# Well-defined NRC queries can be typed

# Well-Defined NRC Queries Can Be Typed
## (extended abstract)

Jan Van den Bussche[1] and Stijn Vansummeren[2]

[1] Hasselt University and Transnational University of Limburg, Belgium
[2] Université Libre de Bruxelles, Belgium

**Abstract.** We study the expressive power of the static type system of the Nested Relational Calculus $\mathcal{NRC}$ and show that on so-called *homogeneous* input and output types, the $\mathcal{NRC}$ type system is *expressively* complete: every untyped but homogeneously well-defined $\mathcal{NRC}$ expression can be equivalently expressed by a well-typed expression. The $\mathcal{NRC}$ static type system hence does not limit the expressive power of the query writer.

*Dedicated to Peter Buneman.*

## 1 Introduction

Peter Buneman has been a longtime advocate of database query languages in the style of functional programming [3, 4, 9]. He has also repeatedly pointed out the relevance of union or variant types in the context of database applications [2,5,8]. Hence it seems fitting to contribute a paper on the expressive power of a typed first-order functional database query language, where we make a heavy use of union and variant types in our technical development.

Conventional wisdom states that programming errors should be caught as soon as possible, preferably at program development time. To this end, most programming languages come equipped with a static type system that accepts only "well-defined" programs that do not "crash" or "go wrong". Unfortunately, however, although decidable static type systems can prove the absence of crashes, they cannot prove their presence. For example, a program like

if <complex test> then <crash>

will be rejected as ill-typed even if <complex test> never terminates and the <crash> expression is never executed, as termination of programs is undecidable and hence cannot be statically checked.

For this reason, practical type systems only need to be sound (i.e., accept only well-defined programs), but not complete (i.e., accept exactly all well-defined programs). Note, however, that devising a type system that only needs to be sound is trivial. It suffices to let every expression be ill-typed, as soundness vacuously holds in the absence of well-typed programs. Of course, such a type

system is useless as it precludes the definition of *all* programs that can be expressed in a well-defined (but untyped) manner. Although real type systems are far from trivial, the question of their expressive power with regard to the class of well-defined programs remains interesting: are there well-defined programs that cannot be expressed by a well-typed program? If so, then the type system limits the expressiveness of the programmer, and more expressive type systems should be considered instead. If not, then we say that the type system is *expressively complete*.

In this paper, we study the expressive power of the static type system of the Nested Relational Calculus $\mathcal{NRC}$ [4]. The $\mathcal{NRC}$ is a database query language that provides a generalization and elegant abstraction of the familiar select-from-where SQL, OQL, and C$^\sharp$ queries. In earlier work [12], we have studied the decision problem of well-definedness for $\mathcal{NRC}$, obtaining that the problem is undecidable for $\mathcal{NRC}$ in general, but decidable for certain restricted fragments. These results motivate the need for an (incomplete) static type system for $\mathcal{NRC}$, such as the one proposed by Buneman et al. [4] during $\mathcal{NRC}$'s inception. Here we show that, on so-called *homogeneous* input and output types, the $\mathcal{NRC}$ type system is *expressively* complete. This hence confirms in the positive a conjecture made by the authors in [11].

Most type systems for imperative Turing complete programming languages are easily shown expressively complete: it suffices to show that one can simulate all Turing machine operations (including encoding and decoding of the programming language objects on Turing machine tapes) in a well-typed manner. Proving expressive completeness for the $\mathcal{NRC}$ type system, in contrast, is more difficult exactly because $\mathcal{NRC}$ is not Turing complete.

Interestingly enough, there are type systems for functional turing complete programming languages that are *not* expressively complete. For example, the untyped lambda calculus can define all computable functions, while in the simply typed lambda calculus only a restricted class of functions, the so-called *extended polynomials*, are definable [1,10]. Moreover, as shown by Kahrs, the type system of the Programmable language for Computable Functions PCF is expressively complete [7], while the type system of ML 1990 is not [6].

This paper is further organized as follows. In Section 2 we formally introduce the $\mathcal{NRC}$ and its static type system. We obtain our main result in Section 3. In this extended abstract, we will omit the detailed proof but we indicate the main steps toward the proof. It is anticipated that the full paper will appear in a scientific journal.

## 2 Preliminaries

From the outset we assume a non-empty set of atomic data constants (which in practice will include integers, strings, and so on). The $\mathcal{NRC}$ operates on *complex objects* $o$, which are nested combinations of atomic data constants $c$; records; and sets:

$$o ::= c \mid () \mid (o, o') \mid \{o, \ldots, o'\}.$$

Here, () is the empty record, and $(o, o')$ is the pair of objects formed by $o$ and $o'$. Note that larger-arity records like, e.g., $(o_1, o_2, o_3)$ can be simulated by nesting pairs as, e.g., $(o_1, (o_2, o_3))$.

The $\mathcal{NRC}$ expressions $e$ themselves are given by the syntax

$$\begin{aligned} e ::= {} & x \mid c \mid () \mid (e, e') \mid \pi_1(e) \mid \pi_2(e) \\ \mid {} & \{\} \mid \{e\} \mid e_1 \cup e_2 \mid \{e_2 \mid x \in e_1\} \mid \bigcup e \\ \mid {} & \mathsf{if}\ e_1 = e_2\ \mathsf{then}\ e_3\ \mathsf{else}\ e_4. \end{aligned}$$

(Parentheses may be used to avoid ambiguity.) Here, $x$ ranges over *variables* that can be bound to input objects; $c$ is data constant formation; () is empty record formation; $(e, e')$ is pair formation; $\pi_1(e)$ and $\pi_2(e)$ is left and right projection on pairs, repsectively; $\{\}$ and $\{e\}$ are empty and singleton set construction, respectively; $e_1 \cup e_2$ is set union; and $\{e_2 \mid x \in e_1\}$ is set comprehension. The set comprehension evaluates $e_2$ for every $x$ in the set returned by $e_1$. For example, $\{\pi_1(x) \mid x \in R\}$ returns the projection on the first component of the set of pairs $R$. The expression $\bigcup e$ flattens the set of sets $e$. Finally, $\mathsf{if}\ e_1 = e_2\ \mathsf{then}\ e_3\ \mathsf{else}$ is a conditional expression that evaluates $e_3$ if $e_1$ and $e_2$ evaluate to the same object, and evaluates $e_4$ otherwise.

