability of such formulas is in NP.

5.1 Named Nested Relational Calculus

Data Model As in Chapter 4, we assume given a sufficiently large set $\{A, B, \dots\}$ of *attribute names*. A *row over* a set S is a function r from a finite set $dom(r)$ of attribute names to S. We write $\hat{\pi}_A(r)$ for the restriction of r to $dom(r) \setminus \{A\}$. We use an intuitive notation for rows, which we illustrate with an example. If r is the row with domain $\{A, B, C\}$ and $r(A) = a$, $r(B) = b$, and $r(C) = c$, then we write r as $\{A\colon a, B\colon b, C\colon c\}$.

As in Chapters 2 and 3, we also assume given a recursively enumerable set $\mathcal{A} = \{a, b, \dots\}$ of atoms. A *named complex object value* v is either an atom, a record $[r]$ with r a row over named complex object values, or a finite set of named complex object values. The natural join $[r] \bowtie [s]$ of two records is defined as follows:

$$[r] \bowtie [s] := \begin{cases} \{[r \cup s]\} & \text{if } r(A) = s(A) \text{ for all } A \in dom(r) \cap dom(s) \\ \emptyset & \text{otherwise.} \end{cases}$$

Note that, since r and s agree on their common attributes, $r \cup s$ is again a row and $[r \cup s]$ is thus indeed a record.

We will abbreviate "named complex object value" by "value" in the rest of this chapter. Furthermore, we will denote named complex object values by

v and w, rows over named complex object values by r and s, and finite sets of named complex object values by V and W.

Syntax The *named nested relational calculus* (NNRC for short) is the set of all expressions generated by the following grammar:

$$\begin{array}{rcl} e & ::= & x \\ & | & [\,] \mid [A\colon e] \mid e.A \mid e \times e \mid e \bowtie e \mid \hat{\pi}_A(e) \\ & | & \emptyset \mid \{e\} \mid e \cup e \mid \bigcup e \mid \{e \mid x \in e\} \\ & | & e = e \;?\; e \;:\; e \end{array}$$

Here, e ranges over NNRC expressions, x ranges over variables, and A ranges over attribute names. 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.

Semantics A *named complex object context* σ is a function from a finite set of variables $dom(\sigma)$ to named complex object values. If $dom(\sigma)$ is a superset of $FV(e)$, then we say that σ is a *named complex object context on* e. Using a similar notation as in the previous chapters, we denote by $x\colon v, \sigma$ the context σ' with domain $dom(\sigma) \cup \{x\}$ such that $\sigma'(x) = v$ and $\sigma'(y) = \sigma(y)$ for $y \neq x$.

The semantics of NNRC expressions is described by means of the *evaluation relation*, as defined in Figure 5.1. Here, we write $\sigma \models e \Rightarrow v$ to denote that e evaluates to value v under named complex object context σ on e. In the rule for $e_1 \times e_2$, note that $dom(r_1)$ and $dom(r_2)$ are required to be disjoint. This implies that $r_1 \cup r_2$ is again a function and that $[r_1 \cup r_2]$ is a record. Further note that, in contrast to the NRC, conditional tests can compare arbitrary values. Equivalently, we could have restricted conditional tests to only compare atoms, and added an emptiness test.

Intuitively, the NNRC is the extension of the NRC to named records. It should hence come as no surprise that the evaluation relation for the NNRC, like the evaluation relation for the NRC, is functional, not total, and only depends on the free variables of an expression. Indeed, it is easy to see that the evaluation relation for the NNRC is functional. Furthermore, we can only inspect the attributes of records, concatenate disjoint records, join records, project out attributes of records, take the union of sets, flatten a set of sets, and iterate over sets. Finally, it is also easy to see that the semantics of an expression only depends on its free variables. 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*.

Example 5.1. Let *friends* and *John* be two variables. Suppose that the value of *friends* is a set of pairs of friends, as a set of records of the form $[\{Name\colon a, Friend\colon b\}]$ where *Name* and *Friend* are attributes. Suppose also that the value of *John* is a name (an atom). The following expression computes the set of all of *John*'s friends:

$$\bigcup\{x.Name = John\ ?\ \{x.Friend\}\ :\ \emptyset \mid x \in friends\}.$$

□

Note 5.2. Although we have not included the analog of the relational algebra renaming operation $\rho_{A/B}$, which renames the attribute A of a record to the attribute B, such an operation is expressible in the NNRC. Indeed, $\rho_{A/B}(x)$ can be expressed as $\hat{\pi}_A(x) \times [B\colon x.A]$. □

Type System In order to ensure that an expression evaluates to a value for every input context in a desired set of contexts, the NNRC comes equipped with a *static type system*. We will focus on a traditional *homogeneous* type system, where values cannot contain heterogeneous sets. A *named complex object type* is a term generated by the following pseudo-grammar:

$$\tau ::= \mathbf{Atom} \mid \mathbf{Record}(\rho) \mid \mathbf{SetOf}(\tau)$$

Here, x ranges over variables and ρ ranges over rows of named complex object types. A named complex object type τ *denotes* a set of values $[\![\tau]\!]$:

- $[\![\mathbf{Atom}]\!] := \mathcal{A}$,
- $[\![\mathbf{Record}(\rho)]\!]$ is the set of all records $[r]$ with $dom(r) = dom(\rho)$ and $r(A) \in [\![\rho(A)]\!]$, for every $A \in dom(r)$; and,
- $[\![\mathbf{SetOf}(\tau)]\!]$ is the set of all finite sets over $[\![\tau]\!]$.

We will abuse notation and do not distinguish between τ and $[\![\tau]\!]$. A *named complex object type assignment* Γ is a function from a finite set $dom(\Gamma)$ of variables to named complex object types. Using a similar notation as in the previous chapters, we denote by $x\colon \tau, \Gamma$ the named complex object type assignment Γ' with domain $dom(\Gamma) \cup \{x\}$ such that $\Gamma'(x) = \tau$ and $\Gamma'(y) = \Gamma(y)$ for $y \neq x$. A named complex object type assignment Γ *denotes* the set of named complex object contexts σ such that $dom(\sigma) = dom(\Gamma)$ and $\sigma(x) \in \Gamma(x)$, for all $x \in dom(\Gamma)$. We will abuse notation and do not distinguish between a type

