MECHANICAL FORMALISATION OF NESTED RELATIONAL CALCULUS QUERY SYNTAX AND SEMANTICS

SUBMITTED TO SWANSEA UNIVERSITY IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF RESEARCH IN LOGIC AND COMPUTATION
SWANSEA UNIVERSITY, DATE 18TH OF DECEMBER 2023
REVISED AT UNIVERSITY OF GLASGOW, DATE 26TH OF JULY 2024

WRITTEN BY

OLIVIA HARTLEY WESTON

SUPERVISED BY

DR. CÉCILIA PRADIC
DR. ARNO PAULY

Abstract

Building on top of existing work by Wong, we exhibit an implementation of the Nested Relational Calculus of Buneman et al. in the Coq interactive theorem prover. We provide a syntax for the language. Additionally, we provide a denotational effective list semantics and a denotational $Prop$-based semantics for the language. We create examples that show the Nested Relational Calculus' capabilities as a declarative programming language. We provide a discussion on works similar to ours, justifying our design decisions. We leave the reader with multiple directions in which the project can be taken, and the expected results.

Declaration

This work has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

13/08/2024

Candidate's signature *Date*

Statement

This thesis is the result of my own investigations, except where otherwise stated. Where correction services have been used, the extent and nature of the correction is clearly marked in a footnote(s).

Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

13/08/2024

Candidate's signature *Date*

Contents

Acknowledgements

I want to thank my best friends and support system, Grem and Rob, for being there with me throughout so much of my life, I hope we will ride it out together until the end. I could not have gotten to this point in my life without support from Dr. Pauly and Dr. Pradic. Both of them encouraged me to explore things I find interesting and provided so many opportunities to network with a welcoming scientific community. Thank you Ashley for helping me stay afloat even in the darkest of days. I apologise to anyone I may have missed, I did not want to drench the paper in my sentimental tears.

List of Figures

Chapter 1

Introduction

Databases are a core concept to modern computing, allowing us to rapidly and reliably transform large quantities of data. There exist many paradigms that describe the operation of a database system, and there exist many languages and theories that allow us to reason within and about these systems. One of these theories is the Nested Relational Calculus, or NRC.

This work is a computer-assisted mechanical formalisation of findings regarding the Nested Relational Calculus of Buneman et al. It mostly draws from the formalisations of NRC found in Wong [34], a thorough PhD thesis on the topic.

We consider NRC to be of interest because it connects the world of relational calculi that are pervasively used in database applications to the world of functional programming. This is because NRC is essentially a typed declarative programming language that uses the lambda calculus as its foundation. As examples, we point to work by Cheney, Lindley, and Wadler [8] as a formulation of language-integrated queries, which are very close relatives of NRC. We also highlight Fowler, Galpin, and Cheney [21] as another example of language-integrated queries in the specific scenario of temporal databases within the LINKS programming language. These use extensions built on NRC as a basis. We showcase an example table written in NRC in Table 1.1.

ID	Name	Salary / total expensing
EMP1	Santana	100000
EMP2	Martyn	150000
CTR1	Chiharu	500000
CTR2	Parveen	200000

Table 1.1: An example employee and contractor table with an associated ID, name, and salary (or projected expenses).

$$(\{(\text{"EMP1"}, \text{"Santana"}, 100000)\} \cup \{(\text{"EMP2"}, \text{"Martyn"}, 150000)\}) \cup (\{(\text{"CTR1"}, \text{"Chiharu"}, 500000)\} \cup \{(\text{"CTR2"}, \text{"Parveen"}, 200000)\})$$

Figure 1.1: The example Table 1.1 shown as an NRC construction, modulo types.

We do not claim to provide new findings regarding the Nested Relational Calculus in the theoretical sense. What we provide is a translation from some of Wong's findings into a dependently typed programming language. To be exact, we provide the core language of NRC without function types, we provide provide two possible denotational semantics for NRC, we give a lemma for compatibility of these two semantics, and we give various extensions of the language. Specifically, type remapping, binding and a select operation. Our choice of proof assistant and dependently typed programming language for these constructions is the Coq[1] proof assistant.

We decided to use the Coq interactive proof assistant for several reasons. First, the project supervisor – Dr. Pradic – has previous experience in this proof assistant, thus research support from her will be more relevant to the project. Second, there already exists a formulation of SQL within Coq, as seen in Benzaken and Contejean [5]. This lends itself to the possibility of a future work that manages to prove conservativity over actual implementations of NRC and

[1]Now known as the Rocq proof assistant. This thesis was written prior to the change of name, and thus the naming reflects the name at the time.

SQL. A prototype written in other proof assistants such as Agda or Lean would need to go through an extra step to conform to the restrictions imposed by the alternative choice. Regardless of the fruition of work on the mechanisation, the guiding thought is to provide an open-ended prototype to the scientific community that can be reused in the future.

In Chapter 2 we discuss the fundamental concepts that this work utilises and builds off of. We discuss the original Codd's relational model in contrast with its nested equivalent (NRC), and finally we provide the reader with formal definitions for concepts that will be used throughout the work.

In Section 2.4 we give a brief historical viewing of interactive theorem provers to give the reader a clearer understanding of the framework and tools that will be used to produce our mechanisation.

In Section 3.1 we show a high-level explanation of the scope of our project in preparation for the coming sections. We begin by showing the syntactic core of NRC, implemented in Coq in Section 3.2. We continue our journey through NRC by showing two possible denotational semantics in Section 3.3. The work finishes off its technical contributions by providing examples of NRC's usage for constructing common elements of programming languages in Section 3.4.

After a thorough exploration of our work, we focus our attention on two primary related works. We provide a discussion about two perspectives on NRC that differ from ours. This can be found in Chapter 5.

We finish our thesis with a discussion of future projects and possibilities of expanding this idea further in three main points, found in Chapter 6.

Chapter 2

Background

2.1 Codd's Relational Calculus & Algebra

We will frequently use the term 'calculus'. When we say this, we refer to a *computational calculus*, which is a system of rules that dictates how to derive computation.

One such way is through the relational calculi. They are a family of systems that treat relations as primitive elements of computation, as proposed by Codd [14]. There are various forms of the relational calculus, such as the relational calculus using three-valued or four-valued logic; however, for this project we consider our model to use the common two-valued first-order, or predicate logic with the appropriate semantic adjustments. This is enriched with the basic relational operators: projection, selection, join (of two forms), union, difference, intersection, division [13].

The calculi serve as components of the greater 'relational model'. The relational model underlies the database theory known as SQL. SQL stands for *Structured Query Language*. It is a declarative programming language used as an interface for relational database systems. At the time of writing it can be considered the most popular database language in the industry, as four of the top five most popular database management systems are all based on it [31].

PupilTable		
Name	**Mark**	**Average Grade**
Jacob	83	79
Marie	70	85
Alex	93	84
Fido	98	46

Table 2.1: Example table which can be represented with relational calculus.

$$
\begin{aligned}
&\text{PupilTable} = \\
&\quad \{j, m, a, f \mid j.\text{Mark} = 83 \quad \wedge j.\text{AverageGrade} = 79 \quad \wedge j.\text{Name} = \text{Jacob} \\
&\qquad\qquad m.\text{Mark} = 70 \quad \wedge m.\text{AverageGrade} = 85 \quad \wedge m.\text{Name} = \text{Marie} \\
&\qquad\qquad a.\text{Mark} = 93 \quad \wedge a.\text{AverageGrade} = 84 \quad \wedge a.\text{Name} = \text{Alex} \\
&\qquad\qquad f.\text{Mark} = 98 \quad \wedge f.\text{AverageGrade} = 46 \quad \wedge f.\text{Name} = \text{Fido}\}
\end{aligned}
$$

Figure 2.1: Example Table 2.1 written in the form of relational calculus.

To illustrate, we show a relation defined using the relational calculus depicting the exam results of a class compared to expected result based on an average of previous grades (Table 2.1 and Figure 2.1). We show a filtering operation in SQL that selects only students who perform better than expected in Figure 2.2.

This work does not go into formalising the relational calculus as proposed by Codd. Rather, this work explores an extension of the relational calculus known as the *Nested Relational Calculus*.

2.2 Nested Relational Calculus, informally

The Nested Relational Calculus — or *NRC* — is a higher-order declarative programming language in the same family as Codd's relational calculus. Originally devised by Buneman et al. [7], later firmly formalised by Wong [34], a large fragment of its syntax is partly inspired by the category-theoretical notion of a monad to nest relations at various depths.

```
FROM PupilTable
SELECT Name
WHERE Mark > AverageGrade
```

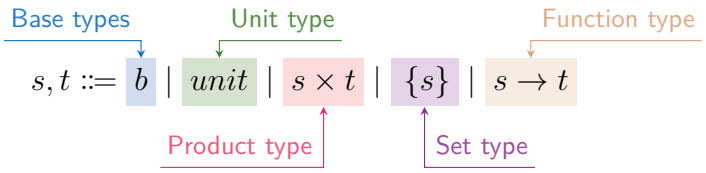
Figure 2.2: An SQL operation filtering out tuples from Table 2.1 and projecting the `Name` column.

NRC is of interest to us because it is a higher-order theory that is capable of expressing the same things that SQL can, while maintaining a connection to the world of the lambda calculus, and therefore the world of programming languages. We will see how this holds in Section 2.3. Connecting these two worlds is one solution to the impedance mismatch problem between programming languages and database languages. This problem was defined in Copeland and Maier [15, 16] as the problem of differing programming paradigms (where programming languages were frequently procedural, while database languages are declarative) and the problem of differing data type definitions between the languages, e.g., an integer in one language may not be the same as an integer in another language).

The semantics classically considered for NRC are that of sets, bags (also known as multisets) and lists [7, 34]. We loosely define the term *semantics* as a 'meaning' that is assigned to each syntactic term of a language. Semantics tell us how a programming language's constructs interact with concrete data structures and/or with the state of a program. There are various forms of semantics, with more discussion on them being found in Section 6.3.

2.3 Nested Relational Calculus, formally

The Nested Relational Calculus can be seen in Buneman et al. [7], and is formalised as a traditional computational calculus with type derivations in Wong [34]. It is a simply typed language for interacting with databases using nested relations. The following are the types in the universe of NRC.



(2.1)

The product type $s \times t$ corresponds to a pair whose first project is an object of type s and its second project is an object of type t. The $unit$ type has one inhabitant, namely $()$. The set type $\{s\}$ corresponds to a finite set of objects of type s. The semantics of objects with the set type vary. Our implementation will employ lists to model NRC with set semantics. In other words, the semantics of our model language's nestable data structures are that of sets, and the programming language implementation semantics of our model's set structure is that of lists.

Function types The function, or arrow type is one that is present in the original NRC, allowing the language to construct intricate programs that can be reduced according to the reduction rules of the lambda calculus. We, however, **will not** be using the function type. This is primarily due to the time constraints of the project. As a result, our NRC is not a fully-fledged programming language, but rather a declarative language defining operations on the contents of a database.

The syntactic structure of our version of NRC is described as follows [34]:

VARIABLE
$$\frac{\Gamma(x) = s}{\Gamma \vdash x{:}s}$$

UNIT
$$\frac{}{\Gamma \vdash ():unit}$$

PRODUCT
$$\frac{\Gamma \vdash e_1{:}s \qquad \Gamma \vdash e_2{:}t}{\Gamma \vdash (e_1, e_2){:}s \times t}$$

PROJ1
$$\frac{\Gamma \vdash e{:}s \times t}{\Gamma \vdash \pi_1 e{:}s}$$

PROJ2
$$\frac{\Gamma \vdash e{:}s \times t}{\Gamma \vdash \pi_2 e{:}t}$$

EMPTY SET
$$\frac{}{\Gamma \vdash \{\}^s{:}\{s\}}$$

SINGLETON
$$\frac{\Gamma \vdash e{:}s}{\Gamma \vdash \{e\}{:}\{s\}}$$

SMALL CUP
$$\frac{\Gamma \vdash e_1{:}\{s\} \qquad \Gamma \vdash e_2{:}\{s\}}{\Gamma \vdash e_1 \cup e_2{:}\{s\}}$$

BIG CUP
$$\frac{\Gamma, x : t \vdash e_1{:}\{s\} \qquad \Gamma \vdash e_2{:}\{t\}}{\Gamma \vdash \bigcup\{e_1 \mid x \in e_2\}{:}\{s\}}$$

INTERSECTION
$$\frac{\Gamma \vdash e_1{:}\{s\} \qquad \Gamma \vdash e_2{:}\{s\}}{\Gamma \vdash e_1 \cap e_2{:}\{s\}}$$

The unit type only produces one constant object. The existence of an empty set is guaranteed, and any existing element of type e can be lifted to a set type. The small cup union operator works as expected based on used semantics. The big cup operator is akin to a bind operator in Haskell, where for all members x^t of e_2 we perform $(\lambda x.e_1)x^t$. As we do not have function types, we interpret a function with its corresponding application just as a substitution. In other words, we feed every member x^t of e_2 as a substitution into each occurrence of $x \in e_1$. The resulting sets are then joined together with a small cup union. This implies that x must be a free variable in e_1. In a more visual manner:

$$\bigcup \{e_1 \mid x \in e_2\} = \texttt{for} x_i \in e_2 \texttt{ where } e_2 = \{x_0, \ldots, x_m\} \\ (e_1\{x_0/x\}) \cup (e_1\{x_1/x\}) \cup \cdots \cup (e_1\{x_m/x\}) \tag{2.2}$$

Properties of NRC All queries created using most members of the NRC family have a polynomial time computational complexity with respect to their size [34, Theorem 2.2.1]. This result, however, excludes certain members of said family. Specifically, the powerset operator takes NRC queries out of polynomial complexity. This is obvious, as the powerset operator has exponential complexity. The author proves this for every 'primitive function' definable in the Nested Relational Algebra, $\mathcal{NRA}$, but also claims that a stronger theorem holds for NRC, given a 'specific' operational semantics. The paper does not go further into this aspect for NRC. Additionally, the nested relational models are conservative over the flat relational model, as long as the queries are flat. This can be seen in Wong [33].

