Well-definedness and semantic type-checking for the nested relational calculus

Jan Van den Bussche\textsuperscript{a}, Dirk Van Gucht\textsuperscript{b}, Stijn Vansummeren\textsuperscript{a,}\textsuperscript{*}

\textsuperscript{a}Hasselt University and Transnational University of Limburg, Agoralaan, Gebouw D, B-3590 Diepenbeek, Belgium
\textsuperscript{b}Indiana University, Computer Science Department, Lindley Hall 215, 150 S. Woodlawn Ave., Bloomington, IN 47405-710, USA

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.

© 2006 Elsevier B.V. All rights reserved.

Keywords: Complex objects; Query equivalence; Semantic type-checking; Well-definedness

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. Most programming languages therefore provide a static type system to detect programming errors \cite{17,18}. These systems ensure “type safety” in the sense that every expression which passes the type system’s tests is guaranteed to be well-defined. Due to the undecidability of the well-definedness problem, these systems are...
necessarily incomplete, i.e., there are expressions which are well-defined, but do not type-check. Such expressions are problematic from a programmer’s point of view, as he must rewrite his code in order to get it to type-check. As such, a major quest in the theory of programming languages consists of finding static type systems for which the set of well-defined but ill-typed expressions is as small as possible.

Although the Holy Grail in this quest (i.e., a 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:

\[
\text{select author: element(b.authors), title: b.title}
\]

\[
\text{from books b}
\]

\[
\text{where b.pub_year > 2000}
\]

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, undecidability of well-definedness is no longer obvious and a study of well-definedness for such languages is therefore desirable. Note that query languages for which well-definedness is decidable in essence possess a type system that 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 causes 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 and Section 5 respectively. 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 $\mathcal{A} = \{ a, b, \ldots \}$ 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 that a set $\mathcal{X} = \{ x, y, \ldots \}$ of variables is given. The Nested Relational Calculus (NRC) is the set of all expressions generated by the following grammar:

$$e \ ::= \ x \mid \langle e, e \rangle \mid \pi_1(e) \mid \pi_2(e) \mid \emptyset \mid \{ e \} \mid e \cup e \mid \bigcup e \mid \{ e \mid x \in e \} \mid e = e \ ? \ e \ :\ e \mid e = \emptyset \ ? \ e \ :\ e$$

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 $\sigma$ 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 $\sigma$ is a context on $e.$ We denote by $x: v, \sigma$ the context $\sigma'$ 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 Fig. 1. Here, we write $\sigma \models e \Rightarrow v$ to denote the fact that $e$ evaluates to value $v$ on context $\sigma$ 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 $\sigma,$ since $\pi_1$ 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 $\sigma$ and $\sigma'$ 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:

$$\tau \ ::= \ Atom \mid Pair(\tau, \tau) \mid SetOf(\tau) \mid \tau \cup \tau.$$

A type $\tau$ denotes a set of values $[\tau]:$

- $[Atom] := \mathcal{A},$
- $[Pair(\tau_1, \tau_2)] := [\tau_1] \times [\tau_2].$
Let \( FV \supseteq \text{We will abuse notation and identify } \tau \sigma \) to types. A type assignment denotes the set of contexts

\[
\Gamma \text{ of John pairs of atoms. Suppose also that the value of } 3. \text{ Well-definedness}
\]

\[
\text{evaluate to a value on } \sigma \text{. This leads us to the following central notion:}
\]

\[
\begin{array}{l}
\text{Variables} \\
\sigma \models x \Rightarrow \sigma(x)
\end{array}
\]

\[
\text{Pair operations}
\]

\[
\begin{array}{l}
\sigma \models e_1 \Rightarrow v_1 \quad \sigma \models e_2 \Rightarrow v_2 \\
\sigma \models (e_1, e_2) \Rightarrow (v_1, v_2)
\end{array}
\]

\[
\begin{array}{l}
\sigma \models e \Rightarrow (v_1, v_2) \\
\sigma \models \pi_1(e) \Rightarrow v_1 \\
\sigma \models \pi_2(e) \Rightarrow v_2
\end{array}
\]

\[
\text{Set operations}
\]

\[
\begin{array}{l}
\sigma \models \emptyset \Rightarrow \emptyset \\
\sigma \models \{e\} \Rightarrow \{v\}
\end{array}
\]

\[
\begin{array}{l}
\sigma \models e \Rightarrow \{V_1, \ldots, V_n\} \\
\sigma \models \bigcup e \Rightarrow \bigcup\{V_1, \ldots, V_n\}
\end{array}
\]

\[
\begin{array}{l}
\sigma \models e \Rightarrow V_1 \quad \sigma \models e_2 \Rightarrow V_2 \\
\sigma \models e_1 \cup e_2 \Rightarrow V_1 \cup V_2
\end{array}
\]