Variables

$$\frac{}{\sigma \models x \Rightarrow \sigma(x)}$$

Record operations

$$\frac{}{\sigma \models [\,] \Rightarrow [\emptyset]} \qquad \frac{\sigma \models e \Rightarrow v}{\sigma \models [A\colon e] \Rightarrow [\{A\colon v\}]} \qquad \frac{\sigma \models e \Rightarrow [r] \quad A \in dom(r)}{\sigma \models e.A \Rightarrow r(A)}$$

$$\frac{\sigma \models e_1 \Rightarrow [r_1] \quad \sigma \models e_2 \Rightarrow [r_2] \quad dom(r_1) \cap dom(r_2) = \emptyset}{\sigma \models e_1 \times e_2 \Rightarrow [r_1 \cup r_2]} \qquad \frac{\sigma \models e_1 \Rightarrow [r_1] \quad \sigma \models e_2 \Rightarrow [r_2]}{\sigma \models e_1 \bowtie e_2 \Rightarrow [r_1] \bowtie [r_2]}$$

$$\frac{\sigma \models e \Rightarrow [r] \quad A \in dom(r)}{\sigma \models \hat{\pi}_A(e) \Rightarrow [\hat{\pi}_A(r)]}$$

Set operations

$$\frac{}{\sigma \models \emptyset \Rightarrow \emptyset} \qquad \frac{\sigma \models e \Rightarrow v}{\sigma \models \{e\} \Rightarrow \{v\}} \qquad \frac{\sigma \models e_1 \Rightarrow V_1 \quad \sigma \models e_2 \Rightarrow V_2}{\sigma \models e_1 \cup e_2 \Rightarrow V_1 \cup V_2}$$

$$\frac{\sigma \models e \Rightarrow \{V_1, \ldots, V_n\}}{\sigma \models \bigcup e \Rightarrow \bigcup \{V_1, \ldots, V_n\}} \qquad \frac{\sigma \models e_1 \Rightarrow V \quad \forall v \in V : (x\colon v, \sigma) \models e_2 \Rightarrow w_v}{\sigma \models \{e_2 \mid x \in e_1\} \Rightarrow \{w_v \mid v \in V\}}$$

Conditional test

$$\frac{\sigma \models e_1 \Rightarrow v_1 \quad \sigma \models e_2 \Rightarrow v_2 \quad \sigma \models e_3 \Rightarrow v \quad v_1 = v_2}{\sigma \models e_1 = e_2 \ ?\ e_3 \ :\ e_4 \Rightarrow v} \qquad \frac{\sigma \models e_1 \Rightarrow v_1 \quad \sigma \models e_2 \Rightarrow v_2 \quad \sigma \models e_4 \Rightarrow v \quad v_1 \neq v_2}{\sigma \models e_1 = e_2 \ ?\ e_3 \ :\ e_4 \Rightarrow v}$$

Figure 5.1: The evaluation relation for NNRC expressions.

assignment and its denotation. Finally, if $dom(\Gamma)$ is a superset of $FV(e)$, then we say that Γ is a named complex object type assignment on e. For convenience, we will abbreviate "named complex object type" and "named complex object type assignment" by "type" respectively "type assignment" in the rest of this chapter.

The *typing relation* for the NNRC is defined in Figure 5.2. Here we write $\Gamma \vdash e : \tau$ to indicate that expression e has type τ under type assignment Γ on e. Note that e has at most one type under Γ, which can easily be derived from Γ by applying the rules in an order determined by the syntax of expression e. If $\Gamma \vdash e : \tau$, then we call (Γ, τ) a *typing* of e.

We note that the type system is sound:

Proposition 5.3 (Soundness). *Let e be an expression, let Γ be a type assignment on e, and let let τ be a type. If $\Gamma \vdash e : \tau$, then $e(\sigma)$ is defined and $e(\sigma) \in \tau$, for every $\sigma \in \Gamma$.*

The proof is by an easy induction on e. Due to the undecidability of well-definedness and semantic type-checking for the NNRC (cf. the undecidability of those problems for the NRC as shown in Chapter 2), the NNRC type system cannot be "complete" however. Indeed, there are examples of e, Γ, and τ such that $e(\sigma) \in \tau$ for every $\sigma \in \Gamma$, but yet $\Gamma \nvdash e : \tau$. A simple example is the expression $e_0 = \emptyset \; ? \; [\,] \; : \; \emptyset$ where e_0 is an expression of set type that is actually unsatisfiable. In Chapter 2 we have studied fragments of the NRC where sound and complete type systems do exist. In the current chapter, however, we will continue with the full language and the present type system which, though necessarily incomplete, is still very natural.

5.2 Type Inference

In this section we show that we can describe the set of all typings of an NNRC expression e by a logical formula. We assume the reader to be familiar with many-sorted first-order logic [20].

Let $\mathcal{L}$ be the many-sorted first-order language over the sorts $\{\mathit{type}, \mathit{row}\}$ in the signature consisting of

- a binary relation symbol $=$ of sort $(\mathit{type}, \mathit{type})$;
- a binary relation symbol $\subseteq$ of sort $(\mathit{row}, \mathit{row})$;
- a binary relation symbol $\#$ of sort $(\mathit{row}, \mathit{row})$;
- a unary function symbol Set of sort $\mathit{type} \to \mathit{type}$;

Variables

$$\frac{}{\Gamma \vdash x : \Gamma(x)}$$

Record operations

$$\frac{}{\Gamma \vdash [\,] : \mathbf{Record}(\emptyset)} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash [A\colon e] : \mathbf{Record}(\{A\colon \tau\})}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{Record}(\rho_1) \quad \Gamma \vdash e_2 : \mathbf{Record}(\rho_2) \quad dom(\rho_1) \cap dom(\rho_2) = \emptyset}{\Gamma \vdash e_1 \times e_2 : \mathbf{Record}(\rho_1 \cup \rho_2)}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{Record}(\rho_1) \quad \Gamma \vdash e_2 : \mathbf{Record}(\rho_2) \quad \rho_1(A) = \rho_2(A) \text{ for all } A \in dom(\rho_1) \cap dom(\rho_2)}{\Gamma \vdash e_1 \bowtie e_2 : \mathbf{SetOf}(\mathbf{Record}(\rho_1 \cup \rho_2))}$$

$$\frac{\Gamma \vdash e : \mathbf{Record}(\rho) \quad A \in dom(\rho)}{\Gamma \vdash e.A : \rho(A)} \qquad \frac{\Gamma \vdash e : \mathbf{Record}(\rho) \quad A \in dom(\rho)}{\Gamma \vdash \hat{\pi}_A(e) : \mathbf{Record}(\hat{\pi}_A(\rho))}$$

Set operations

$$\frac{\tau \text{ a type}}{\Gamma \vdash \emptyset : \mathbf{SetOf}(\tau)} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \{e\} : \mathbf{SetOf}(\tau)}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{SetOf}(\tau) \quad \Gamma \vdash e_2 : \mathbf{SetOf}(\tau)}{\Gamma \vdash e_1 \cup e_2 : \mathbf{SetOf}(\tau)} \qquad \frac{\Gamma \vdash e : \mathbf{SetOf}(\mathbf{SetOf}(\tau))}{\Gamma \vdash \bigcup e : \mathbf{SetOf}(\tau)}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{SetOf}(\tau_1) \quad x\colon \tau_1, \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \{e_2 \mid x \in e_1\} : \mathbf{SetOf}(\tau_2)}$$

Conditional test

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau' \quad \Gamma \vdash e_4 : \tau'}{\Gamma \vdash e_1 = e_2 \ ?\ e_3 \ :\ e_4 : \tau'}$$

Figure 5.2: The typing relation for NNRC expressions.

