Types for Database Query Languages
Polymorphism, Complexity, and Completeness

Stijn Vansummeren
Université Libre de Bruxelles

10 May 2010
Introduction

Extensive study of type systems for:

- General purpose (Turing complete) programming languages:
  - ML [Hindley; 1969 - Milner; 1978 - Damas and Milner; 1982]
  - Many others

- Database programming languages (higher order functions + records + collections + . . .)
  - Extensible records [Wand; 1989 - Rémy; 1989, 1990]
  - Generalized relational operators [Buneman and Ohori; 1996]
  - Constrained types: HM(X) [Odersky, Sulzmann, and Wehr; 1999]
  - Many others

How does this specialize to database query languages?
- Limited expressiveness (not Turing-complete) → complete type systems?
- No higher-order functions, no subtyping → complexity of typability?
- Only records, collections → specialized type inference algorithms?
Introduction

Extensive study of type systems for:

• General purpose (Turing complete) programming languages:
  ○ ML [Hindley; 1969 - Milner; 1978 - Damas and Milner; 1982]
  ○ Many others

• Database programming languages
  (higher order functions + records + collections + ...):
  ○ Extensible records [Wand; 1989 - Rémy; 1989, 1990]
  ○ Generalized relational operators [Buneman and Ohori; 1996]
  ○ Constrained types: HM(X) [Odersky, Sulzmann, and Wehr; 1999]
  ○ Many others

How does this specialize to database query languages?

• Limited expressiveness (not Turing-complete) → complete type systems?

• No higher-order functions, no subtyping → complexity of typability?

• Only records, collections → specialized type inference algorithms?
Introduction

Extensive study of type systems for:

- **General purpose (Turing complete) programming languages:**
  - ML [Hindley; 1969 - Milner; 1978 - Damas and Milner; 1982]
  - Many others

- **Database programming languages**
  (higher order functions + records + collections + ...):
  - Extensible records [Wand; 1989 - Rémy; 1989, 1990]
  - Generalized relational operators [Buneman and Ohori; 1996]
  - Constrained types: HM(X) [Odersky, Sulzmann, and Wehr; 1999]
  - Many others

How does this specialize to database query languages?
Introduction

Extensive study of type systems for:

- **General purpose (Turing complete) programming languages:**
  - ML [Hindley; 1969 - Milner; 1978 - Damas and Milner; 1982]
  - Many others

- **Database programming languages**
  (higher order functions + records + collections + . . .):
  - Extensible records [Wand; 1989 - Rémy; 1989, 1990]
  - Generalized relational operators [Buneman and Ohori; 1996]
  - Constrained types: HM(X) [Odersky, Sulzmann, and Wehr; 1999]
  - Many others

How does this specialize to database query languages?

- Limited expressiveness (not Turing-complete)
- No higher-order functions, no subtyping
- Only records, collections
Introduction

Extensive study of type systems for:

- **General purpose (Turing complete) programming languages**:
  - ML [Hindley; 1969 - Milner; 1978 - Damas and Milner; 1982]
  - Many others

- **Database programming languages**
  (higher order functions + records + collections + ...):
  - Extensible records [Wand; 1989 - Rémy; 1989, 1990]
  - Generalized relational operators [Buneman and Ohori; 1996]
  - Constrained types: HM(X) [Odersky, Sulzmann, and Wehr; 1999]
  - Many others

How does this specialize to database query languages?

- Limited expressiveness (not Turing-complete) \(\rightarrow\) **complete type systems?**
- No higher-order functions, no subtyping
- Only records, collections
Introduction

Extensive study of type systems for:

- **General purpose (Turing complete) programming languages:**
  - ML [Hindley; 1969 - Milner; 1978 - Damas and Milner; 1982]
  - Many others

- **Database programming languages**
  (higher order functions + records + collections + . . .):
  - Extensible records [Wand; 1989 - Rémy; 1989, 1990]
  - Generalized relational operators [Buneman and Ohori; 1996]
  - Constrained types: HM(X) [Odersky, Sulzmann, and Wehr; 1999]
  - Many others

How does this specialize to database query languages?

- Limited expressiveness (not Turing-complete) → complete type systems?
- No higher-order functions, no subtyping → complexity of typability?
- Only records, collections
Introduction

Extensive study of type systems for:

- **General purpose (Turing complete) programming languages:**
  - ML [Hindley; 1969 - Milner; 1978 - Damas and Milner; 1982]
  - Many others

- **Database programming languages**
  (higher order functions + records + collections + . . .):
  - Extensible records [Wand; 1989 - Rémy; 1989, 1990]
  - Generalized relational operators [Buneman and Ohori; 1996]
  - Constrained types: HM(X) [Odersky, Sulzmann, and Wehr; 1999]
  - Many others

How does this specialize to database query languages?

- Limited expressiveness (not Turing-complete) → complete type systems?
- No higher-order functions, no subtyping → complexity of typability?
- Only records, collections → specialized type inference algorithms?
Introduction

Results presented are from the following papers:

- On the Complexity of Deciding Typability in the Relational Algebra
  Acta Informatica, 2005

- Polymorphic Type Inference for the Named Nested Relational Calculus
  ACM TOCL, 2006

- Well-Definedness and Semantic Type-Checking for the Nested Relational
  Calculus
  Theoretical Computer Science, 2007

- Unpublished notes

This is joint work with

- Dirk Van Gucht, Indiana University, USA
- Jan Van den Bussche, Hasselt University, Belgium
Introduction - Nested Relational Calculus $\mathcal{NRC}$

**Canonical Query Language for Complex Objects**

Objects $o := c \mid (A: o, \ldots, B: o') \mid \{o, \ldots, o'\}$
Introduction - Nested Relational Calculus \( \mathcal{NRC} \)

**Canonical Query Language for Complex Objects**

**Objects**

\[ o ::= c | (A: o, \ldots, B: o') | \{o, \ldots, o'\} \]