It should be emphasized that the $x \in e_1$ part in the $\{e_2 \mid x \in e_1\}$ construct is *not* a membership test. It is an abstraction which introduces and binds the variable $x$, whose scope is the expression $e_2$. In light of this view, the *free variables* $FV(e)$ of an expression $e$ are hence inductively defined as follows: $FV(x) = \{x\}$, $FV(o) = \{\}$, $FV(\{e_2 \mid x \in e_1\} = FV(e_1) \cup (FV(e_2) - \{x\})$, and $FV(e)$ is the union of the free variables of $e$'s immediate subexpressions otherwise. We write $e(x, \ldots, y)$ to indicate that $e$ is an expression with $FV(e) \subseteq \{x, \ldots, y\}$. An expression without free variables is *closed*.

Some expressions, like $\pi_1(\{\mathsf{4}\})$ and $\mathsf{5} \cup \{\mathsf{6}\}$, clearly apply primitive operators to inappropriate objects and will therefore crash during evaluation. This intuition is formalized as follows. First, define an *environment* to be a mapping $\alpha$ that maps each variable $x$ to an object $\alpha(x)$. We use the notation $x/o, \alpha$ to stand for the environment that equals $\alpha$ on all variables except $x$, which it maps to $o$. Let $e[\alpha]$ denote the expression obtained from $e$ by replacing all free occurrences of $x$ by $\alpha(x)$, for every $x \in FV(e)$. Clearly, $e[\alpha]$ is fully determined by the free variables of $e$: if $\alpha$ and $\alpha'$ agree on $FV(e)$ then $e[\alpha] = e[\alpha']$. We may therefore write $e[x/o, \ldots, y/o']$ as a shorthand of the more verbose $e[x/o, \ldots, y/o', \alpha]$ when $FV(e) = \{x, \ldots, y\}$. Evaluation of $e(x, \ldots, y)$ on $o, \ldots, o'$ can then be seen as running the operational semantics of Figure 1 on $e[x/o, \ldots, y/o']$. There, we use the notation $e \to o$ to indicate that *closed* expression $e$ evaluates to object $o$. Evaluation *crashes* when there is no $o$ such that $e \to o$.

*Example 1.* Evaluation of the expression $\bigcup\{\{(\pi_1(y), z) \mid z \in x_2\} \mid y \in x_1\}$ with $x_1$ bound to $o_1 = \{(1, 2)\}$ and $x_2$ bound to $o_2 = \{3\}$ is successful:

$$\bigcup\{\{(\pi_1(y), z) \mid z \in \{3\}\} \mid y \in \{(1, 2)\}\} \to \{(1, 3)\}.$$

$$\frac{}{c \to c} \qquad \frac{}{() \to ()} \qquad \frac{e_1 \to o \quad e_2 \to o_2}{(e_1, e_2) \to (o_1, o_2')} \qquad \frac{e \to (o_1, o_2)}{\pi_1(e) \to o_1} \qquad \frac{e \to (o_1, o_2)}{\pi_2(e) \to o_2}$$

$$\frac{}{\{\} \to \{\}} \qquad \frac{e \to o}{\{e\} \to \{o\}} \qquad \frac{e_1 \to \{o_1, \ldots, o_m\} \quad e_2 \to \{o_1', \ldots, o_n'\}}{e_1 \cup e_2 \to \{o_1, \ldots, o_m, o_1', \ldots, o_n'\}}$$

$$\frac{e_1 \to \{o_1', \ldots, o_m'\} \quad e[x/o_i'] \to o_i \text{ for } 1 \leq i \leq m}{\{e \mid x_1 \in e_1\} \to \{o_1, \ldots, o_m\}}$$

$$\frac{e \to \{o_1, \ldots, o_m\} \text{ where each } o_i \text{ is a set}}{\bigcup e \to o_1 \cup \cdots \cup o_m}$$

$$\frac{e_1 \to o_1 \quad e_2 \to o_2 \quad o_1 = o_2 \quad e_3 \to o}{\mathsf{if}\ e_1 = e_2\ \mathsf{then}\ e_3\ \mathsf{else}\ e_4 \to o} \qquad \frac{e_1 \to o_1 \quad e_2 \to o_2 \quad o_1 \neq o_2 \quad e_4 \to o}{\mathsf{if}\ e_1 = e_2\ \mathsf{then}\ e_3\ \mathsf{else}\ e_4 \to o}$$

**Fig. 1.** The operational semantics of $\mathcal{NRC}$.

Evaluation of this expression with $x_1$ bound to $o_1' = (1, 2)$ instead of $o_1$ crashes, however, as no inference rule applies to $\{\{(\pi_1(y), z) \mid z \in \{3\}\} \mid y \in (1, 2)\}$.

Note that crashes only occur when (1) we apply projection to non-pairs, and (2) when we apply set union, comprehension, or flattening to non-sets.

We are interested in the crashing behavior of expressions when the inputs are taken from certain prescribed classes of objects. To this end, let the $\mathcal{NRC}$ *types* be given by the syntax

$$s, t ::= \mathsf{atom} \mid \mathsf{unit} \mid s \times t \mid \{s\} \mid s \vee t.$$

The semantics of a type is just a set of objects: $\mathsf{atom}$ is the set of all atomic data constants; $\mathsf{unit}$ is the type of the empty record $()$; $s \times t$ is the set of all pairs $(o, o')$ with $o$ of type $s$ and $o'$ of type $t$; $\{s\}$ is the set of all finite sets of objects of type $s$; and $s \vee t$ it is the set of all objects of type $s$ or of type $t$. We write $o\colon s$ to indicate that $o$ is an object of type $s$. Note that every object belongs to a type (in fact infinitely many). For example, $\{5, (1, 2)\}$ has type $\{\mathsf{atom} \vee (\mathsf{atom} \times \mathsf{atom})\}$ but also $\{\mathsf{atom} \vee (\mathsf{atom} \times \mathsf{atom}) \vee s\}$ for every $s$.

