A Pattern Calculus for Rule Languages: Expressiveness, Compilation, and Mechanization

Avraham Shinnar, Jérôme Siméon, and Martin Hirzel

IBM Research, USA
{shinnar,simeon,hirzel}@us.ibm.com

Abstract

This paper introduces a core calculus for pattern-matching in production rule languages: the Calculus for Aggregating Matching Patterns (CAMP). CAMP is expressive enough to capture modern rule languages such as JRules, including extensions for aggregation. We show how CAMP can be compiled into a nested-relational algebra (NRA), with only minimal extension. This paves the way for applying relational techniques to running rules over large stores. Furthermore, we show that NRA can also be compiled back to CAMP, using named nested-relational calculus (NNRC) as an intermediate step. We mechanize proofs of correctness, program size preservation, and type preservation of the translations using modern theorem-proving techniques. A corollary of the type preservation is that polymorphic type inference for both CAMP and NRA is NP-complete. CAMP and its correspondence to NRA provide the foundations for efficient implementations of rules languages using databases technologies.

1998 ACM Subject Classification I.2.5 Programming Languages and Software: Expert systems, D.3.3 Language Constructs and Features: Patterns, H.2.3 Languages: Query Languages

Keywords and phrases Rules, Pattern Matching, Aggregation, Nested Queries, Mechanization

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.542

Supplementary Material ECOOP Artifact Evaluation approved artifact available at http://dx.doi.org/10.4230/DARTS.1.1.8

1 Introduction

Production rules are popular for business intelligence (BI) applications, because they can encode complex data-centric policies in a flexible manner [24]. Rules often appeal to business users, as they are easy to understand, extend, and modify. Languages for production rules go back to OPS5 [19], and modern specimens include JRules [24] and Drools [5]. These languages rely heavily on pattern matching and, more recently, on aggregation.

Figure 1 shows an example production rule with aggregation. It consists of a condition (**when**, Lines 2–5) and an action (**then**, Line 6). The condition binds variable `C` to a `Client` working memory element (WME), and aggregates all `Marketer` WMEs `M` for whom `C` belongs to `M`'s bag of clients. The aggregation uses the `collect` operator to create a bag, and binds that bag to `Ms`. Finally, Line 6 creates a new WME that materializes the mapping from `C` to `Ms`. This mapping could then be consulted in other rules, for instance: when a crucial client event happens, then notify all responsible marketers.

While production rule languages are starting to support aggregation, they do not yet have a good story for running aggregates on large and/or distributed data sets. Rules engines are usually centralized, tied to their internal data representation, and not readily applicable to other stores. Ideally, we would like to run production rules over distributed stores efficiently and leverage existing algorithms for large-scale data processing. To accomplish this, we

29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 542–567

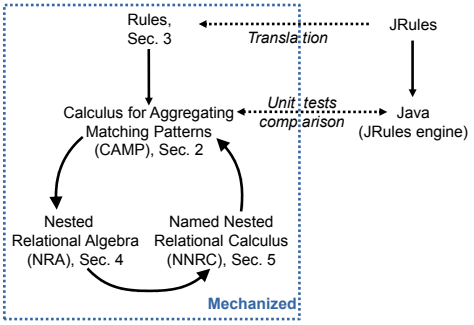
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany


```
1 rule FindMarketers {
2   when {
3     C:  Client();
4     Ms: aggregate { M: Marketer(clients.contains(C.id)); }
5           do { collect {M}; }
6   } then { insert new C2Ms(C, Ms); }
7 }
```

Figure 1 In JRules, compute a reverse mapping from clients to marketers.

translate rules to a database algebra suitable for optimization and distributed execution. Because modern rules languages support complex data models and nesting (e.g., JRules uses an object-oriented data model and supports nested aggregation), we target a Nested Relational Algebra [14] (NRA). Optimization techniques for such algebras extend those already available in relational systems, and have been used to support efficient evaluation for a variety of nested data models, from OO [6, 14] to XML [27, 31].

This paper proposes the Calculus for Aggregating Matching Patterns (CAMP), a core calculus for production rule languages. The essence of patterns is that they have two implicit inputs: the data item that is the subject of the match, and an environment of variables previously bound by the match. The main novelty of CAMP is that it exposes the data flow inherent to the pattern matching of rules. Figure 2 gives an outline for the main languages covered in the paper and their relationships. Figure 2(b) shows the real JRules language which is the motivation for our work, which gets compiled and executed using a Rete-based rules engine [20]. This paper studies the properties of CAMP, proving that it has the same expressiveness as Nested Relational Algebra (NRA) and Named Nested Relational Calculus (NNRC), as depicted in Figure 2(a). The constructive proof includes a translation from CAMP to NRA, suitable as a first step towards an efficient compiler.



(a) Commuting formalisms: each arrow is a translation that preserves semantics, program size, and types.

(b) Outside the mechanization: JRules compiler and runtime.

Figure 2 CAMP formalization & JRules.

From a data processing perspective, the challenges in that translation include encoding the input datum and environment, and on the output side, encoding recoverable errors (caused by non-matching data) to be propagated. From a rules language perspective, the main challenge is to capture a representative yet minimal subset in CAMP. To make the connection between CAMP and productions rules clearer, we provide a Rules language with a syntax that parallels that of JRules and its corresponding encoding in CAMP. Our implementation includes a way to translate JRules into that formal Rules language and runs unit tests to check that the mechanized semantics yields the same results as the real JRules compiler. In that context, our approach is similar to prior work on the database-supported execution of programming languages with embedded queries, such as LINQ [11] compiled to a relational engine using Ferry [22]. One interesting outcome of our work is to show that one can offload the full pattern matching logic underlying production rules to a database engine. The main contributions of the paper are as follows:

- CAMP (Section 2), which captures the matching semantics for production rules with aggregates. The calculus is simple enough for formal reasoning and expressive enough to

support a Rules language (Section 3) that models large subsets of JRules [24].
- A full translation (Section 4) from CAMP to the NRA from [14] with only a minimal extension. This opens the door for taking advantage of the database literature for optimizing and distributing production rule matching over large stores.
- A reverse translation (Sections 5 and 6) from NRA back to CAMP, using the NNRC from [35] as scaffolding, showing that all three languages are equally expressive.
- Type systems for all three languages and proofs that the translations preserve types (Section 7). As a corollary, polymorphic type inference for CAMP is NP-complete.
- A mechanization of all the proofs of semantics preservation, program size, and type correctness using the Coq proof assistant [16] (see peer-reviewed artifact).

2 CAMP

The Calculus for Aggregating Matching Patterns (CAMP) models the query fragment of traditional production rules language such as OPS5 [19], JRules [24], or Drools [5]. The query fragment in these languages is a (possibly nested) pattern that is matched against working memory. To model this query style, CAMP patterns scrutinize an (implicit) datum that we call **it**. Additionally, to support naming matched fragments of **it**, CAMP patterns also refer to an (implicit) environment **env** that maps variables to data. CAMP expressions all return data, the result of the query. This is often a record containing variables bound by the pattern. Patterns can also fail if they do not match the given data. This failure is not fatal, and can trigger alternative pattern matching attempts.

2.1 Syntax

Definition 1 presents the syntax for CAMP. Section 2.3 presents the formal semantics; here we give an informal description.

▶ **Definition 1** (CAMP syntax).

$$\begin{aligned}(\text{patterns})\ p ::= {} & d \mid \oplus p \mid p_1 \otimes p_2 \mid \mathbf{map}\ p \mid \mathbf{assert}\ p \mid p_1 \| p_2 \\ & \mid \mathbf{it} \mid \mathbf{let\ it} = p_1\ \mathbf{in}\ p_2 \mid \mathbf{env} \mid \mathbf{let\ env} \mathrel{+}= p_1\ \mathbf{in}\ p_2\end{aligned}$$

Going in order of presentation, d allows arbitrary (constant) data to be the result of a CAMP pattern. The full data model is presented in Section 2.2. The next two constructs allow unary ($\oplus$) and binary ($\otimes$) operators to process the result of a pattern or patterns. The set of operators is described in Section 2.2 and includes operations for constructing and manipulating records and bags, key components of our data model.

The **map** p construct maps a pattern p over the implicit data **it**. Assuming that **it** is a bag, the result is the bag of results obtained from matching p against each datum in **it**. Failing matches are skipped. The **assert** p construct allows a pattern p to conditionally cause match failure. If p evaluates to false, matching fails, otherwise, it returns the empty record $[]$. The $p_1 \| p_2$ construct allows for recovery from match failure: if p_1 matches successfully, p_2 is ignored; otherwise, if p_1 fails to match, p_2 is evaluated.

Finally, we come to the constructs that deal with the implicit datum being matched and the environment. The **it** construct obtains the datum that is being matched. The **let it** $= p_1$ **in** p_2 construct allows this implicit datum to be altered. Similarly, **env** reifies the current environment as a record. This reified environment can then be manipulated via standard record operators. The final construct, **let env** $+= p_1$ **in** p_2, allows a pattern to add new bindings to the environment. The result of matching p_1 must be a record, which is

interpreted as a reified environment. If the current environment is compatible with the new one (all common attributes have equal values) then they are merged and the pattern p_2 is evaluated with the merged environment. If they are incompatible, the pattern fails. This mimics the standard convention in pattern matching languages that multiple bindings of a pattern variable must all bind to the same value.

As an example, the CAMP version of the JRules pattern `C: Client()` in Line 3 of Figure 1 is **let env** += **assert it**.*type* = "Client" **in let env** += $[C : \textbf{it}]$ **in env**. The pattern first asserts that **it** has type "Client". If **assert** succeeds, it returns the empty record, so the first **let env** leaves the environment unchanged. The next part of the pattern attempts to add $[C : \textbf{it}]$ to **env**. That succeeds if either C is not yet bound, or C is already bound to the same datum. The idiom **let env** += **assert** p_1 **in** p_2 is so common that we write $p_1 \wedge p_2$, for example, **it**.*type* = "Client" $\wedge$ **let env** += $[C : \textbf{it}]$ **in env**.

The example also relies on a fundamental data model and operators. For instance, **it**.*type* retrieves a record attribute, the = operator checks equality, and $[A : d]$ constructs a record. Those are shared across CAMP, NRA, and NNRC and are described next.

2.2 Data Model and Operators

Values in our data model, the set $\mathcal{D}$, are atoms, records, or bags. We assume a sufficiently large set of atoms $a, b, ...$ including numbers, strings, Booleans and a null value written *nil*. A bag is a multiset of values in $\mathcal{D}$; we write $\emptyset$ for the empty bag and $\{d_1, ..., d_n\}$ for the bag containing the values $d_1, ..., d_n$. A record is a mapping from a finite set of attributes to values in $\mathcal{D}$, where attribute names are drawn from a sufficiently large set $A, B, ...$ We write $[\,]$ for the empty record and $[\overline{A_i : d_i}]$ for the record mapping A_i to d_i.