\[
\begin{array}{l}
\sigma \models e \Rightarrow \forall v \in V : (x : v, \sigma) \models e_2 \Rightarrow w_v
\end{array}
\]

\[
\begin{array}{l}
\sigma \models \{e_2 \mid x \in e_1\} \Rightarrow \{w_v \mid v \in V\}
\end{array}
\]

\[
\text{Conditional tests}
\]

\[
\begin{array}{l}
\sigma \models e_1 \Rightarrow a \quad \sigma \models e_2 \Rightarrow b \\
\sigma \models e_3 \Rightarrow v \quad a = b \\
\sigma \models e_1 = e_2 ? e_3 : e_4 \Rightarrow v
\end{array}
\]

\[
\begin{array}{l}
\sigma \models e_1 \Rightarrow V \quad \sigma \models e_2 \Rightarrow v \\
\sigma \models e_1 = \emptyset ? e_2 : e_3 \Rightarrow v
\end{array}
\]

\[
\begin{array}{l}
\sigma \models e_1 \Rightarrow V \quad \sigma \models e_2 \Rightarrow v \\
\sigma \models e_1 = \emptyset ? e_2 : e_3 \Rightarrow v
\end{array}
\]

\[
\begin{array}{l}
\sigma \models e_1 \Rightarrow V \quad \sigma \models e_2 \Rightarrow v \\
\sigma \models e_1 = \emptyset ? e_2 : e_3 \Rightarrow v
\end{array}
\]

\[
\text{Fig. 1. The evaluation relation for NRC expressions.}
\]

- \([\text{SetOf}(\tau)]\) 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 \( \tau \) with \([\tau]\). A type assignment \( \Gamma \) is a function from a finite set of variables \( \text{dom}(\Gamma) \) to types. A type assignment denotes the set of contexts \( \sigma \) for which \( \text{dom}(\sigma) = \text{dom}(\Gamma) \) and \( \sigma(x) \in \Gamma(x) \), for every \( x \in \text{dom}(\sigma) \). Again, we will abuse notation and identify a type assignment with its denotation. Finally, if \( \text{dom}(\Gamma) \) is a superset of \( FV(e) \), then we say that \( \Gamma \) is a type assignment on \( e \).

**Example 1.** Let \( \text{friends} \) and \( \text{John} \) be two variables. Suppose that the value of \( \text{friends} \) is a set of friends, as a set of pairs of atoms. Suppose also that the value of \( \text{John} \) is a name (an atom). The following expression computes the set of all of \( \text{John’s} \) friends:

\[
\bigcup \{\pi_1(x) = \text{John} \? \{\pi_2(x)\} \mid \emptyset | x \in \text{friends}\}.
\]

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

**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 \( \sigma \)). This leads us to the following central notion:
Let It is well-known that the (finite) satisfiability problem for the relational algebra is undecidable. The well-definedness problem for the NRC is undecidable.

Since an actual implementation of the NRC will produce a runtime error on those contexts of $\sigma$ 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 $\phi$ over a relational schema $S$, whether $\phi$ 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:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_1</td>
<td>b_1</td>
<td>c_1</td>
</tr>
<tr>
<td>a_2</td>
<td>b_1</td>
<td>c_2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_1</td>
<td>a_1</td>
</tr>
<tr>
<td>c_2</td>
<td>a_2</td>
</tr>
</tbody>
</table>

Clearly, $D$ can then be encoded as the context $\sigma$ 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 $\phi$ and $S$ there exists an expression $e$ and a type assignment $\Gamma$ such that

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

The fact that $e$ is well-defined under $\Gamma$ stems from the fact that relational algebra expressions are always well-defined with regard to their database schema. It is clear that $\phi$ is satisfiable if, and only if, $e$ is satisfiable on a context in $\Gamma$. Since the expression $\{x \in e \mid \sigma_1(\emptyset)\}$ is not well-defined under $\Gamma$ if, and only if $e$ is satisfiable, we have a reduction from satisfiability to well-definedness. Hence, well-definedness is undecidable. $\square$

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 $S$:

$$\bigcup \bigcup \{x \in x \mid y \in S \} \emptyset \{x \} : \emptyset \mid x \in R \} .$$

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 calculi

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
Let $e$ be a PENRC expression and let

$$\sigma\vdash e.$$ 

The well-definedness problem for the PENRC is decidable. We will show in Section 3.2 that there exists a natural number $k$, computable from $e$, and a context $\sigma'$ with $\sigma' \in \Gamma'$ of width at most $k$ such that $e(\sigma')$ is also undefined. 