**Expressions**

\[ e ::= x | o | (A: e, \ldots, B: e') | e.A \]
\[ | \{\} | \{e\} | e_1 \cup e_2 | \{e \mid x_1 \in e_1, \ldots, x_n \in e_n\} \]

**Binder List**

\[ \Delta ::= x_1 \in e_1, \ldots, x_n \in e_n \]

**Types**

\[ s, t ::= \text{int} | \text{string} | \cdots | (A: s, \ldots, B: t) | \{s\} \]

**Example:**

\[ \{(A: y.C, B: z) \mid y \in x_1, z \in x_2\} \]
Introduction - Nested Relational Calculus $\mathcal{NRC}$

Canonical Query Language for Complex Objects

Objects $o ::= c \mid (A: o, \ldots, B: o') \mid \{o, \ldots, o'\}$

Expressions $e ::= x \mid o \mid (A: e, \ldots, B: e') \mid e.A$
\hline $\mid \{\} \mid \{e\} \mid e_1 \cup e_2 \mid \{e \mid x_1 \in e_1, \ldots, x_n \in e_n\}$

Binder List $\Delta ::= x_1 \in e_1, \ldots, x_n \in e_n$

Types $s, t ::= \text{int} \mid \text{string} \mid \cdots \mid (A: s, \ldots, B: t) \mid \{s\}$

Example:

\{(A: y.C, B: z) \mid y \in x_1, z \in x_2\}

\[
\begin{array}{ccc}
\text{C} & \text{D} \\
1 & 2 \\
3 & 4 \\
\end{array}
\quad \Rightarrow 
\begin{array}{cc}
3 & 8 \\
\end{array}
\]

\[
\begin{array}{cc}
\text{A} & \text{B} \\
1 & 3 \\
1 & 8 \\
3 & 3 \\
3 & 8 \\
\end{array}
\]
Introduction - Nested Relational Calculus \(\mathcal{NRC}\)

**Canonical Query Language for Complex Objects**

- Objects \(o \ ::= \ c \ | \ (A: o, \ldots, B: o') \ | \ \{o, \ldots, o'\}\)

- Expressions \(e \ ::= \ x \ | \ o \ | \ (A: e, \ldots, B: e') \ | \ e.A \)
  \(\ | \ \{\} \ | \ \{e\} \ | \ e_1 \cup e_2 \ | \ \{e \mid x_1 \in e_1, \ldots, x_n \in e_n\}\)

- Binder List \(\Delta \ ::= \ x_1 \in e_1, \ldots, x_n \in e_n\)

- Types \(s, t \ ::= \ \text{int} \ | \ \text{string} \ | \cdots \ | \ (A: s, \ldots, B: t) \ | \ \{s\}\)

**Operational semantics:**

\[
\begin{align*}
  e & \rightarrow o & \ldots & e' & \rightarrow o' \\
  (A: e, \ldots, B: e') & \rightarrow (A: o, \ldots, B: o') & e & \rightarrow (A: o, \ldots, B: o') \\
  e.A & \rightarrow o
\end{align*}
\]
Introduction - Nested Relational Calculus $\mathcal{NRC}$

**Canonical Query Language for Complex Objects**

**Objects**
\[ o ::= c \mid (A: o, \ldots, B: o') \mid \{o, \ldots, o'\} \]

**Expressions**
\[ e ::= x \mid o \mid (A: e, \ldots, B: e') \mid e.A \mid \{\} \mid \{e\} \mid e_1 \cup e_2 \mid \{e \mid x_1 \in e_1, \ldots, x_n \in e_n\} \]

**Binder List**
\[ \Delta ::= x_1 \in e_1, \ldots, x_n \in e_n \]

**Types**
\[ s, t ::= \text{int} \mid \text{string} \mid \cdots \mid (A: s, \ldots, B: t) \mid \{s\} \]

**Operational semantics:**

\[
\begin{align*}
\{\} & \rightarrow \{\} \\
\{e\} & \rightarrow \{o\} \\
\{e_1 \cup e_2\} & \rightarrow \{o_1, \ldots, o_m, o'_1, \ldots, o'_n\} \\
e_1 & \rightarrow \{o_1, \ldots, o_m\} \\
e_2 & \rightarrow \{o'_1, \ldots, o'_n\}
\end{align*}
\]
Introduction - Nested Relational Calculus \(\mathcal{NRC}\)

### Canonical Query Language for Complex Objects

**Objects**  
\[ o ::= c \mid (A : o, \ldots, B : o') \mid \{o, \ldots, o'\} \]

**Expressions**  
\[ e ::= x \mid o \mid (A : e, \ldots, B : e') \mid e.A \]
\[ \mid \{\} \mid \{e\} \mid e_1 \cup e_2 \mid \{e \mid x_1 \in e_1, \ldots, x_n \in e_n\} \]

**Binder List**  
\[ \Delta ::= x_1 \in e_1, \ldots, x_n \in e_n \]

**Types**  
\[ s, t ::= \text{int} \mid \text{string} \mid \cdots \mid (A : s, \ldots, B : t) \mid \{s\} \]

### Operational semantics:

\[
\begin{align*}
  e & \rightarrow o \\
  \{e \mid \} & \rightarrow \{o\}
\end{align*}
\]

\[
\begin{align*}
  e_1 & \rightarrow \{\} \\
  \{e \mid x_1 \in e_1, \Delta\} & \rightarrow \{\}
\end{align*}
\]

\[
\begin{align*}
  e_1 & \rightarrow \{o, \ldots, o'\} \\
  \{e[x_1/o] \mid \Delta[x_1/o]\} \cup \cdots \cup \{e[x_1/o'] \mid \Delta[x_1/o']\} & \rightarrow o'' \\
  \{e \mid x_1 \in e_1, \Delta\} & \rightarrow o''
\end{align*}
\]

15 / 67
In Search of a Complete Static Type System

Static Type Systems for Turing-complete Languages:

• Are sound (i.e., can prove the absence of runtime errors)
• But necessarily incomplete (i.e., cannot prove that an error will occur)

Question:

$NRC$ is not Turing-complete. Does it have a sound and complete static type system?
The question is equivalent to the following decision problem:

**Well-Definedness**

*Input:* Expression $e(x, \ldots, y)$ and types $s, \ldots, t$ for the free variables.

