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

Jan Van den Bussche, Dirk Van Gucht, and Stijn Vansummeren*

January 25, 2006

Abstract

The well-definedness problem for a programming language consists of checking, given an expression and an input type, whether the semantics of the expression is defined for all inputs adhering to the input type. A related problem is the semantic type-checking problem which consists of checking, given an expression, an input type, and an output type whether the expression always returns outputs adhering to the output type on inputs adhering to the input type. Both problems are undecidable for general-purpose programming languages. In this paper we study these problems for the Nested Relational Calculus, a specific-purpose database query language. We also investigate how these problems behave in the presence of programming language features such as singleton coercion and type tests.

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 [17, 18]. These systems

*Research Assistant of the Fund for Scientific Research - Flanders.

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

Although the Holy Grail in this quest (i.e., a 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 [15], OQL [8], and XQuery [5]. 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 paper we study the well-definedness problem for the Nested Relational Calculus (NRC for short), which is well-known from the complex object data model [1, 7, 23]. The NRC is a conservative extension of the relational algebra [22] (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 [6], StruQL [11], and Quilt [9], on which XQuery is based. As such, the study of well-definedness for the NRC serves as a starting point for the study of well-definedness in SQL, OQL, and XQuery.

Specifically, we study the well-definedness problem for the NRC in the standard, set-based, complex object data model. We obtain that the problem is undecidable for the NRC in general, but is decidable for the positive-existential fragment of the NRC (PENRC for short). Next, we study well-definedness for the PENRC in the presence of the singleton coercion operator

extract. This operator, 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 $extract(\{e_1, e_2\})$ 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*.

Certain features of OQL and XQuery are not covered in this paper. For example, 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 study the well-definedness problem for these features in a companion paper [20, 21].

A useful side-effect of having a static type-system is that it computes an output type for every well-typed expression. All outputs generated by the expression are guaranteed to belong to this type. Such an output type is useful in a "producer-consumer" setting where a producer uses a query to generate data, which is processed by a consumer. In order to ensure good operation, the producer is expected to only produce data adhering to a certain type. This can be statically checked by investigating the output type of the query expression. The static type system only computes output types for well-typed expressions however. Moreover, the output type computed by the type system is often "too big" in the sense that it contains values which will never be output. It is therefore interesting to see if we can't solve the *semantic type-checking problem* for the query languages mentioned above. This problem consists of checking, given an expression, an input type and an output type, whether the expression always returns outputs adhering to the output type on inputs in the input type. We study this problem for the NRC and the complex object type system, where we obtain the same (un)decidability results as for well-definedness. The semantic type-checking problem has already been studied extensively in XML-related query languages [2, 3, 13, 14, 16, 19]. 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 the complex object type system in the type system of Alon et al. 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, we will see that semantic type-checking for the PENRC in the complex object type system is decidable.

Organization This paper is further organized as follows. In Section 2 we introduce the nested relational calculus data model and query language. We introduce the well-definedness problem in Section 3, 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 in the presence of singleton coercion and type tests in Section 4 respectively Section 5. We study the semantic type-checking problem in Section 6 and conclude in Section 7.

2 Nested Relational Calculus

Data model We assume 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 *value* is either an atom, a pair of values, or a finite set of values. For example, $\{a, (b, c), (a, \{a, b\})\}$ is a value. We will range over values by u, v, and w and over finite sets of 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 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 *context* σ is a function from a finite set of variables $dom(\sigma)$ to values. If $dom(\sigma)$ is a superset of $FV(e)$, then we say that σ is a *context on* e. 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 NRC expressions is described by means of the *evaluation relation*, as defined in Figure 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