**Lemma 4 (Monotonicity).** Let $e$ be a PENRC expression and let $\sigma$ and $\sigma'$ be contexts on $e$ such that $\sigma \subseteq \sigma'$. If $e(\sigma)$ and $e(\sigma')$ are defined, then $e(\sigma) \subseteq e(\sigma')$. 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 $R \subseteq R$ and $S \subseteq S'$. 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

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

and let the type assignment $\Gamma$ on $e$ be defined by:

$$\Gamma(R) = \text{SetOf(\text{SetOf(Atom)})}$$

$$\Gamma(z) = \text{Atom}.$$ 

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 $\pi_1$ 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 $\sigma'$ 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 $\sigma$ has width at most $k$ if $\sigma(x)$ has width at most $k$, for every $x \in \text{dom}(\sigma)$.

**Proposition 6 (Small Model for Undefinedness).** Let $e$ be a PENRC expression and let $\Gamma$ be a type assignment on $e$ such that $e$ is not well-defined under $\Gamma$. 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 $\rho$ be a permutation of $\mathcal{A}$. We extend $\rho$ to values in the canonical way:

$$\rho((v, w)) := (\rho(v), \rho(w))$$

$$\rho(V) := \{\rho(v) | v \in V\}.$$ 

We also extend $\rho$ component-wise to contexts:

$$\rho(\sigma)(x) := \rho(\sigma(x)).$$

Two contexts $\sigma$ and $\sigma'$ are isomorphic if there exists a permutation $\rho$ 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 $\sigma$ be a context on $e$, and let $\rho$ 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.
Suppose that expression \( e \) is not well-defined under type assignment \( \Gamma \) 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 \( \tau \) of width at most \( k \) can mention by \( \text{rank}(\tau, k) \). Then clearly,

\[
\text{rank}(\text{Atom}, k) = 1 \\
\text{rank}(\text{Pair}(\tau_1, \tau_2), k) = \text{rank}(\tau_1, k) + \text{rank}(\tau_2, k) \\
\text{rank}(\text{SetOf}(\tau'), k) = k \times \text{rank}(\tau', k) \\
\text{rank}(\tau_1 \cup \tau_2, k) = \max\{\text{rank}(\tau_1, k), \text{rank}(\tau_2, k)\}.
\]

Consequently, the maximum number of atoms mentioned in \( \sigma \) is bounded by

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

Note that \( l \) is computable from \( \Gamma \) and \( e \). Now fix some \( l \)-element subset \( A \) of \( \mathcal{A} \). Since the number of different atoms occurring in \( \sigma \) is at most \( l \) there surely exists a renaming \( \rho \) 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 \( \Gamma \), 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 \( \gamma \), from which the result follows. \( \square \)

### 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 \( \tau \) be a type. If \( w \in \tau \) and \( v \sqsubseteq w \), then also \( v \in \tau \).

The proof is by a straightforward induction on \( \tau \). In order to prove Proposition 6 it hence suffices to show that, given an expression \( e \) and a context \( \sigma \) 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 \( \sigma \) (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 \( \sigma \) 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 \( (x : a, \sigma) \) (which is undefined), it follows that \( e(\sigma) \) is also undefined. Clearly, the undefinedness of \( \pi_1(x)(x : a, \sigma) \) 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 \( \sigma' \) 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 \( \sigma'(S) = \emptyset \). \( \square \)

As this example illustrates, we will want to show that, given an expression \( e_1 \), a context \( \sigma \) 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 \( \sigma \). 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:

\[
a \sqcup a := a \\
(u_1, u_2) \sqcup (v_1, v_2) := (u_1 \sqcup v_1, u_2 \sqcup v_2) \\
U \sqcup V := U \cup V.
\]

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 $\sigma$ and $\sigma'$ 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 $x \in \text{dom}(\sigma)$. 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 $\sigma$ has width at most $k$ and $\sigma'$ 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 $\text{minimize}$ on values:

$$\text{minimize}(a) := a$$

$$\text{minimize}(u_1, u_2) := (\text{minimize}(u_1), \text{minimize}(u_2))$$

$$\text{minimize}(V) := \emptyset.$$

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

**Convention.** In what follows we will write $V_k$ for the set of all values of width at most $k$ and $C_k$ for the set of all contexts of width at most $k$.