*Problem:* Decide whether $e$ is well-defined under $s, \ldots, t$, i.e., whether $e[x/o, \ldots, y/o']$ evaluates to an object for all $o: s, \ldots, o': t$. 

Well-Definedness for NRC is decidable.
The question is equivalent to the following decision problem:

**Well-Definedness**

*Input:* Expression $e(x, \ldots, y)$ and types $s, \ldots, t$ for the free variables.

*Problem:* Decide whether $e$ is well-defined under $s, \ldots, t$, i.e., whether $e[x/o, \ldots, y/o']$ evaluates to an object for all $o: s, \ldots, o': t$.

**Theorem**

**Well-Definedness** for $\mathcal{NRC}$ is decidable.
Language Extensions

Consider the Extension of $\mathcal{NRC}$ with

- **Atomic comparisons** $e_1 \text{ eq } e_2$ which can only compare two atomic data values
- This gives us essentially the **conjunctive queries**

**Operational semantics:**

\[
\begin{align*}
\frac{e_1 \rightarrow c_1 \quad e_2 \rightarrow c_1}{e_1 \text{ eq } e_2 \rightarrow \{()\}} \quad \frac{e_1 \rightarrow c_1 \quad e_2 \rightarrow c_2}{e_1 \text{ eq } e_2 \rightarrow \{\}}
\end{align*}
\]

**Example:** return all records in $R$ whose $A$-field is 5

\[
\{x \mid x \in R, y \in (x.A \text{ eq } 5)\}
\]
Language Extensions (2)

**Theorem**

- **Well-Definedness** for $\mathcal{NC}(eq)$ is decidable.
- **Well-Definedness** for $\mathcal{NC}(eq)$ is hard for $\text{Co-Nexptime}$
Language Extensions (2)

**Theorem**

- **Well-Definedness** for $\mathcal{NRC}(\text{eq})$ is decidable.
- **Well-Definedness** for $\mathcal{NRC}(\text{eq})$ is hard for $\text{Co-Nexptime}$

**Hardness follows by reduction from:**

**Satisfiability**

**Input:** Expression $e(x, \ldots, y)$ and types $s, \ldots, t$ such that $e[x/o, \ldots, y/o']$ evaluates to a set for all $o: s, \ldots, o': t$.

**Problem:** Decide whether there exist objects $o: s, \ldots, o': t$ such that $e[x/o, \ldots, y/o']$ evaluates to a non-empty set.
Language Extensions (2)

**Theorem**

- **Well-Definedness** for $\mathcal{NRC}(eq)$ is decidable.
- **Well-Definedness** for $\mathcal{NRC}(eq)$ is hard for **Co-Nexptime**

**Hardness follows by reduction from:**

**Satisfiability**

**Input:** Expression $e(x, \ldots, y)$ and types $s, \ldots, t$ such that $e[x/o, \ldots, y/o']$ evaluates to a set for all $o: s, \ldots, o': t$.

**Problem:** Decide whether there exist objects $o: s, \ldots, o': t$ such that $e[x/o, \ldots, y/o']$ evaluates to a non-empty set.

- Let $e$ be a **closed**, well-defined expression that always outputs a set
Language Extensions (2)

**Theorem**

- **Well-Definedness** for \( \mathcal{NRC}(eq) \) is decidable.
- **Well-Definedness** for \( \mathcal{NRC}(eq) \) is hard for \( \text{Co-NExpTime} \)

**Hardness follows by reduction from:**

**Satisfiability**

**Input:** Expression \( e(x, \ldots, y) \) and types \( s, \ldots, t \) such that \( e[x/o, \ldots, y/o'] \) evaluates to a set for all \( o: s, \ldots, o': t \).

**Problem:** Decide whether there exist objects \( o: s, \ldots, o': t \) such that \( e[x/o, \ldots, y/o'] \) evaluates to a non-empty set.

- Let \( e \) be a **closed**, well-defined expression that always outputs a set
- Then \( \{\}.A \mid x \in e \) is well-def \( \iff e \) is satisfiable
Language Extensions (2)

Theorem

- **Well-Definedness** for $\mathcal{NRC}(eq)$ is decidable.
- **Well-Definedness** for $\mathcal{NRC}(eq)$ is hard for $\text{Co-NExpTime}$

Hardness follows by reduction from:

**Satisfiability**

*Input:* Expression $e(x, \ldots, y)$ and types $s, \ldots, t$ such that $e[x/o, \ldots, y/o']$ evaluates to a set for all $o: s, \ldots, o': t$.

*Problem:* Decide whether there exist objects $o: s, \ldots, o': t$ such that $e[x/o, \ldots, y/o']$ evaluates to a non-empty set.

- Let $e$ be a **closed**, well-defined expression that always outputs a set
- Then $\{\{} \cdot A \mid x \in e \} \text{ is well-def } \iff e \text{ is satisfiable}$
- [Koch; 2006] Satisfiability of closed expressions is $\text{Co-NExpTime}$-hard.
Consider the Extension of $\mathcal{NRC}$ with

- **General comparisons** $e_1 = e_2$ which can compare arbitrary objects
- Gives us at the full power of the relational algebra

**Operational semantics:**

- $e_1 \rightarrow o_1 \quad e_2 \rightarrow o_1 \quad e_1 = e_2 \rightarrow \{()\}$
- $e_1 \rightarrow o_1 \quad e_2 \rightarrow o_2 \quad e_1 = e_2 \rightarrow \{\}$

Satisfiability for Relational Algebra is undecidable.

Therefore, Satisfiability for $\mathcal{NRC}$ is undecidable.