- a unary function symbol *Record* of sort $row \to type$;
- for every attribute A, a unary function symbol $A\colon$ of sort $type \to row$;
- a binary function symbol , of sort $(row, row) \to row$.

We will interpret $\mathcal{L}$ in the many-sorted relational structure $\mathbf{T}$ where

- the universe of *type* is the set of all types;
- the universe of *row* is the set of all rows over types;
- $=$ relates equal types;
- $\subseteq$ relates ρ to ρ' if ρ (as a function, i.e., a set of pairs) is a subset of ρ';
- $\#$ relates ρ to ρ' if $dom(\rho)$ is disjoint with $dom(\rho')$;
- *Set* maps τ to $\mathbf{SetOf}(\tau)$;
- *Record* maps ρ to $\mathbf{Record}(\rho)$;
- $A\colon$ maps τ to the singleton row $\{A\colon \tau\}$; and
- , is the "assymetric" concatenation operation: it maps ρ and ρ' to the row that equals ρ on $dom(\rho)$ and ρ' on $dom(\rho') \setminus dom(\rho)$.

It will be convenient to use the same set $\mathcal{X}$ from the syntax of the NNRC as the set of variables of sort *type* in $\mathcal{L}$. Variables of sort *row* in $\mathcal{L}$ will be denoted using letters from the beginning of the Greek alphabet.

Definition 5.4. A *type formula* is a first-order formula in $\mathcal{L}$ built up from atomic formulas using only existential quantifiers and conjunction.

Example 5.5. The following is an example of a type formula.

$$\phi(x, y) \equiv (\exists\alpha)(\exists\beta)x = Record(\alpha) \wedge y = Record(\beta) \\ \wedge \alpha \# \beta \wedge (\exists z) : (A\colon z) \subseteq \alpha, \beta.$$

Evaluated on the structure $\mathbf{T}$, ϕ defines the set of all pairs of record types $(x = \mathbf{Record}(\rho_1), y = \mathbf{Record}(\rho_2))$ such that $dom(\rho_1) \cap dom(\rho_2) = \emptyset$ and $A \in dom(\rho_1) \cup dom(\rho_2)$. □

Definition 5.6. A type formula ϕ is *principal* for an NNRC expression e if

- ϕ contains no free variables of sort *row*;

- the free variables of sort *type* in ϕ are the free variables of e, plus one additional variable z; and
- $\Gamma \vdash e : \tau$ if, and only if, $\mathbf{T} \models \phi(z\colon \tau, \Gamma)$.

Example 5.7. For a simple example, consider the expression $e_1 = x \cup y$. Then the following is a principal type formula for e_1:

$$(\exists u)x = \mathit{Set}(u) \wedge y = \mathit{Set}(u) \wedge z = \mathit{Set}(u).$$

For a more complicated example, consider the expression:

$$e_2 = \{\{[B\colon t.A]\} \cup \{r \times s\} \mid t \in x \bowtie y\}.$$

Then the following is a principal type formula for e_2:

$$\begin{aligned}(\exists\alpha)(\exists\beta)(\exists\mu)(\exists\nu)x = \mathit{Record}(\alpha) \wedge y = \mathit{Record}(\beta) \wedge (\exists\beta')\alpha \subseteq \beta, \beta' \\ \wedge (\exists\alpha')\beta \subseteq \alpha, \alpha' \wedge t = \mathit{Record}(\alpha, \beta) \wedge r = \mathit{Record}(\mu) \wedge s = \mathit{Record}(\nu) \\ \wedge \mu \# \nu \wedge (\exists q)(A\colon q) \subseteq \alpha, \beta \wedge (B : q) \subseteq \mu, \nu \wedge \mu, \nu \subseteq (B : q) \\ \wedge z = \mathit{Set}(\mathit{Set}(\mathit{Record}(B : q)))\end{aligned}$$

□

Theorem 5.8. *Every NNRC expression e has a principal type formula ϕ_e, of size linear in the size of e, and computable from e in polynomial time.*

Proof. Let e be an expression and let $x_1, \ldots, x_n$ be the free variables of e. Let z be a variable different from $x_1, \ldots, x_n$. We construct the type formula $\phi_e(z, x_1, \ldots, x_n)$ on e by induction on e.

- If $e = x$, then $\phi_e := (z = x)$.
- If $e = [\,]$, then e does not have any free variables. Define

$$\phi_e := (\exists\alpha)z = \mathit{Record}(\alpha) \wedge \alpha \# \alpha.$$

- If $e = [A\colon e']$, then $x_1, \ldots, x_n$ are also the free variables of e'. Define

$$\phi_e := (\exists x_0)\phi_{e'}(x_0, x_1 \ldots, x_n) \wedge z = \mathit{Record}(A\colon x_0).$$

- If $e = e'.A$, then $x_1, \ldots, x_n$ are also the free variables of e'. Define

$$\phi_e := (\exists\alpha)\phi_{e'}(\mathit{Record}(\alpha), x_1 \ldots, x_n) \wedge (A\colon z) \subseteq \alpha.$$

- If $e = e_1 \times e_2$, then let $\{y_1, \ldots, y_k\}$ be the subset of $\{x_1, \ldots, x_n\}$ which are free in e_1 and let $\{y'_1, \ldots, y'_l\}$ be the subset of $\{x_1, \ldots, x_n\}$ which are free in e_2. Define

$$\begin{aligned}\phi_e := (\exists\alpha)\phi_{e_1}(\mathit{Record}(\alpha), y_1 \ldots, y_k) \\ \wedge (\exists\alpha')\ \phi_{e_2}(\mathit{Record}(\alpha'), y'_1, \ldots, y'_l) \\ \wedge \alpha \# \alpha' \wedge z = \mathit{Record}(\alpha, \alpha').\end{aligned}$$