2.4 Programming language theory

This short section serves as exposition for concepts that are commonly seen in the field of programming language theory, leading up to interactive theorem proving. Everything mentioned in this section is used in some manner throughout the project, but is not the subject of it. Readers experienced with programming language theory will likely not gain new information from this section.

Lambda calculus In 1932 Church provided foundations for a formal system, a subset of which used higher-order anonymous functions as its foundation, termed the Lambda Calculus [12]. This contrasted Turing's foundations of instructions on an infinite double-ended tape. Lambda calculus uses term substitutions and reduction operations as its definition of term evaluation. It is considered the foundation for modern functional programming languages [2].

Type systems An effect of having an untyped lambda calculus is the ability to create recursive functions through applying a function to itself. This feature can lend itself to core logical issues because there is nothing stopping us from writing nonterminating programs. Historically, computer scientists and computational mathematicians have been interested in computable programs and functions. A type system can help achieve that. One popular system for modelling computation in a typed manner is the Simply Typed Lambda Calculus, or STLC [3].

Dependent type theory Ever since the inception of the Lambda Calculus there have been countless type systems created to express computations. A type system that rose to prominence was written by Martin-Löf, which also connected to the foundations of mathematics by introducing logical connectives that had a one-to-one correspondence with those in intuitionistic logic. This is called Martin-Löf Type Theory, or MLTT, as seen in Martin-Löf [26]. By combining a connection to logic and using constructivist principles, the type signatures of functions became correspondent to existential proofs of witnesses for said functions.

Coq Alongisde Martin-Löf's Intuitionistic Type Theory there also exists the Calculus of Constructions, by Coquand and Huet [17]. It is also a type theory that is based in constructive approaches. It contributed to the creation of the Coq proof assistant, which we use to mechanically formalise the Nested Relational Calculus. The Calculus of Constructions has a chain of type universes called $Type$ with each element indexed by the natural numbers, e.g. $Type_0$, $Type_1$ etc. A strict subset of this chain of universes is the type of propositions

$Prop$, the values of which get discarded during code extraction. $Prop$ is an impredicative universe of types, whereas $Type$ is a predicative.

2.5 A background in Coq syntax

We describe core components of the Coq proof assistant's programming language, Gallina. The syntax is similar to OCaml.

Pattern matching Coq supports a very rich form of pattern matching. It supports simple pattern matching, allowing us to destruct a term into its constructors, but it also allows us to use additional keywords such as `return` in the header of the pattern match, which specifies a return type of the pattern match. Additionally, we can use `as` to annotate the subject of the pattern match with a binder, and `in` to give the typechecker additional information about the type shape of the structure we are matching over.

Inductively defined structures In Coq we can create inductively defined structures using the `Inductive` keyword. Every constructor (in the example, `case_1` and `case_2`) is a function that builds an element of our inductive structure. This is the primary mechanism through which we introduce the syntax of NRC.

```
1 Inductive structure_name (argument1 : type1) : return_type :=
2 | case_1 (argument2 : type2) : structure_name arg1 arg2
3 | case_2 (argument3 : type3) : structure_name arg1 arg2
4 ...
```

Standard functions To define a function in Coq we use the `Definition` keyword.

```
1 Definition function_name (argument1 : type1) : return_type := ...
```

Recursive functions To define a recursive function in Coq we use the `Fixpoint` keyword.

```
Fixpoint function_name (argument1 : type1) : return_type := ...
```

Records To define records in Coq we use the `Record` keyword.

```
Record record_name (argument1 : type1) : return_type :=
  { param1 : ptype1 ;
    param2 : ptype2 ;
    ...             }
```

Proof functions Coq sees a distinction between functions that are relevant to computation and functions that are used in proofs. We have discussed typical functions for computations in the form of `Definition` and `Fixpoint` keywords. Proof functions can be used via keywords like `Theorem`, `Lemma` and `Axiom`. The `Axiom` keyword is special because it does not require a proof after its definition, rather its contents are assumed to be true. The other keywords require the provision of a proof after stating the definition.

```
Theorem theorem_name (argument1 : type1) : return_type = ...
```

Chapter 3

Implementation of Nested Relational Calculus

3.1 High-level overview

Urelements We will be using the term *urelement*. This term can be interchangeably used with the term *atom*. It is a term originating from set theory, discussing objects that are not sets but may be elements of a set. The importance lies in the indivisiblity of the urelement, compared to sets.

Our implementation of the Nested Relational Calculus is that of a type system, an inductive type of syntactic terms and a semantics. Alongside this, some minor examples and theorems are added.

The in-universe type system of the Nested Relational Calculus is fairly simple. It consists of atomic types ι which represent the urelements of the language, the unit type 1, the set type $\beta\,\tau$[1] which takes some in-universe type τ as parameter and the product type $\tau_1 \times \tau_2$, which takes in-universe types τ_1 and τ_2 as parameters.

[1]We use the set constructor β, as opposed to, say, Set for primarily cultural reasons, as the supervisor of this project, Dr. Cécilia Pradic, has previously used the same notation.

The syntax system of NRC is nearly identical to the one found in Wong [34, Section 2.1]. Our syntax differs in two ways, one being that we do not include function types in our formulation. A lack of function types necessitates removing lambda abstraction and function application from the core set of syntactic expressions. This is done primarily due to the time constraints of the project. The other difference is the addition of a difference operator. This is done because the difference operator adds expressivity to NRC at no cost to its complexity. More discussion on these decisions can be found in Section 5.2.

To implement the difference operator — and in fact any non-monotonic operations — we will need a defined equality in the language of NRC. This has been previously achieved in Wong [34, Section 2.4]. As a result of this, we have imposed a restriction on the semantics of types in NRC: we only allow (decidably equal) types with a linear order.

For showing equality we note that Wong — alongside other literature on relational calculi [24] — utilise the already-existing unit type to express booleans in computable predicates. This is because the set monad of the unit type effectively becomes a type with two inhabitants: $\{\}$ the empty set, and $\{()\}$ the set containing the unit. We can regard these as false and true respectively. With these conventions and restrictions, we are now ready to formalise semantics for NRC.

This work formalises two denotational semantics. The first is semantics based on the $Prop$ type in Coq, giving us a 'logical' representation of NRC from the point of view of the Coq type system. The second is a more pragmatic, list-based semantics that lets NRC work with concrete data structures. The compatibility of these two semantics is proven.

With a fully formalised syntax and semantics, we can start seeing outlines of NRC as a fully-fledged programming language. To further its arsenal, we add definitions to aid in term substitution by means of remapping contexts of NRC variables and by binding contexts to new ones. We define the notion of a

predicate using the set-unit type in NRC in accordance with the $unit$-boolean semantics we discussed above. This approach was taken from Wong [34], justified by previous works on relational model theory not using booleans as first-class elements [24].

With the notion of a predicate defined, we are ready to show a generalised *select* operator which filters data based on a provided predicate.

3.2 Syntax

Coq pseudocode alterations Here, we list all alterations made to the Coq code that would make the code non-functional[2]. These alterations are present for the sake of readability and easier code explainability. For a clearer code reference, the code may be found in the associated GitHub repository.

Pseudocode	Coq	Pseudocode	Coq
→	->	∃	exists
⇒	=>	⋆	*
∧	/\	¬	~
∨	\/	π_1	nrc_proj1
⊥	False	π_2	nrc_proj2
λ x ⇒	fun x =>	()	tt
∀	forall		

This section contains the core Coq constructs used in this project. To make reading the Coq code easier, syntactic elements have been adjusted, thus the code is actually pseudocode that will not run. For a list of adjustments made see paragraph above. Unless stated otherwise, the following code definitions can be found in the file `Syntax.v` of the formalisation subdirectory of the project.

Definition 1.

[2]This could, perhaps, be stepped over with the `Utf8` module from the Coq standard library.

The following is the definition of our core in-universe types[3] of NRC.

$$\text{NRC}_{\text{TYPE}} = \tau, \sigma ::= \iota \mid 1 \mid \beta\,\tau \mid \tau \times \sigma$$

The types are as follows: ι stands for an urelement of NRC. 1 refers to the unit type with a sole inhabitant $()$. $\beta\,\tau$ refers to a type of sets of τ. $\tau \times \sigma$ refers to a type of pairs of τ and σ. Note that this is the same as the definition given in Equation 2.1. In Figure 3.1 we find the same definition in Coq pseudocode.

```
Inductive nrc__type : Type :=
| nrcT_atom : nrc__type
| nrcT_unit : nrc__type
| nrcT_pow  : nrc__type → nrc__type
| nrcT_prod : nrc__type → nrc__type → nrc__type.
```

Figure 3.1: The type system of NRC.

Because the constructor names are unwieldy, they have been shortened to `ι`, `1`, `τ × σ` and `β τ`, standing for atom, unit, pair and set, respectively. This was done using Coq's notation system.

Variable type X We note that we use an arbitrary type X in Figure 3.2 which works as the namespace of our variables. As an example, $X := \mathbb{N}$ would allow us to use the natural numbers as variables which, when evaluated with semantics in mind, point to some data of some, potentially different type. This variable type is where we could implement other forms of binding, such as de Bruijn indices.

Sigma We use $\Sigma : X \to \text{NRC}_{\text{TYPE}}$ as our typing context. In other words, Σ is a function mapping variables of some arbitrary type X to sorts (in other words, elements of NRC_{TYPE}) in the NRC universe. This keeps track of typechecking in Coq.

[3] In the codebase these are called `sort`.

```
Inductive nrc__stx {X : Type} (Σ : X → nrc__type)
  : nrc__type → Type :=
  | nrc_var (x : X) : nrc__stx Σ (Σ x)
  | nrc_pair (τ σ : nrc__type)
             (H1 : nrc__stx Σ τ)
             (H2 : nrc__stx Σ σ)
             : nrc__stx Σ (τ × σ)
  | nrc_proj1 (τ σ : nrc__type)
              (H : nrc__stx Σ (τ × σ))
              : nrc__stx Σ τ
  | nrc_proj2 (τ σ : nrc__type)
              (H : nrc__stx Σ (τ × σ))
              : nrc__stx Σ σ
  | nrc_unit : nrc__stx Σ 1
  | nrc_empty (τ : nrc__type) : nrc__stx Σ (β τ)
  | nrc_sing (τ : nrc__type)
             (H : nrc__stx Σ τ)
             : nrc__stx Σ (β τ)
  | nrc_cup (τ : nrc__type)
            (H1 : nrc__stx Σ (β τ))
            (H2 : nrc__stx Σ (β τ))
            : nrc__stx Σ (β τ)
  | nrc_bigcup (τ σ : nrc__type)
               (H1 : nrc__stx Σ (β σ))
               (H2 : @nrc__stx ^X (Σ & σ) (β τ))
               : nrc__stx Σ (β τ)
  | nrc_diff (τ : nrc__type)
             (H1 : nrc__stx Σ (β τ))
             (H2 : nrc__stx Σ (β τ))
             : nrc__stx Σ (β τ).
```

Figure 3.2: The syntax of NRC.