Hence, Well-Definedness for $\mathcal{NRC}$ is undecidable.
Consider the Extension of $\mathcal{NRC}$ with

- **General comparisons** $e_1 = e_2$ which can compare arbitrary objects
- **Gives us at the full power of the relational algebra**

Operational semantics:

$$
\frac{e_1 \rightarrow o_1 \quad e_2 \rightarrow o_1}{e_1 = e_2 \rightarrow \{(\)}}, \quad \frac{e_1 \rightarrow o_1 \quad e_2 \rightarrow o_2}{e_1 = e_2 \rightarrow \{\}}
$$

**Theorem**

- **Satisfiability** for Relational Algebra is undecidable.
- Therefore **Satisfiability** for $\mathcal{NRC}(=)$ is undecidable.
- Hence, **Well-Definedness** for $\mathcal{NRC}(=)$ is undecidable.
Consider the Extension of $\mathcal{NRC}$ with

- Singleton extraction $\text{extract}(e)$ that extract the value from a singleton set
- present in OQL

Operational semantics:

$$e \rightarrow \{o\}$$

$$\text{extract}(e) \rightarrow o$$

This allows us to model some features of SQL

- SQL: select ... where ($5 = \text{select distinct } A \text{ from } R$)
- $\mathcal{NRC}(eq, extract)$: $5 \text{ eq } (\text{extract } \{x.A \mid x \in R\})$
Consider the Extension of $\mathcal{NRC}$ with

- **Singleton extraction** $\text{extract}(e)$ that extract the value from a singleton set
- present in OQL

Operational semantics:

$$
\frac{e \rightarrow \{o\}}{\text{extract}(e) \rightarrow o}
$$

This allows us to model some features of SQL

- **SQL**: select ... where ($5 = \text{select distinct} A \text{ from } R$)
- $\mathcal{NRC}(\text{eq}, \text{extract})$: $5 \text{ eq } (\text{extract} \{x.A \mid x \in R\})$

**Theorem**

Well-Definedness for $\mathcal{NRC}(\text{eq}, \text{extract})$ is undecidable
In Search of a Complete Static Type System (4)

**Conclusion:**

- Complete Static Type Systems exist for restricted query languages (the conjunctive queries)
- But these systems have high complexity
In Search of a Complete Static Type System (4)

**Conclusion:**

- Complete Static Type Systems exist for restricted query languages (the conjunctive queries)
- But these systems have high complexity

**Solution:** adopt the standard (incomplete) static type system
Conclusion:

- Complete Static Type Systems exist for restricted query languages (the conjunctive queries)
- But these systems have high complexity

**Solution:** adopt the standard (incomplete) static type system

Typing rules:

\[
\begin{align*}
T \vdash x : T(x) & \quad T \vdash o : s & \quad T \vdash e : (A : s, \ldots, B : t) \\
\end{align*}
\]

\[
\begin{align*}
T \vdash e : s & \quad \ldots & \quad T \vdash e' : t \\
\end{align*}
\]

\[
T \vdash (A : e, \ldots, B : e') : (A : s, \ldots, B : t)
\]
Conclusion:

- Complete Static Type Systems exist for restricted query languages (the conjunctive queries)
- But these systems have high complexity

Solution: adopt the standard (incomplete) static type system

Typing rules:

\[
\begin{align*}
T & \vdash \{ \} : \{ s \} \\
T & \vdash \{ e \} : \{ s \} \\
T & \vdash e : s \\
T & \vdash e_1 : \{ s \} \quad T \vdash e_2 : \{ s \} \\
T & \vdash e_1 \cup e_2 : \{ s \}
\end{align*}
\]

\[
\begin{align*}
T & \vdash \{ x_1 \in e_1, \ldots, x_n \in e_n \} : \{ s \} \\
T & \vdash x_1 : s_1, \ldots, x_i : s_i, T \vdash e_{i+1} : \{ s_{i+1} \} \text{ for } 0 \leq i < n \\
T & \vdash x_1 : s_1, \ldots, x_n : s_n, T \vdash e : s
\end{align*}
\]
Expressiveness?

Static typing sometimes limits expressiveness:

- Untyped Lambda Calculus: all computable functions
- Simply Typed Lambda Calculus: restricted to extended polynomials
Expressiveness?

Static typing sometimes limits expressiveness:

- Untyped Lambda Calculus: all computable functions
- Simply Typed Lambda Calculus: restricted to extended polynomials

Is the same true for queries?

- Language-integrated queries in statically typed languages (LINQ, links, ...)
- Language-integrated queries in dynamically typed languages (Python, Ruby, ...)

Is the static type system for NRC expressively complete?

Can all well-typed queries be equivalently written in a well-typed way?
Expressiveness?

Static typing sometimes limits expressiveness:
- Untyped Lambda Calculus: all computable functions
- Simply Typed Lambda Calculus: restricted to extended polynomials

Is the same true for queries?
- Language-integrated queries in statically typed languages (LINQ, links, …)
- Language-integrated queries in dynamically typed languages (Python, Ruby, …)

Question:
Is the static type system for $NRC$ expressively complete?
- Can all well-typed queries be equivalently written in a well-typed way?
Expressiveness? (2)

**Theorem**

The static type system for $\mathcal{NRC}(=)$ is expressively complete:

- Every $\mathcal{NRC}(=)$ expression $e(x, \ldots, y)$ that is well-defined under $r, \ldots, s$ and only produces outputs in a type $t$ has an equivalent expression $e'(x, \ldots, y)$ such that $x : r, \ldots, y : s \vdash e' : t$.

- Moreover, $e'$ is of size linear in $e$. 

Some ways in which expressions can be well-defined, but ill-typed:

- Unreachable code: $\{ \}$

- Creating heterogeneous objects: $\{ z \cdot A | z \in (x \cup y) \}$

- Can be rewritten as $\{ z \cdot A | z \in x \} \cup \{ z \cdot A | z \in y \}$

- General case more difficult: $\{ z \cdot A | z \in e \}$ ill-typed when $e$ is another comprehension that returns a heterogeneous set.

I would appreciate any pointers to the literature on similar results for general-purpose (functional) programming languages!
The static type system for $\text{NRC}(=)$ is expressively complete:

- Every $\text{NRC}(=)$ expression $e(x, \ldots, y)$ that is well-defined under $r, \ldots, s$ and only produces outputs in a type $t$ has an equivalent expression $e'(x, \ldots, y)$ such that $x: r, \ldots, y: s \vdash e': t$.