Types of the form $s \vee t$ are called *union types*. A type in which no union type occurs is called a *homogeneous type*. So, $\{\mathsf{atom}\}$ is a homogeneous type, but $\{\mathsf{atom} \vee (\mathsf{atom} \times \mathsf{atom})\}$ is not. An object is *homogeneous* if it has a homogeneous type. It is *heterogenous* otherwise. So, $\{5, 1, 2\}$ is a homogeneous set object (of homogeneous type $\{\mathsf{atom}\}$), but $\{5, (1, 2)\}$ is heterogenous.

**Definition 1.** *A type assignment is a mapping* T *that assigns a type* $\mathrm{T}(x)$ *to each variable* $x$*. An environment* $\alpha$ *is compatible with a type assignment* T*, written* $\alpha\colon \mathrm{T}$ *if* $\alpha(x)\colon \mathrm{T}(x)$*, for every* $x$*. An* $\mathcal{NRC}$ *expression* $e$ *is said to be* well-defined *under* T *if for every* $\alpha\colon \mathrm{T}$ *there exists* $o$ *with* $e[\alpha] \to o$*. We write* $\mathrm{T} \models e\colon t$ *to indicate that* $e$ *is well-defined under* T *and, moreover, every output* $o$ *of* $e$ *under all* $\alpha\colon \mathrm{T}$ *is of type* $t$*.*

$$\frac{}{\mathrm{T} \vdash x\colon \mathrm{T}(x)} \qquad \frac{}{\mathrm{T} \vdash c\colon \mathsf{atom}} \qquad \frac{}{\mathrm{T} \vdash ()\colon \mathsf{unit}}$$

$$\frac{\mathrm{T} \vdash e_1\colon s_1 \quad \mathrm{T} \vdash e_2\colon s_2}{\mathrm{T} \vdash (e_1, e_2)\colon s_1 \times s_2} \qquad \frac{\mathrm{T} \vdash e\colon s_1 \times s_2 \quad i = 1, 2}{\mathrm{T} \vdash \pi_i(e)\colon s_i}$$

$$\frac{}{\mathrm{T} \vdash \{\}\colon \{s\}} \qquad \frac{\mathrm{T} \vdash e\colon s}{\mathrm{T} \vdash \{e\}\colon \{s\}} \qquad \frac{\mathrm{T} \vdash e_1\colon \{s\} \quad \mathrm{T} \vdash e_2\colon \{s\}}{\mathrm{T} \vdash e_1 \cup e_2\colon \{s\}}$$

$$\frac{\mathrm{T} \vdash e_1\colon \{s\} \quad x\colon s, \mathrm{T} \vdash e_2\colon t}{\mathrm{T} \vdash \{e_2 \mid x \in e_1\}\colon \{t\}} \qquad \frac{\mathrm{T} \vdash e\colon \{\{s\}\}}{\mathrm{T} \vdash \bigcup e\colon \{s\}}$$

$$\frac{\mathrm{T} \vdash e_1\colon s \quad \mathrm{T} \vdash e_2\colon s \quad \mathrm{T} \vdash e_3\colon t \quad \mathrm{T} \vdash e_4\colon t}{\mathrm{T} \vdash \mathsf{if}\ e_1 = e_2\ \mathsf{then}\ e_3\ \mathsf{else}\ e_4\colon t}$$

**Fig. 2.** Static type system of $\mathcal{NRC}$.

Traditionally, the $\mathcal{NRC}$ is defined to operate only on homogeneous objects, and its type system (which we will define shortly) hence considers only homogeneous types. For the discussion that follows, however, it will be convenient to be able to assign a type also to heterogenous objects. Whence our inclusion of the union types.

The static type system for $\mathcal{NRC}$ is given in Fig. 2. There, the notation $x\colon s, \mathrm{T}$ stands for the type assignment that equals T on all variables except $x$, which it maps to $s$. As usual, the notation $\mathrm{T} \vdash e\colon s$ indicating that $e$ *has type* $s$ *under* T should be read as "assuming that the free variables $x$ of $e$ are bound to objects of type $\mathrm{T}(x)$, $e$ outputs objects of type $s$". Observe that this relation only depends on the free variables of an expression: if T and $\mathrm{T}'$ agree on $FV(e)$ and $\mathrm{T} \vdash e\colon s$, then also $\mathrm{T}' \vdash e\colon s$. We may therefore write $x\colon r, \ldots, y\colon s \vdash e\colon t$ as a shorthand of the more verbose $x\colon r, \ldots, y\colon s, T \vdash e\colon t$ when $FV(e) = \{x, \ldots, y\}$.

The obvious property one expects from a type system is *soundness*:

**Theorem 1.** *The static type system of Fig. 2 is sound. That is, if* $\mathrm{T} \vdash e\colon t$ *then* $\mathrm{T} \models e\colon t$.

Well-typedness hence implies well-definedness. The converse implication does not hold however, as the static type system rejects certain well-defined expressions. For example, $\{\pi_1(\{\}) \mid x \in \{\}\}$ is well-defined, but is not *well-typed* (i.e., there is no $s$ such that $\vdash e\colon s$). In the following Section we will show, however, that $e$ can equivalently be expressed by a well-typed $\mathcal{NRC}$ expression.

*A note on the $\mathcal{NRC}$ static type system.* As mentioned earlier, traditionally the type system of $\mathcal{NRC}$ does not include union types. Formally, this means that, traditionally, in Fig. 2 the meta-variables $s$ and $t$ are restricted to range over homogeneous types, and T is restricted to *homogeneous type assignments* (i.e.,

a mapping from variables to homogeneous types). We do not require this restriction, and are hence able to derive, for example $x\colon \{r_1 \vee r_2\}, y\colon \{r_1 \vee r_2\} \vdash x \cup y\colon \{r_1 \vee r_2\}$.

Still, our setting treats union types in a conservative way, in the following sense.

**Proposition 1.** *Let* $\mathrm{T}$ *be a homogeneous type assignment (i.e., a mapping from variables to homogeneous types) and let* $e$ *be an* $\mathcal{NRC}$ *expression. If* $\mathrm{T} \vdash e\colon t$ *then* $t$ *is also a homogeneous type.*

This implies that on homogeneous type assignments, the type system of Fig. 2 hence coincides with the traditional one.