Our data model uses bags instead of sets to naturally support aggregation. For instance, given employee records, a projection can obtain the bag of salaries, which can then be averaged. A set would yield the wrong result, as it would omit duplicate salaries. Other work uses bags for similar reasons [21]. Our data model supports arbitrary nesting of bags and records to model object-oriented rule languages such as JRules. This nesting increases expressiveness and simplifies the treatment of aggregation, group-by, and nested queries.

Records x and y are compatible if $\forall A \in \mathrm{dom}(x) \cap \mathrm{dom}(y), x(A) = y(A)$. We define the sum $x+y$ of compatible records as their union $x \cup y$, and leave it undefined for non-compatible records. An *operator* is a pure function defined on only its explicit operands; it does not access any implicit input (e.g., **it** or **env** in CAMP). Given operands of the correct types, operators are total. We factor out operators from each of our three languages (CAMP, NRA, and NNRC) to keep the languages small and focused on the essentials. Definition 2 presents a set of basic unary and binary operators on our data model.

▶ **Definition 2** (Operators).

$$\begin{array}{lll} \text{(uops)} \ \oplus d & ::= & \mathit{identity}\ d \mid \neg d \mid \{d\} \mid \#d \mid \mathit{flatten}\ d \mid [A\!:\!d] \mid d.A \mid d\!-\!A \\ \text{(bops)} \ d_1 \otimes d_2 & ::= & d_1 = d_2 \mid d_1 \in d_2 \mid d_1 \cup d_2 \mid d_1 * d_2 \mid d_1 + d_2 \end{array}$$

In order of presentation, the unary operators do the following:

identity d	returns d.
$\neg d$	negates a Boolean.
$\{d\}$	constructs a singleton bag containing the value d.
$\#d$	returns the number of elements in a bag.
flatten d	flattens a bag of bags.

$[A:d]$	constructs a record with a single attribute A containing the value d.
$d.A$	accesses the value associated with attribute A in record d.
$d-A$	returns a record with all attributes of d except A.

Each of the last three operators involves a datum d and a statically given attribute A. Once the attribute is fixed, they are unary operators on the datum. For instance, $d.type$ is the unary operator for retrieving attribute *type* of data d.

In order of presentation, the binary operators do the following:

$d_1 = d_2$	compares two data for equality.
$d_1 \in d_2$	returns true if and only if d_1 is an element of bag d_2.
$d_1 \cup d_2$	returns the union of two bags.
$d_1 * d_2$	concatenates two records, favoring d_1 if there are overlapping attributes.
$d_1 + d_2$	returns a singleton bag containing the concatenation of the two records if they are compatible, and returns $\emptyset$ otherwise.

In [35], the record concatenation operators are denoted $\times$ and $\bowtie$, but we use $*$ and $+$ instead to reserve $\times$ and $\bowtie$ for the NRA cross product and join on bags (see Section 4.1). Some of the operators, such as bag and record construction, are fundamental, since we need them to properly manipulate our data model. But since operators are simple functions, it is easy to add more. For instance, # counts elements of a bag, and we could easily add other unary reduction operators for summation or finding a minimum. Note that even record concatenation with $d_1 + d_2$ is total, returning the empty bag if the records are incompatible. We defer the discussion of type errors, which can be detected statically, to Section 7.

2.3 Semantics

The syntax and an informal description of CAMP were given in Section 2.1. Figure 3 presents a big-step operational semantics for CAMP. A mechanization of these semantics using the Coq proof assistant [16] is available in the companion artifact for this paper.

The relation $\sigma \vdash p \,@\, d \Downarrow_r d?$ relates an environment (record) σ, a pattern p, and a datum d (the implicit datum **it**) with an output $d?$. The subscript $_r$ on the relation stands for "rules", to distinguish the semantics from those of NRA and NNRC presented later in the paper. The output is a member of the lifted data domain $\mathcal{D}? = \mathcal{D} + \mathbf{err}$, representing either the returned datum or match failure.

The $\Downarrow_r$ relation is partial; malformed patterns and patterns that are given the wrong type of data may not admit any derivations. This is distinct from match failure, which is recoverable, and is internalized in the relation. Section 7 will present a type system for CAMP with a soundness result guaranteeing that well-typed patterns matching appropriately typed data can always derive a result (possibly **err**, indicating match failure).

The rules for constants and operators are standard. The rules for **map** conspire to evaluate the given pattern over each element of the datum. Pattern match failures are ignored and the rest of the results are gathered as the result. The Assert and OrElse rules allow patterns to explicitly cause and recover from an **err**. To enable easy sequencing with **let env**, **assert** true returns an empty record.

The data that is being matched is obtained via **it**, and temporarily replaced using **let it** $= p_1$ **in** p_2, where p_2 is evaluated with **it** set to the result of evaluating p_1.

The environment is obtained using **env**, which reifies it as a record. New bindings are added to the environment using **let env** $+= p_1$ **in** p_2. If p_1 evaluates to a record that is compatible (defined in Section 2.2) with the current environment, they are joined and used as

$$\frac{}{\sigma \vdash d_0 \mathbin{@} d \Downarrow_r d_0}\ (\text{Constant}) \qquad \frac{\sigma \vdash p_1 \mathbin{@} d \Downarrow_r d_1 \quad \sigma \vdash p_2 \mathbin{@} d \Downarrow_r d_2 \quad d_1 \otimes d_2 = d_3}{\sigma \vdash p_1 \otimes p_2 \mathbin{@} d \Downarrow_r d_3}\ (\text{Binary Op})$$

$$\frac{\sigma \vdash p \mathbin{@} d \Downarrow_r d_0 \quad \oplus d_0 = d_1}{\sigma \vdash \oplus p \mathbin{@} d \Downarrow_r d_1}\ (\text{Unary Op}) \qquad \frac{\sigma \vdash p \mathbin{@} d \Downarrow_r \textbf{err} \quad \sigma \vdash \textbf{map}\ p \mathbin{@} s \Downarrow_r s_0}{\sigma \vdash \textbf{map}\ p \mathbin{@} \{d\} \cup s \Downarrow_r s_0}\ (\text{Map } \textbf{err})$$

$$\frac{}{\sigma \vdash \textbf{map}\ p \mathbin{@} \emptyset \Downarrow_r \emptyset}\ (\text{Map } \emptyset) \qquad \frac{\sigma \vdash p \mathbin{@} d \Downarrow_r d_0 \quad \sigma \vdash \textbf{map}\ p \mathbin{@} s \Downarrow_r s_0}{\sigma \vdash \textbf{map}\ p \mathbin{@} \{d\} \cup s \Downarrow_r \{d_0\} \cup s_0}\ (\text{Map})$$

$$\frac{\sigma \vdash p \mathbin{@} d \Downarrow_r \mathsf{true}}{\sigma \vdash \textbf{assert}\ p \mathbin{@} d \Downarrow_r []}\ (\text{Assert True}) \qquad \frac{\sigma \vdash p \mathbin{@} d \Downarrow_r \mathsf{false}}{\sigma \vdash \textbf{assert}\ p \mathbin{@} d \Downarrow_r \textbf{err}}\ (\text{Assert False})$$

$$\frac{\sigma \vdash p_1 \mathbin{@} d \Downarrow_r d_1}{\sigma \vdash p_1 \mid\mid p_2 \mathbin{@} d \Downarrow_r d_1}\ (\text{OrElse 1}) \qquad \frac{\sigma \vdash p_1 \mathbin{@} d \Downarrow_r \textbf{err} \quad \sigma \vdash p_2 \mathbin{@} d \Downarrow_r d_2?}{\sigma \vdash p_1 \mid\mid p_2 \mathbin{@} d \Downarrow_r d_2?}\ (\text{OrElse 2})$$

$$\frac{}{\sigma \vdash \textbf{it} \mathbin{@} d \Downarrow_r d}\ (\textbf{it}) \qquad \frac{}{\sigma \vdash \textbf{env} \mathbin{@} d \Downarrow_r \sigma}\ (\textbf{env})$$

$$\frac{\sigma \vdash p_1 \mathbin{@} d \Downarrow_r d_1 \quad \sigma \vdash p_2 \mathbin{@} d_1 \Downarrow_r d_2?}{\sigma \vdash \textbf{let it} = p_1 \textbf{ in } p_2 \mathbin{@} d \Downarrow_r d_2?}\ (\text{Let } \textbf{it})$$

$$\frac{\sigma \vdash p_1 \mathbin{@} d \Downarrow_r \sigma_1 \quad \sigma + \sigma_1 \vdash p_2 \mathbin{@} d \Downarrow_r d_2?}{\sigma \vdash \textbf{let env} \mathrel{+}= p_1 \textbf{ in } p_2 \mathbin{@} d \Downarrow_r d_2?}\ (\text{Let } \textbf{env})$$

$$\frac{\sigma \vdash p_1 \mathbin{@} d \Downarrow_r \sigma_1 \quad \neg\mathsf{compatible}(\sigma, \sigma_1)}{\sigma \vdash \textbf{let env} \mathrel{+}= p_1 \textbf{ in } p_2 \mathbin{@} d \Downarrow_r \textbf{err}}\ (\text{Let } \textbf{env err})$$

err propagation:

$$\frac{\sigma \vdash p \mathbin{@} d \Downarrow_r \textbf{err}}{\begin{array}{l}\sigma \vdash \oplus p \mathbin{@} d \Downarrow_r \textbf{err} \\ \sigma \vdash p \otimes p_2 \mathbin{@} d \Downarrow_r \textbf{err} \\ \sigma \vdash p_1 \otimes p \mathbin{@} d \Downarrow_r \textbf{err} \\ \sigma \vdash \textbf{assert}\ p \mathbin{@} d \Downarrow_r \textbf{err} \\ \sigma \vdash \textbf{let it} = p \textbf{ in } p_2 \mathbin{@} d \Downarrow_r \textbf{err} \\ \sigma \vdash \textbf{let env} \mathrel{+}= p \textbf{ in } p_2 \mathbin{@} d \Downarrow_r \textbf{err}\end{array}}$$

Figure 3 CAMP semantics. $\boxed{\sigma \vdash p \mathbin{@} d \Downarrow_r d?}$

the environment for evaluating p_2 (recall that "+" is defined only for compatible records). Incompatibility results in match failure.

Figure 3 also presents six rules enabling **err** propagation, written in compressed form. The antecedent of the given rule separately implies all six of the consequents.