- If $e = e_1 \bowtie e_2$, then let $\{y_1, \ldots, y_k\}$ be the subset of $\{x_1, \ldots, x_n\}$ which are free in e_1 and let $\{y'_1, \ldots, y'_l\}$ be the subset of $\{x_1, \ldots, x_n\}$ which are free in e_2. Define

$$\begin{aligned}\phi_e := (\exists\alpha)\phi_{e_1}(\mathit{Record}(\alpha), y_1 \ldots, y_k) \\ \wedge (\exists\alpha')\ \phi_{e_2}(\mathit{Record}(\alpha'), y'_1, \ldots, y'_l) \\ \wedge (\exists\beta')\alpha \subseteq \alpha', \beta' \wedge (\exists\beta)\alpha' \subseteq \alpha, \beta \\ \wedge z = \mathit{Set}(\mathit{Record}(\alpha, \alpha')).\end{aligned}$$

- If $e = \hat{\pi}_A(e')$, then $x_1, \ldots, x_n$ are also the free variables of e'. Define

$$\begin{aligned}\phi_e := (\exists\alpha)\phi_{e'}(\mathit{Record}(\alpha), x_1 \ldots, x_n) \wedge (\exists\beta)(\exists y)(A\colon y) \# \beta \\ \wedge \alpha \subseteq (A\colon y), \beta \wedge (A\colon y), \beta \subseteq \alpha \wedge z = \mathit{Record}(\beta).\end{aligned}$$

- If $e = \emptyset$, then $\phi_e := (\exists y) z = \mathit{Set}(y)$.

- If $e = \{e'\}$ then $x_1, \ldots, x_n$ are also the free variables of e'. Define

$$\phi_e := (\exists x_0)\phi_{e'}(x_0, x_1, \ldots, x_n) \wedge z = \mathit{Set}(x_0).$$

- If $e = e_1 \cup e_2$, then let $\{y_1, \ldots, y_k\}$ be the subset of $\{x_1, \ldots, x_n\}$ which are free in e_1 and let $\{y'_1, \ldots, y'_l\}$ be the subset of $\{x_1, \ldots, x_n\}$ which are free in e_2. Define

$$\phi_e := \phi_{e_1}(z, y_1 \ldots, y_k) \wedge \phi_{e_2}(z, y'_1, \ldots, y'_l) \wedge (\exists y) z = \mathit{Set}(y).$$

- If $e = \bigcup e'$, then $x_1, \ldots, x_n$ are also the free variables of e'. Define

$$\phi_e := \phi_{e'}(\mathit{Set}(z), x_1, \ldots, x_n) \wedge (\exists y) z = \mathit{Set}(y).$$

- If $e = \{e_2 \mid x \in e_1\}$, then let $\{y_1, \ldots, y_k\}$ be the subset of $\{x_1, \ldots, x_n\}$ which are free in e_1 and let $\{x, y'_1, \ldots, y'_l\}$ be the subset of $\{x_1, \ldots, x_n\}$ which are free in e_2. Define

$$\begin{aligned}\phi_e := (\exists x)\phi_{e_1}(\mathit{Set}(x), y_1, \ldots, y_k) \\ \wedge (\exists y'_0)\phi_{e_2}(y'_0, x, y'_1, \ldots, y'_l) \wedge z = \mathit{Set}(y'_0).\end{aligned}$$

- If $e = e_1 = e_2 \,?\, e_3 \,:\, e_4$, then let $u_1, \ldots, u_k$ be the free variables of e_1, let $u'_1, \ldots, u'_l$ be the free variables of e_2, let $y_1, \ldots, y_p$ be the free variables of e_3, and let $y'_1, \ldots, y'_q$ be the free variables of e_4. Define ϕ_e as

$$(\exists z')\phi_{e_1}(z', u_1, \ldots, u_k) \wedge \phi_{e_2}(z', u'_1, \ldots, u'_l) \\ \wedge \phi_{e_3}(z, y_1, \ldots, y_p) \wedge \phi_{e_4}(z, y'_1, \ldots, y'_q).$$

Clearly, ϕ_e is computable from e in polynomial time. It is easy to see that ϕ_e is indeed a principal type formula for e and that ϕ_e is indeed of size linear in the size of e. □

5.3 Typability

Some expressions, such as for example $[].A$, $x \cup x.A$, $[A\colon x].B$, and $x.A \bowtie (x \times [A\colon y])$ do not have any typing. We will refer to such expressions as *untypable*.

Definition 5.9. An NNRC expression e is called *typable* if there exists a type assignment Γ on e and a type τ such that $\Gamma \vdash e : \tau$. Deciding whether a given expression e is typable is called the *typability problem for the NNRC*.

It follows from Theorem 5.8 that deciding whether an expression e is typable is equivalent to computing the principal type formula ϕ_e for e and then deciding whether ϕ_e is satisfiable in $\mathbf{T}$. We will now show that deciding the latter is in the complexity class NP. Since ϕ_e is computable from e in polynomial time, it then follows that the typability problem is also in NP.

We first note that, since ϕ_e is an conjunctive formula, it is very easily put in existential prenex normal form $(\exists x_1) \ldots (\exists x_n)\psi$ with ψ quantifier free. Clearly, ϕ_e is satisfiable in $\mathbf{T}$ if, and only if, ψ is. We will therefore restrict our attention to quantifier free type formulas.

Definition 5.10. The set $\mathit{Specattrs}(\phi)$ of a type formula ϕ is the set of attributes A for which a term of the form $A\colon t$ occurs in ϕ.

Definition 5.11. The *restriction* $\rho|_S$ of a row ρ to a set of attributes S is the row ρ' with domain $dom(\rho) \cap S$ such that for each $A \in dom(\rho) \cap S$, $\rho'(A)$ is the restriction of the type $\rho(A)$ to S. Here, the restriction $\tau|_S$ of a type τ to S is the type obtained from τ by restricting every row occurring in τ to S. So, this is a recursive definition. In addition, we define the restriction $h|_S$ of a valuation h to S as the valuation h' such that $h'(x) = h(x)|_S$ for every $x \in dom(h)$.

Lemma 5.12. *If ϕ is a type formula and h is a valuation such that $\mathbf{T} \models \phi(h)$, then also $\mathbf{T} \models \phi(h|_{Specattrs(\phi)})$.*

Proof. It is easy to see by induction on t that, for any term t in $\mathcal{L}$ we have $h|_{Specattrs(\phi)}(t) = h(t)|_{Specattrs(\phi)}$. The lemma then follows by an easy induction on ϕ. □

Theorem 5.13. *Deciding satisfiability in* $\mathbf{T}$ *of a quantifier free type formula is in* NP*.*

Proof. Let ψ be a quantifier free type formula. Let, for every attribute name A and every variable α of sort *row* in ψ, x_A^α be a distinct type variable not in ψ. An *attribute assignment* on ψ is a function f which maps each variable α of sort *row* in ψ to a term in $\mathcal{L}$ of sort *row* of the form

$$A\colon x_A^\alpha, \ldots, B\colon x_B^\alpha$$

where $\{A, \ldots, B\} \subseteq Specattrs(\psi)$. Note that, in particular, the size of f is polynomial in the size of ψ. Let ψ_f be the quantifier-free type formula obtained from ψ by replacing each variable α of sort *row* in ψ by the term $f(\alpha)$. Clearly, ψ_f can be computed from e in polynomial time.