## 3 Completeness

Our goal in this section is to obtain the following result. Let $e \equiv^{\mathrm{T}} f$ denote that $e$ and $f$ are *equivalent* on inputs of type $\mathrm{T}$. That is, for every $\alpha\colon \mathrm{T}$, either $e[\alpha]$ and $f[\alpha]$ both crash or they evaluate to the same object $o$.

**Theorem 2.** *Homogeneous well-defined* $\mathcal{NRC}$ *expressions can be expressed in a well-typed way, in the following sense. For every* $\mathcal{NRC}$ *expression* $e$*, every* homogeneous *type assignment* $\mathrm{T}$ *and every* homogeneous *type* $t$ *such that* $\mathrm{T} \models e\colon t$ *there exists an* $\mathcal{NRC}$ *expression* $h$ *such that (1)* $\mathrm{T} \vdash h\colon t$ *and (2)* $e \equiv^{\mathrm{T}} h$*.*

The proof is effective and allows us to transform $e$ into $h$, given $\mathrm{T}$ and $t$. Intuitively, there are two problems to overcome. The first problem is that well-defined expressions may contain ill-defined subexpressions. For example, $e = \{\pi_1(\{\}) \mid x \in \{\}\}$ is well-defined, but its subexpression $e' = \pi_1(\{\})$ is not. Of course, $e'$ is "dead code" (it is never executed) and we can therefore alternatively express $e$ by $\{\bot_t \mid x \in \{\}\}$ where $\bot_t$ is an arbitrary constant object of type $t$, the desired output type. Our transformation of $e$ into $f$ will therefore need to detect dead subexpressions, and replace them by harmless constants. The second problem is that even when evaluated on homogeneous inputs, well-defined expressions may manipulate heterogeneous objects while well-typed expressions cannot. For example, $e = \{\pi_1(z) \mid z \in (x \cup y)\}$, is well-defined under $x\colon \{s \times s\}, y\colon \{s \times t)\}$ with $s$ and $t$ are two different types. It is not well-typed under this type assignment, however, as the type rule for $x \cup y$ requires $x$ and $y$ to have the same set type. Nevertheless, the same query is expressed by $e' = \{\pi_1(z) \mid z \in x\} \cup \{\pi_1(x) \mid z \in y\}$, which is well-typed. In general, we will deal with this problem by simulating heterogeneous objects by homogeneous ones.

The proof of Theorem 2 is divided into three steps as follows, where step 1 and 2 deal with the first problem, and step 2 and 3 deal with the second problem.

1. First, we show that homogeneous well-defined $\mathcal{NRC}$ expressions can be defined in a well-typed way in $\mathcal{NRC}(\mathit{cast})$, an extension of $\mathcal{NRC}$ with a typecast operator.

2. Next, we show that well-typed $\mathcal{NRC}(\mathit{cast})$ can be simulated in well-typed $\mathcal{NRC}^+$, a variant of the $\mathcal{NRC}$ in which we disallow union types but add *sum types* (also known as *variant types*) instead. In particular, since $\mathcal{NRC}^+$ does not have a typecasting operator, we show that casts can be simulated on sum types.
3. We are thus left with well-typed $\mathcal{NRC}^+$ expressions with homogeneous input and output types, which are already known to be expressible in a well-typed manner in $\mathcal{NRC}$ itself [13].

We now develop each of these steps in turn.

### 3.1 Adding casts

Let $\mathcal{NRC}(\mathit{cast})$ be the extension of $\mathcal{NRC}$ with expressions of the form $\langle s \rangle\, e$, for every type $s$:

$$e ::= \cdots \mid \langle s \rangle\, e.$$

Semantically, $\langle s \rangle\, e$ returns the same object as $e$ if that object is of type $s$, otherwise it returns an arbitrary (but fixed) object $\bot_s$ of type $s$.

$$\frac{e \to o \quad o\colon s}{\langle s \rangle\, e \to o} \qquad \frac{e \to o \quad \neg(o\colon s)}{\langle s \rangle\, e \to \bot_s}$$

Note that the output of $\langle s \rangle\, e$ is always of type $s$. We therefore add the following type rule to the typesystem of $\mathcal{NRC}$:

$$\frac{\mathrm{T} \vdash e\colon t}{\mathrm{T} \vdash \langle s \rangle\, e\colon s}$$

It is easy to see that with this addition the typesystem of $\mathcal{NRC}(\mathit{cast})$ is sound.

The following proposition shows that the typecast operator allows us to extend *any* expression into a well-typed (and therefore, well-defined) expression. Intuitively, this is because we can always typecast subexpressions that do not meet the type rule constraints of Fig. 2 into a type that does meet these constraints. Consider, for example, that we are type-checking $\pi_1(e)$ and suppose that we have already derived $\mathrm{T} \vdash e\colon t$ with $t = \{\mathsf{unit}\} \vee (s_1 \times s_2)$. Then clearly, $\pi_1(e)$ cannot be well-typed under T since the type rule for $\pi_1$ requires $e$ to have a pair type. Suppose, however, that we know that $\pi_1(e)$ is well-defined under T. Then clearly, although the type system derives $\{\mathsf{unit}\} \vee (s_1 \times s_2)$ as the output type of $e$, we know that $e$ can never output an object of type $\{\mathsf{unit}\}$, otherwise $\pi_1(e)$ would crash. Hence, $\pi_1(e)$ can equivalently be expressed on T by $\pi_1(\langle s_1 \times s_2 \rangle\, e)$, which is well-typed under T.

Similarly, suppose that we have derived $\mathrm{T} \vdash e\colon t$ with $t = (s_1 \times s_2) \vee (s_1' \times s_2')$. Again, $\pi_1(e)$ is not well-typed under T since the type rule for $\pi_1$ requires $e$ to have a pair type $t_1 \times t_2$ instead of a union type. In this case, however, it suffices to recognize that all objects of type $(s_1 \times s_2) \vee (s_1' \times s_2')$ also have type $t' := (s_1 \vee s_1') \times (s_2 \vee s_2')$. Hence, $\pi_1(e)$ can equivalently be expressed on T by $\pi_1(\langle t' \rangle\, e)$, which is well-typed under T.

These two simple ideas form the basis of the following proposition.