To illustrate, consider the JRules pattern `M: Marketer(clients.contains(C.id));` from Line 4 in Figure 1. This corresponds to the CAMP pattern in Equation 1:

$$\textbf{it}.\mathit{type} = \text{“}\mathsf{Marketer}\text{”} \wedge \textbf{env}.C.\mathit{data.id} \in \textbf{it}.\mathit{data.clients} \wedge \textbf{let env} \mathrel{+}= [M : \textbf{it}] \textbf{ in env} \quad (1)$$

In words: if **it** is a record whose *type* attribute is "Marketer", and if the environment already has a binding for C with a *data.id* attribute that is an element of the bag **it**.*data.clients*, then bind **it** to M in the environment and return the enriched environment reified as a record.

3 Rules

This section describes CAMP Rules, a set of rules macros on top of the CAMP core calculus from Section 2, which we also implemented in the Coq proof assistant [16]. As shown in Figure 2, the rules macros bridge the gap between CAMP and real-world production rule languages such as JRules [24]. This section provides practical context for this work, describing the connection between CAMP rules and JRules.

3.1 JRules

This paper grew out of our work on IBM's "Operational Decision Manager: Decision Server Insights" product, or *ODM Insights* for short [29], a middleware for integrating event

processing with analytics. Figure 4 depicts the architecture of ODM Insights. Everything is driven by *events*, which are (possibly nested) objects in motion. When ODM Insights receives an incoming event, it routes it to an event processor. An *event processor* is a rule engine that can read individual entities from a shared store, and can also consult the result of analytics. An *entity* is a (possibly nested) object at rest, for instance, a JSON document [18]. The *store* holds all the entities. The event processor reacts to an event with two kinds of side effects: it can write to the store (to modify an existing entity or create a new entity), and it can send zero or more output events (also known as *actions*). On each firing, the event processor accesses at most a handful of entities. In contrast, the *analytics processor* periodically scans all the entities in the store to derive global insights useful to guide future actions. When the out-of-band analytics finish, the resulting insights become available so the reactive event processors can consult them.

Another important goal was to integrate the authoring experience: code for the event processor is written in JRules with extensions for spatiotemporal concepts. As state-of-the-art production rule technology did not offer a way to program the analytics, we needed to find a way to use production rules for queries over a large, and possibly distributed, data store. To fill this gap, this paper shows how to translate rules to NRA, for which there are known evaluation techniques over large stores.

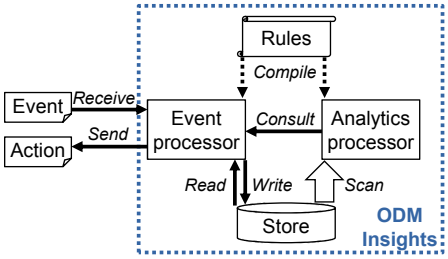


Figure 4 ODM Insights.

To demonstrate that CAMP accurately models a real-world production rule language, we implemented a compiler from JRules to CAMP rules. Our compiler reuses the front-end of the JRules compiler to get an abstract syntax tree, then traverses that tree to generate CAMP rules in Coq, along with a corresponding test harness. For a given rule, the test harness runs both the JRules engine and the translated Coq code against the same test input and compares the two results. With this technique, we tested several examples, to confirm that the JRules engine and the CAMP rules return the same result (modulo permutation over the collection of items being returned). While this testing does not have the rigor and completeness of mechanized proofs, it connects our mechanized theory to the real world.

3.2 Basic Production Rules in CAMP

Rules r are an intermediate language that bottoms out in CAMP patterns p from Definition 1. Rules r are defined via macros – meta-level functions that statically compute CAMP patterns p. For now, we will focus on basic rules drawn from the grammar $r ::= \textbf{\textit{when}}\ p; r \mid \textbf{\textit{return}}\ p$. Aggregation and negation are covered in Section 3.3.

The input to a JRules rule consists of a working memory. Our translation to CAMP encodes this as an environment mapping *WORLD* to a bag of working memory elements. We define the macro $\mathbf{WW}\,(p) := \mathbf{let\ it} = \mathbf{env}.\mathit{WORLD}\ \mathbf{in}\ p$ to apply a pattern over the working memory. Each working memory element is a typed object. We encode each such object as a record with two attributes, *type* (a string) and *data* (a record). This simple encoding of objects does not account for subtyping, but suffices for our purposes.

A production rule engine finds all ways in which a rule can match. We express this using a ***when*** macro, **map**ping each pattern over working memory as follows:

$$[\![\textbf{\textit{when}}\ p; r]\!] = \mathit{flatten}\,(\mathbf{WW}\,(\mathbf{map}\ (\mathbf{let\ env} \mathrel{+}= p\ \mathbf{in}\ [\![r]\!])))$$

In words: for each working memory element, attempt pattern p. If it succeeds, add the

```
1 when {
2   C: Client();
3   M: Marketer(
4        clients.contains(C.id));
5 } then {
6   insert new C2M(C, M);
7 }
```

Figure 5 JRules with no aggregation.

$$
\begin{array}{lll}
\textbf{\textit{when}} & & \textbf{it}.\mathit{type} = \text{“Client”} \\
& \wedge & \textbf{let env} \mathrel{+}= [C : \textbf{it}] \textbf{ in env}; \\
\textbf{\textit{when}} & & \textbf{it}.\mathit{type} = \text{“Marketer”} \\
& \wedge & \textbf{env}.C.\mathit{data.id} \in \textbf{it}.\mathit{data.clients} \\
& \wedge & \textbf{let env} \mathrel{+}= [M : \textbf{it}] \textbf{ in env}; \\
\textbf{\textit{return}} & & [\mathit{type} : \text{“C2M”}]* \\
& & [\mathit{data} : [\mathit{client} : \textbf{env}.C]* \\
& & \quad\ [\mathit{marketer} : \textbf{env}.M]]
\end{array}
$$

Figure 6 CAMP rule for Figure 5.

$$
\begin{array}{rl}
p_1 \wedge p_2 & := \textbf{let env} \mathrel{+}= \textbf{assert } p_1 \textbf{ in } p_2 \\
\textbf{WW}\,(p) & := \textbf{let it} = \textbf{env}.\mathit{WORLD} \textbf{ in } p \\
\textbf{mapall } p & := \textbf{let env} \mathrel{+}= [x : \textbf{map } p] \textbf{ in } \big((\#(\textbf{env}.x) = \#\textbf{it}) \wedge \textbf{env}.x\big) \qquad x \text{ is } \textbf{fresh} \\
\textbf{mapsnone } p & := \#(\textbf{map } p) = 0
\end{array}
$$

Figure 7 Auxiliary definitions for rules.

resulting bindings to **env** and run the translated tail rule $[\![r]\!]$. This results in a bag of bags, where the inner bags are either empty for non-matches or singletons for matches. The final *flatten* returns the matches.

The output of a production rule consists of actions. We encode this in CAMP by returning a singleton bag containing the result of the action caused by a rule. This is done via a ***return*** macro: $[\![\textbf{\textit{return}}\ p]\!] = \{p\}$. The rule macros defined so far (**WW**, ***when***, ***return***, and $\wedge$ from Section 2.1) suffice to translate production rules without aggregation or negation. Figures 5 and 6 show a JRules program and the CAMP rule for it. Note that the **`when`** clause from JRules yields two ***when*** macros in the CAMP rule, each performing its own implicit loop over working memory. Since CAMP rules do not have side effects, we model the insertion by ***return***ing the computed value.

3.3 Rules with Aggregation and Negation

This section completes the language of rules macros from Section 3.2. Figure 7 presents definitions used in the semantics for rule macros. We already saw $\wedge$ and **WW**. The other two definitions, **mapall** and **mapsnone**, help encode aggregation and negation, respectively.

The **mapall** macro is similar to **map**, but ensures that the given pattern matches on all the data in the bag. Unlike **map**, which allows (and ignores) **err**s, **mapall** propagates any such **err**s. It does this by counting: if there are any **err**s, the result of **map** will be smaller than its input. To avoid recomputing the **map**, it stores its result in the environment, using a variable x that is **fresh**, meaning not used elsewhere in the program. The **mapsnone** macro also employs **map**. By counting to check that the resulting bag is empty it ensures that the given pattern does not match *any* data in the bag.

Definition 3 defines a rule as a sequence of patterns, each marked to indicate how the pattern should be interpreted. Patterns that are meant to (implicitly) be matched against each datum in working memory are signaled using ***when***. In contrast, ***global*** introduces patterns that should be run against the working memory itself (reified as a bag of data). Patterns marked with ***not*** are tested to ensure that they do not successfully match against *any* working memory data. A rule sequence is always terminated with ***return***.

$$[\![\textbf{\textit{when}}\ p; r]\!] = \mathit{flatten}\,(\mathbf{WW}\,(\mathbf{map}\ (\mathbf{let}\ \mathbf{env}\ +\!=\ p\ \mathbf{in}\ [\![r]\!])))$$
$$[\![\textbf{\textit{global}}\ p; r]\!] = \mathbf{let}\ \mathbf{env}\ +\!=\ \mathbf{WW}\,(p)\ \mathbf{in}\ [\![r]\!]$$
$$[\![\textbf{\textit{not}}\ p; r]\!] = \mathbf{WW}\,(\mathbf{mapsnone}\ p) \wedge [\![r]\!]$$
$$[\![\textbf{\textit{return}}\ p]\!] = \{p\}$$

$$\frac{[\mathit{WORLD} : w] \vdash [\![r]\!] \mathbin{@} \mathit{nil} \Downarrow_r d?}{r \mathbin{@} w \Downarrow_r d?}\ \text{(Rule Evaluation)}$$

$$\mathbf{aggregate}(r, \oplus, p) := \mathbf{let}\ \mathbf{it} = [\![r]\!]\ \mathbf{in}\ \oplus\mathbf{mapall}\ (\mathbf{let}\ \mathbf{env}\ +\!=\ \mathbf{it}\ \mathbf{in}\ p)$$

Figure 8 Evaluating rules against working memory.