We now claim that ψ is satisfiable in $\mathbf{T}$ if, and only if, there exists an attribute assignment f on ψ such that ψ_f is satisfiable in $\mathbf{T}$. Indeed, suppose that ψ is satisfiable in $\mathbf{T}$. By Lemma 5.12 there exists a valuation h of ψ such that $\mathbf{T} \models \psi(h)$ and such that $dom(h(\alpha)) \subseteq Specattrs(\psi)$, for all variables α of sort *row* in ψ. Then let f be the attribute assignment on ψ defined by

$$f(\alpha) := A\colon x_A^\alpha, \ldots, B\colon x_B^\alpha$$

where $dom(h(\alpha)) = \{A, \ldots, B\}$. Let h_f be the valuation on ψ_f which equals h on type variables in ψ and for which $h_f(x_A^\alpha) = h(\alpha)(A)$. It is easy to see that $\mathbf{T} \models \psi_f(h_f)$.

Conversely, suppose that there exists an attribute assignment f on ψ such that ψ_f is satisfiable in $\mathbf{T}$. Then let h_f be a valuation of ψ_f such that $\mathbf{T} \models \psi_f(h_f)$. Let h be the valuation on ψ which equals h_f on the type variables in ψ and for which $h(\alpha)$ is the row ρ with domain $\{A, \ldots, B\}$ where $f(\alpha) = A\colon x_A^\alpha, \ldots, B : x_B^\alpha$ such that $\rho(A) = h_f(x_\alpha^A)$. It is easy to see that $\mathbf{T} \models \psi(h)$.

Hence, in order to check satisfiability of ψ, it suffices to guess an attribute assignment on ψ (which is polynomial in the size of ψ) and check whether ψ_f is satisfiable. The latter can be done in polynomial time, as we show in the following theorem. □

Theorem 5.14. *Satisfiability in* $\mathbf{T}$ *of quantifier free type formulas without variables of sort row can be decided in polynomial time.*

Proof. Let ψ be a quantifier free type formula without variables of sort *row*. Since ψ is a conjunction of atomic formulas, we can view ψ as a set of atomic formulas. Moreover, since there are no variables of sort *row* in ψ, every term of sort *row* in ψ is of the form $A\colon t, \ldots, B\colon t'$. We assume without loss of generality that an attribute occurs at most once in such a term. Indeed, for example the term $A\colon x, B\colon y, A\colon z, C\colon u$ is equivalent to the term $A\colon x, B\colon y, C\colon u$. Without loss of generality we will then treat such terms as rows over terms of sort *type*. Let ψ_1 be the subset of ψ defined by

$$\psi_1 := \{u_1 \# u_2 \mid (u_1 \# u_2) \in \psi\} \cup \{u_1 \subseteq u_2 \mid (u_1 \subseteq u_2) \in \psi\}.$$

Let ψ_2 be defined by

$$\begin{aligned}\psi_2 := &\{t_1 = t_2 \mid (t_1 = t_2) \in \psi\} \\ &\cup \{u_1(A) = u_2(A) \mid (u_1 \subseteq u_2) \in \psi \text{ and } A \in \mathit{dom}(u_1) \cap \mathit{dom}(u_2)\}.\end{aligned}$$

It is clear that ψ_1 and ψ_2 can be computed from ψ in polynomial time. Let us call ψ_1 *consistent* if for every $u_1 \# u_2$ in ψ_1 we have $\mathit{dom}(u_1) \cap \mathit{dom}(u_2) = \emptyset$ and for every $u_1 \subseteq u_2$ we have $\mathit{dom}(u_1) \subseteq \mathit{dom}(u_2)$.

We claim that ψ is satisfiable in $\mathbf{T}$ if, and only if, ψ_1 is consistent and ψ_2 is satisfiable in $\mathbf{T}$. Indeed, it is easy to see that if ψ is satisfiable, then ψ_1 must be consistent. Furthermore, if h is a valuation for which $\mathbf{T} \models \psi(h)$, then $h(t_1) = h(t_2)$ for every $t_1 = t_2$ in ψ and $h(u_1)(A) = h(u_2)(A)$ for every $u_1 \subseteq u_2$ in ψ and every $A \in \mathit{dom}(u_1) \cap \mathit{dom}(u_2)$. Hence, $\mathbf{T} \models \psi_2(h)$.

Conversely, suppose that ψ_1 is consistent and that ψ_2 is satisfiable in $\mathbf{T}$. Let h be a valuation such that $\mathbf{T} \models \psi_2(h)$. Then $h(t_1) = h(t_2)$ for every $t_1 = t_2$ in ψ. Furthermore, since $\mathit{dom}(u_1) \cap \mathit{dom}(u_2) = \emptyset$ for every $u_1 \# u_2$ in ψ (as ψ_1 is consistent), and since there are no variables of sort *row* in ψ, it follows that $\mathit{dom}(h(u_1)) \cap \mathit{dom}(h(u_2)) = \emptyset$ for every $u_1 \# u_2$ in ψ. Finally, since $\mathit{dom}(u_1) \subseteq \mathit{dom}(u_2)$ for every $u_1 \subseteq u_2$ in ψ (as ψ_1 is consistent) and since $h(u_1(A)) = h(u_2(A))$ for every $A \in \mathit{dom}(u_1) \cap \mathit{dom}(u_2)$ (as $\mathbf{T} \models \psi_2(h)$), it follows that $h(u_1) \subseteq h(u_2)$ for every $u_1 \subseteq u_2$ in ψ. Hence, $\mathbf{T} \models \psi(h)$.

In order to check satisfiability of ψ, it hence suffices to check consistency of ψ_1 and satisfiability of ψ_2. Consistency of ψ_1 can clearly be checked in polynomial time. We now show that satisfiability of ψ_2 in $\mathbf{T}$ can also be checked in polynomial time. Let $\prec$ be some arbitrarily fixed order on the special attributes of ψ_2. We assume without loss of generality that every term of sort *row* in ψ_2 is of the form $(A_1\colon t_1), (A_2\colon t_2), \ldots, (A_m\colon t_m)$ with $A_1 \prec A_2 \prec \cdots \prec A_m$ (as such terms can clearly be reordered in polynomial time without affecting satisfiability otherwise). Note that ψ_2 is simply a set of equations between terms of sort type. It is then easy to see that checking satisfiability of ψ_2 in $\mathbf{T}$ amounts to finding a substitution θ of variables in ψ_2

to terms in $\mathcal{L}$ of sort *type* such that $\theta(t_1)$ and $\theta(t_2)$ are syntactically equal for every equation $t_1 = t_2$ in ψ_2. Hence, satisfiability of ψ_2 reduces to finding a unifier of every equation in ψ_2, which is known to be decidable in polynomial time [4, 36, 48]. □

The complexity upper bound of NP provided by Theorem 5.13 is actually tight:

Proposition 5.15. *Typability for the NNRC is* NP*-complete.*

Proof. Since typability of an expression e is equivalent to computing the type formula ϕ_e for e and then deciding whether ϕ_e is satisfiable in $\mathbf{T}$, it follows from Theorems 5.8 and 5.13 that typability for the NNRC is in NP.