**Proposition 2.** *For every $\mathcal{NRC}$ expression $e$ and every type assignment $\mathrm{T}$ there exists an expression $f^{e,\mathrm{T}} \in \mathcal{NRC}(cast)$ and type $t^{e,\mathrm{T}}$ such that*

*(a)* $\mathrm{T} \vdash f^{e,\mathrm{T}} : t^{e,\mathrm{T}}$*; and*
*(b) for every* $\alpha : \mathrm{T}$*, if* $e[\alpha] \to o$ *then* $f^{e,\mathrm{T}}[\alpha] \to o$*.*

**Corollary 1.** *For every $\mathcal{NRC}$ expression $e$, every type assignment $\mathrm{T}$, and every type $t$ with $\mathrm{T} \models e : t$ there exists a well-typed $\mathcal{NRC}(cast)$ expression $\mathrm{T} \vdash f : t$ such that then $e \equiv^{\mathrm{T}} f$.*

### 3.2 Simulating union types and casts

In this subsection we explain how the union types and casts of $\mathcal{NRC}(cast)$ can be simulated in $\mathcal{NRC}^+$, a variant of $\mathcal{NRC}$ in which we disallow union types but add *sum types* (also known as *variant types*) instead. Note that $\mathcal{NRC}^+$ does not have a typecasting operator.

$\mathcal{NRC}^+$ extends $\mathcal{NRC}$ on three levels: on the level of objects themselves, on the level of types, and on the level of expressions. On the level of objects, $\mathcal{NRC}^+$ adjoins the atomic, records, and set objects of $\mathcal{NRC}$ with *tagged* objects of the form *left* $o$ and *right* $o$:

$$o ::= c \mid () \mid (o, o') \mid \{o, \ldots, o'\} \mid \mathit{left}\ o \mid \mathit{right}\ o.$$

One can see tagged objects as objects paired with either the label *left* or the label *right*.

On the level of types, $\mathcal{NRC}^+$ adjoins the atomic, record, and set types of $\mathcal{NRC}$ with sum types, as given by the syntax:

$$\sigma, \tau ::= \mathsf{atom} \mid \mathsf{unit} \mid \sigma \times \tau \mid \{\sigma\} \mid \sigma{+}\tau.$$

Note that every homogeneous type $s$ as defined in Section 2 is syntactically also an $\mathcal{NRC}^+$ type. Like $\mathcal{NRC}$ types, the semantics of a $\mathcal{NRC}^+$ type is just a set of objects: $\mathsf{atom}$ is the set of all atomic data constants; $\mathsf{unit}$ is the type of the empty record $()$; $\sigma \times \tau$ is the set of all pairs $(o, o')$ with $o$ of type $\sigma$ and $o'$ of type $\tau$; $\{\sigma\}$ is the set of all finite sets of objects of type $\sigma$; and $\sigma{+}\tau$ is the set of all objects *left* $o$ and *right* $o$ with $o$ of type $\sigma$ and $\tau$, respectively.

On the level of expressions $\mathcal{NRC}^+$ extends $\mathcal{NRC}$ with two tagged object assembly operations, and one dissembly operation:

$$e ::= \ldots \mid \mathtt{left}^{\sigma,\tau}\, e \mid \mathtt{right}^{\sigma,\tau}\, e \mid \mathsf{when}\ e_1\ \mathsf{is\ left}\, x\ \mathsf{do}\ e_2\ \mathsf{or\ right}\, y\ \mathsf{do}\ e_3$$

where $\sigma$ and $\tau$ range over $\mathcal{NRC}^+$ types. Intuitively, applying assembly operation $\mathtt{left}^{\sigma,\tau}$ to object $o$ adds the *left* label to $o$, returning *left* $o$. $\mathtt{right}^{\sigma,\tau}$ works similarly. The dissembly operation $\mathsf{when}\ e_1\ \mathsf{is\ left}\, x\ \mathsf{do}\ e_2\ \mathsf{or\ right}\, y\ \mathsf{do}\ e_3$ first inspects the result of $e_1$. If this is a tagged object *left* $o$ then it evaluates $e_2$ with $x$ bound to $o$. If this is a tagged object *right* $o$ then it evaluates $e_3$ with $y$ bound to $o$. The free variables of $\mathtt{left}^{\sigma,\tau}\, e$ and $\mathtt{right}^{\sigma,\tau}\, e$ are hence simply the free variables of $e$. In contrast, the free variables of $\mathsf{when}\ e_1\ \mathsf{is\ left}\, x\ \mathsf{do}\ e_2\ \mathsf{or\ right}\, y\ \mathsf{do}\ e_3$ is $FV(e_1) \cup (FV(e_2) - \{x\}) \cup (FV(e_3) - \{y\})$.

$\mathcal{NRC}^+$ OPERATIONAL SEMANTICS.

$$\frac{e \to o}{\texttt{left}^{\sigma,\tau}\ e \to \mathit{left}\ o} \qquad \frac{e \to o}{\texttt{right}^{\sigma,\tau}\ e \to \mathit{right}\ o}$$

$$\frac{e_1 \to \mathit{left}\ o_1 \quad e_2[x/o_1] \to o}{\mathsf{when}\ e_1\ \mathsf{is\ left}\ x\ \mathsf{do}\ e_2\ \mathsf{or\ right}\ y\ \mathsf{do}\ e_3 \to o}$$

$$\frac{e_1 \to \mathit{right}\ o_1 \quad e_3[y/o_1] \to o}{\mathsf{when}\ e_1\ \mathsf{is\ left}\ x\ \mathsf{do}\ e_2\ \mathsf{or\ right}\ y\ \mathsf{do}\ e_3 \to o}$$

$\mathcal{NRC}^+$ STATIC TYPE SYSTEM.