▶ **Definition 3** (Rules).

$$\text{(rules)}\quad r \quad ::= \quad \textbf{\textit{when}}\ p; r \mid \textbf{\textit{global}}\ p; r \mid \textbf{\textit{not}}\ p; r \mid \textbf{\textit{return}}\ p$$

Figure 8 presents a translation from rules to patterns. Rule evaluation proceeds by evaluating the translated pattern in an environment with *WORLD* bound to the desired working memory. Note that the translated pattern ignores the initial input, only accessing the working memory. Figure 8 also defines an **aggregate** macro with three parameters: a rule r, a reduction (unary) operator $\oplus$, and a transformer pattern p. First, the pattern for r is executed. Second, the result of r is transformed by **mapall**ing the transformer pattern p. Finally, the results of p are aggregated using $\oplus$. This is meant to be used with patterns marked ***global***, enabling nested aggregation over working memory. Consider the JRules code from Lines 4–5 of Figure 1: `Ms:aggregate {M:Marketer(clients.contains(C.id));} do collect{M};`. The CAMP rule for it is ***global***$[Ms : \mathbf{aggregate}(\textbf{\textit{when}}\ p_1; \textbf{\textit{return}}\ \mathbf{env}, \mathit{identity}, \mathbf{env}.M)]$;, where p_1 is defined by Equation 1 at the end of Section 2.3. Note the use of ***when*** nested inside ***global*** to match all working memory elements again for aggregation.

4 CAMP to NRA

This section describes how to translate production rules to nested relational algebra (NRA). Since production rule languages support nesting both code and data, we use NRA which generalizes the well-known relational algebra (RA). In some cases, this generalization actually enables simplification. For instance, instead of RA's projection with attribute assignments, NRA offers a general map operator; and instead of RA's global table names, NRA can place tables with attributes of a top-level database record.

4.1 Nested Relational Algebra

We use the NRA from [14] with a bag semantics and extended with a conditional default operation. We consider a core set of queries, sufficient to express the full algebra. Richer queries (e.g., joins, unnest) can be built on top of this core.

▶ **Definition 4** (NRA syntax).

$$\begin{aligned}\text{(queries)}\ q \quad ::= \quad & d \mid \mathbf{In} \mid \oplus q \mid q_1 \otimes q_2 \mid \chi_{\langle q_2 \rangle}(q_1) \mid \sigma_{\langle q_2 \rangle}(q_1) \\ \mid \quad & q_1 \times q_2 \mid \bowtie^{d}_{\langle q_2 \rangle}(q_1) \mid q_1 \parallel q_2\end{aligned}$$

In this grammar, d returns constant data, **In** returns the context value (usually a bag or a record), and $\oplus$ and $\otimes$ are unary and binary operators from Section 2.2. χ is the map operation on bags (in simple cases, map degenerates to conventional relational projection π), σ is selection, $\times$ is Cartesian product, and $\bowtie^d$ is the *dependent join* which evaluates q_2 with the context set one at a time to the results of q_1 and concatenates pairs of records resulting from q_1 and q_2. (Dependent join is written $q_1\langle q_2\rangle$ in [14] and MapConcat$\{q_2\}(q_1)$ in [31].) Sub-queries in subscripts with angle brackets $\langle ...\rangle$ are *dependent*, applied one at a time to the results of their sibling in the query plan.

The last operator, which we call *default*, is the only operator not originally proposed in [14], and is used to encode error propagation in CAMP. It evaluates its first operand and returns its value, unless that value is $\emptyset$, in which case it returns the value of its second operand (as default). For NRA to be equivalent to NNRC (a well-known correspondence [34]), some form of conditional must be included. Prior work included a similar default operation for null values, testing if the first data is null. Since we chose not to complicate our presentation with three-valued logic to implicitly propagate null values, we will instead use $\emptyset$ for errors.

4.2 Semantics

Figure 9 presents the semantics for NRA. The subscript $_a$ on the relation $\Downarrow_a$ stands for algebra. Previous work [14] used a denotational semantics; we chose a big-step operational semantics for consistency with CAMP. [1]

Evaluating a query q against input data d is written $q \mathbin{@} d$. The **In** query returns that data, whereas the constant query returns the specified constant. Unary and binary operators evaluate the provided queries, and then evaluate the operator on the result(s).

A map query, $\chi_{\langle q_2\rangle}(q_1)$, is evaluated recursively using the two rules Map and Map $\emptyset$. The Map rule evaluates the query q_1, producing a bag s_1. One element is transformed using q_2 and the result is unioned with the result of mapping q_2 over the rest of s_1 (expressed as a constant query). The Map $\emptyset$ rule terminates this recursion with $\emptyset$.

Selection queries, $\sigma_{\langle q_2\rangle}(q_1)$, are evaluated recursively, similar to map queries. Each element in the bag produced by q_1 is kept only if applying the predicate q_2 returns true.

Product queries, $q_1 \times q_2$, require a double recursion. The Prod rule evaluates both q_1 and q_2 to produce bags s_1 and s_2. Elements d_1 and d_2 are chosen from them, and their product is unioned with the product of d_1 with the remainder of s_2 and the product of the remainder of s_1 with all of s_2.

Dependent join queries, $\bowtie^d{}_{\langle q_2\rangle}(q_1)$, first evaluate q_1 to produce a bag s_1. For each element d_1 of s_1, they evaluate q_2 with the context data set to d_1, yielding a bag s_2. The result is a bag consisting of all the record concatenations $d_1 \times d_2$ of records resulting from q_1 and q_2. Like product queries, dependent join queries use double recursion, with the main difference being that the inner recursion (over q_2) depends on the outer recursion (over q_1) by using its result as the context data.

Finally, the last query provided is the default query $q_1 \| q_2$. The rules for this query are straightforward, evaluating q_1 and looking at the result. If it is an empty bag, q_2 is evaluated and its result returned, otherwise the result of q_1 is returned.

[1] Our mechanization actually defines the semantics for all three languages via a computational denotational semantics. However, we chose to use a more conventional style for presentation.

$$\frac{}{d_0 \ @ \ \ d \Downarrow_a d_0}\ (\text{Constant}) \qquad \frac{}{\mathbf{In} \ @ \ \ d \Downarrow_a d}\ (\text{ID}) \qquad \frac{q \ @ \ d \Downarrow_a d_0 \quad \oplus d_0 = d_1}{\oplus q \ @ \ d \Downarrow_a d_1}\ (\text{Unary Operator})$$

$$\frac{q_1 \ @ \ d \Downarrow_a d_1 \quad q_2 \ @ \ d \Downarrow_a d_2 \quad d_1 \otimes d_2 = d_3}{q_1 \otimes q_2 \ @ \ d \Downarrow_a d_3}\ (\text{Binary Operator}) \qquad \frac{q_1 \ @ \ d \Downarrow_a \emptyset}{\chi_{\langle q_2 \rangle}(q_1) \ @ \ d \Downarrow_a \emptyset}\ (\text{Map } \emptyset)$$

$$\frac{q_1 \ @ \ d \Downarrow_a \{d_1\} \cup s_1 \quad q_2 \ @ \ d_1 \Downarrow_a d_2 \quad \chi_{\langle q_2 \rangle}(s_1) \ @ \ d \Downarrow_a s_2}{\chi_{\langle q_2 \rangle}(q_1) \ @ \ d \Downarrow_a \{d_2\} \cup s_2}\ (\text{Map})$$

$$\frac{q_1 \ @ \ d \Downarrow_a \{d_1\} \cup s_1 \quad q_2 \ @ \ d_1 \Downarrow_a \mathsf{true} \quad \sigma_{\langle q_2 \rangle}(s_1) \ @ \ d \Downarrow_a s_2}{\sigma_{\langle q_2 \rangle}(q_1) \ @ \ d \Downarrow_a \{d_1\} \cup s_2}\ (\text{Select True})$$

$$\frac{q_1 \ @ \ d \Downarrow_a \{d_1\} \cup s_1 \quad q_2 \ @ \ d_1 \Downarrow_a \mathsf{false} \quad \sigma_{\langle q_2 \rangle}(s_1) \ @ \ d \Downarrow_a s_2}{\sigma_{\langle q_2 \rangle}(q_1) \ @ \ d \Downarrow_a s_2}\ (\text{Select False})$$

$$\frac{q_1 \ @ \ d \Downarrow_a \emptyset}{\sigma_{\langle q_2 \rangle}(q_1) \ @ \ d \Downarrow_a \emptyset}\ (\text{Select } \emptyset) \qquad \frac{q_1 \ @ \ d \Downarrow_a \emptyset}{q_1 \times q_2 \ @ \ d \Downarrow_a \emptyset}\ (\text{Product } \emptyset_1) \qquad \frac{q_2 \ @ \ d \Downarrow_a \emptyset}{q_1 \times q_2 \ @ \ d \Downarrow_a \emptyset}\ (\text{Product } \emptyset_2)$$

$$\frac{q_1 @ d \Downarrow_a \{d_1\} \cup s_1 \quad q_2 @ d \Downarrow_a \{d_2\} \cup s_2 \quad \{d_1\} \times s_2 @ d \Downarrow_a s_3 \quad s_1 \times (\{d_2\} \cup s_2) @ d \Downarrow_a s_4}{q_1 \times q_2 \ @ \ d \Downarrow_a \{d_1 * d_2\} \cup s_3 \cup s_4}\ (\text{Prod})$$

$$\frac{q_1 \ @ \ d \Downarrow_a \{d_1\} \cup s_1 \quad q_2 \ @ \ d_1 \Downarrow_a \{d_2\} \cup s_2 \quad \bowtie^d_{\langle s_2 \rangle}(\{d_1\}) \ @ \ d \Downarrow_a s_3 \quad \bowtie^d_{\langle q_2 \rangle}(s_1) \ @ \ d \Downarrow_a s_4}{\bowtie^d_{\langle q_2 \rangle}(q_1) \ @ \ d \Downarrow_a \{d_1 * d_2\} \cup s_3 \cup s_4}\ (\text{DJ})$$

$$\frac{q_1 \ @ \ d \Downarrow_a \emptyset}{\bowtie^d_{\langle q_2 \rangle}(q_1) \ @ \ d \Downarrow_a \emptyset}\ (\text{DJ } \emptyset_1) \qquad \frac{q_1 \ @ \ d \Downarrow_a \{d_1\} \cup s_1 \quad q_2 \ @ \ d_1 \Downarrow_a \emptyset \quad \bowtie^d_{\langle q_2 \rangle}(s_1) \ @ \ d \Downarrow_a s_2}{\bowtie^d_{\langle q_2 \rangle}(q_1) \ @ \ d \Downarrow_a s_2}\ (\text{DJ } \emptyset_2)$$

$$\frac{q_1 \ @ \ d \Downarrow_a d_1 \quad d_1 \neq \emptyset}{q_1 \parallel q_2 \ @ \ d \Downarrow_a d_1}\ (\text{Default } \neg\text{Null}) \qquad \frac{q_1 \ @ \ d \Downarrow_a \emptyset \quad q_2 \ @ \ d \Downarrow_a d_2}{q_1 \parallel q_2 \ @ \ d \Downarrow_a d_2}\ (\text{Default Null})$$

Figure 9 NRA Semantics. $q \ @ \ d \Downarrow_a d$

4.3 Translating CAMP to NRA

There are two key mismatches between the evaluation of CAMP (introduced in Section 2) and the evaluation of NRA that must be addressed by any translation. CAMP is parameterized by both an environment and input data whereas NRA is parameterized only by input data. Additionally, the output of CAMP is part of the lifted domain $\mathcal{D}? = \mathcal{D} + \mathbf{err}$, allowing any CAMP pattern to return a recoverable error. In contrast, NRA always returns data and has no concept of recoverable errors.