We already know that typability for the relational algebra is NP-complete from Chapter 4. Furthermore, it is well-known that the relational algebra can be simulated in the NNRC [9, 60]. It is not difficult to see that this simulation preserves typability. Hence, typability for the NNRC is also NP-complete. □

By the reduction of typability of an NNRC expression to satisfiability in $\mathbf{T}$ of a type formulas it also follows:

Corollary 5.16. *Deciding satisfiability in* $\mathbf{T}$ *of a type formula is* NP*-complete.*

6

Conclusions

In this dissertation we have studied problems related to the static detection of programming errors in database query languages. Such errors are perhaps even more critical for databases than for programming languages. Indeed, if a runtime error occurs during a transaction, an expensive recovery procedure has to be invoked to keep the database in a consistent state.

To detect such programming errors as early as possible, we have studied the well-definedness and semantic type-checking problems in the context of database query languages. Specifically, we have shown the well-definedness and semantic type-checking problems to be undecidable in general. In Chapter 3 we have identified several sources of undecidability for well-definedness: (1) non-monotonic behavior; (2) interpretation of atomic data values; (3) non-local behavior; or (4) non-local undefinedness behavior. In contrast, when we restricted our set of base operations to include only monotone, generic, local, and locally-undefined base operations, well-definedness became decidable. Although these results were obtained for first-order, object-creating languages interpreted in a tree-structured, list-based data model, we are confident that they can easily be transfered to first-order languages interpreted in other data models. For example, 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 3.25, 3.27, 3.44, and 3.45. Hence, the well-definedness problem remains decidable in this case.

From a practical point of view, however, it is clear that the number of

possible counter-examples we need to check according to the decision procedure outlined in the proofs of Theorems 2.8 and 3.46 can grow huge very fast. A future study of the computational complexity of well-definedness is hence desirable in order to obtain a practical algorithm.

Since both the well-definedness problem and the semantic type-checking problem are undecidable for database query languages that are expressive enough to simulate the relational algebra, a sound and complete type system for such languages does not exist. This observation can be seen as a formal motivation for the common use of incomplete static type systems in database query languages such as SQL, OQL, and XQuery. Nevertheless, it would be interesting to see if a hybrid approach can be made to work in practice: use the well-definedness and semantic type-checking decision procedures when possible, and use the static type system otherwise.

We have also studied classic problems from the theory of programming languages, such as type inference and typability, in the context of database query languages. Specifically, we have shown that typability is NP-complete for the relational algebra, even under various restrictions. On the positive side, typability does not become more complex when we move from the flat relational algebra to the named nested relational calculus.

Compared to the EXPTIME-completeness of typability for ML [31, 34], the NP-completeness of typability for the relational algebra or the NNRC may not seem that bad. It is important to note, however, that the relational algebra and the NNRC are essentially *simply* typed languages, i.e., languages without parametric polymorphism [42, 49]. It is exactly this feature that causes the typability problem for ML [31, 34] to be EXPTIME-hard. The correct programming language to compare against is therefore the simply typed lambda calculus, for which typability is P-complete [19].

Bibliography

[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] Franz Baader and Wayne Snyder. Unification theory. In *Handbook of Automated Reasoning*, pages 445–532. Elsevier and MIT Press, 2001.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] Peter Buneman and Atsushi Ohori. Polymorphism and type inference in database programming. *ACM Transactions on Database Systems*, 21(1):30–76, 1996.

[11] R. G. G. Cattell, Douglas K. Barry, Mark Berler, Jeff Eastman, David Jordan, Craig Russell, Olaf Schadow, Torsten Stanienda, , and Fernando Velez, editors. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann, 2000.

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

[13] Don Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An XML query language for heterogeneous data sources. In *The World Wide Web and Databases: WebDB 2000. Selected Papers*, volume 1997 of *Lecture Notes in Computer Science*, pages 1–25. Springer-Verlag, 2001.

[14] Ashok K. Chandra and Moshe Y. Vardi. The implication problem for functional and inclusion dependencies is undecidable. *SIAM Journal on Computing*, 14(3):671–677, 1985.

[15] 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/`.

[16] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley, 6th edition edition, 1995.

[17] 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.

[18] 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.

[19] Cynthia Dwork, Paris C. Kanellakis, and John C. Mitchell. On the sequential nature of unification. *Journal of Logic Programming*, 1(1):35–50, 1984.

[20] Herbert B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 2001. Second Edition.

[21] Mary F. Fernández, Daniela Florescu, Alon Levy, and Dan Suciu. Declarative specification of Web sites with Strudel. *The VLDB Journal*, 9:38–55, 2000.

[22] 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.

[23] 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.

[24] M. R. Garey and D. S. Johnson. *Computer and Intractability, A Guide to the Theory of NP-Completeness*. Freeman, 1979.

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

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

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

[28] 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.

[29] 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.

[30] Simon Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.

[31] Paris C. Kanellakis, Harry G. Mairson, and John C. Mitchell. Unification and ML-type reconstruction. In *Computational Logic - Essays in Honor of Alan Robinson*, pages 444–478. MIT Press, 1991.

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

[33] Alon Y. Levy and Dan Suciu. Deciding containment for queries with complex objects (extended abstract). In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of Database Systems*, pages 20–31. ACM Press, 1997.

[34] Harry G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 1990.

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

[36] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2), 1982.

[37] 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.

[38] 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.

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

[40] Jim Melton and Alan R. Simon. *SQL 1999: Understanding Relational Language Components*. Morgan Kaufmann, 2002.

[41] 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.

[42] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

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

[44] 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.

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

[46] A. Ohori and P. Buneman. Type inference in a database programming language. In *Proceedings of the 1988 ACM conference on LISP and functional programming*. ACM Press, 1988.

[47] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, 1995.

[48] Mike S. Paterson and Mark N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16:158–167, 1978.

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

[50] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993.

[51] T. J. Schaefer. The complexity of satisfiability problems. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 216–226. ACM Press, 1978.