- Moreover, $e'$ is of size linear in $e$.

Some ways in which expressions can be well-defined, but ill-typed:

- Unreachable code: $\{\{\}.A \mid x \in \{\}}$
Theorem

The static type system for $\mathcal{NRC}(=)$ is expressively complete:

- Every $\mathcal{NRC}(=)$ expression $e(x, \ldots, y)$ that is well-defined under $r, \ldots, s$ and only produces outputs in a type $t$ has an equivalent expression $e'(x, \ldots, y)$ such that $x: r, \ldots, y: s \vdash e': t$.
- Moreover, $e'$ is of size linear in $e$.

Some ways in which expressions can be well-defined, but ill-typed:

- Unreachable code: $\{\{} . A \mid x \in \{\} \} \rightarrow \{\}$
Expressiveness? (2)

Theorem

The static type system for $\mathcal{NR}(\equiv)$ is expressively complete:

- Every $\mathcal{NR}(\equiv)$ expression $e(x, \ldots, y)$ that is well-defined under $r, \ldots, s$ and only produces outputs in a type $t$ has an equivalent expression $e'(x, \ldots, y)$ such that $x: r, \ldots, y: s \vdash e': t$.
- Moreover, $e'$ is of size linear in $e$.

Some ways in which expressions can be well-defined, but ill-typed:

- Unreachable code: $\{\{\}.A \mid x \in \{\}\} \rightarrow \{\}$

- Creating heterogeneous objects: $\{z.A \mid z \in (x \cup y)\}$
  - ill-typed under $x \mapsto \{(A: r, B: s)\}$, $y \mapsto \{(A: r, C: t)\}$.

I would appreciate any pointers to the literature on similar results for general-purpose (functional) programming languages!
Expressiveness? (2)

**Theorem**

The static type system for $\mathcal{NRC}(\simeq)$ is expressively complete:

- Every $\mathcal{NRC}(\simeq)$ expression $e(x, \ldots, y)$ that is well-defined under $r, \ldots, s$ and only produces outputs in a type $t$ has an equivalent expression $e'(x, \ldots, y)$ such that $x : r, \ldots, y : s \vdash e' : t$.

- Moreover, $e'$ is of size linear in $e$.

**Some ways in which expressions can be well-defined, but ill-typed:**

- Unreachable code: $\{\{\}.A \mid x \in \{\}\} \rightarrow \{\}$

- Creating heterogeneous objects: $\{z.A \mid z \in (x \cup y)\}$
  - ill-typed under $x \mapsto \{(A : r, B : s)\}$, $y \mapsto \{(A : r, C : t)\}$.
  - Can be rewritten as $\{z.A \mid z \in x\} \cup \{z.A \mid z \in y\}$.

I would appreciate any pointers to the literature on similar results for general-purpose (functional) programming languages!
Expressiveness? (2)

**Theorem**

The static type system for $NC(=)$ is expressively complete:

- Every $NC(=)$ expression $e(x,\ldots,y)$ that is well-defined under $r,\ldots,s$ and only produces outputs in a type $t$ has an equivalent expression $e'(x,\ldots,y)$ such that $x: r,\ldots,y: s\vdash e': t$.
- Moreover, $e'$ is of size linear in $e$.

**Some ways in which expressions can be well-defined, but ill-typed:**

- Unreachable code: $\{\emptyset\}.A \mid x \in \emptyset \rightarrow \emptyset$

- Creating heterogeneous objects: $\{z.A \mid z \in (x \cup y)\}$
  - ill-typed under $x \mapsto \{(A: r, B: s)\}$ $y \mapsto \{(A: r, C: t)\}$.
  - Can be rewritten as $\{z.A \mid z \in x\} \cup \{z.A \mid z \in y\}$

- General case more difficult: $\{z.A \mid z \in e\}$
  - ill-typed when $e$ is another comprehension that returns a heterogeneous set.
Expressiveness? (2)

**Theorem**

The static type system for $\mathcal{NRC}(=)$ is expressively complete:

- Every $\mathcal{NRC}(=)$ expression $e(x, \ldots, y)$ that is well-defined under $r, \ldots, s$ and only produces outputs in a type $t$ has an equivalent expression $e'(x, \ldots, y)$ such that $x: r, \ldots, y: s \vdash e': t$.

- Moreover, $e'$ is of size linear in $e$

Some ways in which expressions can be well-defined, but ill-typed:

- Unreachable code: $\{\{\}.A \mid x \in \{\}\} \rightarrow \{\}$

- Creating heterogeneous objects: $\{z.A \mid z \in (x \cup y)\}$
  - ill-typed under $x \mapsto \{(A: r, B: s)\}$, $y \mapsto \{(A: r, C: t)\}$.
  - Can be rewritten as $\{z.A \mid z \in x\} \cup \{z.A \mid z \in y\}$

- General case more difficult: $\{z.A \mid z \in e\}$
  - ill-typed when $e$ is another comprehension that returns a heterogeneous set

I would appreciate any pointers to the literature on similar results for general-purpose (functional) programming languages!
Polymorphic Expressiveness

Consider the Extension of $\mathcal{NRC}$ with

- **Complement projection** $\text{drop}_A(e)$ that retains all of $e$’s fields but $A$

Operational semantics and typing rule:

$$
\begin{align*}
    e & \rightarrow (A: o, B: o', \ldots, C: o'') & T \vdash e: (A: r, B: s, \ldots, C: t) \\
    \text{drop}_A e & \rightarrow (B: o', \ldots, C: o'') & T \vdash \text{drop}_A e: (B: s, \ldots, C: t)
\end{align*}
$$

Note that:

- We can easily simulate $\text{drop}_A x$ in $\mathcal{NRC}(=)$ if we know the type of $x$
Polymorphic Expressiveness

Consider the Extension of \( \mathcal{NRC} \) with

- Complement projection \( \text{drop}_A(e) \) that retains all of \( e \)'s fields but \( A \)

Operational semantics and typing rule:

- \( e \rightarrow (A: o, B: o', \ldots, C: o'') \)
- \( \text{drop}_A \ e \rightarrow (B: o', \ldots, C: o'') \)
- \( T \vdash e: (A: r, B: s, \ldots, C: t) \)
- \( T \vdash \text{drop}_A \ e: (B: s, \ldots, C: t) \)

Note that:

- We can easily simulate \( \text{drop}_A x \) in \( \mathcal{NRC}(=) \) if we know the type of \( x \)
- But not if \( x \)'s type is unknown!