Big cup The big cup operator is defined as a bind operator that evaluates e_1 under a context enriched with $x \in e_2$, then performs a small cup operation on the resulting sets. The intuition for this can be found in Equation 2.2. In Figure 3.2 we find the same definition in Coq pseudocode.

Context enrichment As seen in the case of `nrc_bigcup`, we require a context enriched by σ. This is achieved through Coq's `option_rect`, which generates a function with an extra object in its domain. Thus the context becomes either an evaluation of Σ, or an evaluation of the $+1$ element, which in this case

is σ. In practice we can decide whether to access Σ or σ through the use of the `option` type, also commonly known as the `maybe` type. The explicit type argument `^X` is shorthand for `option X`. The @ symbol states that the call to `nrc__stx` should treat all parameters as explicit.

Intrinsic vs extrinsic typing When implementing a programming language, we must decide whether we want an intrinsically typed or an extrinsically typed system. In an intrinsically typed system, we have one abstract data type responsible for the data, as well as for its well-typedness. With an extrinsically typed system, we build a core data type of the language that is then passed through a typechecking function to ensure it's well-typed. We chose an intrinsically-typed system primarily as an exercise in the correct-by-construction programming paradigm pioneered by users of dependently typed languages (spanning back to Church-style intrinsic interpretations of terms). As our programming language of choice is dependently typed, we see no reason to use multiple structures to show well-typed terms of NRC when the programming language gives us the tooling to create a singular, precise and mathematically cohesive structure describing our desired object. Naturally, as a tradeoff, we force the abstract syntax trees of our language to be more complex to write, since we are forced to include more information by definition. We do, however, get a guarantee of type preservation for any new operation we may need to add to the language. If we were to use an extrinsic reresentation, we would increase our workload further by needing to provide more proofs, and we would need to 'carry' many proofs first through a well-typedness check. In return, we would get a much cleaner abstract syntax tree for our language, allowing our language definition to be more digestible for a new reader. We believe that we could alleviate some of the issues with extrinsic typing by using proof automation, but this would be an extra learning curve for us, as this is the first proper exposure to the Coq proof assistant for the author of this paper. For more information on intrinsic and extrinsic semantics of terms we recommend Reynolds [29].

Variable binding There are many strategies for tackling binders, many of which can be found in a survey in Aydemir et al. [1]. Our choice of binding is to use single arbitrary type for variables. At first in this project we assume `string` for easily readable variable names. Then, once we delve into Chapter 4, we switch to using natural numbers due to bugs with Coq's normalisation, as well as due to the cumbersome nature of Coq strings (see Section 4.2.1). At the time of early design for NRC we were considering de Bruijn indices as a candidate for representing variables. A known issue with de Bruijn indexing is the lack of human readability, which we consider important enough that we would not want to use it (de Bruijn [18]). It would, however, eliminate any issues with capture-avoiding substitutition and α-equivalence. We felt that the simplest and least time-consuming variable representation would suit the project best. We have not considered higher-order approaches at the time as we were not aware of them, but even after review, we believe that Coq's lack of primitive support for strong higher-order abstract syntax (HOAS) would cause more trouble than necessary for a project already limited on time. To see a tool implementing HOAS in Coq and the associated challenges we advise reading Felty and Momigliano [20] to see the engineering work associated with such a technique. Instead, if we were intent on using higher-order approaches for binding we would give more consideration to LF systems such as Twelf or Beluga.

3.3 Semantics

3.3.1 $Prop$-based

Definition 2.
In Figure 3.3, our semantics show a simple pattern match over a given type from NRC_{TYPE}. A set type has been rewritten into a logical implication producing a type from the type of propositions, $Prop$. The variable X is set as implicit for cleaner code. It represents a concrete type we are interested in interpreting into. Unless specified otherwise, the code in this section can be found in the file `Semantics.v` of the repository's formalisation subdirectory.

```
Fixpoint interpTypeProp {X} (τ : nrc_type) : Type :=
  match τ with
    ι     ⇒ X
  | 1     ⇒ unit
  | τ × σ ⇒ interpTypeProp τ * interpTypeProp σ
  | β τ   ⇒ interpTypeProp τ → Prop
  end.
```

Figure 3.3: The interpretation of NRC_{TYPE} in $Prop$.

Definition 3.
In Figure 3.4 are $Prop$ semantics for the terms of NRC_{STX}. As we can see, these semantics require a general type variable $X : Type$, which we previously also used for interpreting elemnets of NRC_{TYPE}. It is assumed for common usage of this NRC code that this variable is always the same, thus it has been made implicit, as seen by the curly braces in `X`.

The arguments Σ and τ are implicit arguments that arise from the contents of argument t, which is a term in NRC_{STX}. The argument ρ represents a dependently typed function that is capable of interpreting the variables from the X type into concrete values. It is indexed by the variable $x : X$, depending on the value of x after being checked for its type. This ensures that ρ cannot interpret a variable x in a manner that is inconsistent with Σ's interpretation of x's type.

Variable `nrc_var` is an instance of interpreting a variable `x` under context `Σ:X → nrc_type`. This is shown as `(Σ x)` interpreted by the function `ρ`.

Pair `nrc_pair` is interpreted by using Coq's pair construct, after which a recursive call is made to interpret the content of each component of the pair.

Projections Both of the projections are interpreted by using Coq's projection functions `fst` and `snd`, after which a recursive call is made to interpret the contents of the projections.

```
Fixpoint interpStxProp {X} {Σ : X → nrc_type} {τ}
                       (t : nrc__stx Σ τ)
                       (ρ : ∀ x, interpTypeProp (Σ x))
                     : interpTypeProp τ :=
  match t with
    nrc_var    Σ x         ⇒ ρ x
  | nrc_pair   Σ τ σ t u   ⇒ (interpStxProp t ρ, interpStxProp u ρ)
  | nrc_proj1  Σ τ σ t     ⇒ π₁ interpStxProp t ρ
  | nrc_proj2  Σ τ σ t     ⇒ π₂ interpStxProp t ρ
  | nrc_unit   Σ           ⇒ ()
  | nrc_empty  Σ τ         ⇒ λ _ ⇒ ⊥
  | nrc_sing   Σ τ t       ⇒ λ z ⇒ z = interpStxProp t ρ
  | nrc_cup    Σ τ t u     ⇒ λ z ⇒
                               interpStxProp t ρ z ∨
                               interpStxProp u ρ z
  | nrc_bigcup Σ τ σ t u   ⇒ λ z ⇒
                               ∃ y,
                                 interpStxProp t ρ y ∧
                                 interpStxProp u (ρ & y) z
  | nrc_diff   Σ τ t u     ⇒ λ z ⇒
                               interpStxProp t ρ z ∧
                               (¬ interpStxProp u ρ z)
  end.
```

Figure 3.4: The interpretation of NRC_{STX} in $Prop$.

Unit The interpretation provides the inhabitant of Coq's unit type known as `tt`.

Empty set `nrc_empty` provides an implication in the form of a lambda abstraction which then goes to the `False` type of Coq, written ⊥. By definition, ⊥ has no inhabitants. It is also a member of the Coq type $Prop$, thus the semantics of NRC_{TYPE} hold.

Singleton set `nrc_sing` also uses a lambda abstraction over a variable `z`, which is then taken to the recursing proposition `z = interpStxProp t ρ`. This proposition is a member of $Prop$, thus the semantics of NRC_{TYPE} hold.

Small cup `nrc_cup` lambda abstracts over `z` and provides the recursing assertion `interpStxProp t ρ z ∨ interpStxProp u ρ z`. It is in Coq's

$Prop$ and states that the element `z` is either in the left set `t` or the right set `u`, which corresponds to the intuitive understanding of small cup.

Big cup `nrc_bigcup` lambda abstracts over `z`. It provides a witness `y` using Coq's constructive `∃` operator. This witness is subject to the recursing assertion that `y` can be applied to `interpStxProp t ρ` to provide a meaningful NRC term with semantics. This is followed by a logical conjunction with the recursing assertion that given our input set `z`, we can interpret it with respect to its parent NRC term `u` with the assumption that the context is enriched by `y`. This is written `interpStxProp u (ρ & y) z`. Once again, these assertions are in Coq's $Prop$ type, thus the semantics of NRC_{TYPE} hold.

Difference `nrc_diff` lambda abstracts over `z` and states `interpStxProp t ρ z ∧ (¬ interpStxProp u ρ z)`. In other words, the statement asserts that our resulting set `z` is simultaneously within the interpretation of term `t`, while not being in the interpretation of term `u`. This, once again, is in $Prop$, and therefore follows NRC_{TYPE}.

3.3.2 List

While semantics utilising $Prop$ are well-defined and proving common programming language properties[4] over them is possible, they can be considered more of a tool for proofs in Coq rather than computationally meaningful data. This is due to the nature of the $Prop$ type in Coq. To see a more pragmatic aspect of NRC we will use the list datatype which is commonly found in most programming languages. We are interested in expressing the set monad as sets with a linear order imposed on them, so our lists will have to emulate their behaviour: there are no duplicate elements (normalisation) and every compound datatype such as sets and pairs must be linearly ordered. We use a lexicographic ordering to achieve this.

[4] For example, soundness and completeness with respect to another language, type preservation, etc.

Decidable equality Prior to defining a linear order over interpreted terms, we are interested in the simpler notion of decidable equality. This is used for auxiliary list functions which are useful for normalisation, e.g. `removeAll`, which removes all occurrences of a given element in a list. In that example, we need to be able to test for equality to be able to remove all instances of a specific element. These definitions can be found in the file `DecidableEquality.v`.

Definition 4.
Decidable equality is defined as $\forall x(\forall y(x = y \vee x \neq y))$ constructively. We implement this in Coq in Figure 3.5.

```
1 Definition decEq (A : Type) := ∀ (x y : A), {x = y} + {x ≠ y}.
```

Figure 3.5: Decidable equality defined in Coq.

As we can see, Coq uses `{term1} + {term2}` to represent a constructive or, rather than the classical or. Whereas the commonly used $\vee$ can be used on terms of type $Prop$, if we desire computationally significant results from this, i.e. to actually figure out whether or not the witness to our or-question is $term1$ or $term2$, we must use a constructive or. This is why decidable equality is written without $\vee$.

Ordered types We will need ordered types to be able to consistently represent the semantics of NRC using lists. These lists need to be normalised, and thus sorted. To be able to sort these elements, we will need an ordering imposed over them. The majority of generic work on ordered types can be found in the file `OrderedTypes.v` of the formalisation subfolder of the repository. We define a linearly ordered type as a record type with the following fields:

- An ordering function
- Reflexivity proof of the function
- Transitivity proof of the function

- Antisymmetry proof of the function
- Totality (connexity) proof of the function

Definition 5.
We use the shorthands `x ⊑? y` and `x ⊑A? y` to represent the ordering function `ordFunction` and the explicitly typed ordering function respectively in Figure 3.6.

```
Record orderedType (X : Type) : Type ≔ mkOrdered
  { ordFunction : X → X → bool;
    ordReflexivity : ∀ (x : X),
                       x ⊑? y = true;
    ordTransitivity : ∀ (x y z : X),
                        x ⊑? y = true
                      → y ⊑? z = true
                      → x ⊑? z = true;
    ordAntisymmetry : ∀ (x y : X),
                        x ⊑? y = true
                      → y ⊑? x = true
                      → x = y;
    ordTotality : ∀ (x y : X),
                    x ⊑? y = true ∨ y ⊑? x = true
  }.
```

Figure 3.6: Definition of an ordered type.

Now that we have defined decidable equality and a linear order, we may proceed to define an interpretation of NRC_{TYPE} and NRC_{STX} using lists as the underlying data structure.

Definition 6.
Once again, we use an ambient `X:Type` that represents the type of used variables. An interpretation of NRC_{TYPE} in list semantics may be defined as in Figure 3.7.

For our decidability and ordering requirements to be imposed onto interpreted terms of NRC_{STX}, we show that terms of NRC_{TYPE} are beholden to a decidable equality function, titled `decEqInterpSortEff`, and a lexicographi-

```
Fixpoint interpSortEff {X : Type} (τ : nrc__type) : Type :=
  match τ with
    ι     ⇒ X
  | 1     ⇒ unit
  | τ × σ ⇒ interpSortEff τ * interpSortEff σ
  | β τ   ⇒ list (interpSortEff τ)
  end.
```

Figure 3.7: Interpretation of NRC_{TYPE} using list.

cal ordering function, titled `lexOrderInterpSortEff`. Both functions show that their argument is a type that upholds said properties. Then, in code we show that every term of NRC_{TYPE} maintains these two properties for all NRC_{TYPE} terms we may write, so long as we use a type of variables that has decidable equality and a lexicographical order.