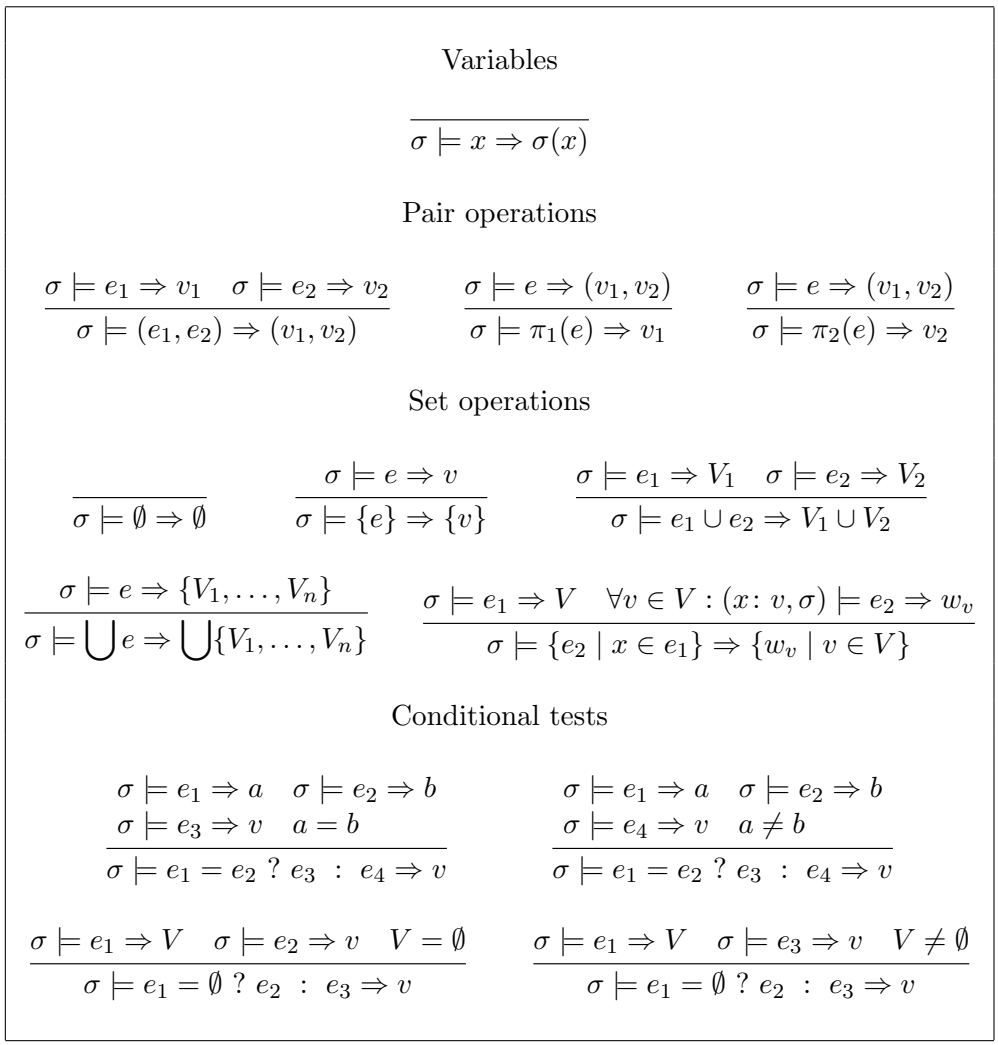

Figure 1: The evaluation relation for NRC expressions.

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, $\sigma' \models e \Rightarrow v$.

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

Example 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 $\Gamma(\mathit{John}) = \mathbf{Atom}$. □

3 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. Let e be an 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* consists of checking, given an expression e and a type assignment Γ on e, 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 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 [7, 23] 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. Actually, *every* relational algebra expression (with equality predicates only) in this fragment is satisfiable. 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.

3.1 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 [4]:

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

Note that $\sqsubseteq$ is only a pre-order, not a partial order. Indeed, $\sqsubseteq$ fails to be anti-symmetric: $\{\{a,b\}\} \sqsubseteq \{\{a\},\{a,b\}\}$ and $\{\{a\},\{a,b\}\} \sqsubseteq \{\{a,b\}\}$, but $\{\{a,b\}\} \neq \{\{a\},\{a,b\}\}$.