Figure 10 presents a compiler from CAMP to NRA that addresses both of these mismatches. The translation assumes (and preserves) a special encoding of both input and output to allow the semantics of a CAMP pattern to be expressed in NRA. The input of a compiled pattern is always a record with two components, E and D, storing the current environment and data. The output is always a bag. This bag is guaranteed to be either empty (representing a recoverable error) or a singleton of the data.

We will explain the translation starting with the simpler cases, not in the order presented. The rule for constants is trivial. The translations of **it** and **env** are a simple lookup, since they are both components of the input record.

Unary and binary operators ensure proper error propagation by taking advantage of the invariant that the returned data is a bag with zero or one elements. Mapping an operation over such a bag evaluates it on the data if present, and propagates the error otherwise

$$
\begin{aligned}
[\![d]\!] &= \{d\} \\
[\![\oplus p]\!] &= \chi_{\langle \oplus \mathbf{In} \rangle}([\![p]\!]) \\
[\![p_1 \otimes p_2]\!] &= \chi_{\langle \mathbf{In}.T_1 \otimes \mathbf{In}.T_2 \rangle}\left(\chi_{\langle [T_1 : \mathbf{In}] \rangle}([\![p_1]\!]) \times \chi_{\langle [T_2 : \mathbf{In}] \rangle}([\![p_2]\!])\right) \\
[\![\mathbf{map}\ p]\!] &= \left\{ \mathit{flatten}\left(\chi_{\langle [\![p]\!] \rangle}\left(\rho_{D/\{T\}}\left(\{[E : \mathbf{In}.E] * [T : \mathbf{In}.D]\}\right)\right)\right)\right\} \\
[\![\mathbf{assert}\ p]\!] &= \chi_{\langle [] \rangle}\left(\sigma_{\langle \mathbf{In} \rangle}([\![p]\!])\right) \\
[\![p_1 \| p_2]\!] &= [\![p_1]\!] \| [\![p_2]\!] \\
[\![\mathbf{it}]\!] &= \{\mathbf{In}.D\} \\
[\![\mathbf{let\ it} = p_1\ \mathbf{in}\ p_2]\!] &= \mathit{flatten}\left(\chi_{\langle [\![p_2]\!] \rangle}\left(\rho_{D/\{T\}}\left(\{[E : \mathbf{In}.E] * [T : [\![p_1]\!]]\}\right)\right)\right) \\
[\![\mathbf{env}]\!] &= \{\mathbf{In}.E\} \\
[\![\mathbf{let\ env} \mathrel{+}= p_1\ \mathbf{in}\ p_2]\!] &= \mathit{flatten}\Big(\pi_{\langle [\![p_2]\!] \rangle}\big(\\
&\qquad \pi_{\langle [E:\mathbf{In}.E_2] * [D:\mathbf{In}.D] \rangle}\big(\\
&\qquad\quad \rho_{E_2/\{T_2\}}\big(\pi_{\langle \mathbf{In} * [T_2:\mathbf{In}.E + \mathbf{In}.E_1] \rangle}\big(\\
&\qquad\qquad \rho_{E_1/\{T_1\}}\left(\{\mathbf{In} * [T_1 : [\![p_1]\!]]\}\right)\big)\big)\big)\Big) \\
\rho_{B/\{A\}}(q) &= \chi_{\langle \mathbf{In} - A \rangle}\left(\bowtie^{d}{}_{\langle \chi_{\langle [B:\mathbf{In}] \rangle}(\mathbf{In}.A) \rangle}(q)\right)
\end{aligned}
$$

Figure 10 Compiling CAMP to NRA.

(mapping $\emptyset$ to $\emptyset$). Binary operators store the two partial results in a record, then extract the components and apply the operator.

The translations of **assert** and orElse ($p_1 \| p_2$) take advantage of the translation mapping **err** to $\emptyset$. The selection operator is used for **assert**, along with a map that, in case of success, replaces true with the expected empty record.

To simplify the translation of the remaining patterns (**map**, **let it**, and **let env**), we introduce a derived operation, unnesting. $\rho_{B/\{A\}}(q)$ in Figure 10 unnests the nested bag stored in record attribute A, and renames its elements to attribute B. Given a bag $\{[A : \{a_1, ..., a_n\}, \overline{C_i : c_i}]\}$, unnest returns the bag $\{[B : a_1, \overline{C_i : c_i}], ..., [B : a_n, \overline{C_i : c_i}]\}$, using an intermediate bag $\{[B : a_1, A : \{a_1, ..., a_n\}, \overline{C_i : c_i}], ..., [B : a_n, A : \{a_1, ..., a_n\}, \overline{C_i : c_i}]\}$ created via a dependent join, then subtracting out the superfluous A attribute. This intermediate step is why we use a temporary name and have ρ rename as it unnests.

With unnesting in hand, let us look at the translation of **map** p from CAMP, which assumes that the current input data is a bag. In the NRA translation, that means that $\mathbf{In}.D$ is a bag. The compiler cannot, however, use the NRA map operation directly ($\chi_{\langle [\![p]\!] \rangle}(\mathbf{In}.D)$), as this would not preserve an important invariant of our translation: the input to a compiled pattern must be a record with the data and the environment. Instead, the map translation creates a singleton bag out of a record containing the environment (E) and the data bag. For the data it uses a temporary name, T. Unnesting allows us to obtain the required input data, a bag of records, each with the (appropriate part of the) data and the environment. This input can now be used with map (χ). Since the result is a bag of singleton bags, we finish the translation by flattening the result down to a bag of the data, and then lifting it back into a singleton bag (to preserve the output invariant).

The **let it** and **let env** translations use the unnesting operation to similar effect. Note that the translation of p_1 is a bag with at most one element, so the map accomplishes the

required sequencing and error propagation automatically. The **let env** rule needs to use unnesting twice: once to calculate and unnest the environment returned by p_1, and once to unnest the concatenation of that environment with the current environment. Note that this concatenation is done using the + operation, which returns $\emptyset$ if the environments are not compatible. This is exactly the desired semantics, since $\emptyset$ in NRA represents **err** in CAMP.

4.4 Correctness

Theorem 5 asserts that the translation from CAMP to NRA in Figure 10 preserves semantics. A CAMP pattern that evaluates to d becomes an NRA query that evaluates to $\{d\}$ and a CAMP pattern that evaluates to **err** becomes an NRA query that evaluates to $\emptyset$.

▶ **Theorem 5** (Correctness of compiler from CAMP to NRA)**.**

$$\sigma \vdash p \mathbin{@} d_1 \Downarrow_r d_2 \iff [\![p]\!] \mathbin{@} ([E : \sigma] * [D : d_1]) \Downarrow_a \{d_2\}$$
$$\sigma \vdash p \mathbin{@} d_1 \Downarrow_r \mathbf{err} \iff [\![p]\!] \mathbin{@} ([E : \sigma] * [D : d_1]) \Downarrow_a \emptyset$$

This theorem is verified by the accompanying mechanization. The proof is straightforward, relying on the invariants that the translation assumes and ensures. Due to the computational nature of our mechanized semantics, much of the apparent complexity of the proofs is reduced to simple computation. We also prove that given a CAMP pattern, the translated NRA query is at most a constant factor larger.

5 NRA to NNRC

Having introduced CAMP and a translation from CAMP to NRA, we now wish to go in the other direction, showing how to translate NRA back to CAMP. Rather than go directly from NRA to CAMP, we stage the translation through the named nested-relational calculus (NNRC), which provides a more declarative model and is well known to have the same expressivity as NRA [34]. This section introduces NNRC and a translation from NRA to NNRC. Section 6 completes the cycle, introducing a translation from NNRC back to CAMP.

Staging through NNRC allows us to split apart different aspects of the translation. As we will see in Section 5.3, all of the dependent queries in NRA get translated into a single NNRC construct, the comprehension. Section 6, which introduces a translation from NNRC back to CAMP, can then focus on the comprehension, without needing to separately handle all the dependent constructs provided by NRA. Taking this detour also allows us to mechanize the well-known equivalence between NRA and NNRC (staging the translation through CAMP).

5.1 Named Nested Relational Calculus

We assume a sufficiently large set of variables $\{x, y, ...\}$. The calculus is similar to the one used in [35], with a bag semantics.

▶ **Definition 6** (NNRC syntax)**.**

$$\text{(exprs) } e \quad ::= \quad x \mid d \mid \oplus e_1 \mid e_1 \otimes e_2 \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid \ \{e_2 \mid x \in e_1\} \mid e_1\ ?\ e_2 : e_3$$

In this grammar, x is a variable, d is constant data and $\oplus$ and $\otimes$ are unary and binary operators, as defined in Section 2.2. The **let** expression allows for dependent sequencing: expression e_1 is evaluated and its result bound to x in the environment, which is then used to evaluate e_2. The bag comprehension $\{e_2 \mid x \in e_1\}$ first evaluates expression e_1, producing

$$\frac{\sigma(x) = d}{\sigma \vdash x \Downarrow_c d}\ \text{(Variable)} \qquad \frac{}{\sigma \vdash d \Downarrow_c d}\ \text{(Constant)} \qquad \frac{\sigma \vdash e \Downarrow_c d_0 \quad \oplus d_0 = d_1}{\sigma \vdash \oplus e \Downarrow_c d_1}\ \text{(Unary Operator)}$$

$$\frac{\sigma \vdash e_1 \Downarrow_c d_1 \quad \sigma \vdash e_2 \Downarrow_c d_2 \quad d_1 \otimes d_2 = d_3}{\sigma \vdash e_1 \otimes e_2 \Downarrow_c d_3}\ \text{(Binary Operator)}$$

$$\frac{\sigma \vdash e_1 \Downarrow_c d_1 \quad (x : d_1, \sigma) \vdash e_2 \Downarrow_c d_2}{\sigma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \Downarrow_c d_2}\ \text{(Let)} \qquad \frac{\sigma \vdash e_1 \Downarrow_c \emptyset}{\sigma \vdash \{e_2 \mid x \in e_1\} \Downarrow_c \emptyset}\ \text{(For } \emptyset\text{)}$$

$$\frac{\sigma \vdash e_1 \Downarrow_c \{d\} \cup s \quad (x : d, \sigma) \vdash e_2 \Downarrow_c d_0 \quad \sigma \vdash \{e_2 \mid x \in s\} \Downarrow_c s_0}{\sigma \vdash \{e_2 \mid x \in e_1\} \Downarrow_c \{d_0\} \cup s_0}\ \text{(For)}$$

$$\frac{\sigma \vdash e_1 \Downarrow_c \mathsf{true} \quad \sigma \vdash e_2 \Downarrow_c d_2}{\sigma \vdash e_1 \ ?\ e_2 : e_3 \Downarrow_c d_2}\ \text{(If True)} \qquad \frac{\sigma \vdash e_1 \Downarrow_c \mathsf{false} \quad \sigma \vdash e_3 \Downarrow_c d_3}{\sigma \vdash e_1 \ ?\ e_2 : e_3 \Downarrow_c d_3}\ \text{(If False)}$$

Figure 11 NNRC Semantics. $\boxed{\sigma \vdash e \Downarrow_c d}$

a bag, then expression e_2 is evaluated with x bound to the current element. The result of the comprehension is a bag of these results. The conditional $e_1 \ ?\ e_2 : e_3$ first evaluates e_1; if the result is true, it evaluates e_2, otherwise it evaluates e_3.