$$\frac{}{\mathrm{T} \vdash^+ x\colon \mathrm{T}(x)} \qquad \frac{}{\mathrm{T} \vdash^+ c\colon \mathsf{atom}} \qquad \frac{}{\mathrm{T} \vdash^+ ()\colon \mathsf{unit}}$$

$$\frac{\mathrm{T} \vdash^+ e_1\colon \sigma_1 \quad \mathrm{T} \vdash^+ e_2\colon \sigma_2}{\mathrm{T} \vdash^+ (e_1, e_2)\colon \sigma_1 \times \sigma_2} \qquad \frac{\mathrm{T} \vdash^+ e\colon \sigma_1 \times \sigma_2 \quad i = 1, 2}{\mathrm{T} \vdash^+ \pi_i(e)\colon \sigma_i}$$

$$\frac{}{\mathrm{T} \vdash^+ \{\}\colon \{\sigma\}} \qquad \frac{\mathrm{T} \vdash^+ e\colon s}{\mathrm{T} \vdash^+ \{e\}\colon \{\sigma\}} \qquad \frac{\mathrm{T} \vdash^+ e_1\colon \{\sigma\} \quad \mathrm{T} \vdash^+ e_2\colon \{s\}}{\mathrm{T} \vdash^+ e_1 \cup e_2\colon \{\sigma\}}$$

$$\frac{\mathrm{T} \vdash^+ e_1\colon \{\sigma\} \quad x\colon \sigma, \mathrm{T} \vdash^+ e_2\colon \tau}{\mathrm{T} \vdash^+ \{e_2 \mid x \in e_1\}\colon \{\tau\}} \qquad \frac{\mathrm{T} \vdash^+ e\colon \{\{\sigma\}\}}{\mathrm{T} \vdash^+ \bigcup e\colon \{\sigma\}}$$

$$\frac{\mathrm{T} \vdash^+ e_1\colon \sigma \quad \mathrm{T} \vdash^+ e_2\colon \sigma \quad \mathrm{T} \vdash^+ e_3\colon \tau \quad \mathrm{T} \vdash^+ e_4\colon \tau}{\mathrm{T} \vdash^+ \mathsf{if}\ e_1 = e_2\ \mathsf{then}\ e_3\ \mathsf{else}\ e_4\colon \tau}$$

$$\frac{\mathrm{T} \vdash^+ e\colon \sigma}{\mathrm{T} \vdash^+ \texttt{left}^{\sigma,\tau}\ e\colon \sigma{+}\tau} \qquad \frac{\mathrm{T} \vdash^+ e\colon \tau}{\mathrm{T} \vdash^+ \texttt{right}^{\sigma,\tau}\ e\colon \sigma{+}\tau}$$

$$\frac{\mathrm{T} \vdash^+ e_1\colon \sigma_1{+}\sigma_2 \quad x\colon \sigma_1, \mathrm{T} \vdash^+ e_1\colon \tau \quad y\colon \sigma_2, \mathrm{T} \vdash^+ e_2\colon \tau}{\mathrm{T} \vdash^+ \mathsf{when}\ e_1\ \mathsf{is\ left}\ x\ \mathsf{do}\ e_2\ \mathsf{or\ right}\ y\ \mathsf{do}\ e_3\colon \tau}$$

**Fig. 3.** The operational semantics and type system of $\mathcal{NRC}^+$.

In the type system, $\texttt{left}^{\sigma,\tau}\ e$ will have type $\sigma{+}\tau$, provided that $e$ has type $\sigma$. Similarly, $\texttt{right}^{\sigma,\tau}\ e$ has type $\sigma{+}\tau$ provided that $e$ has type $\tau$. Finally, $\mathsf{when}\ e_1\ \mathsf{is\ left}\ x\ \mathsf{do}\ e_2\ \mathsf{or\ right}\ y\ \mathsf{do}\ e_3$ has type $\tau$ provided that $e_1$ has type $\sigma_1{+}\sigma_2$, and $e_2$ and $e_3$ both have type $\tau$ under the assumption that $x\colon \sigma_1$ and $y\colon \sigma_2$, respectively.

The formal evaluation rules, as well as the rules of the $\mathcal{NRC}^+$ static type system are given in Fig. 3. There, we use the notation $\mathrm{T} \vdash^+ e\colon \tau$ that $\mathcal{NRC}^+$ expression $e$ has $\mathcal{NRC}^+$ type $\tau$ under $\mathcal{NRC}^+$ type assignment T. It is straightforward to show that this type system is sound.

*Example 2.* The expression

$$e = \mathsf{when}\ z\ \mathsf{is\ left}\ x\ \mathsf{do}\ \pi_1(x)\ \mathsf{or\ right}\ y\ \mathsf{do}\ \bigcup y$$

is well-typed under the type assignment $z\colon (\{\mathsf{atom}\} \times \mathsf{atom})+\{\{\mathsf{atom}\}\}$. Intuitively, this type assignment indicates that $z$ can take values that are either of type $\{\mathsf{atom}\} \times \mathsf{atom}$ or of type $\{\{\mathsf{atom}\}\}$. The expression evaluates $\pi_1$ on its input object $z$ if that object is of pair type $\{\mathsf{atom}\} \times \mathsf{atom}$. (To be precise, it evaluates $\pi_1$ on $o$ when $z$ is of the form *left* $o$). It evaluates $\bigcup$ on its input object if it is of type $\{\{\mathsf{atom}\}\}$ (i.e., $z$ is of the form *right* $o$).

As this example illustrates, one can see sum types as a (better-behaved) variant of union types. The crucial difference between union types and sum types lies in the fact that, by means of the *left* and *right* labels, objects of a sum type carry runtime type information, whereas objects of a union type (not having labels) do not. Indeed, an expression similar to $e$ in the example above can intuitively not be defined in a well-defined manner in $\mathcal{NRC}(\mathit{cast})$ under the type assignment $z\colon (\{\mathsf{atom}\} \times \mathsf{atom}) \vee \{\{\mathsf{atom}\}\}$ since we have no means in $\mathcal{NRC}(\mathit{cast})$ to inspect whether $z$ is of type $\{\mathsf{atom}\} \times \mathsf{atom}$ or $\{\{\mathsf{atom}\}\}$. We will exploit the fact that $\mathcal{NRC}^+$ has this form of runtime type information to show that well-typed $\mathcal{NRC}(\mathit{cast})$ can be simulated in well-typed $\mathcal{NRC}^+$. The simulation is based on the following encoding.