The containment relation 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 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 $R = \{a\}$ 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 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 3.2 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 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 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 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 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 $rank(\tau, k)$. Then clearly,

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

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

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

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 occurring in σ is at most l there surely exists a renaming ρ 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. □

3.2 Small Model Properties

In this section we prove the small model property for undefinedness (Proposition 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 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 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 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 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 $(u \sqcup v) \sqsubseteq w$.*

Recall that $\sqsubseteq$ is not anti-symmetric, so least upper bounds according to $\sqsubseteq$ need not be unique, as already illustrated by the remark we made after defining $\sqsubseteq$ in Section 3.1.

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 $x \in dom(\sigma)$. It follows from Lemma 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 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.

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

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 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 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 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:

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

Since an arithmetic expression defining $c_{e'}$ is computable from e' by Proposition 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 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 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 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 11. Let $\sigma' = \sigma_1 \sqcup \sigma_2$. Note that $\sigma_1 \sqsubseteq \sigma'$ and $\sigma_2 \sqsubseteq \sigma'$ by Lemma 11. By monotonicity $\{u\} \sqsubseteq e_1(\sigma')$. Hence, there exists some $u' \in e_1(\sigma')$ such that $u \sqsubseteq u'$. Then $x\colon u, \sigma_2 \sqsubseteq x\colon u', \sigma'$, 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 6 now follows by Lemma 13 and Lemma 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 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 9. Hence the proposition.

4 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" [23]. 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.[1]

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

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:

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

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 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 15. *The equivalence problem for PENRC is undecidable.*[2]

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

[2]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 [12].

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 $\mathbf{Pair}(\mathbf{Atom}, \mathbf{Pair}(\mathbf{Atom}, \mathbf{Atom}))$. 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, 10]. 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, 10].

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 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 show in our companion paper [20].

5 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 the PENRC extended with such a type test expression:

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

Here, τ ranges over types. The resulting language will be denoted by PENRC(*type*). 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 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 3 that well-definedness for PENRC(*type*) is undecidable.

Corollary 18. *The well-definedness problem for PENRC(type) is undecidable.*

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

Example 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 write PENRC(*kind*) for the obtained language. 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 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 4). We can therefore extend the proofs of Proposition 12 and Lemma 13 to show that the PENRC(*kind*) also has the small model properties for definedness and undefinedness.

Proposition 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 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 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 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 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 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 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 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 9, the small model property for undefinedness continues to hold in the presence of kind tests. It readily follows (cf. the proof of Theorem 8):

Theorem 23. *The well-definedness problem for PENRC(kind) is decidable.*

6 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 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 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 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 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 25. By Proposition 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 9. Since $\sigma \in \Gamma$, also $\sigma' \in \Gamma$ by Lemma 9. □

It readily follows (cf. the proof of Theorem 8):

Proposition 27. *The semantic type-checking problem for PENRC(kind) is decidable.*

7 Conclusion

We have shown that the well-definedness problem, which is undecidable for any general-purpose programming language, remains undecidable for special-purpose query languages powerful enough to simulate the relational algebra. Specifically, we have shown that the well-definedness problem for the NRC is undecidable. In contrast, this problem becomes decidable when one limits the NRC to its positive-existential fragment. The core reason for this decidability is a small model property. If we add a singleton coercion operator to the PENRC, then well-definedness becomes undecidable again. The well-definedness for the PENRC also becomes undecidable if we add a type test construct. Fortunately, well-definedness remains decidable if we limit ourselves to kind tests, a limited form of type tests. Finally, we have shown that the semantic type-checking problem is also undecidable for the full NRC, although it is decidable for the PENRC with kind tests.

It is clear that for the settings where the well-definedness problem is decidable, the proposed algorithm of enumerating all possible counter-examples is computationally expensive. Therefore, a precise analysis of the computational complexity of the well-definedness problem is desirable, as is an investigation on how to obtain an algorithm which performs well in practice.

References

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

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

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

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

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

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

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

[19] 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.

[20] Stijn Vansummeren. Deciding well-definedness of first-order, object-creating operations over tree-structured data. `http://alpha.uhasselt.be/~lucg5855/pubs.html`.

[21] Stijn Vansummeren. Deciding well-definedness of XQuery fragments. In *PODS*, 2005.

[22] 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.

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