[52] 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.

[53] Martin Sulzmann. *A General Framework for Hindley/Milner Type Systems with Constraints.* PhD thesis, Yale University, 2000.

[54] Martin Sulzmann. A general type inference framework for Hindley/Milner style systems. In *Functional and Logic Programming: FLOPS 2001*, volume 2024 of *Lecture Notes in Computer Science*, pages 248–263. Springer-Verlag, 2001.

[55] Jeffrey D. Ullman. *Elements of ML Programming, ML97 Edition*. Prentice-Hall, 1998.

[56] J. Van den Bussche and E. Waller. Polymorphic type inference for the relational algebra. *Jounal of Computer and System Sciences*, 64:694–718, 2002.

[57] Stijn Vansummeren. On the complexity of deciding typability in the relational algebra. *Acta Informatica*, To Appear.

[58] Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93(1):1–15, 1991.

[59] Limsoon Wong. Normal forms and conservative properties for query languages over collection types. In *Proceedings of the twelfth symposium on Principles of Database Systems*, pages 26–36. ACM Press, 1993.

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

[61] 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.

Samenvatting

De operaties van algemene programmeertalen zoals C of Java zijn slechts op bepaalde invoeren gedefinieerd. Zo is bijvoorbeeld de array-indexatie $a[i]$ enkel gedefinieerd wanneer i binnen het bereik van de array a ligt. Wanneer, gedurende de uitvoering van een programma, een operatie uitgevoerd wordt op een foutieve invoer, dan is de uitvoer van het programma ongedefinieerd. Inderdaad, in dat geval stopt het programma met een uitvoeringsfout of, nog erger, berekent het een onjuist resultaat.

Om zulke programmeerfouten zo vroeg mogelijk te detecteren is het interessant om na te gaan of we het *welgedefinieerdheidsprobleem* algoritmisch kunnen oplossen. Dit probleem bestaat er uit om, gegeven een expressie en een invoertype, te beslissen of de semantiek van de expressie gedefinieerd is voor elke invoer in het invoertype. Volgens de stelling van Rice is dit probleem jammer genoeg onbeslisbaar. De meeste programmeertalen bieden daarom een *statisch typesysteem* aan om bovenstaande programmeerfouten te detecteren [42, 49]. Deze typesystemen verzekeren "typeveiligheid" in de zin dat elke expressie die door het typesysteem aanvaard wordt gegarandeerd welgedefinieerd is. Wegens de onbeslisbaarheid van het welgedefinieerdheidsprobleem zijn deze typesystemen noodzakelijkerwijze incompleet: er zijn welgedefinieerde expressies die niet door het typesysteem aanvaard worden. Zulke expressies zijn problematisch vanuit het standpunt van de programmeur, aangezien hij zijn code moet herschrijven vooraleer die door het typesysteem aanvaard wordt. In de theorie der programmeertalen is men dan ook steeds op zoek naar typesystemen waarvoor de verzameling van welgedefinieerde maar niet-aanvaarde expressies zo klein mogelijk is.

Alhoewel de heilige graal in deze zoektocht (met name, een typesysteem dat precies alle welgedefinieerde expressies aanvaardt) onvindbaar is voor algemene programmeertalen, is dat niet noodzakelijk zo voor *gespecialiseerde gegevensbanktalen* zoals SQL [40], OQL [11] en XQuery [6].[1] Ook in deze ta-

[1] XQuery is in principe een volledige programmeertaal. De meeste XQuery programma's zijn echter van de vorm "for-let-where-return", wat we als de gegevensbankstaal van XQuery beschouwen.

len kunnen expressies ongedefinieerd zijn. Beschouw bijvoorbeeld de volgende OQL expressie:

```
select author: element(b.authors), title: b.title
from books b
where b.pub_year > 2000
```

Deze expressie zal voor elk boek dat na het jaar 2000 gepubliceerd is de auteur en titel teruggeven. De deelexpressie `element(b.authors)` gaat na dat het boek `b` door precies één auteur geschreven is. Indien dat zo is wordt deze unieke auteur teruggegeven, anders is het resultaat van deze deelexpressie ongedefinieerd.

Aangezien gegevensbanktalen een beperktere uitdrukkingskracht hebben dan algemene programmeertalen is de stelling van Rice niet op hen van toepassing. Bijgevolg is het interessant om na te gaan of het welgedefinieerdheidsprobleem voor zulke talen beslisbaar is. In dit proefschrift bestuderen we daarom het welgedefinieerdheidsprobleem voor gegevensbanktalen. We beginnen onze studie met welgedefinieerdheid voor de geneste relationele calculus (afgekort als NRC). De NRC is een canonieke gegevensbanktaal voor het complexe object datamodel [1, 9, 60]. Het is een conservatieve extensie [59] van de relationele algebra (dewelke het hart van SQL vormt), en kan zelf gezien worden als het hart van OQL. Bovendien inspireerde de NRC ook het ontwerp van verscheidene gegevensbanktalen voor semi-gestructureerde data zoals UnQL [8], StruQL [21], en Quilt [13] waarop XQuery gebaseerd is. Onze studie van welgedefinieerdheid voor de NRC is dan ook een goed startpunt voor de studie van welgedefinieerdheid in SQL, OQL en XQuery.

Concreet gezien bestuderen we het welgedefinieerdheidsprobleem voor de NRC in het standaard, verzamelingengebaseerde, complexe object datamodel [1, 9, 60]. We tonen aan dat het probleem onbeslisbaar is voor de NRC in zijn algemeenheid, maar beslisbaar wordt wanneer we ons beperken tot diens positief existentieel fragment (dewelke we als PENRC afkorten). Vervolgens bestuderen we welgedefinieerdheid voor de PENRC in de aanwezigheid van *extract*, een operator die, net zoals OQL's elementoperator, ongedefinieerd is op niet-singleton verzameling invoeren. We tonen aan dat deze operator het welgedefinieerdheidsprobleem opnieuw onbeslisbaar maakt. Tenslotte bestuderen we het welgedefinieerdheidsprobleem voor de PENRC in de aanwezigheid van *typetesten*, die onder andere in XQuery voorkomen. Aan de hand van zulke testen kan men tijdens de uitvoering van een programma het type van een waarde inspecteren. We tonen aan dat ook typetesten het probleem opnieuw onbeslisbaar maken. We identificeren wel een beperkte vorm van typetesten waarvoor het probleem beslisbaar blijft.