*The encoding* We will use $\mathcal{NRC}^+$'s sum types to simulate $\mathcal{NRC}$'s union types by means of the following one-to-one correspondence between the syntax of sum types and union types. Let $s^+$ be the $\mathcal{NRC}^+$ type obtained by recursively replacing every union type $t_1 \vee t_2$ in $\mathcal{NRC}$ type $s$ by ${t_1}^+ + {t_2}^+$.

$$\mathsf{atom}^+ = \mathsf{atom} \qquad \mathsf{unit}^+ = \mathsf{unit} \qquad (s \times t)^+ = s^+ \times t^+$$
$$\{s\}^+ = \{s^+\} \qquad (s \vee t)^+ = s^+ + t^+$$

Similarly, let $\overline{\sigma}$ be the $\mathcal{NRC}$ type obtained by recursively replacing every sum type $\tau_1+\tau_2$ occurring in $\mathcal{NRC}^+$ type $\sigma$ by the union type $\overline{\tau_1} \vee \overline{\tau_2}$.

$$\mathsf{atom}^+ = \mathsf{atom} \qquad \mathsf{unit}^+ = \mathsf{unit} \qquad (s \times t)^+ = s^+ \times t^+$$
$$\{s\}^+ = \{s^+\} \qquad (s \vee t)^+ = s^+ + t^+$$

Clearly, $\overline{s^+} = s$ and $\overline{\sigma}^+ = \sigma$. Moreover, if $s$ is a homogeneous type, then $s^+ = \overline{s} = s$. We extend these operations pointwise to type assignments and write, for example, $\mathrm{T}^+$ for $\mathcal{NRC}^+$ type assignment that maps $x \mapsto \mathrm{T}(x)^+$, for every $x$. The type assignment $\overline{\mathrm{T}}$ is defined similarly.

Finally, let the erasure $\overline{o}$ of $\mathcal{NRC}^+$ object $o$ be the object obtained by recursively replacing all subobjects of the form *left* $o'$ and *right* $o'$ in $o$ by $\overline{o'}$.

$$\overline{c} := c \qquad \overline{()} = () \qquad \overline{(o_1, o_2)} = (\overline{o_1}, \overline{o_2})$$
$$\overline{\{o, \ldots, o'\}} := \{\overline{o}, \ldots, \overline{o'}\} \qquad \overline{\mathit{left}\ o} := \overline{o} \qquad \overline{\mathit{right}\ o} := \overline{o}$$

Note that if $o$ is a homogeneous object, then $\overline{o} = o$. We extend erasure pointwise to $\mathcal{NRC}^+$ environments, and write $\overline{\alpha}$ for the environment with domain $\mathit{dom}(\alpha)$ that maps $x \mapsto \overline{\alpha(x)}$, for every $x \in \mathit{dom}(\alpha)$.

**Definition 2.** *Let $s$ be an $\mathcal{NRC}$ type. We say that $u\colon s^+$ is an* encoding *of $o\colon s$ with respect to $s$ if $\overline{u} = o$.*

It is easy to see that for every $\mathcal{NRC}$ type $s$ and every $o\colon s$ there is at least one encoding. Hence, for every $\alpha\colon \mathrm{T}$ there always an environment $\alpha'\colon \mathrm{T}^+$ encoding $\alpha$ (i.e., $\overline{\alpha'} = \alpha$).

**Lemma 1.** *Type casts can be simulated in $\mathcal{NRC}^+$. That is, for all $\mathcal{NRC}$ types $s$ and $t$ there exists an $\mathcal{NRC}^+$ expression $cast^{s,t}(x)$ such that:*

*(a)* $x\colon s^+ \vdash^+ cast^{s,t}(x)\colon t^+$*; and*
*(b)* $cast^{s,t}[x/o] \to o'$ *implies* $\langle t\rangle\, \overline{o} \to \overline{o'}$*, for every* $o\colon s^+$.

Let us illustrate the proof idea by means of the following example.

*Example 3.* Let $\mathsf{atom}^n$ with $n \geq 1$ stand for the type of $n$-ary tuples of atomic data constants: $\mathsf{atom}^1 = \mathsf{atom}$; $\mathsf{atom}^2 = \mathsf{atom} \times \mathsf{atom}$; $\mathsf{atom}^3 = \mathsf{atom} \times \mathsf{atom}^2$; and so on. Let $s = \mathsf{atom}^2 \vee \mathsf{atom}^3$ and $t = \mathsf{atom}^3 \vee \mathsf{atom}^4$. Suppose that $\bot_t = (c, (c, (c, c)))$. Then $cast^{s,t}(x)$ is given by

$$\mathsf{when}\ x\ \mathsf{is\ left}\ y\ \mathsf{do}\ \mathtt{right}^{\mathsf{atom}^3,\mathsf{atom}^4}(c, (c, (c, c)))\ \mathsf{or\ right}\ z\ \mathsf{do}\ \mathtt{left}^{\mathsf{atom}^3,\mathsf{atom}^4}(z).$$

**Lemma 2.** *For all $\mathcal{NRC}$ types $s$ and $t$ there exists an expression $eq^{s,t}(x, y)$ in $\mathcal{NRC}^+$ that checks equality modulo encodings:*

*(a)* $x\colon s^+, y\colon t^+ \vdash^+ eq^{s,t}(x, y)\colon \{\mathsf{unit}\}$*; and*
*(b)* $eq^{s,t}[x/o_x, y/o_y] \to \{()\}$ *iff* $\overline{o_x} = \overline{o_y}$*, for every* $o_x\colon s^+, o_y\colon t^+$.

Let us illustrate the proof idea by means of the following example.

*Example 4.* Using the notation of Example 3, let $s = \mathsf{atom}^2 \vee \mathsf{atom}^3$ and $t = \mathsf{atom}^3 \vee \mathsf{atom}^4$. Then $eq^{s,t}(x, y)$ is given by

$$\begin{array}{l}\mathsf{when}\ x\ \mathsf{is\ left}\ u\ \mathsf{do}\ \{\}\\ \mathsf{or\ right}\ u\ \mathsf{do}\\ \quad \mathsf{when}\ y\ \mathsf{is\ left}\ v\ \mathsf{do}\ (\mathsf{if}\ u = v\ \mathsf{then}\ \{()\}\ \mathsf{else}\ \{\})\\ \quad \mathsf{or\ right}\ v\ \mathsf{do}\ \{\}\end{array}$$

**Proposition 3.** *For every $\mathcal{NRC}(cast)$ expression $e$, type assignment $\mathrm{T}$ and type $t$ with $\mathrm{T} \vdash e\colon t$ there exists $\mathcal{NRC}^+$ expression $e^+$ such that:*

*(a)* $\mathrm{T}^+ \vdash^+ e^+\colon t^+$*; and*
*(b)* $e^+[\alpha] \to o$ *iff* $e[\overline{\alpha}] \to \overline{o}$*, for every* $\alpha\colon \mathrm{T}^+$.