**Proposition 12 (Small Model for Definedness).** Let $e$ be a PENRC expression. There exists a computable function $c_e$ mapping natural numbers to natural numbers such that for every natural number $k$, every context $\sigma$ on $e$ for which $e(\sigma)$ is defined, and every $u \subseteq e(\sigma)$ of width at most $k$, there exists a context $\sigma' \sqsubseteq \sigma$ of width at most $c_e(k)$ such that $u \subseteq e(\sigma')$. 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.

$$c_x(k) := k$$

$$c_{(e_1, e_2)}(k) := c_{e_1}(k) + c_{e_2}(k)$$

$$c_{\pi_1(e')}(k) := c_{e'}(k)$$

$$c_{\pi_2(e')}(k) := c_{e'}(k)$$

$$c_\emptyset(k) := 0$$

$$c_{[e_1]}(k) := k \times c_{e_1}(k)$$

$$c_{[\mathit{true}_e]}(k) := c_{e_1}(k) + c_{e_2}(k)$$

$$c_{[\mathit{false}_e]}(k) := c_{e'}(k)$$

$$c_{[\mathit{true}_e \land \mathit{false}_e]}(k) := c_{e_1}(k) + c_{e_2}(k)$$

$$c_{[\mathit{true}_e \lor \mathit{false}_e]}(k) := c_{e'}(k)$$

$$c_{[\mathit{true}_e \lor \mathit{false}_e] ? e_3 : e_4}(k) := \max\{c_{e_3}(k), c_{e_4}(k)\}.$$

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 $\sigma$ be a context on $e$ for which $e(\sigma)$ is defined, and let $u \subseteq e(\sigma)$ be a value of width at most $k$. Define the predicate $P(u, e, \sigma, k)$ as follows:

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

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 $\sigma'$ by

$$\sigma'(y) = \begin{cases} u & \text{if } y = x \\ \text{minimize}(\sigma(y)) & \text{otherwise.} \end{cases}$$
• If \( e = \emptyset \), then we take \( \sigma^\prime = \text{minimize}(\sigma) \).

• If \( e = (e_1, e_2) \), then \( e(\sigma) \) is a pair. Hence, \( u = (u_1, u_2) \) for some \( u_1, u_2 \in V_k \). 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 \( \sigma_1 \cup \sigma_2 \in C_{e_1(k)} + c_{e_2(k)} = C_{e(k)} \). Moreover, by monotonicity:

\[
(u_1, u_2) \subseteq (e_1(\sigma_1), e_2(\sigma_2)) \subseteq (e_1(\sigma_1 \cup \sigma_2), e_2(\sigma_1 \cup \sigma_2)) = e(\sigma_1 \cup \sigma_2).
\]

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

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

Define,

\[
u_1 := \{ v \in u \mid w_v \in e(\sigma_1) \}, \]

\[
u_2 := \{ v \in u \mid w_v \in e(\sigma_2) \}.
\]

Then \( u = u_1 \cup u_2, u_1 \subseteq e(\sigma) \), and \( u_2 \subseteq e(\sigma) \). Moreover, \( u_1, u_2 \in V_k \). 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 \( u' = (u, \text{minimize}(w)) \). Then \( u' \subseteq (v, w) \) since \( u \subseteq v \) and \( \text{minimize}(w) \subseteq w \). Moreover, \( u' \in V_k \) because \( \text{minimize}(w) \in V_0 \). Hence there exists \( \sigma' \in P(u', e', \sigma, k) \) by the induction hypothesis. Hence,

\[
u = \pi_1(u') \subseteq \pi_1(e'(\sigma')) = e(\sigma').
\]

Since also \( C_{e'(k)} = C_{e'(k)} \), 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 \( u \subseteq e(\sigma') \) for any \( \sigma' \subseteq \sigma \). Hence, it suffices to take \( \sigma' = \text{minimize}(\sigma) \), which is in \( C_0 \subseteq C_{e(k)} \). 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 \( v \subseteq e'(\sigma) \). 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 \( \sigma_v \subseteq \sigma' \) for every \( v \in u \), and \( \sigma' \subseteq \sigma \). By monotonicity we then have \( v \subseteq e'(\sigma_v) \subseteq e'(\sigma') \), and hence \( u \subseteq e'(\sigma') = e(\sigma') \). Moreover, \( \sigma' \in C_{k \times c_{e(k)}} = C_{e(k)} \). Hence, \( \sigma' \in P(u, e, \sigma, k) \).

• If \( e = \bigcup e' \), then \( e'(\sigma) \) is a set of sets. Since \( u \subseteq e(\sigma) \) there exists, for every \( v \in u \), a \( w_v \in e(\sigma) \) such that \( v \subseteq w_v \).

Let \( e'(\sigma) = \{ V_1, \ldots, V_n \} \). Note that \( e(\sigma) = V_1 \cup \cdots \cup V_n \). Define, for each \( i \in [1, n] \),

\[
U_i := \left\{ v \in u \mid w_v \in V_i \setminus \bigcup_{j < i} V_j \right\}.
\]

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, \( U_i \subseteq V_i \). Let \( u' \) be the set of all non-empty \( U_i \)’s. Then \( u' \subseteq e'(\sigma) \) and \( u' \in V_k \). The result then follows from the induction hypothesis.