5.2 Semantics

An environment σ is a mapping from a finite set of variables to values. We write $(x : d, \sigma)$ for the environment σ extended with variable x mapped to data d. Figure 11 describes the semantics of NNRC expressions using the judgment $\sigma \vdash e \Downarrow_c d$, meaning: under environment σ, expression e evaluates to d. The subscript $_c$ on the relation $\Downarrow_c$ stands for calculus. Unlike CAMP and NRA, expressions are not queries over input data, but are parameterized solely by their environment.

The rule for variables looks up the given variable in the environment and returns the associated data. Constant expressions return the given constant, irrespective of the environment. Unary and binary operator expressions evaluate the given expressions in the current environment, and then apply the given operator to the results.

Let expressions evaluate the first expression in the current environment and then evaluate the second expression in an environment enriched with a binding from the given variable to the result of evaluating the first expression.

Comprehensions, $\{e_2 \mid x \in e_1\}$, are similar to let expressions, except that e_1 returns a bag, and e_2 is evaluated with x bound to each element of that bag in turn. Rule For encodes this recursion, evaluating e_1 and then picking an element of the resulting bag and running e_2 on it. The result is unioned with the evaluation of a comprehension of e_2 over the remainder of the bag. Rule For $\emptyset$ enables this recursion to terminate.

The rules for the final type of expression, the conditional, are straightforward. The first expression is evaluated and its result used to determine which branch to evaluate.

5.3 Translation from NRA to NNRC

When compiling NRA queries to NNRC expressions, there are two main problems that need to be addressed: the different contexts and the need for variable names.

$$
\begin{aligned}
[\![d]\!]_x &= d \\
[\![\mathbf{In}]\!]_x &= x \\
[\![\oplus q]\!]_x &= \oplus [\![q]\!]_x \\
[\![q_1 \otimes q_2]\!]_x &= [\![q_1]\!]_x \otimes [\![q_2]\!]_x \\
[\![\chi_{\langle q_2 \rangle}(q_1)]\!]_x &= \{[\![q_2]\!]_t \mid t \in [\![q_1]\!]_x\} && t \text{ is } \mathbf{fresh} \\
[\![\sigma_{\langle q_2 \rangle}(q_1)]\!]_x &= \mathit{flatten}(\{[\![q_2]\!]_t \ ?\ \{t\} : \emptyset \mid t \in [\![q_1]\!]_x\}) && t \text{ is } \mathbf{fresh} \\
[\![q_1 \times q_2]\!]_x &= \mathit{flatten}(\{\{t_1 * t_2 \mid t_2 \in [\![q_2]\!]_x\} \mid t_1 \in [\![q_1]\!]_x\}) && t_1 \text{ is } \mathbf{fresh} \wedge t_2 \text{ is } \mathbf{fresh} \\
[\![\bowtie^d{}_{\langle q_2 \rangle}(q_1)]\!]_x &= \mathit{flatten}(\{\{t_1 * t_2 \mid t_2 \in [\![q_2]\!]_{t_1}\} \mid t_1 \in [\![q_1]\!]_x\}) && t_1 \text{ is } \mathbf{fresh} \wedge t_2 \text{ is } \mathbf{fresh} \\
[\![q_1 \parallel q_2]\!]_x &= \mathbf{let}\ t = [\![q_1]\!]_x\ \mathbf{in}\ ((t = \emptyset)\ ?\ [\![q_2]\!]_x : t) && t \text{ is } \mathbf{fresh}
\end{aligned}
$$

Figure 12 Compiling NRA to NNRC.

NRA queries and NNRC expressions run in different contexts. NRA queries do not have an environment, but are run against specified data. In contrast, NNRC expressions have only an environment and no other implicit data. The translation from NRA queries to NNRC expressions needs to store the input data for the query in the environment of the compiled expression. We parameterize the translation with the variable used for this mapping (subscript $_x$ in Figure 12).

The compiler also needs a way to manufacture variable names for use with NNRC let expressions and comprehensions. To simplify the presentation, we assume that we have a way to generate sufficiently fresh variables (variables that are distinct from any other variables used in the expression). We call such variables **fresh**. We revisit this in Section 5.4.

Figure 12 presents a compiler from NRA queries to NNRC expressions. Since NNRC provides declarative comprehensions, the compiler resembles a denotational semantics for NRA. The translation translates the identity query, **In**, as returning the value of x, the variable that holds the current data. The rest of the translation ensures that this association is preserved. Constants are translated to constants, and the sub-queries of unary and binary operator queries are translated and the specified operator reapplied.

The map query, $\chi_{\langle q_2 \rangle}(q_1)$, is expressed as a straightforward comprehension using a fresh variable t. The selection query $\sigma_{\langle q_2 \rangle}(q_1)$ is similarly translated to a comprehension. The body of the comprehension either returns a singleton bag containing the element of the bag returned by q_1 or $\emptyset$. The result is thus a bag of (empty or singleton) bags, which is flattened using the *flatten* operator.

Both the product $q_1 \times q_2$ and the dependent join $\bowtie^d{}_{\langle q_2 \rangle}(q_1)$ are expressed as two nested comprehensions. The resulting bag of bags is then flattened. The crucial difference between them is which data is used in the translation of q_2 in the inner comprehension. For the product, the same data, bound to x, is used. For the dependent join, the variable used in the outer comprehension is used, so the translation uses the current element of q_1 as the current data. This difference succinctly captures the dependency.

The final type of query, the default query, is translated into a conditional expression that inspects the result t of q_1, and evaluates q_2 if it is $\emptyset$ or returns t otherwise.

The usage of let expressions when translating default is the main reason we extend the traditional NNRC with let expressions. Existing work generally assumes that all the data is a bag. In that case, the let expression is not needed, since in that case we can express $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$ as $\mathit{flatten}(\{e_2 \mid x \in \{e_1\}\})$. However, that works only if we assume that e_2

returns a bag (otherwise the flatten may not even be well-typed). Since we did not want to limit the data our programs manipulate, we elected to add in let expressions to handle the case where the data may not be a bag.[2]

5.4 Correctness

The accompanying mechanization proves Theorem 7: our translation preserves semantics. NRA query q against data d evaluates to the same result (or lack thereof) as its translation into NNRC, $[\![q_1]\!]_x$, with an environment that associates x with the input d.

▶ **Theorem 7** (Correctness of compiler from NRA to NNRC).

$$\textit{If } \sigma(x) = d_1 \textit{ then } q \mathbin{@} d_1 \Downarrow_a d_2 \iff \sigma \vdash [\![q]\!]_x \Downarrow_c d_2$$

The proof of Theorem 7 is straightforward, taking advantage of the previously mentioned similarity between the denotational semantics of NRA and the translation to NNRC. The proof essentially formalizes and verifies that correspondence.

The main additional challenge in the mechanization relates to the use of fresh variables. In Figure 12 we assumed the existence of **fresh**, which produces a "sufficiently unique" variable. In the mechanized semantics, this is formalized as a variety of freshness functions that produce variables fresh with respect to the relevant parts of sub-expressions and the input variable. The proof then verifies that each picked variable is indeed sufficiently fresh. Full details are included in the accompanying mechanization.

The mechanization also establishes that the compiler in Figure 12 produces a pattern at most a constant times larger than the input NRA. As a consequence, the compiler can clearly be observed to run in time polynomial in the input size.

6 NNRC to CAMP

This section completes the cycle from Figure 2, showing how to compile NNRC, as defined and presented in Section 5, back into the CAMP language from Section 2. This establishes the equivalence of all the languages, exhibiting compilers from CAMP to NRA to NNRC and back to CAMP. All these compilers cause at most a linear increase in code size, so the result of a full cycle of compilation yields a program semantically equivalent to the original and at most a constant times larger.

6.1 Translation from NNRC to CAMP

Figure 13 presents a compiler from NNRC to CAMP. Compilation is mostly straightforward, using CAMP's environment (**env**) as a target for NNRC's environment (σ). Since NNRC does not represent a query over data, the compilation does not need **it** except as part of the translation scheme. In particular, a compiled NNRC expression returns the same result regardless of the input data to the pattern (verified in the mechanization).

We need to be careful of shadowing (i.e., an inner variable definition hiding the previous definition for a variable with the same name), however, as the two languages' environments have different semantics when a variable appears twice. In CAMP, adding a variable to the environment ensures that its value is equal to the value of the previous binding for that

[2] Note that our compilation of CAMP to NRA (Section 4.3) actually enforces this restriction, since it lifts all data into a singleton bag. Nonetheless, we prefer the more general presentation.

$$
\begin{aligned}
[\![x]\!] &= \mathbf{env}.x \\
[\![d]\!] &= d \\
[\![\oplus e]\!] &= \oplus [\![e]\!] \\
[\![e_1 \otimes e_2]\!] &= [\![e_1]\!] \otimes [\![e_2]\!] \\
[\![\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2]\!] &= \mathbf{let\ it} = [\![e_1]\!]\ \mathbf{in\ let\ env} \mathrel{+}= [x : \mathbf{it}]\ \mathbf{in}\ [\![e_2]\!] \\
[\![\{e_2 \mid x \in e_1\}]\!] &= \mathbf{let\ it} = [\![e_1]\!]\ \mathbf{in\ mapall}\ (\mathbf{let\ env} \mathrel{+}= [x : \mathbf{it}]\ \mathbf{in}\ [\![e_2]\!]) \\
[\![e_1\ ?\ e_2 : e_3]\!] &= \mathbf{let\ env} \mathrel{+}= [x : [\![e_1]\!]]\ \mathbf{in} \qquad\qquad x \text{ is } \mathbf{fresh} \\
&\qquad ((\neg\neg \mathbf{env}.x) \wedge [\![e_2]\!]) \,\|\, ((\neg \mathbf{env}.x) \wedge [\![e_3]\!])
\end{aligned}
$$

Figure 13 Compiling NNRC to CAMP.

variable. To resolve this, we disallow shadowing, and assume that all comprehensions and let bindings in an NNRC expression use distinct variable names. It is always possible to rename the variables used in comprehensions and let expressions so that they are all distinct. This is verified by our mechanization.