**Corollary 2.** *For every $\mathcal{NRC}(cast)$ expression $e$, every homogeneous type assignment $\mathrm{T}$, and every homogeneous type $t$ with $\mathrm{T} \vdash e\colon t$ there exists a well-typed $\mathcal{NRC}^+$ expression $\mathrm{T} \vdash^+ e^+\colon t$ such that $e^+ \equiv^{\mathrm{T}} e$.*

### 3.3 Removing sum types

To finalize the proof, we recall the following result by Wong [13, Corollary 2.3.5].

**Proposition 4 (Wong [13]).** *For every $\mathcal{NRC}^+$ expression $e$, every homogeneous type assignment $\mathrm{T}$ and every homogeneous type $t$ with $\mathrm{T} \vdash^+ e : t$ there exists a well typed $\mathcal{NRC}$ expression $\mathrm{T} \vdash f : t$ such that $e \equiv^{\mathrm{T}} f$.*

We hence obtain the following proof of Theorem 2.

*Proof (of Theorem 2).* Let $e$ be an $\mathcal{NRC}$ expression; let T be a homogeneous type assignment; and let $t$ be a homogeneous type such that $\mathrm{T} \models e : t$. By Corollary 1 there exists $\mathcal{NRC}(\mathit{cast})$ expression $f \equiv^{\mathrm{T}} e$ with $\mathrm{T} \vdash f : t$. By Corollary 2 there exists $\mathcal{NRC}^+$ expression $g \equiv^{\mathrm{T}} f$ with $\mathrm{T} \vdash g : t$. Then, by Proposition 4 there exists $\mathcal{NRC}$ expression $h \equiv^{\mathrm{T}} f$ such that $\mathrm{T} \vdash f : t$, as desired. □

## 4 Discussion

One may wonder whether Theorem 2 can be strengthened to the case where $e$ is well-typed under a *heterogeneous* type assignment T with output in a *heterogeneous* type $t$. It turns out that the static type system of Fig. 2 is too weak for this purpose since it never introduces or manipulates union types; it merely propagates them from the input type assignment to output type. Indeed, the following proposition is straightforward to obtain by induction on $e$.

**Proposition 5.** *Let $\mathrm{T} \vdash e : t$. Then every union type $t_1 \vee t_2$ that occurs in $t$ also occurs in $\mathrm{T}(x)$ for some $x \in FV(e)$.*

Hence, we cannot find a well-typed equivalent of the well-defined $x : \{s\}, y : \{t\} \models x \cup y : \{s \vee t\}$ when $s$ and $t$ are distinct types.

Alternatively, can Theorem 2 be strengthened to the case where $e$ is well-typed under a *heterogeneous* type assignment T with output in a *homogeneous* type $t$? Proposition 5 does not exclude this possibility. We conjecture, however, that it is also not possible to strengthen Theorem 2 in this sense.

*Conjecture 1.* There exists a heterogeneous type assignment T, $\mathcal{NRC}$ expression $e$ and homogeneous type $t$ with $\mathrm{T} \models e : t$ that cannot be equivalently expressed by a well-typed $\mathcal{NRC}$ expression.

## References

[1] H. Barendregt. *The Lambda Calculus: its Syntax and Semantics.* North-Holland, 1984.
[2] P. Buneman, S. Davidson, and A. Watters. A semantics for complex objects and approximate answers. *J. Comput. Syst. Sci.*, 43(1):170–218, 1991.
[3] P. Buneman, R. Frankel, and R. Nikhil. An implementation technique for database query languages. *ACM Trans. Database Syst.*, 7(2):164–186, 1982.

[4] P. Buneman, S. A. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theor. Comput. Sci.*, 149(1):3–48, 1995.

[5] P. Buneman and B. Pierce. Union types for semistructured data. In R. Connor and A. Mendelzon, editors, *Research Issues in Structured and Semistructured Database Programming*, volume 1949 of *Lecture Notes in Computer Science*, pages 184–207. Springer, 2000.

[6] S. Kahrs. Limits of ML-definability. In *Programming Languages: Implementations, Logics, and Programs, 8th International Symposium, PLILP'96, Aachen, Germany, September 24-27, 1996, Proceedings*, volume 1140 of *Lecture Notes in Computer Science*, pages 17–31. Springer, 1996.

[7] S. Kahrs. Well-going programs can be typed. In *Typed Lambda Calculi and Applications, 6th International Conference, TLCA 2003, Valencia, Spain, June 10-12, 2003, Proceedings*, volume 2701 of *Lecture Notes in Computer Science*, pages 167–179. Springer, 2003.

[8] A. Ohori and P. Buneman. Polymorphism and type inference in database programming. *ACM Trans. Database Syst.*, 21(1):30–76, 1996.

[9] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database programming in Machiavelli—a polymorphic language with static type inference. In J. Clifford, B. Lindsay, and D. Maier, editors, *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, volume 18:2 of *SIGMOD Record*, pages 46–57. ACM Press, 1989.

[10] H. Schwichtenberg. Definierbare funktionen in $\lambda$-kalkül mit typen. *Archiv für mathematische Logik und Grundlagenforschung*, 174:113–114, 1976.

[11] J. Van den Bussche, D. Van Gucht, and S. Vansummeren. A crash course on database queries. In *Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 143–154. ACM, 2007.

[12] J. Van den Bussche, D. Van Gucht, and S. Vansummeren. Well-definedness and semantic type-checking for the nested relational calculus. *Theor. Comput. Sci.*, 371(3):183–199, 2007.

[13] L. Wong. *Querying Nested Collections.* PhD thesis, University of Pennsylvania, 1994.