• If \( e = \{ e_2 \mid x \in e_1 \} \), then \( e(\sigma) \) is a set. Since \( u \subseteq e(\sigma) \) there exists, for every \( v \in u \), a \( w_v \in e(\sigma) \) such that \( v \subseteq w_v \).

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 \( v = e_2(x: z_v, \sigma) \). Hence there exists, for every \( v \in u \), a context \( x: z_v, \sigma_v \in P(v, e_2, x: z_v, \sigma, k) \) by the induction hypothesis. Let \( u' = \{ z_v \mid v \in u \} \). Then \( u' \) contains at most \( k \) elements of \( V_{e_2(k)} \). Hence, \( u' \in V_m \) with \( m = \max(k, c_{e_2(k)}) \). Moreover, \( x: z_v, \sigma_v \subseteq x: z_v, \sigma \) by the induction hypothesis, so \( z_v \subseteq z_v \), and hence \( u' \subseteq e_1(\sigma) \).

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

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

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

\[
v \subseteq e_2(x: z''_v, \sigma'_v) \subseteq e_2(x: z''_v, \sigma').
\]

Since this holds for every \( v \in u \), we have \( u \subseteq e(\sigma') \).
If \( e = e_1 = e_2 \neq e_3 : e_4 \), 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 \( \sigma' \in \mathcal{C}_{e_3(k)} \subseteq \mathcal{C}_{e_2(k)} \), we have \( \sigma' \in P(u, e, \sigma, k) \). The case where \( e_1(\sigma) \neq e_2(\sigma) \) is similar. □

**Lemma 13.** For every PENRC expression \( e \) there exists a natural number \( k_e \), computable from \( e \), such that for every context \( \sigma \) 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 \( \sigma \) 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:

\[
\begin{align*}
k_e &:= 0 \\
k_{(e_1, e_2)} &:= \max\{k_{e_1}, k_{e_2}\} \\
k_{\pi_1(e')} &:= k_e' \\
k_{\pi_2(e')} &:= k_e' \\
k_\emptyset &:= 0 \\
k_{e'} &:= k_e' \\
k_{e_1 \cup e_2} &:= \max\{k_{e_1}, k_{e_2}\} \\
k_{\bigcup e'} &:= \max\{k_{e'}, c_e(1)\} \\
k_{e_1 = e_2 \ ? \ e_3 : e_4} (k) &:= \max\{k_{e_1}, k_{e_2}, k_{e_3}, k_{e_4}\}.
\end{align*}
\]

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 \( \sigma \) 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 \( \sigma' = \text{minimize}(\sigma) \). By monotonicity \( e'(\sigma') \) cannot be a pair and hence \( e(\sigma') \) is also undefined. Moreover, \( \sigma' \in \mathcal{C}_0 \subseteq \mathcal{C}_k \).
- 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 \( \sigma' = \text{minimize}(\sigma) \). By monotonicity \( e_1(\sigma) \) cannot be a set and hence \( e(\sigma') \) is undefined. Moreover, \( \sigma' \in \mathcal{C}_0 \subseteq \mathcal{C}_k \). 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 \( \sigma' = \text{minimize}(\sigma) \). By monotonicity, \( e'(\sigma') \) cannot be a set and hence \( e(\sigma) \) is undefined. Moreover, \( \sigma' \in \mathcal{C}_0 \subseteq \mathcal{C}_k \). If \( e'(\sigma) \) is a set, but not a set of sets, then there exists some \( u \in e'(\sigma) \) that is not a set. Then \( \{\text{minimize}(u)\} \in \mathcal{V}_1 \) and \( \{\text{minimize}(u)\} \subseteq e'(\sigma) \). By Proposition 12 there exists \( \sigma'' \in \mathcal{C}_{e'(1)} \subseteq \mathcal{C}_k \) with \( \sigma'' \sqsubseteq \sigma \) such that \( \{\text{minimize}(u)\} \subseteq e'(\sigma'') \). Hence, \( e'(\sigma'') \) is not a set of sets and \( e(\sigma'') \) is also undefined.
- 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 \( \sigma' = \text{minimize}(\sigma) \). By monotonicity, \( e_1(\sigma') \) cannot be a set and hence \( e(\sigma') \) is undefined. Moreover, \( \sigma' \in \mathcal{C}_0 \subseteq \mathcal{C}_k \).
3. Otherwise, \( e_1(\sigma) \) is defined and a set, but there is some \( v \in e_1(\sigma) \) such that \( e_2(\sigma) \) is undefined. By the induction hypothesis, there exists \( x : u, \sigma_2 \in \mathcal{C}_{e_2} \) with \( x : u, \sigma_2 \subseteq x : v, \sigma \) such that \( e_2(\sigma) \) is undefined. Then \( \{u\} \in \mathcal{V}_{\max\{1, k_0\}} \) and \( \{u\} \subseteq e_1(\sigma) \). By Proposition 12 there exists \( \sigma_1 \in \mathcal{C}_{e_1(\max\{1, k_0\})} \) with \( \sigma_1 \subseteq \sigma \) such that \( \{u\} \subseteq e_1(\sigma_1) \). Since both \( \sigma_1 \subseteq \sigma \) and \( \sigma_2 \subseteq \sigma \), \( \sigma_1 \cup \sigma_2 \) is defined by Lemma 11. Let \( \sigma' = \sigma_1 \cup \sigma_2 \). Note that
Lemma 11. By monotonicity \{u\} \subseteq e_1(\sigma') and \{\sigma_2\} \subseteq \sigma' by Lemma 11. By monotonicity \{u\} \subseteq e_1(\sigma'). Hence, there exists some \(u' \in e_1(\sigma')\) such that \(u \subseteq u'\). Then \(x : u, \sigma_2 \subseteq x : u', \sigma'\), and hence \(e_2(x : u', \sigma')\) is also undefined by monotonicity. Hence, \(e(\sigma')\) is undefined. Moreover, \(\sigma' \in C_{e_1(\max\{1,k_2\})+k_2} \subseteq C_{k_e}\).