Theorem

- A typing of an expression \( e \) is a pair \( (T, s) \) such that \( T \vdash e: s \)
- Say that two expressions \( e(x, \ldots, y) \) and \( e'(x, \ldots, y) \) are polymorphically equivalent if they have the same set of typings and, for each such typing \( (T, s) \), \( e_1 \) and \( e_2 \) evaluate to the same output on each input of type \( T \)
- No expression in \( \mathcal{NRC}(=) \) is polymorphically equivalent to \( \text{drop}_A x \)
Polymorphic Expressiveness (2)

Consider the Extension of \( \mathcal{NRC} \) with

- **Cartesian Product** \( e_1 \times e_2 \)
- **Join** \( e_1 \bowtie e_2 \)

Typing rules: \((\phi_1, \phi_2, \psi \text{ are record types, } + \text{ is record type concatenation})\)

\[
T \vdash e_1 : \{\phi_1\} \quad T \vdash e_2 : \{\phi_2\} \\
\phi_1 \text{ and } \phi_2 \text{ have disjoint sets of attributes} \quad T \vdash e_1 \times e_2 : \{\phi_1 + \phi_2\}
\]

\[
T \vdash e_1 : \{\phi_1 + \psi\} \quad T \vdash e_2 : \{\phi_2 + \psi\} \\
\phi_1 \text{ and } \phi_2 \text{ have disjoint sets of attributes} \quad T \vdash e_1 \bowtie e_2 : \{\phi_1 + \phi_2 + \psi\}
\]
Polymorphic Expressiveness (2)

Consider the Extension of $\mathcal{NRC}$ with

• **Cartesian Product** $e_1 \times e_2$
• **Join** $e_1 \Join e_2$

Typing rules: ($\phi_1, \phi_2, \psi$ are record types, $+$ is record type concatenation)

\[
\begin{align*}
T \vdash e_1 : \{\phi_1\} & \quad T \vdash e_2 : \{\phi_2\} \\
\phi_1 \text{ and } \phi_2 \text{ have disjoint sets of attributes} \\
T \vdash e_1 \times e_2 : \{\phi_1 + \phi_2\}
\end{align*}
\]

\[
\begin{align*}
T \vdash e_1 : \{\phi_1 + \psi\} & \quad T \vdash e_2 : \{\phi_2 + \psi\} \\
\phi_1 \text{ and } \phi_2 \text{ have disjoint sets of attributes} \\
T \vdash e_1 \Join e_2 : \{\phi_1 + \phi_2 + \psi\}
\end{align*}
\]

**Theorem**

• No expression in $\mathcal{NRC}(\_=, \text{drop}, \times)$ is polymorphically equivalent to $e_1 \Join e_2$.
• No expression in $\mathcal{NRC}(\_=, \text{drop}, \Join)$ is polymorphically equivalent to $e_1 \times e_2$. 
Polymorphic Expressiveness (3)

Open Research Questions

- Is there a reasonable notion when a query language is “polymorphically complete”?
- What operators are needed to obtain such a language?
Typability and Type Inference

Two classical problems

**Typability**

*Input:* Expression $e(x, \ldots, y)$

*Problem:* Do there exist $T$ and $t$ such that $T \vdash e : t$?

**Type Inference**

*Input:* Expression $e(x, \ldots, y)$

*Problem:* Give an explicit description of the set of all typings $(T, s)$ for which $T \vdash e : s$?

**Practical Motivation:**

- Complexity of Typability tells us something about the complexity of typechecking queries in implicitly typed programming languages.
- Type inference is essential for query optimization in the absence of schema information (Kleisli, ...).
Typability and Type Inference

What notion of type formulae is “just right” for a given query language?
Typability and Type Inference

What notion of type formulae is “just right” for a given query language?

**Theorem** [Buneman and Ohori; 1996]

There exists a polynomial time algorithm that, given an expression $e(x, \ldots, y)$ in $\mathcal{NRC}(=)$, returns false if $e$ is untypable, and otherwise returns a kinded type formula describing all of $e$’s typings.
Typability and Type Inference

What notion of type formulae is “just right” for a given query language?

**Theorem** [Buneman and Ohori; 1996]

There exists a polynomial time algorithm that, given an expression $e(x, \ldots, y)$ in $\mathcal{NRC}(=)$, returns false if $e$ is untypable, and otherwise returns a kinded type formula describing all of $e$’s typings.

**Theorem** [Rémy; 1993]

There exists a polynomial time algorithm that, given an expression $e(x, \ldots, y)$ in $\mathcal{NRC}(=, \text{drop})$, returns false if $e$ is untypable, and otherwise returns a type formula with row variables describing all of $e$’s typings.
Typability and Type Inference for $\mathcal{NRC}(=, \text{drop}, \times, \triangleright)$

Our proposal: a type inference algorithm based on constrained types, in the spirit of $\text{HM}(X)$ [Odersky, Sulzmann, and Wehr; 1999]

Signature of principal type formulas

A principal type formula for $e(x_1, \ldots, x_n)$ is a conjunctive, many-sorted, first-order logic formula $\varphi(x_1, \ldots, x_n, z)$ that, interpreted in the structure of all possible types $T$, defines all typings of $e$:

\[
x_1 : s_1, \ldots, x_n : s_n \vdash e : t \quad \iff \quad T \models \varphi(s_1, \ldots, s_n, t)
\]