Bepaalde eigenschappen van OQL en XQuery zijn echter niet aanwezig in de standaard, verzamelinggebaseerde NRC. Inderdaad, OQL werkt ook op bags en lijsten, terwijl XQuery op lijsten werkt. Beide talen bezitten objectidentiteit en de mogelijkheid om nieuwe objecten aan te maken. Daarom bestuderen we ook welgedefinieerdheid voor een familie QL(B) van gegevenbanktalen die in een boomgestructureerd, lijstengebaseerd datamodel geïnterpreteerd worden. Hierbij is B een verzameling van *basisoperaties* (zoals een gelijkheidstest, het berekenen van de kinderen van een bepaalde knoop in een boom, ...). De gegevensbanktaal QL(B) wordt dan bekomen door aan B variabelen, constanten, conditionele testen, variabelebindingen en iteratie toe te voegen. Als dusdanig is elke QL(B) een eerste orde objectcreërende gegevensbanktaal. We identificeren eigenschappen van basisoperaties die het welgedefinieerdheidsprobleem mogelijkerwijze onbeslisbaar maken en stellen beperkingen voor die beslisbaarheid garanderen. De behaalde resultaten zijn rechtstreeks toepasbaar op OQL en XQuery.

Een probleem dat gerelateerd is met welgedefinieerdheid is het semantisch typeberekeningsprobleem. Dat probleem bestaat er uit om, gegeven een expressie, een invoertype en een uitvoertype, na te gaan of de expressie enkel resultaten in het uitvoertype produceert wanneer we ze evalueren op invoeren in het invoertype. Het semantisch typeberekeningsprobleem is nuttig in een "producent-consument" scenario waarin een producent data genereert, dewelke verwerkt wordt door een consument. Om een goede communicatie te verzekeren mag de producent enkel data behorende tot een zeker type produceren. Spijtig genoeg is dit probleem ook onbeslisbaar voor algemene programeertalen volgens de stelling van Rice. In de praktijk is de producent echter vaak een expressie in een gegevensbanktaal. Het is daarom nuttig om het semantisch typeberekeningsprobleem voor gegevensbanktalen te onderzoeken. In dit proefschrift doen we dat voor de NRC. Voor XQuery en andere XML-gerelateerde gegevensbanktalen werd dat probleem reeds uitvoerig bestudeerd [2, 3, 37, 38, 41, 52]. We tonen aan dat ook het semantisch typeberekeningsprobleem onbeslisbaar is voor de NRC in zijn algemeenheid, maar beslisbaar wordt voor de PENRC.

Aangezien zowel het welgedefinieerdheidsprobleem als het semantisch typeberekeningsprobleem in het algemeen onbeslisbaar blijven voor gegevensbanktalen, volgt hieruit dat ook in gegevensbanktalen de detectie van programmeerfouten door middel van een incompleet statisch typesysteem moet gebeuren. In het tweede gedeelte van dit proefschrift bestuderen we daarom klassieke problemen voor typesystemen in de context van gegevensbanktalen.

Vooreerst bestuderen we de complexiteit van typeerbaarheid voor de relationele algebra. Dat probleem bestaat er uit om, gegeven een relationele algebra-expressie, na te gaan of de expressie door het statisch typesysteem

van de relationele algebra aanvaard wordt. Typeerbaarheid in de relationele algebra is het equivalent van typechecking in impliciet getypeerde programmeertalen met polymorfe typesystemen, zoals ML [55] en Haskell [30]. Het is daarom ook interessant om na te gaan wat de complexiteit ervan is. Zo is het bijvoorbeeld bekend dat typeerbaarheid voor de simpel getypeerde lambda calculus P-compleet is, terwijl het EXPTIME-compleet is voor ML [31, 34]. Van den Bussche en Waller hebben reeds aangetoond dat typeerbaarheid voor de relationele algebra in NP zit. De precieze complexiteit was echter onbekend. We tonen aan dat het probleem in zijn algemeenheid NP-compleet is. In het bijzonder tonen we aan dat het probleem NP-hard wordt door (1) de productoperator; (2) de selectie-operator op willekeurige verzamelingen van getypeerde predikaten; en (3) de selectie-operator op "brave" verzamelingen van getypeerde predikaten tezamen met join en projectie of hernoeming. Het probleem zit in P wanneer (1) we enkel unie, verschil, join en selectie op "brave" verzamelingen van getypeerde predikaten toelaten; of (2) we alle operaties behalve product toelaten en de verzameling van selectiepredikaten ten hoogste één basistype vermeldt. De meeste van deze resultaten volgen uit een hecht verband van typeerbaarheid met niet-uniforme constraint-satisfaction.

Vervolgens bestuderen we het statisch typesysteem van de genaamde versie van de geneste relationele calculus (afgekort als NNRC). De basisoperaties van de NNRC zijn *polymorf* ten opzichte van dit typesysteem. Zo kunnen we het attribuut A inspecteren van eender welk record, zolang dat een attribuut A bevat. We kunnen het product nemen van eender welke twee records, zolang hun verzameling van attributen disjunct is. We kunnen de unie nemen van eender welke twee verzamelingen van hetzelfde type. Deze typeringsvoorwaarden worden complexer wanneer we operators in expressies combineren. Beschouw bijvoorbeeld de expressie $\{(x \times y).A \mid x \in R\}$. Opdat deze expressie door het typesysteem van de NNRC aanvaard zou worden moet R een verzamelingtype hebben dewelke het type van x bevat; moeten x en y recordtypes hebben wiens verzamelingen van attributen disjunct zijn; en moet één van deze verzamelingen van attributen A bevatten.

Dan stelt zich de vraag onder welke toekenningen van types aan de vrije variabelen van een NNRC-expressie e de expressie door het typesysteem aanvaard wordt. En wat is het overeenkomstige type van de expressie onder deze toekenningen? Kunnen we in het bijzonder een expliciete beschrijving geven van deze, typisch oneindige, verzameling van *typeringen*? Dit probleem is niets anders dan de NNRC-versie van het klassieke *type-inferentie* probleem. Type-inferentie is een uitvoerig bestudeerd onderwerp in de theorie van programmeertalen [42, 49] en wordt in geavanceerde functionele programmeertalen zoals ML [55] en Haskell [30] gebruikt.

In dit proefschrift stellen we een expliciete beschrijving van de verzameling

van alle mogelijke typeringen van een NNRC-expressie e voor door middel van een conjunctieve logische formule ϕ_e dewelke in het universum van alle mogelijke types geïnterpreteerd wordt. Deze formule ϕ_e is efficiënt berekenbaar uit e. Vervolgens tonen we aan dat het satisfiability-probleem van zulke conjunctieve formules in NP zit. Hieruit volgt dat typeerbaarheid voor de NNRC ook in NP zit. Aangezien de NNRC een uitbreiding is van de relationele algebra, waarvoor typeerbaarheid reeds NP-compleet is, is typeerbaarheid voor de NNRC dus niet moeilijker dan voor het speciale geval van de relationele algebra.