- 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 \(\sigma' = \text{minimize}(\sigma)\). By monotonicity, \(e_1(\sigma')\) cannot be an atom and hence \(e(\sigma')\) is undefined. Moreover, \(\sigma' \in C_0 \subseteq C_{k_e}\). 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 \(\sigma' \in C_{k_e} \subseteq C_{k_e}\) such that \(\sigma' \subseteq \sigma\) and \(e(\sigma')\) 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. □

Proposition 6 now follows by Lemmas 13 and 9. Indeed, let \(e\) be a PENRC expression, let \(\Gamma\) be a type assignment on \(e\), and let \(\sigma \in \Gamma\) such that \(e(\sigma)\) is undefined. By Lemma 13 there is a natural number \(k_e\), computable from \(e\) alone, and \(\sigma' \subseteq \sigma\) of width at most \(k_e\) such that \(e(\sigma')\) is also undefined. Since \(\sigma \in \Gamma\), it follows that \(\sigma'\) is also in \(\Gamma\) 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 \(\pi_1\) and \(\pi_2\) 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.\(^1\)

Formally, we denote by PENRC(\(\text{extract}\)) the version of the PENRC to which we add \(\text{extract}\) as an expression:

\[e ::= \cdots | \text{extract}(e)\]

The semantics of \(\text{extract}\) is defined as follows:

\[
\begin{align*}
\sigma & \models e \Rightarrow \{v\} \\
\sigma & \models \text{extract}(e) \Rightarrow v
\end{align*}
\]

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

Although \(\text{extract}\) 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 \(\sigma \subseteq \sigma'\), then \(e(\sigma')\) is also undefined. For example, take \(e = \text{extract}(x)\), \(\sigma(x) = \{\{a\}, \{a, b\}\}\), and \(\sigma'(x) = \{\{a, b\}\}\). It is clear that \(e(\sigma)\) is undefined, but \(e(\sigma')\) is not. Note however that \(\{\{a\}, \{a, b\}\} \subseteq \{\{a, b\}\}\) 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(\(\text{extract}\)) 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 \(\Gamma\). We say that \(e_1\) and \(e_2\) are equivalent under \(\Gamma\) 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 \(\Gamma\), whether \(e_1\) and \(e_2\) are equivalent under \(\Gamma\).

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

\(^1\) We note that OQL, the object-oriented cousin of SQL, also has such a destructor, written \(\text{element}(e)\).
The equivalence problem for PENRC is undecidable.\footnote{We note that, in contrast, the containment problem for the PENRC (with regard to \(\subseteq\), not ordinary set-containment) in the absence of union is decidable \cite{12}.}

**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 \texttt{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, the tuple type \texttt{Tuple(Atom, Atom, Atom)} can be simulated by \texttt{Pair(Atom, Pair(Atom, 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 \cite{10}. This problem is defined as follows. Let \(x\) be a variable, let \(n\) be a natural number, and let \(\Gamma\) be the type assignment with domain \(\{x\}\) such that

\[
\Gamma(x) = \text{SetOf(Tuple(Atom, Atom, \ldots, Atom))}. \tag{\ref{eq:1000}}
\]