Compiling variables, data, and operators is straightforward, using **env** to explicitly access the environment as needed. The **let** expression uses an outer **let it** to handle the required dependent sequencing, and then uses **let env** to add the calculated data to the environment, bound to the requested variable. As this variable is assumed unique (in the program), the compatibility clause of **let env** will always be satisfied.

Comprehensions ($\{e_2 \mid x \in e_1\}$) are compiled similarly to **let** sequencing expressions, except that **mapall**[3] is used to map $[\![e_2]\!]$ over each element in the collection produced by $[\![e_1]\!]$, with the variable x bound to the appropriate element.

Compiling the conditional expression $e_1\ ?\ e_2 : e_3$ is slightly more complicated, as the pattern language only provides a (single-branched) orElse pattern ($p_1 \| p_2$), which handles errors. We use a **let env** pattern to evaluate and remember the compiled first expression. We assert that it evaluated to true and sequence the true branch e_2 with that assertion. If the assertion fails, that branch will not be executed. If this series of patterns fails, we use the orElse pattern to recover. In this case, either the initial assertion failed (e_1 either failed to match or did not return true), or e_2 failed. To distinguish among these possibilities, the recovery branch of the orElse first checks that the negation of (the remembered value of) e_1 is true. If it is true, then e_3 is evaluated. Since this branch ensures that e_1 evaluates to a Boolean (since the negation operation is defined only on Booleans), we use double negation to enforce that property for the first branch as well.

When using this let statement we need to be careful that we always pick a "fresh" (sufficiently unique) variable name. As we discussed earlier in the context of shadowing, the environment used by CAMP enforces that variables bound multiple times are all bound to the same data. This is not the semantics that we want. To avoid problems with duplicate variables in the source NNRC, we employed alpha-renaming. To avoid a similar type of problem in our compilation, we assume a way to generate a **fresh** variable name that does not conflict with any other variable name. This is used both in our translation of conditional expressions, and in our previous definition of **mapall**.

[3] Recall that **mapall** as well as the $\wedge$ operation were defined in Figure 7.

6.2 Correctness

The mechanization proves Theorem 8, verifying that the compiler is correct.

▶ **Theorem 8** (Correctness of compiler from NNRC to CAMP). *Assuming that σ is well-formed (formalized in the accompanying mechanization):*

$$\sigma \vdash e \Downarrow_c d \iff \sigma \vdash [\![e]\!] @ d_0 \Downarrow_r d$$

Note that d_0 is unconstrained, as a compiled NNRC expression does not access **it**.[4]

The proof is staged: we first prove correct a naive compiler that does not use **let env** or fresh variables, instead performing redundant computation. We then prove it produces a pattern semantically equivalent to the pattern produced by the compiler in Figure 13, where redundant computation is avoided to ensure the compiler preserves the original complexity. Semantics preservation of our compiler follows by composing these results.

Once more, formalizing these proofs requires a proper treatment of freshness. To do this, the mechanization formalizes the concept of free and bound variables, and provides a way to obtain a new variable that is not in a provided set. Ensuring that all required freshness conditions are met is a major challenge of the proofs and mechanization, requiring many well-formedness conditions to be defined on the environment and verified as preserved by the induction. Our mechanization also provides a method "unshadow" that renames variables in an NNRC expression to ensure that all bound variables are distinct (both within expression and from a provided environment). The mechanization proves that this transformation preserves semantics. Thus, our actual compiler runs "unshadow" and compiles the result, allowing all NNRC expressions to be compiled.

We have also mechanized a proof that the compiler in Figure 13 produces a pattern at most a constant times larger than the input NNRC. The compiler also clearly runs in time polynomial in the input size, although this property is not mechanized.

7 Type Checking

This paper has introduced a new language, CAMP, as well as formalized variants of existing languages, NRA and NNRC. All the semantics presented were partial; ill-typed programs may not evaluate. All of our compilers were careful to translate ill-typed programs into ill-typed programs. The correctness theorems all guarantee that a translation can evaluate successfully if and only if the original could evaluate successfully.

This section introduces types for data and operators (Section 7.1) and formalizes the type systems for CAMP (Section 7.2), NRA (Section 7.3), and NNRC (Section 7.4). Each type system is sound: well-typed programs always evaluate to a result. Note that in the case of CAMP, a recoverable error is a valid result of pattern matching. Furthermore, as formalized in Section 7.5, all of our compilers are type-preserving. Well-typed programs are compiled to well-typed programs (with an associated type) and vice versa.

Section 7.6 applies this result to type inference for the languages. In particular, this lets us build on related work [35] to equip CAMP with a polymorphic type inference algorithm, and prove that polymorphic type inference for CAMP is NP-complete.

[4] This is of course verified by the accompanying mechanization.

7.1 Types for Data Model and Operators

Our types for data include primitive types `NIL`, `INT`, `BOOL`, and `STRING`. The type of a (homogeneous) bag with elements of type τ is written $\{\tau\}$, punning the notation used for data bags. Similarly, $[\overline{A_i : \tau_i}]$ is the type of a record with attributes A_i, each with data of type τ_i. We use the notation $d :_d \tau$ to mean that data d has type τ.

As with data records, we define a notion of compatibility for record types. Two type records are considered compatible if any overlapping attributes have the same type. Note that two records can have compatible types but incompatible data, causing recoverable match failure. We define $*$ and $+$ as for data records: $[\overline{A_i : \tau_{A_i}}] * [\overline{B_j : \tau_{B_j}}]$ concatenates two record types, favoring the types of A's attributes in case of conflict. Compatible concatenation, $[\overline{A_i : \tau_{A_i}}] + [\overline{B_j : \tau_{B_j}}]$, also concatenates the record types, but is defined only if the two records are compatible.

The types of the unary operators, written $\oplus :_o \tau_1 \to \tau_2$, written here for a given value d of type τ, are as follows:

$$
\begin{array}{ll}
\mathit{identity}\ d :_o \tau \to \tau & \mathit{flatten}\ d :_o \{\{\tau\}\} \to \{\tau\} \\
\neg d :_o \texttt{BOOL} \to \texttt{BOOL} & [A : d] :_o \tau \to [A : \tau] \\
\{d\} :_o \tau \to \{\tau\} & d.A :_o [A : \tau, \overline{B_i : \tau_i}] \to \tau \\
\#d :_o \{\tau\} \to \texttt{INT} & d - A :_o [A : \tau, \overline{B_i : \tau_i}] \to [\overline{B_i : \tau_i}]
\end{array}
$$

The types of the binary operators, written $\otimes :_o \tau_1 \to \tau_2 \to \tau_3$, given here for two given values d_1 and d_2 of types τ_1 and τ_2 respectively, are as follows:

$$
\begin{array}{l}
d_1 = d_2 :_o \tau \to \tau \to \texttt{BOOL} \\
d_1 \in d_2 :_o \tau \to \{\tau\} \to \texttt{BOOL} \\
d_1 \cup d_2 :_o \{\tau\} \to \{\tau\} \to \{\tau\} \\
d_1 * d_2 :_o [\overline{A_i : \tau_{A_i}}] \to [\overline{B_j : \tau_{B_j}}] \to [\overline{A_i : \tau_{A_i}}] * [\overline{B_j : \tau_{B_j}}] \\
d_1 + d_2 :_o [\overline{A_i : \tau_{A_i}}] \to [\overline{B_j : \tau_{B_j}}] \to \{[\overline{A_i : \tau_{A_i}}] + [\overline{B_j : \tau_{B_j}}]\}
\end{array}
$$

7.2 CAMP Type System

Figure 14 presents a type system for CAMP. Where the CAMP semantics use a data environment σ, the CAMP type system uses a type context Γ. And where the CAMP semantics use an input datum **it**, the types of CAMP constructs are functions from an input type τ_0. We employ the same environment reification at the type level as we did for data, allowing the context Γ to be reified as a record type. The environment and type context can thus be related via a data typing relation, $\sigma :_d \Gamma$.

Most of the type rules are standard. We assume that **err** can be given any type, and so the rules do not need to account for recoverable errors, since **err** unifies with the type ascribed when a pattern does not have a recoverable error.

The Assert rule has an empty record type since **assert** returns an empty record if the assertion holds. The **env** rule reifies the context as a record type, just like the semantics reify the environment. The Let **env** rule checks the type of the second pattern with a context enriched by the bindings in the type record that type the first expression. Recall that the "+" operator ensures that these new bindings are compatible with the current context.

Theorem 9 asserts that the type system for CAMP given in Figure 14 is sound with respect to the semantics given in Figure 3. Given a well typed pattern, evaluating that pattern with an environment and input data that are appropriately typed will always yield a

$$\frac{d :_d \tau}{\Gamma \vdash d :_r \tau_0 \to \tau}\ (\text{Constant}) \qquad \frac{\Gamma \vdash p :_r \tau_0 \to \tau_1 \quad \oplus :_o \tau_1 \to \tau_2}{\Gamma \vdash \oplus p :_r \tau_0 \to \tau_2}\ (\text{Unary Operator})$$

$$\frac{\Gamma \vdash p_1 :_r \tau_0 \to \tau_1 \quad \Gamma \vdash p_2 :_r \tau_0 \to \tau_2 \quad \otimes :_o \tau_1 \to \tau_2 \to \tau_3}{\Gamma \vdash p_1 \otimes p_2 :_r \tau_0 \to \tau_3}\ (\text{Binary Operator})$$

$$\frac{\Gamma \vdash p :_r \tau_0 \to \tau_1}{\Gamma \vdash \mathbf{map}\ p :_r \{\tau_0\} \to \{\tau_1\}}\ (\text{Map}) \qquad \frac{\Gamma \vdash p :_r \tau_0 \to \texttt{BOOL}}{\Gamma \vdash \mathbf{assert}\ p :_r \tau_0 \to [\,]}\ (\text{Assert})$$

$$\frac{\Gamma \vdash p_1 :_r \tau_0 \to \tau_1 \quad \Gamma \vdash p_2 :_r \tau_0 \to \tau_1}{\Gamma \vdash p_1 \| p_2 :_r \tau_0 \to \tau_1}\ (\text{OrElse}) \qquad \frac{}{\Gamma \vdash \mathbf{it} :_r \tau_0 \to \tau_0}\ (\mathbf{it})$$

$$\frac{\Gamma \vdash p_1 :_r \tau_0 \to \tau_1 \quad \Gamma \vdash p_2 :_r \tau_1 \to \tau_2}{\Gamma \vdash \mathbf{let}\ \mathbf{it} = p_1\ \mathbf{in}\ p_2 :_r \tau_0 \to \tau_2}\ (\text{Let } \mathbf{it}) \qquad \frac{}{\Gamma \vdash \mathbf{env} :_r \tau_0 \to \Gamma}\ (\mathbf{env})$$

$$\frac{\Gamma \vdash p_1 :_r \tau_0 \to [\overline{A_i : \tau_{A_i}}] \quad \left(\Gamma + [\overline{A_i : \tau_{A_i}}]\right) \vdash p_2 :_r \tau_0 \to \tau_2}{\Gamma \vdash \mathbf{let}\ \mathbf{env} \mathrel{+}= p_1\ \mathbf{in}\ p_2 :_r \tau_0 \to \tau_2}\ (\text{Let } \mathbf{env})$$

Figure 14 Type rules for CAMP. $\boxed{\Gamma \vdash p :_r \tau_0 \to \tau_1}$

result – data or a recoverable error. If the result is data, then it will be appropriately typed. The proof proceeds by induction and is included in the mechanization.