Normalisation of elements To perform operations with set-like behaviour such as the difference (or interesection) operator, we need to define the notion of a normal form list semantics. Normalisation is done on the type level to members of NRC_{TYPE}, as well as to arbitrary lists with our ordering.

List normalisation All normalisation of lists comes down to the definition of `normaliseListRepr`, as seen in Figure 3.8. The function uses `nodup`, a Coq standard library function that removes duplicate elements from a list, `ordDecEq`, which extracts proofs of decidable equality and total (lexicographic, in our case) order of the used type, and finally `insertSort`, which is a simple sorting algorithm. The precise sorting algorithm used is irrelevant.

```
Definition normaliseListRepr {X : Type} (O : orderedType X)
                             (l : list X) : list X :=
  insertSort O (nodup (ordDecEq O) l).
```

Figure 3.8: List normalisation function.

Definition 7.
In Figure 3.9 is a definition of converting an interpreted type of NRC_{TYPE} to its

normal form. This is structured as a pattern match on the possible constructors of NRC_{TYPE}. It re-abstracts a given interpreted term of NRC_{TYPE} to change the internal structure of said term without affecting its type at the end (the input argument is `x:interpSortEff τ`, which is the same as the output type).

```
Fixpoint normaliseInterpSortEff {τ : nrc__type}
                                (x : interpSortEff τ)
                                : interpSortEff τ :=
  match τ return interpSortEff τ → interpSortEff τ with
  | ι     ⇒ λ x ⇒ x
  | 1     ⇒ λ x ⇒ tt
  | β σ   ⇒ λ x ⇒ normaliseListRepr (lexOrderInterpSortEff σ)
                                    (map normaliseInterpSortEff x)
  | τ × σ ⇒ λ x ⇒ let (y,z) := x
                  in (normaliseInterpSortEff y,
                      normaliseInterpSortEff z)
  end x.
```

Figure 3.9: Normalisation of NRC_{TYPE} with list semantics.

Normalisation of lists A second step on our way to ensuring we always work with consistent data is the definition of list normalisation. This definition consists of a duplicate element checking function and a sorting function. Our implementation uses Coq's builtin `nodup` function to deduplicate entires and a simple insert sort function. The selection of an appropriate sorting algorithm is left up to the reader.

With that, all that is left is to define an interpretation of NRC_{STX} using lists as the underlying structural datatype.

Definition 8.
In Figure 3.2 is a function that interprets a term t of NRC_{STX} using lists as its underlying domain of semantics. As we can see, the non-set-like operations are mostly trivial transformations into Coq's language, with the exception of the variable case which ensures that the interpreted variable is normalised. Defining a singleton in `nrc_sing` is a matter of definiting a list with one element. Defining a small cup operation is a matter of normalising the results

```
Fixpoint interpStxEff {X : Type}
                      {Σ : X → nrc__type}
                      {τ : nrc__type}
                      (t : nrc__stx Σ τ)
                      (ρ : ∀ x,
                        interpSortEff (Σ x))
                      : interpSortEff τ :=
  match t with
    nrc_var   Σ x         ⇒ normaliseInterpSortEff (ρ x)
  | nrc_pair  Σ τ σ t u   ⇒ ((interpStxEff t ρ), (interpStxEff u ρ))
  | nrc_proj1 Σ τ σ t     ⇒ fst (interpStxEff t ρ)
  | nrc_proj2 Σ τ σ t     ⇒ snd (interpStxEff t ρ)
  | nrc_empty Σ τ         ⇒ nil
  | nrc_unit  Σ           ⇒ tt
  | nrc_sing  Σ τ t       ⇒ (interpStxEff t ρ :: nil)
  | nrc_cup   Σ τ t u     ⇒ normaliseListRepr
                              (lexOrderInterpSortEff _)
                              (interpStxEff t ρ ++ interpStxEff u ρ)
  | nrc_bigcup Σ τ σ t u ⇒ normaliseListRepr
                              (lexOrderInterpSortEff _)
                              (concat
                                (map
                                  (λ x ⇒ interpStxEff u (ρ & x))
                                  (interpStxEff t ρ)))
  | nrc_diff   Σ τ t u    ⇒ listSubtr
                              (decEqInterpSortEff _)
                              (interpStxEff t ρ)
                              (interpStxEff u ρ)
  end.
```

Figure 3.10: Interpretation of NRC_{STX} using list.

of a list append operation (++). A big cup operation more closely reminds us of the intuition to big cup provided in Equation 2.2, where we concatenate the results of individual function applications to the fixed set term u, with an interpretation function enriched by x in the same manner that we enriched the context previously. Finally, defining the difference of two NRC terms is left to a generic list subtraction function that removes all instances of the elements in the first argument from the second argument. This is done with an assumption of decidable equality over our given variable type X. As the difference operation is subtractive, it does not affect normal form.

We have now defined the notion of list semantics for NRC_{STX}. As a small corollary to both our works on semantics, we unify them through a proof of compatibility. First, we define a function called `eff2Prop` which converts an effective list semantic representation of NRC_{STX} to a $Prop$-based one.

Definition 9.
We define a function `eff2Prop` that converts a member of type NRC_{TYPE} in list semantics to $Prop$ semantics in Figure 3.11. We write an explicit return for the pattern match to assist Coq's typechecker.

```
Fixpoint eff2Prop {τ : sort}
                  : interpSortEff τ → interpSortProp τ :=
  match τ return interpSortEff τ → interpSortProp τ with
    ι     ⇒ λ x    ⇒ x
  | 1     ⇒ λ x    ⇒ tt
  | τ × σ ⇒ λ x    ⇒ let (x1, x2) := x
                     in  (eff2Prop x1, eff2Prop x2)
  | β τ   ⇒ λ xs y ⇒ In y (map eff2Prop xs)
  end.
```

Figure 3.11: Conversion of list semantics to $Prop$ semantics.

The definition (type signature) of our proof of compatibility is found in Figure 3.12. We note that this proof requires both propositional and functional extensionality.

Definition 10.
Note that function composition ∘ is defined as follows: `f ∘ g` is equivalent to `λ x ⇒ f (g x)`.

Proof structure The proof `effPropCompat` is one of structural induction over an arbitrary term of type NRC_{STX}. The trivial cases are easily reduced down to statements that hold true by their reflexivity.

Some cases, such as the pair case, require the list semantics to normalise the underlying data, where normalisation is defined as a deduplication and sorting algorithm applied to the list in order to maintain set properties. For

```
Lemma effPropCompat {X : Type}
                    {Σ : X → nrc__type}
                    {τ : nrc__type}
                    (t : nrc__stx Σ τ)
                    (ρ : ∀ x,
                           interpSortEff (Σ x))
  : eff2Prop (interpStxEff t ρ) = interpStxProp t (eff2Prop ∘ ρ).
```

Figure 3.12: Semantic compatibility definition.

this, we have additional lemmas which prove that performing `eff2Prop` on normalised data is computationally equal to performing it on unnormalised data. We can see the name and type of this lemma in Figure 3.13.

```
  eff2Prop_normalise_compat : forall {τ : sort}
                                     (x : interpSortEff τ),
    eff2Prop (normaliseInterpSortEff x) = eff2Prop x.
```

Figure 3.13: Type of `eff2Prop_normalise_compat`.

Once we get into the set-based cases, such as singleton, we frequently come to a proof in which both sides of the equality use a lambda due to the definition of $Prop$-based semantics' set being that of a function from the set element to $Prop$. Under the assumption of functional extensionality[5] resolving this would not be an issue. To do this, we use the meta-language's functional extensionality tooling, allowing us to locally bind a new variable for cases when there are two lambdas at the top level of the term using the `extensionality` tactic. To give example, we present the proof state before and after applying the relevant functional extensionality and propositional extensionality, in that order in Figure 3.14.

In cases where we join sets, such as small cup, we are also forced to use propositional extensionality, as the notion of equality is too strong for terms in $Prop$. Thus, we separete our proof into two proofs of implication. Most of the proofs revolve around moving terms in and out of Coq's built-in `In` inductive structure, which is defined inductively based on whether or not a given term

[5]That is, a function is the same as another if they have the same outputs for the same inputs.

```
(* Proof state prior to applying tactics. *)
(fun y : interpSortProp τ =>
  In y
    (map eff2Prop
      (normaliseListRepr (lexOrderInterpSortEff τ)
          (interpStxEff t1 ρ ++ interpStxEff t2 ρ)))) =
(fun z : interpSortProp τ =>
  interpStxProp t1 (fun x : X => eff2Prop (ρ x)) z \/
  interpStxProp t2 (fun x : X => eff2Prop (ρ x)) z)

(* Proof state after applying tactics. *)
In z
  (map eff2Prop
      (normaliseListRepr (lexOrderInterpSortEff τ)
        (interpStxEff t1 ρ ++ interpStxEff t2 ρ))) <->
interpStxProp t1 (fun x : X => eff2Prop (ρ x)) z \/
interpStxProp t2 (fun x : X => eff2Prop (ρ x)) z
```

Figure 3.14: Before (above) and after (below) applying functional extensionality and propositional extensionality to the singleton set proof.

is at the head of the list, or somewhere along the tail of the list. [6]

The big cup case is, as expected, the most complex part of this proof. It involves proving the equivalence of a representation that is written using Coq's In structure, and a representation using an existentially quantified variable. We direct the reader to Appendix A for code of the proof, and to the repository for the proof in context of additional lemmas.

3.4 Extensions

We have now defined the syntax, as well as semantics for NRC. Now that the core constructions are in place, we exhibit a definition of a select function, such as the one found in the relational algebra and calculus. In order to arrive at defining this, we need to first introduce the notion of type remapping and binding in our queries.

[6] In other words, x is In a list L if it is either at the head of L, or it is In the tail.

Remapping types generally We start by showing our definition of a type map that correlates variables from one domain, X, with variables from a second domain, Y, to produce an equivalent interpretation using a new context.

Definition 11.
In Figure 3.15 we see that a `mapTy` function, given two types and two contexts that act on those types, is a dependent co-product construction. This means that given types X and Y, as well as contexts Σ and Γ, there will exist a function `f` that is capable of transforming `x : X` to `y : Y` in such a way that the evaluation of contexts Σ and Γ is equivalent.

```
Definition mapTy {X Y : Type}
                 (Σ : X → nrc__type)
                 (Γ : Y → nrc__type) :=
  sigT (λ f ⇒ ∀ x, Γ (f x) = Σ x).
```

Figure 3.15: Function mapping types alongside corresponding contexts.

We can see that if we add an extra type via $+1$ to our two types, we need only claim that remapping it leads to itself. This is shown in Figure 3.16 and proven mechanically in code.

Definition 12.
Adding a binder to a type map is defined as follows in Figure 3.16.

```
Definition map_add_binder {X Y : Type}
                          (Σ : X → nrc__type)
                          (Γ : Y → nrc__type)
  : mapTy Σ Γ → ∀ σ, mapTy (Σ & σ) (Γ & σ).
```

Figure 3.16: Adding binders to type maps.

Remapping in $\mathrm{NRC}_{\mathrm{STX}}$ With the preliminary definitions out of the way, we may now show how to remap types within terms of $\mathrm{NRC}_{\mathrm{STX}}$ in Figure 3.17.

Definition 13.
Due to the size of this definition, we have minimised the amount of necessary explicit arguments. Arguments deemed missing when comparing with previous definitions may be considered implicitly inferred[7].

Variable `nrc_var` may seem a little confusing. To explain more verbosely, `projT2` is a function that extracts the right element of a dependent coproduct. In this case, the dependent coproduct is an ordered pair `(f,p)` where `f` translates elements of `X` to elements of `Y` such that `p` holds. In this case, `p` is the statement `Γ (f x) = Σ x`. Our directive `match ... in ...` informs the Coq type checker that the shape of types we destruct is roughly equal to something equal to `varSort`. Our directive `match ... return ...` informs the Coq type checker that the output type will be `nrc__stx Γ varSort`. Finally, as there is only one shape that our proposition can take (that is, an equality), we can destruct it with the `eq_refl` constructor for equalities. Now that we have this equality destructed, Coq will not raise an error when type checking whether or not the given `x` with an `f` applied to it corresponds to the original `x` given.

The other cases are fairly trivial transformations of NRC syntax to accommodate newly remapped types. For big cup we use `map_add_binder`, which we shortened to `mapAB`, as context enrichment is guaranteed to occur.

Binding terms We define an operation that binds a context of a new type to an NRC expression in Figure 3.18.

Much like in the case of mapping types, we need to also adjust for the case of enriched contexts by claiming that a bind has no impact on additional binders added via $+1$. This is shown in Figure 3.19.