A functional dependency is a rule of the form \(X \rightarrow Y\) where \(X\) and \(Y\) are sequences over \([1, n]\). We say that a context \(\sigma \in \Gamma\) satisfies \(\sigma \models X \rightarrow Y\), denoted by \(\sigma \models X \rightarrow Y\), 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 \(\sigma \models X \subseteq Y\), 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 \(\Sigma\) be a finite set of functional and inclusion dependencies. We say that \(\sigma \in \Gamma\) satisfies \(\Sigma\), denoted by \(\sigma \models \Sigma\), if \(\sigma\) satisfies every dependency in \(\Sigma\). Let \(\rho\) be an additional target functional dependency. We say that \(\Sigma\) implies \(\rho\) if every context \(\sigma\) which satisfies \(\Sigma\) also satisfies \(\rho\). The implication problem for functional and inclusion dependencies consists of checking, given \(n, \Sigma,\) and \(\rho\), whether \(\Sigma\) implies \(\rho\). It is well-known that this problem is undecidable \cite{10}.

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

\[
\bigcup \left\{ \Pi_X(t_1) = \Pi_X(t_2) \land \Pi_Y(t_1) \neq \Pi_Y(t_2) \mid \{x\} : \emptyset \mid t_2 \in x \mid t_1 \in x \right\}.
\]

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

\[
\bigcup \left\{ \Pi_X(t_1) = \Pi_Y(t_2) \mid \emptyset \mid t_2 \in x \mid t_1 \in x \right\} \cup \left\{\{x\}\right\}.
\]

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 \(\Sigma\) and let \(\psi_1, \ldots, \psi_l\) be the inclusion dependencies in \(\Sigma\). By construction, \(\Sigma\) implies \(\rho\) 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

\[(r_1, \ldots, r_k, s_1, \ldots, s_l, \tau),\]
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 \( \Sigma \) implies \( \rho \) if, and only if, for every \( \sigma \in \Gamma \) there exists \( j \in [1, p] \) such that \( f_0(\sigma) = f_j(\sigma) \). Hence \( \Sigma \) implies \( \rho \) if, and only if,
\[
\left( \{f_0 \cup \{f_1\} \cup \cdots \cup \{f_p\}\}(\sigma) \right) = \left( \{f_1 \cup \cdots \cup \{f_p\}\}(\sigma) \right),
\]
for every \( \sigma \in \Gamma \). \( \square \)

**Corollary 16.** The well-definedness problem for PENRC\((\text{extract})\) is undecidable.

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
\[
x \in \text{Pair}(\text{Atom, Atom}) \ ? \ \{\pi_1(x)\} : \emptyset
\]
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:
\[
e := \cdots | e \in \tau \ ? \ e : e.
\]
Here, \( \tau \) ranges over types. We denote the resulting language by PENRC\((\text{type})\). The semantics of a type test is the obvious one:
\[
\begin{align*}
\sigma \models e_1 & \Rightarrow v_1 \quad v_1 \in \tau & \sigma \models e_2 & \Rightarrow v \\
\sigma \models e_1 \in \tau \ ? \ e_2 & : e_3 \Rightarrow v \\
\sigma \models e_1 & \Rightarrow v_1 \quad v_1 \not\in \tau & \sigma \models e_3 & \Rightarrow v \\
\sigma \models e_1 \in \tau \ ? \ e_2 & : e_3 \Rightarrow v
\end{align*}
\]

**Proposition 17.** The NRC is semantically contained in PENRC\((\text{type})\); 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:
\[
\{ (x, x) \mid x \in e_1 \} \in \text{SetOf}(\text{Atom}) \ ? \ e_2 : e_3.
\]
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\((\text{type})\) is undecidable.

**Corollary 18.** The well-definedness problem for PENRC\((\text{type})\) 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 \( \Gamma \) with \( \Gamma(R) = \text{SetOf}(\text{Pair}(\text{Atom, Atom})) \), but is undefined under the type assignment \( \Gamma' \) with
\[
\Gamma'(R) = \text{SetOf}(\text{Pair}(\text{Atom, Atom}) \cup \text{Atom}).
\]
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:
\[
e' = \bigcup \{x \in \text{Pair} \ ? \ \{\pi_1(x)\} : \emptyset \mid x \in R\}.
\]
It is clear that \( e' \) computes the same result as \( e \) on contexts \( \Gamma \) and that \( e' \) is well-defined under \( \Gamma' \). 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. \( \square \)
As a limited form of type tests, we propose the following. A kind is a term generated by the following grammar:

\[ \kappa ::= \text{Atom} \mid \text{Pair} \mid \text{Set} . \]

Here, \( \kappa \) 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:

\[ e ::= \cdots \mid e \in \kappa \mid e : e . \]

Here, \( \kappa \) ranges over kinds. We write PENRC\((kind)\) for the obtained language. The semantics of kind tests is the obvious one:

\[
\begin{align*}
\sigma \models e_1 \Rightarrow v_1 \quad v_1 \in \kappa & \quad \sigma \models e_2 \Rightarrow v \\
\sigma \models e_1 \in \kappa \?? e_2 \Rightarrow e_3 \Rightarrow v
\end{align*}
\]

Lemma 20. Let \( \kappa \) 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 \( \kappa \). 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 \( \sigma \) 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 \( \sigma \) 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 21.