Example: principal type formula for $x \cup y$

\[(\exists u) x = \text{Set}(u) \land y = \text{Set}(u) \land z = \text{Set}(u)\]
Typability and Type Inference for $\mathcal{NR}(=, \text{drop}, \times, \bowtie)$

The signature of the logic and its interpretation:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Arity</th>
<th>Interpretation in $\mathcal{T}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x, y, \ldots$</td>
<td>type</td>
<td></td>
</tr>
<tr>
<td>$\rho, \rho', \ldots$</td>
<td>row</td>
<td></td>
</tr>
<tr>
<td>$=$</td>
<td>type $\times$ type</td>
<td>equality relation on types</td>
</tr>
<tr>
<td>$\subseteq$</td>
<td>row $\times$ row</td>
<td>containment of functions e.g. $A: s \subseteq B: t$, $A: s$</td>
</tr>
<tr>
<td>$#$</td>
<td>row $\times$ row</td>
<td>relates rows with disjoint domains</td>
</tr>
<tr>
<td>Set</td>
<td>type $\rightarrow$ type</td>
<td>maps $s$ to ${s}$</td>
</tr>
<tr>
<td>Record</td>
<td>row $\rightarrow$ type</td>
<td>maps $A: s, \ldots, B: t$ to $(A: s, \ldots, B: t)$</td>
</tr>
<tr>
<td>$A$</td>
<td>type $\rightarrow$ row</td>
<td>maps $s$ to $A: s$</td>
</tr>
<tr>
<td>$,$</td>
<td>row $\times$ row $\rightarrow$ row</td>
<td>maps $(\rho_1, \rho_2)$ to $\rho_1 \cup \hat{\pi}_{\text{dom}(\rho_1)}(\rho_2)$</td>
</tr>
</tbody>
</table>

Example: principal type formula for $\{v.A \mid v \in (x \times y)\}$

$$(\exists \rho)(\exists \rho') x = \text{Set}(\text{Record}(\rho)) \land y = \text{Set}(\text{Record}(\rho')) \land \rho \not\# \rho'$$

$$\land (\exists u) z = \text{Set}(u) \land A(u) \subseteq \rho, \rho'$$
Typability and Type Inference for $\mathcal{NR}(\equiv, \text{drop}, \times, \otimes)$

Theorem

Every $\mathcal{NR}(\equiv, \text{drop}, \times, \otimes)$ expression $e$ has a principal type formula $\varphi_e$, of size linear in the size of $e$, and computable from $e$ in polynomial time.
Typability and Type Inference for $\mathcal{NRC}(\simeq, \text{drop}, \times, \boxtimes)$

**Theorem**

Every $\mathcal{NRC}(\simeq, \text{drop}, \times, \boxtimes)$ expression $e$ has a principal type formula $\varphi_e$, of size linear in the size of $e$, and computable from $e$ in polynomial time.

**Main application:**

Typability of $\mathcal{NRC}(\simeq, \text{drop}, \times, \boxtimes)$-expressions is in NP.
Typability and Type Inference for $\mathcal{NRC}(=, \text{drop}, \times, \Join)$

**Theorem:**

Typability of $\mathcal{NRC}(=, \text{drop}, \times, \Join)$-expressions is in NP.

**Proof sketch:**

**Step 1.** Expression $e \rightarrow$ principal type formula $\varphi_e$

- $e$ is *typable* $\iff$ $\varphi_e$ is *satisfiable*

**Example:**

$$\{v.A \mid v \in (x \times y)\} \cup \{v.B \mid v \in x\}$$

$\iff$

$$(\exists u)(\exists \rho_1)(\exists \rho_2) z = \text{Set}(u) \land x = \text{Set(Record}(\rho_1)) \land y = \text{Set(Record}(\rho_2))$$

$$\land \rho_1 \neq \rho_2 \land A(u) \subseteq \rho_1, \rho_2 \land B(u) \subseteq \rho_1$$
Typability and Type Inference for $\mathcal{NRC}(=, \text{drop}, \times, \lhd)$

**Theorem:**

Typability of $\mathcal{NRC}(=, \text{drop}, \times, \lhd)$-expressions is in NP.

**Proof sketch:**

**Step 2.** Principal type formula $\varphi_e \rightarrow$ quantifier free formula $\psi_e$

$\varphi_e$ is satisfiable $\iff \psi_e$ is satisfiable

**Example:**

$$(\exists u)(\exists \rho_1)(\exists \rho_2) z = \text{Set}(u) \land x = \text{Set}(\text{Record}(\rho_1)) \land y = \text{Set}(\text{Record}(\rho_2))$$

$$\land \rho_1 \not\# \rho_2 \land A(u) \subseteq \rho_1, \rho_2 \land B(u) \subseteq \rho_1$$

$\iff$

$z = \text{Set}(u) \land x = \text{Set}(\text{Record}(\rho_1)) \land y = \text{Set}(\text{Record}(\rho_2))$

$$\land \rho_1 \not\# \rho_2 \land A(u) \subseteq \rho_1, \rho_2 \land B(u) \subseteq \rho_1$$
Typability and Type Inference for $\mathcal{NRC}(=, \text{drop, } \times, \bowtie)$

**Theorem:**
Typability of $\mathcal{NRC}(=, \text{drop, } \times, \bowtie)$-expressions is in $\text{NP}$.

**Proof sketch:**

**Step 3.** Quantifier free formula $\psi_e \rightarrow \text{guess}$ quantifier free formula $\theta_e$ without row variables

$\psi_e$ is satisfiable $\iff$ $\theta_e$ is satisfiable

**Example:**

$z = \text{Set}(u) \land x = \text{Set}(\text{Record}(\rho_1)) \land y = \text{Set}(\text{Record}(\rho_2))$

$\land \rho_1 \not\equiv \rho_2 \land A(u) \subseteq \rho_1, \rho_2 \land B(u) \subseteq \rho_1$

$\iff$

$z = \text{Set}(u) \land x = \text{Set}(\text{Record}(B(u_1))) \land y = \text{Set}(\text{Record}(A(u_2)))$

$\land B(u_1) \not\equiv A(u_2) \land A(u) \subseteq B(u_1), A(u_2) \land B(u) \subseteq B(u_1)$
Typability and Type Inference for $\mathcal{NRC}(=, \text{drop}, \times, \Join)$

**Theorem:**

Typability of $\mathcal{NRC}(=, \text{drop}, \times, \Join)$-expressions is in NP.