We show the definition of a bind operator to be an application of `bindTy` from Figure 3.18 to a given variable of the NRC term. Once again, `bind_add_binder` is shortened to `bindAB` for the sake of brevity.

[7] Additionally, the full Coq version of this code requires `{struct t}` as an argument to tell the Coq type checker that `t` is the structural fixed point.

```
Fixpoint map {X Y}
             {Σ : X → nrc__type}
             {Γ : Y → nrc__type}
             {τ : nrc__type}
             (f : mapTy Σ Γ)
             (t : nrc__stx Σ τ)
             : nrc__stx Γ τ :=
  match t with
    nrc_var    x      ⇒ match (projT2 f x) in (_ = varSort)
                             return nrc__stx Γ varSort with
                           eq_refl ⇒ nrc_var ((projT1 f) x)
                         end
  | nrc_pair   p1 p2  ⇒ nrc_pair   (map f p1)
                                   (map f p2)
  | nrc_proj1  x      ⇒ nrc_proj1  (map f x)
  | nrc_proj2  x      ⇒ nrc_proj2  (map f x)
  | nrc_unit          ⇒ nrc_unit
  | nrc_empty         ⇒ nrc_empty
  | nrc_sing   x      ⇒ nrc_sing   (map f x)
  | nrc_cup    x y    ⇒ nrc_cup    (map f x)
                                   (map f y)
  | nrc_bigcup σ x y  ⇒ nrc_bigcup (map f x)
                                   (map (mapAB Σ Γ f σ) y)
  | nrc_diff   x y    ⇒ nrc_diff   (map f x)
                                   (map f y)
  end.
```

Figure 3.17: Remapping members of NRC_{STX}.

3.4.1 Select

We show the construction of a select operator using predicates over NRC, referred to as NRC_{PRED}.

Definition 14.
As we can see in Figure 3.21, a predicate over NRC can be defined as an NRC term with a boolean–like value of `{}` or `{()}` (as explained in Section 3.1) with a context that is enriched by the data we wish to apply the predicate to.

Definition 15.
Our definition of the syntax-level `select` operator is written using Coq's proof mode, where we apply tactics that alter the state of the proof in a way closer to common logical reasoning. The definition can be found in Figure 3.22.

```
Definition bindTy {X Y : Type}
                  (Σ : X → nrc__type)
                  (Γ : Y → nrc__type) :=
  ∀ (x : X), nrc__stx Γ (Σ x).
```

Figure 3.18: Binding new types to an NRC_{STX} expression.

```
Definition bind_add_binder {X Y : Type}
                           (Σ : X → nrc__type)
                           (Γ : Y → nrc__type) :
  bindTy Σ Γ → ∀ σ, bindTy (Σ & σ) (Γ & σ).
```

Figure 3.19: Adding binders to type bindings.

The proof begins by unfolding the definition of `nrc_pred` of the given argument `p`. This lets Coq (and us) interactively see what `p` is defined as in terms of `nrc__stx`. The definition of `nrc_pred` can be found in Figure 3.21. We then apply `nrc_bigcup` twice, attempting to produce an object of type `nrc__stx Σ (β τ)`.

The first application utilises `t` as the binding argument that the operation iterates over, with the second argument being our goal to solve. The second utilises `p` as the binding argument, and the second argument once again needs to be solved for.

Now we need to prove the existence of an NRC term with context `Σ & τ & 1`, producing an element of `β τ`. We assert the existence of a map from our original `Σ` to this `Σ & τ & 1`. This is possible because we can extract `Σ` out of this enriched context via applying constructors of `option`, i.e. `Some` in this case.

Finally, we apply this map to our term and are left to provide the 'origin' of this map, which is our term `t`. `p` as the binding argument.

```
Fixpoint bind {X Y : Type}
              {Σ : X → nrc__type}
              {Γ : Y → nrc__type}
              {τ : sort}
              (f : bindTy Σ Γ)
              (t : nrc__stx Σ τ)
              : nrc__stx Γ τ ≔
  match t with
    nrc_var    x     ⇒ f x
  | nrc_pair   p1 p2 ⇒ nrc_pair  (bind f p1)
                                 (bind f p2)
  | nrc_proj1  x     ⇒ nrc_proj1 (bind f x)
  | nrc_proj2  x     ⇒ nrc_proj2 (bind f x)
  | nrc_unit         ⇒ nrc_unit
  | nrc_empty        ⇒ nrc_empty
  | nrc_sing   x     ⇒ nrc_sing  (bind f x)
  | nrc_cup    x y   ⇒ nrc_cup   (bind f x)
                                 (bind f y)
  | nrc_bigcup σ x y ⇒ nrc_bigcup σ (bind f x)
                                    (bind (bindAB Σ Γ f σ) y)
  | nrc_diff   x y   ⇒ nrc_cup   (bind f x)
                                 (bind f y)
  end.
```

Figure 3.20: Binding a new $\mathrm{NRC}_{\mathrm{STX}}$ expression.

```
Definition nrc__pred {X : Type}
                     (Σ : X → nrc__type)
                     (τ : nrc__type) ≔
  nrc__stx (Σ & τ) (β 1).
```

Figure 3.21: Definition of a predicate over an $\mathrm{NRC}_{\mathrm{STX}}$ term.

```
Definition nrc_select {X : Type}
                      {Σ : X → nrc__type}
                      {τ : nrc__type}
                      (t : nrc__stx Σ (β τ))
                      (p : nrc__pred Σ τ)
                      : nrc__stx Σ (β τ).
unfold nrc__pred in p.
refine (nrc_bigcup _ t _).
refine (nrc_bigcup _ p _).
assert (H_map : mapTy Σ ((Σ & τ) & 1)).
{ unfold mapTy.
  exists (λ x ⇒ Some (Some x)).
  intros.
  reflexivity. }
refine (map H_map _).
apply t.
Defined.
```

Figure 3.22: Syntax-level select operator.

Chapter 4

Evaluation

In this chapter we discuss computable examples built using our model of NRC, show their code, and reflect on limitations.

4.1 Encoding concrete data

In the NRC formalisation that is presented prior, we sacrificed the ability to bind multiple types for a simpler, unityped presentation. All terms of NRC require a background type that they are interpreted within, represented as $X : Type$. With this information being carried around by the syntactic terms, an interpretation of an NRC_{STX} term is simply a recovering of some $x : X$ that fits within the position of a ι urelement. This carries with it a considerable issue: if X is only allowed to be one type, we do not allow a complex term, e.g. $\iota \times \iota$, to have different types for each instance of ι. In other words, all our attributes in a given relation are forced to share a type.

To solve this issue, there exist many solutions, but we admit that all of them are quite ad-hoc with respect to our existing structure. A proper resolution would require a ground-up rewrite of the typing system. We may solve this by passing in a product type as our type background for interpretation, but by doing so we have essentially forfeited the usage of product types in NRC sorts, and we have also lost meaning in the notion of an NRC atom. We could sim-

ply choose to add individual concrete base types into our NRC sorts, but this contaminates the NRC sorts with specification-oriented information. A minor adjustment to this approach, however, leads us to a solution that we find the least destructive in the broad picture.

For this specific chapter, we have decided to add a new construction in the form of base types for Booleans, natural numbers, and strings. To sew this figurative patch of types onto our existing NRC fabric, we add a minor change of definition for what a NRC_{TYPE} is, in the form of parametrising an ι, or `sort_atom`, as it is in code, by one of our base types $\mathcal{B}$, or `baseTy`, as it is in code. To then interpret a `baseTy`, we define an $interpBase$ function that translates this variant type into concrete Coq types such as `nat`, `string`, and `bool`.

```
Variant baseTy : Type := ty_str | ty_nat | ty_bool.
Definition interpBase (b : baseTy) : Type :=
  match b with
    ty_str => string
  | ty_nat => nat
  | ty_bool => bool
  end.
Inductive sort : Type :=
    sort_atom : baseTy -> sort
  | sort_unit : sort
  | sort_pair : sort -> sort -> sort
  | sort_pow  : sort -> sort.
```

The pattern exhibited here, of a central `baseTy` that carries types interpretable in the host languages, is called a 'universe pattern', which was introduced in Oury and Swierstra [28].

This change requires modifying proofs that refer to the background type X (or, when in variable form, usually referred to as U), and instead redirecting them to the three given base types. This means we had to prove new proofs regarding decidable equality and a total order with respect to the $\leq$ operator.

With these changes made, we can move on to show examples of Coq data being interpreted into the languge of NRC sorts, NRC_{TYPE}. We show three examples of data being hardcoded into memory as an interpreted form of NRC_{TYPE}, with the first example being the same that can be found in Chapter 1.

```
Definition nrc_example_1 :
  interpSortEff (β (ι ty_str × ι ty_str × ι ty_nat)) :=
    ("EMP1", "Santana", 100000) :: ("EMP2", "Martyn", 150000) ::
    ("CTR1", "Chiharu", 500000) :: ("CTR2", "Parveen", 200000) ::
    nil.
Definition nrc_example_2 :
  interpSortEff (β (β (ι ty_nat) × ι ty_str × ι ty_bool)) :=
    ((10 :: 29 :: 54 :: nil), "Project Iris", true) ::
    ((10 :: 94 :: 95 :: 96 :: nil), "Project Cornea", false) ::
    ((nil), "Project Sclera", true) ::
    nil.
Definition nrc_example_3 :
  interpSortEff (β (ι ty_str × ι ty_nat × ι ty_nat)) :=
    ("CTR1", 10, 5000) :: ("CTR1", 29, 1000) :: ("EMP2", 94, 500) ::
    ("EMP2", 95, 500) :: ("EMP2", 96, 500) ::
    nil.
```

Figure 4.1: Three examples of Coq data being encoded as interpreted NRC_{TYPE}.

We believe that the unityped design was fine, and perhaps even ideal for proving properties of NRC due to its inherent simplicity, but was also one of the primary hurdles of realising practical computable examples that would show how NRC relates to real data structures one might see in databases. Given the chance to start the project from scratch, I believe one needs to reflect on whether or not they intend to use the formulation for the purposes of mathematical formalism, or for the purposes of a usable programming language. Given the former was the original goal of this project, we would likely not change this, or at the very least, we would change the positioning of the type background in a way that is more expandable for future endeavours.

4.2 Syntactic terms

Having shown that it is possible to encode data in our NRC model, we now move on to provide examples showing that it is possible to encode entire NRC terms in it, as well as their semantic interpretations. We begin by showing encodings of syntactic terms.

To recall the definition of a term in NRC_{STX}, we need to provide an ambient interpretation type $X : Type$, as well as an ambient context $\Sigma : X \to sort$, which takes values of our interpretation type, and translates them into values of NRC_{TYPE}, or `sort`. For simplicity's sake, we will be using Coq's type of natural numbers `nat` as the 'namespace' for our variables. We provide a simple example context in Figure 4.2. As we can see, we make space for four variables of type `nat`, two variables of type `bool`, and the rest of the namespace (therefore, the rest of the natural numbers) corresponds to variables of type `string`.

```
Definition nat_ctxt (n : nat) : sort :=
  match n with
  | 0 => ι ty_nat  | 1 => ι ty_nat | 2 => ι ty_nat | 3 => ι ty_nat
  | 4 => ι ty_bool | 5 => ι ty_bool
  | _ => ι ty_str
  end.
```

Figure 4.2: Example type context.

This context gives us a way to identify variables (which will be natural numbers, as we have set out by the definition of $X := \texttt{nat}$) as things of specific types in the NRC language. With this, we can now show an example NRC query in Figure 4.3.

Due to the intrinsically typed nature of our syntax, the typechecker does not allow us to write ill-typed terms, as each term carries with it the conditions for well-typedness.

```
Definition nrc_stx :
  nrc__stx nat_ctxt (β (ι ty_str × ι ty_str × ι ty_nat)) :=
    (@nrc_cup _ nat_ctxt _
      (nrc_sing
        (nrc_pair (nrc_pair (nrc_var 8) (nrc_var 6)) (nrc_var 0)))
      (nrc_sing
        (nrc_pair (nrc_pair (nrc_var 9) (nrc_var 7)) (nrc_var 2)))).
```

Figure 4.3: Example NRC_{STX} query.

4.2.1 An interlude for bugs

We note that in Figure 4.3 we are forced to use the explicit representation of the `nrc_cup` term by means of the `@` symbol. This makes every argument explicit, even though in the file we have made most of the arguments implicit. The reason for this is due to a yet-to-be-explained bug with Coq that appears to cause the typechecker unable to normalise the NRC terms given after the definition delimiter `:=`. As a result of this, every term that should be normalised by means of being applied to the Σ ends up staying as it is, without being evaluated. Because the terms remain the same, Coq's typechecker does not compare the meaning of each `nrc_var` based on its underlying type, but rather based on equality of the presented constants.