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 \( c_e(k) \) 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 \( \sigma' \in \mathcal{C}_{c_e}(k) \subseteq \mathcal{C}_e(k) \), the result follows. The case where \( e_1(\sigma) \notin \kappa \) is similar. \( \square \)

Lemma 22. For every PENRC\((kind)\) expression \( e \) there exists a natural number \( k_e \), computable from \( e \), such that for every context \( \sigma \) 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:

\[ k_{e_1\in\kappa \?? e_2 : e_3}(k) := \max\{k_{e_1}, k_{e_2}, k_{e_3}\} . \]

It is clear that \( k_e \) remains computable from \( e \). Let \( \sigma \) 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 \( \sigma' \in \mathcal{C}_{c_2}(k) \subseteq \mathcal{C}_e(k) \) 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. \( \square \)
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 $\Gamma$ under which $e$ is well-defined, and an output type $\tau$, check that $e(\sigma) \in \tau$ for every $\sigma \in \Gamma$. If so, then we say that $e$ has output type $\tau$ under $\Gamma$.

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 $\text{SetOf}(\Gamma(x))$ under type assignment $\Gamma$ 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 $\tau$ is a type and $v \notin \tau$, then there exists a natural number $k$, computable from $\tau$, and a value $u \subseteq v$ of width at most $k$ such that $u \notin \tau$.

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

\[
c(\text{Atom}) := 0 \\
c(\text{Pair}(\tau_1, \tau_2)) := \max(c(\tau_1), c(\tau_2)) \\
c(\text{SetOf}(\tau')) := \max(1, c(\tau')) \\
c(\tau_1 \cup \tau_2) := c(\tau_1) + c(\tau_2).
\]

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

- If $\tau = \text{Atom}$, then take $u = \text{minimize}(v)$.
- If $\tau = \text{Pair}(\tau_1, \tau_2)$, then either
  1. $v$ is not a pair, in which case we take $u = \text{minimize}(v)$; 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 $\tau = \text{SetOf}(\tau')$, then either
  1. $v$ is not a set, in which case we take $u = \text{minimize}(v)$, or
  2. there exists some $v' \subseteq v$ such that $v' \notin \tau'$. By the induction hypothesis there exists $u' \in V_{c(\tau')}$ such that $u' \subseteq v'$ and $u' \notin \tau'$. Then $\{u'\} \subseteq v$ 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 $u_1 \subseteq v$ and $u_2 \subseteq v$ such that $u_1 \notin \tau_1$ and $u_2 \notin \tau_2$. Take $u = u_1 \cup u_2$ and suppose that $u \in \tau$. Then either $u \in \tau_1$ or $u \in \tau_2$. If $u \in \tau_1$, then also $u_1 \subseteq u$ would have to be in $\tau_1$ by Lemma 9, which is a contradiction. If $u \in \tau_2$, then also $u_2 \subseteq u$ would have to be in $\tau_2$, which is also a contradiction. Hence, $u \notin \tau$. Moreover, $u \in V_{c(\tau_1) + c(\tau_2)} = V_{c(\tau)}$. \qed

Corollary 26 (Small Model for Semantic Type-Checking). For every expression $e$ in PENRC(kind), every type assignment $\Gamma$ on $e$ and every type $\tau$ such that $e$ does not have output type $\tau$ under $\Gamma$, there exists a natural number $k$, computable from $e$ and $\tau$, 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 \( \tau \) under \( \Gamma \). Then there exists a context \( \sigma \in \Gamma \) such that \( e(\sigma) \not\in \tau \). There exists a natural number \( l \), computable from \( \tau \), and a value \( u \preceq e(\sigma) \) of width at most \( l \) such that \( u \not\in \tau \) by Lemma 25. By Proposition 21 there exists a computable function \( c_e \), computable from \( e \), and a context \( \sigma' \preceq \sigma \) of width at most \( k := c_e(l) \) such that \( u \preceq e(\sigma') \). Since \( u \not\in \tau \), \( e(\sigma') \) is also not in \( \tau \) 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.

Acknowledgement

The second author was supported by NSF Grant IIS-0082407. The last author is the Postdoctoral Fellow of the Research Foundation – Flanders (FWO – Vlaanderen).

References