**Proof sketch:**

**Step 4.** Quantifier free formula $\theta_e$ without row variables $\rightarrow$ simplified formula $\sigma_e$

$\theta_e$ is **satisfiable** $\iff$ $\sigma_e$ is **satisfiable**

**Example:**

\[
\begin{align*}
z &= \text{Set}(u) \land x = \text{Set}(\text{Record}(B(u_1))) \land y = \text{Set}(\text{Record}(A(u_2))) \\
&\quad \land B(u_1) \neq A(u_2) \land A(u) \subseteq B(u_1), A(u_2) \land B(u) \subseteq B(u_1) \\
&\quad \iff \\
&\quad z = \text{Set}(u) \land x = \text{Set}(\text{Record}(B(u_1))) \land y = \text{Set}(\text{Record}(A(u_2))) \\
&\quad \land A(u) \subseteq B(u_1), A(u_2) \land B(u) \subseteq B(u_1)
\end{align*}
\]
Typability and Type Inference for $\mathcal{NRC}(=, \text{drop}, \times, \Join)$

**Theorem:**
Typability of $\mathcal{NRC}(=, \text{drop}, \times, \Join)$-expressions is in NP.

**Proof sketch:**

**Step 4.** Quantifier free formula $\theta_e$ without row variables $\rightarrow$ simplified formula $\sigma_e$

$\theta_e$ is satisfiable $\iff$ $\sigma_e$ is satisfiable

**Example:**

\[
\begin{align*}
z &= \text{Set}(u) \land x = \text{Set(Record}(B(u_1))) \land y = \text{Set(Record}(A(u_2))) \\
&\quad \land A(u) \subseteq B(u_1), A(u_2) \land B(u) \subseteq B(u_1)
\end{align*}
\]

$\iff$

\[
\begin{align*}
z &= \text{Set}(u) \land x = \text{Set(Record}(B(u_1))) \land y = \text{Set(Record}(A(u_2))) \\
&\quad \land u = u_2 \land u = u_1
\end{align*}
\]

The latter formula can efficiently be solved by unification
Typability and Type Inference for \( \mathcal{NRC}(=, \text{drop}, \times, \boxtimes) \)

**Theorem:**

Typability of \( \mathcal{NRC}(=, \text{drop}, \times, \boxtimes) \)-expressions is NP-hard.

**Proof sketch:** By a reduction from Positive one-in-three 3SAT.

Abbreviate \( \pi_A(e) := \{v.A \mid v \in e\} \).
Typability and Type Inference for $\mathcal{NRC}(=, \text{drop}, \times, \Join)$

**Theorem:**

Typability of $\mathcal{NRC}(=, \text{drop}, \times, \Join)$-expressions is NP-HARD.

**Proof sketch:** By a reduction from Positive one-in-three 3SAT.

Abbreviate $\pi_A(e) := \{ \nu.A \mid \nu \in e \}$.

\[
(x_1 \lor y_1 \lor z_1) \land (x_2 \lor y_1 \lor z_2) \land (x_2 \lor y_3 \lor z_1) \text{ is satisfiable} \\
\Leftrightarrow \\
\pi_A(x_1 \times y_1 \times z_1) \cup \pi_A(x_2 \times y_1 \times z_2) \cup \pi_A(x_2 \times y_3 \times z_1) \text{ is typable}
\]
Typability and Type Inference for $\mathcal{NRC}(=,\text{drop},\times,\boxtimes)$

What about typability for expressions without cartesian product operator?

Typability of NNRC-expressions without cartesian product operator is $\text{NP}$-hard.

**Theorem**

**Proof sketch:**

• Uses proof idea from Ohori and Buneman (1988), who showed that typability for “generalized join” is $\text{NP}$-hard.

• Reduction from $\text{Monotone 3SAT}$.

Reductions transfer to programming languages with symmetric record concatenation, join, or mixin modules.
Typability and Type Inference for $\mathcal{NRC}(=, \text{drop}, \times, \ltimes)$

What about typability for expressions without cartesian product operator?

**Theorem**

Typability of NNRC-expressions without cartesian product operator is NP-hard.

**Proof sketch:**

- Uses proof idea from Ohori and Buneman (1988), who showed that typability for “generalized join” is NP-hard.
- Reduction from **Monotone 3SAT**.
What about typability for expressions without cartesian product operator?

**Theorem**

Typability of NNRC-expressions without cartesian product operator is NP-hard.

**Proof sketch:**

- Uses proof idea from Ohori and Buneman (1988), who showed that typability for “generalized join” is NP-hard.
- Reduction from **Monotone 3SAT**.

Reductions transfer to programming languages with symmetric record concatenation, join, or mixin modules.
The complexity of type checking DBPL’s with the $NRC(=, \text{drop}, \times, \Join)$ as the embedded QL.

When the ambient language is the simply typed $\lambda$-calculus:

- Type checking moves from $P$-complete to $NP$-hard.
The complexity of type checking DBPL’s with the $\mathcal{NRC}(=, \text{drop}, \times, \triangleright)$ as the embedded QL

When the ambient language is the simply typed $\lambda$-calculus

- Type checking moves from $\mathsf{P}$-complete to $\mathsf{NP}$-hard

When the ambient language is ML:

- Type checking was already $\mathsf{EXPTIME}$-complete
- However, $\mathsf{EXPTIME}$-hardness is only due to peculiar programs which rarely occur in practice
- Type checking ML is typically in linear time in practice
- In contrast, $\mathsf{NP}$-hardness for $\mathcal{NRC}(=, \text{drop}, \times, \triangleright)$ is due to cartesian product and join, which do occur in practice