To show a simplified example, we first assume that the natural number variable 1 maps to $(\iota\ \mathbb{N})$, that is, an NRC atom representing a natural number. We assume that the same applies to the natural number variable 2. In other words, $1 \mapsto (\iota\mathbb{N}) \wedge 2 \mapsto (\iota\mathbb{N})$. Now, when looking at the NRC term $\{1\} \cup \{2\}$, we see that, when the variables' types are examined, the two types of the cup-expression match (both are $(\iota\ \mathbb{N})$), and therefore the expression is legal. Coq, on the other hand, appears incapable of performing this reduction[1], and defaults to seeing whether or not the left-hand and the right-hand side of $\{1\} \cup \{2\}$ match. When examined without the knowledge of the context's mapping, we indeed can say that $1 \neq 2$, and reject the term. This entire issue is resolved by explicitly pro-

[1]To be specific, Coq is unable to infer the meaning of the function Σ that reduces variables to types. In code, this is expected to be inferred from the output type of the expression which we are writing.

viding the context Σ at the level of every set-based combinatorial expression, allowing Coq to deduce it in further nested expressions, and apply it to the 1 and 2 presented in the example above.

4.3 Semantically-interpreted terms

Now that we have created an example term, we finally show an NRC_{STX} term that is interpreted into common representable Coq types. We recall the definition of `interpStxEff`, our effective list interpretation. For it, we require an NRC term, alongside all of the things that form it (a background type for variable interpretaton, a context, and some arbitrary NRC type), and an interpretation function that can evaluate variables into concrete Coq values. We show this interpretation function in Figure 4.4.

```
(* ρ : forall x : X, interpSortEff (Σ x) *)
Definition interpFun : forall x : nat, interpSortEff (nat_ctxt x) :=
  fun s => match s with
           0 => 100000    | 1 => 150000   | 2 => 500000
         | 3 => 200000    | 4 => true     | 5 => false
         | 6 => "Santana" | 7 => "Martyn" | 8 => "EMP1"
         | 9 => "EMP2"    | _ => "Uncharted territory"
        end.
```

Figure 4.4: Value interpretation function.

The interpretation function is a dependent product that is indexed (depends on) an x : `nat`. This is our type of variables. Resulting from it is a value that must align with the type of passing x into `nat_ctxt`, and then into `interpSortEff`. Without the usage of a dependent type here we would be capable of producing, e.g. a `string` value for the variable 0, even though we can see according to our context in Figure 4.2, that the variable 0 maps to something of type $(\iota\,\mathbb{N})$, which is, in turn, interpreted by `interpSortEff` as `nat`.

With both a syntactic term and an interpretation function in hand, we may now easily show what the term evaluates to with the following definition in

Figure 4.5. We also show what the result of evaluating said term is in a comment below.

```
Definition nrc_example__stx := interpStxEff nrc_stx interpFun.

Compute (nrc_example__stx).
(* Result of computing nrc_example__stx:
   = ("EMP1", "Santana", 100000) :: ("EMP2", "Martyn", 150000) ::
     nil
   : interpSortEff (β (ι ty_str × ι ty_str × ι ty_nat))
*)
```

Figure 4.5: Example of interpreted NRC_{STX} term.

Set multiplicity Because we use set semantics, adding extra 3-tuples to our existing set in the above examples that match exactly another existing one will not change the output. To explain with a simpler example, a query calling for $\{1\} \cup \{1\} \cup \{2\}$ will be interpreted as $\{1, 2\}$, rather than $\{1, 1, 2\}$. This is due to the normalisation function found in Figure 3.8.

Ease of writing Due to the aforementioned Coq bug, as well as an intrinsic typing discipline, we believe that writing NRC queries in this system is rather tedious. This can be accelerated with the use of Coq's notations, allowing us to specify custom-symbol operators for each query operation. We believe that for purposes of better understanding of the material, it would have been wise to maintain a Coq file that acts as a playground for the currently-developed features of NRC, which would focus on ergonomic aspects of the language as well.

Name shadowing A worry that may come up when examining NRC is for the behaviour of name shadowing in the context of reusing attribute names for other purposes. Due to us not having named attributes, and due to our implementation of NRC not having internal names be definable from within the syntax, this is an issue that is entirely a question for Coq's attitude towards

name shadowing. Once a name in Coq is defined, it is immutable and not redefinable, thus shadowing is prevented.

4.4 Select operator

Due to there not being enough time to finish the `select` operator, we can not provide a full example with semantics. On the other hand, we can provide an example, purely syntactic term, that we can annotate with data that we could potentially use for semantic interpretation. We apologise for the complexity of the following explanation, as it is difficult to speak of abstract syntactic terms without concrete semantics.

```
Definition nrc_select_atom__stx :
  nrc__stx (/0 & β (ι ty_str × ι ty_str) & ι ty_str)
           (β (ι ty_str × ι ty_str)) :=
  nrc_select (@nrc_var _
               ((/0 & β (ι ty_str × ι ty_str)) & ι ty_str)
               (Some None))
             (@nrc_equal _
               ((((/0 & β (ι ty_str × ι ty_str))
                     & ι ty_str)
                     & (ι ty_str × ι ty_str)) _
               (nrc_var (Some None))
               (nrc_proj1 (nrc_var (None)))).
```

Figure 4.6: Select over NRC_{STX} term without semantics.

The term in Figure 4.6 can be understood in prose as a term that selects only the left projection of a set of terms $\beta(\iota \times \iota)$, so long as the equality predicate holds between a term of type ι, and the left projection of the aforementioned set $\beta(\iota \times \iota)$. Thus, one interpretation (the one this term is designed around), is that we are given something of type ι, and we 'browse' a data structure of type $\beta(\iota \times \iota)$, only returning whenever the predicate matches the left ι of each of the data structure's tuple with the given input ι.

We now give a technical and thorough examination of the specified term. Starting from the Coq type, fix a term in NRC_{STX} that uses the empty context $\emptyset$, or `/0`. This context is enriched by a bound term of type $\beta(\iota \times \iota)$, and is then enriched once again by a bound term of type ι. For the purposes of brevity, let us call this context ξ.

The resulting NRC type structure of this term will be that of $\beta(\iota \times \iota)$. The concrete meaning of such a syntactic term is defined by means of the `nrc_select` operator. The term subject to the select operator is a variable that is retrieved from the ξ context, more precisely, it is the part of the context that refers to something of type $\beta(\iota \times \iota)$. It is over this term that the predicate acts. The predicate then proceeds to check the equality of a variable term of the form of ι, selected using `option` constructors from the enriched context, going from right to left. That is, it selects the second element of the predicate's triple-enriched context from the right, which is something of type ι. The thing compared against this ι is something that is a variable retrieved from a left projection of a pair of $(\iota \times \iota)$, thus it, too, is an ι. This is to be expected, as we expect to provide an equality test something of matching types. In this case, the thing being compared is something of an atomic sort ι.

Chapter 5

Related Work

In this chapter we discuss works that relate to the notions we mechanically formalised.

5.1 Wisnesky

Much of Wisnesky's 2013 thesis, *'Functional Query Languages with Categorical Types'* utilises NRC, but it is not the subject of study. Their interest lies primarily in using NRC for its inherent similarities with Higher-order Logic (HOL), as one of the work's goals is the exploration of HOL as a querying language.

Because our focus is on NRC and its intricacies as expressed in Wong's thesis, we take a conservative approach to adding new structures and definitions that change the expressivity of NRC at the cost of other beneficial properties. A notable addition to Wisnesky's formulation of NRC is the power set operator pow.

While it is possible to augment NRC in such a manner that the conservative extension property is upheld even with pow (see [34, Corollary 3.3.5]), their form of NRC does not contain the necessary constructions to do so. More concretely, we would need a summation operator, arithmetic operations $\cdot, \div, +, -$

and linear orders for base types. None of these are present in the core construction used by Wisnesky [32].

The second property we are interested in is that of the polynomial complexity bound on queries with respect to their size, as described in Section 2.3. Unfortunately, this property cannot be kept while the pow operator is in play. This is due to the exponential size of the output of a pow operation. Even if we decided to forgo the polynomial complexity bound, we would still be taking NRC beyond the scope of the classic relational model, thus rendering any work on connecting the two useless without extra reconciliation steps.

Similarly to us, Wisnesky has a Coq formalisation of the NRC they used [32, Section 3.13]. This formalisation does not include context manipulation, with the assumption that it will be used in a pipeline from HOL to the relational calculus. By contrast, our definition of NRC is more agnostic towards input.

5.2 Wong

Wong's doctoral thesis is a large body of work describing their journey creating the Kleisli system and the associated CPL language. For us, the results of Wong define contemporary NRC and unify previous works on nested relations. Work that treats NRC as the centre of the stage primarily appears in the second part. We note that this work discusses multiple instances of the NRC family, whereas we are interested in a singular interpretation of NRC. Our interpretation is closest to what the literature refers to as $\mathcal{NRC}(-)$. This refers to NRC enriched with the set difference operator.

Our version of NRC is a conservative extension of the core NRC. This is shown by Wong in [34, Theorem 2.4.2], where Wong shows expressive equivalence of $\mathcal{NRC}(=)$ and $\mathcal{NRC}(-)$, among others. We also know that adding operators such as $=$ and $-$ does not take NRC out of polynomial time.

We also know that our implementation using simulated booleans is equivalent to $\mathcal{NRC}(\mathbb{B})$ according to Wong [34, Corollary 2.3.6], i.e. $\mathcal{NRC} = \mathcal{NRC}(\mathbb{B})$.

According to Wong [34, Claim 2.5.5], $\mathcal{NRC}(\mathbb{B}, =)$ is the 'right' nested relational language. We note that while our form of it is not the same, it is by all means expressively equivalent, as shown by the above two paragraphs. We use this form of NRC as our core because of its simplicity and powerful expressivity. For an equal project with a bigger timeframe that would allow the inclusion of function types, we believe that $\mathcal{NRC}(\mathbb{B}, =)$ would be a more appropriate choice.

5.3 Language-integrated querying

Wong used NRC as a stepping stone that forms the basis of Kleisli and CPL for their domain-specific purposes (optimisation of queries in genetic databases). Wisnesky used NRC as an embedded querying language that functions as a translation layer between an alternative theory, such as higher-order logic. This general approach to embedding querying languages comes from Meijer, Beckman, and Bierman [27], at that time working at Microsoft, who created the notion of the Language-Integrated Query, or LINQ. It is a system for writing queries in a language such as SQL or XML (there exist a wide variety of LINQ styles), while staying within the original host language (originally C# and Visual Basic). While the formal theory behind LINQ has not fully crystallised in the public eye at this time of the paper's release, there was considerable effort to produce one. A materialisation of these efforts can be found in Cheney, Lindley, and Wadler [8], which describes a formal model heavily inspired by NRC.

We believe that LINQ was a great success for alleviating the mismatch between programming languages and database querying languages, but its results do not come with mechanically proven guarantees of correctness, nor do they guarantee any sort of query minimality or optimisation. LINQ, much like other papers which do not focus on NRC by itself, also does not do justice to

a thorough mechanical examination of NRC. This is, obviously, because NRC was not the primary subject of examination when designing LINQ. We do, however, believe that examining papers on LINQ is a helpful way to gain a better understanding of what NRC can do in a practical setting. As such, we also recommend Fehrenbach and Cheney [19] as an exemplary show of NRC's capabilities within the realm of the LINKS programming language.

5.4 Relational models & SQL

There are a number of projects that work on the relational calculus, relational algebra and SQL. All of these are relevant to our work on NRC due to the relationship between the families of relational calculi. We exhibit Chu et al.'s HoTTSQL [11], Malecha et al.'s verified relational database management system [25], Ricciotti and Cheney's formalisation of SQL with nulls (NullSQL) [30], and Benzaken, Contejean, and Dumbrava's SQL_{Coq} [5].

5.4.1 HoTTSQL

HoTTSQL is a novel formalisation of SQL using a modern approach – homotopy type theory. Specifically, HoTTSQL uses univalent types that carry proofs of equality and membership with them. The main goal of HoTTSQL is to provide an easy to use form of SQL for verifying query optimisation. As such, it comes with tooling to easily check two SQL queries' equality. To allow this, the work generalises th of a relation to allow infinite attributes and, in the case of multisets, to allow tuples with infinite multiplicity.

HoTTSQL is aware of Malecha et al. [25], as well as Benzaken, Contejean, and Dumbrava [6]. It shows a considerable improvement in the complexity of proving query equality for the purpose of optimisation.

- Its semantics allow both sets and bags (multisets), with the infinitary caveat mentioned prior.
- It is capable of representing nested relations.

- It is implemented in Coq, with the addition of the univalence theorem to enable univalent types.

Given that the paper states HoTTSQL is capable of representing nested relations[1], we believe that there is an alleyway for NRC to be constructed within a univalent framework; however, writing a formalisation of NRC in such a manner would require a full rewrite from the ground-up, and thus is not feasible for the current software artefact presented by this project.

5.4.2 Verified RDBMS

We show Malecha et al.'s verified relational database management system, or RDBMS [25]. It is a fairly comprehensive work that goes into providing a full relationally equivalent model which functions as a foundation for other components of an RDBMS, namely a query parser, query optimiser, a database engine that manipulates a concrete underlying data structure (here it's B+ trees), and storage method of said structure. All of these are verified and written in Coq.

The work does not concern itself with an abstract, generalised notion of querying the way we do. It imposes a strict structure, coupling semantic denotation with syntax depicting schemata of concrete types which correspond to Coq's internal types. Because said types do not come with decidable equality, and because the semantics are set-based, the work provides a requirement similar to ours in terms of only allowing types that have decidable equality and total ordering. They also impose a requirement of serialisability[2] of every type used in a schema to allow permanently storing data.

[1] The paper announces that it is capable of representing nested relations; however, it does not go deeper into the topic.

[2] Serialisability is defined as a proof of a function that translates objects of a given type into a string, which can be written to disc.

This work uses a lot of scaffolding work from Coq's FSet library[3], which we regretfully have not used in our project, as we believe that it could have eased some tedious tasks with regard to proving decidability and ordering. Simultaneously, if we were to take work on HoTTSQL into consideration, there is significant drawback in compatibility if we were to use FSet (finite sets), as HoTTSQL requires allowing a degree of transfinitism (see Section 5.4.1).

We believe that, while the work is much more extensive in terms of its goals (we formalise NRC queries only, while their work formalises an entire database management system), there is inspiration to be taken away from the practical approach. We think defining schemata as lists of types could be an avenue that is more concrete than our univariable approach of using an arbitrary type $X : Type$, but we simultaneously recognise that introducing new ad-hoc structures such as lists of types would detract from the generality and simplicity of a more abstract presentation that uses foundational elements of type theory such as product types. We believe that anyone who may be interested in implementing a relational model in Coq should pay close attention to the design decisions of their type system implementation with respect to their needs.

The work addresses an interesting point, which we wish to emphasise in this work to motivate further action.

'Unfortunately, it can often take significant time to fully understand the Coq-engineering consequences of a seemingly inconsequential design choice; for instance, whether to represent schema as lists of type names or as functions from column numbers to type names. At present it is unclear how best to explore the design space.' (Malecha et al. [25, Section 3.3])

In other words, the full extent of the impact of implementation decision minutiae is not properly identified, and there is more desire for work on formalising database systems, as well as coming up with a good meta-analysis of methodologies for formalising relational models.

[3]With the caveat that the authors reworked a lot of it to be first-class with respect to type-checking and compilation.

5.4.3 NullSQL

Work by Ricciotti and Cheney [30] shows that many of the aforementioned formalisations lack a critical point that is key to formalising SQL, that being the presence of a three-valued logic using NULLs in SQL. SQL commonly uses NULL to represent the non-presence of a value, and it is enabled by default unless the programmer chooses to disabled it. As a result, a lot of attempts at formalising SQL do not fully capture the correctness errors that may occur in the case of NULL values combining with logical predicates that the user assumes Boolean logic for. The authors of NullSQL present a Coq formalisation that tackles this issue, and it discusses the difficulties of adding such functionality to previously written formalisations.

For instance, source work by Wong simply does not discuss the potential for three-valued logic to be included [34]. Work by Chu et al. currently appears to be quite incompatible with the addition of three-valued logic, or at the very least requires extensive work [11]. Malecha et al. [25] does not provide three-valued logic semantics, and only Benzaken and Contejean's SQL_{Coq} tackles the subject [5] prior to this work.

We believe that applying the knowledge from this work would also require a deeper rework of NRC's core semantics in order to accommodate NULL values, and as such we think that it would be difficult to accomplish without a ground-up rewrite, much like HoTTSQL in Section 5.4.1. If such a rewrite were to be attempted, we highly recommend reviewing Cheney and Ricciotti [10], which proposes two models of NRC with NULL values: one realised with option types ('explicit NULLs'), and one realised with NULL as an extra value added to every base type ('implicit NULLs').

5.4.4 SQL_{Coq}

A contemporary result in the formalisation of the relational model and SQL is Benzaken and Contejean's SQL_{Coq}. To be precise, there exists previous work

on the relational model being formalised in Coq, as seen in Benzaken, Contejean, and Dumbrava [6]. After this work, we see the release of Benzaken and Contejean [5]. The latter is a much more specific and realistic implementation of SQL queries which includes the majority of common SQL statements, such as aggregation and group-by statements, as well as the aforementioned `NULL` values. The former is a work describing more pure aspects of the relational model, being closer in line to Wong's work.

Benzaken and Contejean [5] uses a piecemeal approach to describing queries, with many structures being described in their own `Inductive` Coq statements. In contrast to our work, the syntax does not appear to be defined with a type context in mind, and the only usage of an arbitrary type parameter is to provide a universe of discourse for formulae. The type context is defined separately and then fed into a typechecking function. In other words, a large portion of SQL_{Coq} uses an extrinsic typing paradigm, which contrasts with our project.

Chapter 6

Conclusion & Future Works

This section discusses directions in which the project can be taken, as well as potential future results we anticipate.

6.1 Full implementation of Δ_0 formulae

We may be interested in exploring an implementation of formulae within NRC because of the inherent need to provide reasoning over predicates in a more practical setting, such as SQL. Different formalisms of the relational model have approached formulae in different ways [11, 25, 30, 5]. As such we, too, are given many options to choose from.

For our current work, in Section 3.4.1 we discuss a `select` operator written in NRC. We have chosen to define a predicate NRC_{PRED} as an NRC syntactic term that has a boolean-like content (simulated booleans with $unit$) and its context is enriched by the term we are interested in. This is a rather impractical approach for writing `select` queries, as the predicate always has to be custom-made for our requirements using core NRC terms.

This shortcoming can be alleviated by broadly implementing predicates in the form of a conversion between Δ_0 formulae and elements of NRC_{PRED}. Previous work on Δ_0 formulae within NRC exists in Benedikt and Pradic [4]. The

authors show that, given a proof of functionality, Δ_0 formulas are capable of implicitly defining transformations within NRC. We believe that this result can be constructed in our language as well.

6.2 NRC plus the relational model

According to Wong [34], it is possible to simulate relational calculus queries with flattened nested relational ones. Wisnesky [32] produces a pipeline that transforms NRC into the language of the relational calculi. Additionally, Cheney, Lindley, and Wadler [9] show that nested queries can be 'shredded' into multiple flat SQL queries. Because of this result, we hypothesise that a transformation between NRC and SQL queries can be encoded with further improvements to our model. As of writing, we do not believe there are constructive proofs that fully connect these models through said means.

All of these results go to show that the nested relational languages and the flat relational languages have a close connection, and many examples exist that bridge this connection. Our work shows potential for another form of bridging on the side of mechanised proofs and programming language design utilising dependently typed languages. By now, we have examined several existing relational model formalisations in Section 5.4, and we leave the reader with an open window peering into the potential for joining NRC and the relational model.

We believe that given more time and development, one of the final milestones of this project would be proving existing results of NRC with respect to one of these formalisms. This would include, but not be limited to proofs of polynomial query size bounds, conservative extension properties that span the nested and flat calculi and verification of database systems that utilise SQL indirectly, through the use of a conversion pipeline in the style of Wisnesky, or a generalised database querying model such as Kleisli in the style of Wong. We cannot comment on the precise nature of such a connection because of the wide variety of design decisions that have been put into previous formalisms.

6.3 Operational semantics

This section expects a baseline understanding of semantics in programming languages, specifically the difference between denotational and operational semantics. For a resource on the semantics of programming language we recommend *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics* by Hennessy [23].

All our semantics in Section 3.3 are denotational in nature, since they do not refer to a particular interpretation of language elements as changes in the state of the program, but rather changes of underlying modelled data structures.

Denotational semantics are useful for understanding NRC intuitively, without excessive worries about evaluation order and state management. Denotational semantics are not useful for a reference implementation of NRC in a compiler. We currently rely on Coq's underlying data structures, and even if we were to extract our code into the programming language OCaml, we would rely on OCaml's underlying data structures. This does not give us much strength in terms of optimising queries, which is a strong component of database theory and query studies.

Operational semantics (small-step and big-step) would allow us to formally define NRC and its operations as direct changes of state to the program. We also anticipate that operational semantics could allow for more precise complexity bounds, since we are allowed to calculate the cost of operations.

For a reference on operational semantics for NRC, we point to Fehrenbach and Cheney [19] (small-step) and Fowler, Galpin, and Cheney [22] (big-step). Both of these works use the programming language LINKS, which takes a lot from NRC when it comes to handling database queries (the language's purpose, however, is wider than just for querying databases).

Bibliography

[1] Brian Aydemir et al. 'Engineering Formal Metatheory'. In: *SIGPLAN Not.* 43.1 (Jan. 2008), pp. 3–15. ISSN: 0362-1340. DOI: 10.1145/1328897.1328443. URL: https://doi.org/10.1145/1328897.1328443.

[2] Henk Pieter Barendregt. 'Introduction to Lambda Calculus'. In: *Nieuw Archief voor Wiskunde* 4.2 (1984), pp. 337–372. ISSN: 0028-9825. URL: https://hdl.handle.net/2066/17289.

[3] Henk Pieter Barendregt, Wil Dekkers, and Richard Statman. 'Lambda Calculus with Types'. In: *Cambridge University Press* (June 2013). DOI: 10.1017/cbo9781139032636. URL: https://cir.nii.ac.jp/crid/1361418520366711680.

[4] Michael Benedikt and Cécilia Pradic. 'Generating Collection Transformations from Proofs'. In: *Proceedings of the ACM on Programming Languages* 5 (POPL Jan. 2021), pp. 1–28. DOI: 10.1145/3434295. URL: https://doi.org/10.1145%2F3434295.

[5] Véronique Benzaken and Évelyne Contejean. 'A Coq Mechanised Formal Semantics for Realistic SQL Queries: Formally Reconciling SQL and Bag Relational Algebra'. In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. Cpp 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 249–261. ISBN: 978-1-4503-6222-1. DOI: 10.1145/3293880.3294107. URL: https://doi.org/10.1145/3293880.3294107.

[6] Véronique Benzaken, Évelyne Contejean, and Stefania Dumbrava. 'A Coq Formalization of the Relational Data Model'. In: *Programming Lan-*

guages and Systems. Ed. by Zhong Shao. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 189–208. ISBN: 978-3-642-54833-8.

[7] Peter Buneman et al. 'Principles of Programming with Complex Objects and Collection Types'. In: *Theoretical Computer Science* 149.1 (1995), pp. 3–48. ISSN: 0304-3975. DOI: 10.1016/0304-3975(95)00024-Q. URL: https://www.sciencedirect.com/science/article/pii/030439759500024Q.

[8] James Cheney, Sam Lindley, and Philip Wadler. 'A Practical Theory of Language-Integrated Query'. In: *SIGPLAN Not.* 48.9 (Sept. 2013), pp. 403–416. ISSN: 0362-1340. DOI: 10.1145/2544174.2500586. URL: https://doi.org/10.1145/2544174.2500586.

[9] James Cheney, Sam Lindley, and Philip Wadler. 'Query Shredding: Efficient Relational Evaluation of Queries over Nested Multisets'. In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 1027–1038. ISBN: 978-1-4503-2376-5. DOI: 10.1145/2588555.2612186. URL: https://doi.org/10.1145/2588555.2612186.

[10] James Cheney and Wilmer Ricciotti. 'Comprehending Nulls'. In: *The 18th International Symposium on Database Programming Languages*. Dbpl '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 3–6. ISBN: 978-1-4503-8646-3. DOI: 10.1145/3475726.3475730. URL: https://doi.org/10.1145/3475726.3475730.

[11] Shumo Chu et al. 'HoTTSQL: Proving Query Rewrites with Univalent SQL Semantics'. In: *SIGPLAN Not.* 52.6 (June 2017), pp. 510–524. ISSN: 0362-1340. DOI: 10.1145/3140587.3062348. URL: https://doi.org/10.1145/3140587.3062348.

[12] Alonzo Church. 'A Set of Postulates for the Foundation of Logic'. In: *Annals of Mathematics* 33.2 (1932), pp. 346–366. ISSN: 0003486X. DOI: 10.2307/1968337. JSTOR: 1968337. URL: http://www.jstor.org/stable/1968337 (visited on 11/17/2023).

[13] E. F. Codd. *The Relational Model for Database Management: Version 2*. USA: Addison-Wesley Longman Publishing Co., Inc., 1990. ISBN: 0-201-14192-2.

[14] Edgar Frank Codd. 'A Relational Model of Data for Large Shared Data Banks'. In: *Commun. ACM* 13.6 (June 1970), pp. 377–387. ISSN: 0001-0782. DOI: 10.1145/362384.362685. URL: https://doi.org/10.1145/362384.362685.

[15] George Copeland and David Maier. 'Making Smalltalk a Database System'. In: *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*. Sigmod '84. New York, NY, USA: Association for Computing Machinery, 1984, pp. 316–325. ISBN: 0-89791-128-8. DOI: 10.1145/602259.602300. URL: https://doi.org/10.1145/602259.602300.

[16] George Copeland and David Maier. 'Making Smalltalk a Database System'. In: *Sigmod Record* 14.2 (June 1984), pp. 316–325. ISSN: 0163-5808. DOI: 10.1145/971697.602300. URL: https://doi.org/10.1145/971697.602300.

[17] Thierry Coquand and Gérard Huet. *The Calculus of Constructions*. INRIA, May 1986. URL: https://inria.hal.science/inria-00076024.

[18] N.G de Bruijn. 'Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem'. In: *Indagationes Mathematicae (Proceedings)* 75.5 (Jan. 1, 1972), pp. 381–392. ISSN: 1385-7258. DOI: 10.1016/1385-7258(72)90034-0. URL: https://www.sciencedirect.com/science/article/pii/1385725872900340.

[19] Stefan Fehrenbach and James Cheney. *Language-Integrated Provenance*. 2017. arXiv: 1607.04104 [cs.PL]. URL: https://arxiv.org/abs/1607.04104.

[20] Amy Felty and Alberto Momigliano. 'Hybrid'. In: *Journal of Automated Reasoning* 48.1 (Jan. 1, 2012), pp. 43–105. ISSN: 1573-0670. DOI: 10.1007/s10817-010-9194-x. URL: https://doi.org/10.1007/s10817-010-9194-x.

[21] Simon Fowler, Vashti Galpin, and James Cheney. 'Language-Integrated Query for Temporal Data'. In: *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. Gpce 2022. New York, NY, USA: Association for Computing Machinery, 2022, pp. 5–19. ISBN: 978-1-4503-9920-3. DOI: 10.1145/3564719.3568690. URL: https://doi.org/10.1145/3564719.3568690.

[22] Simon Fowler, Vashti Galpin, and James Cheney. *Language-Integrated Query for Temporal Data (Extended Version)*. 2022. arXiv: 2210.12077 [cs.PL]. URL: https://arxiv.org/abs/2210.12077.

[23] Matthew Hennessy. *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics*. USA: John Wiley & Sons, Inc., 1990. ISBN: 0-471-92772-4.

[24] David Maier. *The Theory of Relational Databases*. Computer Science Press, 1983. ISBN: 0-914894-42-0.

[25] Gregory Malecha et al. 'Toward a Verified Relational Database Management System'. In: *SIGPLAN Not.* 45.1 (Jan. 2010), pp. 237–248. ISSN: 0362-1340. DOI: 10.1145/1707801.1706329. URL: https://doi.org/10.1145/1707801.1706329.

[26] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1980.

[27] Erik Meijer, Brian Beckman, and Gavin Bierman. 'LINQ: Reconciling Object, Relations and XML in the .NET Framework'. In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. Sigmod '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 706. ISBN: 1-59593-434-0. DOI: 10.1145/1142473.1142552. URL: https://doi.org/10.1145/1142473.1142552.

[28] Nicolas Oury and Wouter Swierstra. 'The Power of Pi'. In: *SIGPLAN Not.* 43.9 (Sept. 2008), pp. 39–50. ISSN: 0362-1340. DOI: 10.1145/1411203.1411213. URL: https://doi.org/10.1145/1411203.1411213.

[29] John C. Reynolds 1935-. *Theories of Programming Languages*. Cambridge: Cambridge University Press, 1998. ISBN: 0-521-59414-6 978-0-521-59414-1.

[30] Wilmer Ricciotti and James Cheney. 'A Formalization of SQL with Nulls'. In: *Journal of Automated Reasoning* 66.4 (Nov. 1, 2022), pp. 989–1030. ISSN: 1573-0670. DOI: 10.1007/s10817-022-09632-4. URL: https://doi.org/10.1007/s10817-022-09632-4.

[31] Petroc Taylor. *Most Popular Database Management Systems Worldwide 2023*. Statista, Sept. 2023. URL: https://www.statista.com/statistics/809750/worldwide-popularity-ranking-database-management-systems/.

[32] Ryan Wisnesky. 'Functional Query Languages with Categorical Types'. Cambridge, Massachusetts: Harvard University, Nov. 2013. 138 pp. URL: https://dash.harvard.edu/bitstream/handle/1/11744455/Wisnesky_gsas.harvard_0084L_11288.pdf (visited on 01/06/2023).

[33] Limsoon Wong. 'A Conserative Property of a Nested Relational Query Language'. In: 1992. URL: https://repository.upenn.edu/bitstreams/74ce3844-b45f-45f4-9e60-e28187777789/download.

[34] Limsoon Wong. 'Querying Nested Collections'. University of Pennsylvania 3401 Walnut Street, Suite 400C Philadelphia, PA 19104-6228: University of Pennsylvania, May 1994. 293 pp. URL: https://core.ac.uk/download/pdf/76359979.pdf (visited on 11/23/2023).

Appendix A

effPropCompat proof

Here we provide the full proof of `effPropCompat`. We also provide the type signatures of lemmas used in the proof that are not in the standard library of Coq.

```
Lemma effPropCompat {X} {Σ : X -> _} {τ} (t : nrc__stx Σ τ) ρ :
  eff2Prop (interpStxEff t ρ) = interpStxProp t (eff2Prop ∘ ρ).
Proof.
  (* Trivial cases resolved by reflexivity *)
  induction t; simpl; try reflexivity.
  - apply eff2Prop_normalise_compat.
  (* Pair case *)
  - apply f_equal2; auto.
  (* Projection 1 case *)
  - rewrite <- IHt. simpl. destruct interpStxEff.
    simpl. reflexivity.
  (* Projection 2 case *)
  - rewrite <- IHt. simpl. destruct interpStxEff.
    simpl. reflexivity.
  (* Singleton set case *)
  - fold interpSortProp.
    extensionality z.
    apply propositional_extensionality.
    split; intros H.
    + destruct H as [H | H].
      * rewrite <- IHt. symmetry. apply H.
```

```
      * destruct H.
    + left. rewrite IHt. symmetry. apply H.
  (* Small cup case *)
  - extensionality z. apply propositional_extensionality.
    split; intros H.
    + rewrite <- IHt1. rewrite <- IHt2.
      apply in_map_iff in H. destruct H as [x [E_x H]].
      subst. apply normaliseList_member in H. apply in_app_or in H.
      destruct H as [H | H].
      * left. simpl. apply in_map. apply H.
      * right. simpl. apply in_map. apply H.
    + destruct H as [H | H].
      * rewrite <- IHt1 in H. simpl in H.
        apply in_map_iff in H. destruct H as [x [E_x H]].
        subst. apply in_map. apply normaliseList_member.
        apply in_or_app. left. apply H.
      * rewrite <- IHt2 in H. simpl in H.
        apply in_map_iff in H. destruct H as [x [E_x H]].
        subst. apply in_map. apply normaliseList_member.
        apply in_or_app. right. apply H.
  (* Big cup case *)
  - simpl. fold interpSortProp. extensionality z.
    apply propositional_extensionality.
    split; intros H; simpl.
    + apply in_map_iff in H. destruct H as [x [E_x H]]. subst.
      apply normaliseList_member in H. apply in_concat in H.
      destruct H as [xs [H1 H2]]. apply in_map_iff in H1.
      destruct H1 as [y [H11 H12]]. subst. exists (eff2Prop y).
      split.
      * rewrite <- IHt1. simpl. apply in_map. apply H12.
      * simpl in IHt2. specialize (IHt2 (ρ & y)).
        apply (f_equal (fun y => y (eff2Prop x))) in IHt2.
        assert (HInterpFunEq :
                  (fun x : ^ X => eff2Prop (option_rect
                  ↪  (interpSortEff ∘ Σ & σ) ρ y x))
                  = (option_rect (interpSortProp ∘ Σ & σ)
                                 (fun x0 : X => eff2Prop (ρ x0))
                                 ↪  (eff2Prop y))
        ).
```

```
59          { extensionality z. destruct z; simpl; reflexivity. }
60          rewrite HInterpFunEq in IHt2. rewrite <- IHt2. apply in_map.
61          apply H2.
62        + destruct H as [y [H1 H2]]. apply in_map_iff.
63          specialize (IHt1 ρ). rewrite <- IHt1 in H1.
64          simpl in H1. apply in_map_iff in H1. destruct H1 as [x [H1
            ↪  H12]].
65          subst. rename H12 into H1. specialize (IHt2 (ρ & x)).
66          assert (H_e2P_ordering :
67                  (option_rect (interpSortProp ∘ Σ & σ) (fun x : X =>
                    ↪  eff2Prop (ρ x)) (eff2Prop x))
68                    = fun y => eff2Prop (option_rect (interpSortEff ∘ Σ
                      ↪  & σ) ρ x y)).
69          { extensionality p. destruct p as [p | ]; simpl; reflexivity.
            ↪  }
70          rewrite <- H_e2P_ordering in IHt2. rewrite <- IHt2 in H2.
            ↪  simpl in H2.
71          apply in_map_iff in H2. destruct H2 as [y [H21 H22]]. exists
            ↪  y.
72          split.
73          * apply H21.
74          * apply normaliseList_member. apply in_concat. subst.
75            fold interpSortEff in *.
76            eexists; split; [|exact H22].
77            apply in_map_iff.
78            eexists; split; [reflexivity|].
79            exact H1.
80      - simpl. extensionality q. fold interpSortProp in q.
81        rewrite <- IHt1. rewrite <- IHt2.
82        apply propositional_extensionality.
83        split; simpl; intros H.
84        + apply (in_map_iff) in H. destruct H as [q' [H1 H2]]. subst.
85          apply listSubtr_in in H2. split.
86          * apply in_map. destruct H2 as [H1 H2]. apply H1.
87          * intros Hneg. destruct H2 as [H1 H2]. apply in_map_iff in
            ↪  Hneg.
88            destruct Hneg as [x [Hneg1 Hneg2]].
89            destruct (Normal_interpStxEff t1 ρ) as [? _]; subst.
90            destruct (Normal_interpStxEff t2 ρ) as [? _]; subst.
```

```
        fold interpSortEff in *.
        apply eff2Prop_Normal_injec in Hneg1; subst;
        (apply H; auto; fail) || (apply H0; auto; fail) || auto.
    + destruct H as [H H']; apply in_map_iff in H; destruct H as [?
    ↪  [? ?]]; subst.
      apply in_map, listSubtr_in; split; auto.
      intro; apply H'; apply in_map; auto.
  Qed.
```

Below are the additional lemmas that we have defined and proved in our code repository. All other lemmas one may not recognise are to be found in Coq's standard library.

```
eff2Prop_normalise_compat : forall {τ : sort} (x : interpSortEff τ),
  eff2Prop (normaliseInterpSortEff x) = eff2Prop x

normaliseList_member : forall {X : Type} (O : orderedType X)
                         (x : X) (l : list X),
  In x l <-> In x (normaliseListRepr O l)

Normal_interpStxEff : forall {X : Type} {τ : sort} {Σ : X -> sort}
                      (t : nrc__stx Σ τ)
                      (ρ : forall x : X, interpSortEff (Σ x)),
  Normal (interpStxEff t ρ)

eff2Prop_Normal_injec : forall {τ : sort} (x y : interpSortEff τ),
  Normal x -> Normal y ->
  eff2Prop x = eff2Prop y ->
  x = y

listSubtr_in : forall {X : Type} {D : decEq X}
               {A B : list X} (x : X),
  In x (listSubtr D A B) <-> In x A /\ ~ In x B
```