▶ **Theorem 9** (Soundness of type system for CAMP).

$$\textit{if}\ (\Gamma \vdash p :_r \tau_0 \to \tau_1) \wedge (\sigma :_d \Gamma) \wedge (d_0 :_d \tau_0)\ \ \textit{then}\ \exists d_1?, (\sigma \vdash p \mathbin{@} d_0 \Downarrow_r d_1?) \wedge (d_1? :_d \tau_1)$$

7.3 NRA Type System

Figure 15 presents a type system for NRA (defined in Section 4.1). A well-typed query has an input type τ_0 and an output type τ_1, written $q :_a \tau_0 \to \tau_1$. Note how the difference between product and dependent join manifests as the difference in the input type of q_2. This is analogous to their different translations to NNRC in Figure 12.

The type system for NRA given in Figure 15 is sound with respect to the semantics given in Figure 9. Well typed queries, when applied to appropriately typed input data, will always evaluate successfully with appropriately typed output data. This is formalized by Theorem 10, which is verified in the accompanying mechanization.

▶ **Theorem 10** (Soundness of type system for NRA).

$$\textit{if}\ (q :_a \tau_0 \to \tau_1) \wedge (d_0 :_d \tau_0)\ \ \textit{then}\ \exists d_1, (q \mathbin{@} d_0 \Downarrow_a d_1) \wedge (d_1 :_d \tau_1)$$

7.4 NNRC Type System

Figure 16 presents a type system for the third language under discussion, NNRC (introduced in Section 5.1). This type system uses a type context to type an expression, similarly to CAMP. Unlike our system for CAMP, however, we do not need to reify contexts as records. Instead, we treat contexts as ordered lists. Adding x with type τ to the context Γ, written $(x : \tau, \Gamma)$, always succeeds, masking any previous binding for x. An NNRC environment σ has type Γ, written $\sigma : \Gamma$, if corresponding elements have the same variables, and the data in σ is typed by the corresponding type in Γ.

This type system for NNRC (Figure 16) is sound with respect to the semantics for NNRC given in Figure 11. Evaluating a well typed expression in an appropriately typed context

$$\frac{}{\mathbf{In} :_a \tau_0 \to \tau_0}\ (\text{ID}) \qquad \frac{d :_d \tau}{d :_a \tau_0 \to \tau}\ (\text{Constant}) \qquad \frac{q :_a \tau_0 \to \tau_1 \quad \oplus :_o \tau_1 \to \tau_2}{\oplus q :_a \tau_0 \to \tau_2}\ (\text{Unary Operator})$$

$$\frac{q_1 :_a \tau_0 \to \tau_1 \quad q_2 :_a \tau_0 \to \tau_2 \quad \otimes :_o \tau_1 \to \tau_2 \to \tau_3}{q_1 \otimes q_2 :_a \tau_0 \to \tau_3}\ (\text{Binary Operator})$$

$$\frac{q_1 :_a \tau_0 \to \{\tau_1\} \quad q_2 :_a \tau_1 \to \tau_2}{\chi_{\langle q_2 \rangle}(q_1) :_a \tau_0 \to \{\tau_2\}}\ (\text{Map}) \qquad \frac{q_1 :_a \tau_0 \to \{\tau_1\} \quad q_2 :_a \tau_1 \to \texttt{BOOL}}{\sigma_{\langle q_2 \rangle}(q_1) :_a \tau_0 \to \{\tau_1\}}\ (\text{Select})$$

$$\frac{q_1 :_a \tau_0 \to \{[\overline{A_i : \tau_{A_i}}]\} \quad q_2 :_a \tau_0 \to \{[\overline{B_j : \tau_{B_j}}]\}}{q_1 \times q_2 :_a \tau_0 \to \{[\overline{A_i : \tau_{A_i}}] * [\overline{B_j : \tau_{B_j}}]\}}\ (\text{Product})$$

$$\frac{q_1 :_a \tau_0 \to \{[\overline{A_i : \tau_{A_i}}]\} \quad q_2 :_a [\overline{A_i : \tau_{A_i}}] \to \{[\overline{B_j : \tau_{B_j}}]\}}{\bowtie^d_{\langle q_2 \rangle}(q_1) :_a \tau_0 \to \{[\overline{A_i : \tau_{A_i}}] * [\overline{B_j : \tau_{B_j}}]\}}\ (\text{DJ})$$

$$\frac{q_1 :_a \tau_0 \to \{\tau_1\} \quad q_2 :_a \tau_0 \to \{\tau_1\}}{q_1 \parallel q_2 :_a \tau_0 \to \{\tau_1\}}\ (\text{Default})$$

Figure 15 Type rules for the Nested Relational Algebra. $q :_a \tau_0 \to \tau_1$

$$\frac{}{\Gamma \vdash x :_c \Gamma(x)}\ (\text{Variable}) \qquad \frac{d :_d \tau}{\Gamma \vdash d :_c \tau}\ (\text{Constant}) \qquad \frac{\Gamma \vdash e :_c \tau_1 \quad \oplus :_o \tau_1 \to \tau_2}{\Gamma \vdash \oplus e :_c \tau_2}\ (\text{Unary Operator})$$

$$\frac{\Gamma \vdash e_1 :_c \tau_1 \quad \Gamma \vdash e_2 :_c \tau_2 \quad \otimes :_o \tau_1 \to \tau_2 \to \tau_3}{\Gamma \vdash e_1 \otimes e_2 :_c \tau_3}\ (\text{Binary Operator})$$

$$\frac{\Gamma \vdash e_1 :_c \tau_1 \quad (x : \tau_1, \Gamma) \vdash e_2 :_c \tau_2}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 :_c \tau_2}\ (\text{Let}) \qquad \frac{\Gamma \vdash e_1 :_c \{\tau_1\} \quad (x : \tau_1, \Gamma) \vdash e_2 :_c \tau_2}{\Gamma \vdash \{e_2 \mid x \in e_1\} :_c \{\tau_2\}}\ (\text{For})$$

$$\frac{\Gamma \vdash e_1 :_c \texttt{BOOL} \quad \Gamma \vdash e_2 :_c \tau \quad \Gamma \vdash e_3 :_c \tau}{\Gamma \vdash e_1\ ?\ e_2 : e_3 :_c \tau}\ (\text{If})$$

Figure 16 Type rules for the Named Nested Relational Calculus. $\Gamma \vdash e :_c \tau$

will always succeed with appropriately typed data. This is formalized by Theorem 11, which is verified in the accompanying mechanization.

▶ **Theorem 11** (Soundness of type system for NNRC).

if $(\Gamma \vdash e :_c \tau) \wedge (\sigma :_d \Gamma)$ *then* $\exists d, (\sigma \vdash e \Downarrow_c d) \wedge (d :_d \tau)$

7.5 Type Preservation

As mentioned earlier, all our compilers are semantics preserving. In addition to preserving output data upon successful evaluation, they also preserve errors. A compiled program evaluates successfully if and only if the source program does.

All the compilers presented in this paper also preserve types. This is formalized in Theorem 12 and verified in the accompanying mechanization. This theorem asserts that the compilers preserve types in both directions: well-typed source programs guarantee well-typed compiled programs and well-typed compiled programs can only result from well-typed source programs. The forwards direction is the more traditional type preservation result, and ensures

that type-checking the source program suffices. We will discuss the backwards direction in Section 7.6.

▶ **Theorem 12** (Type Preservation). *Assuming that* Γ *is well-formed (formalized in the accompanying mechanization):*

$$\begin{array}{lll} \textit{CAMP}\leftrightarrow\textit{NRA}: & & \Gamma \vdash p :_r \tau_0 \to \tau_1 \quad \Leftrightarrow \quad [\![p]\!] :_a [E : \Gamma, D : \tau_0] \to \{\tau_1\} \\ \textit{NRA} \leftrightarrow \textit{NNRC}: & \textit{if } \Gamma(x) = \tau_0 \quad \textit{then} & q :_a \tau_0 \to \tau_1 \quad \Leftrightarrow \quad \Gamma \vdash [\![q]\!]_x :_c \tau_1 \\ \textit{NNRC}\leftrightarrow\textit{CAMP}: & & \Gamma \vdash e :_c \tau_1 \quad \Leftrightarrow \quad \Gamma \vdash [\![e]\!] :_r \tau_0 \to \tau_1 \end{array}$$

The accompanying mechanization formalizes the definition of well-formedness, which is needed to ensure that the context does not interfere with the variables introduced by the translations. We have chosen to omit these details from the paper, leaving them to the accompanying mechanization. The empty context is always well-formed.

For all of the compilations, when proving type preservation, it is helpful to prove that the typing rules are generally invertible. For the translation from CAMP to NRA, it is also helpful to derive an (invertible) type rule for the unnesting operator from Figure 10:

$$\frac{q : \tau_0 \to \{[A : \{\tau_A\}, \overline{C_i : \tau_{C_i}}]\}}{\rho_{B/\{A\}}\,(q) : \tau_0 \to \{[B : \tau_A, \overline{C_i : \tau_{C_i}}]\}}\ (\text{Unnest})$$

In Section 6.1 we observed that the translation from NNRC to CAMP produces a pattern that ignores the input. This is apparent in the type preservation theorem, which allows the input type of an NNRC expression compiled to CAMP to be ascribed any type. This polymorphic type expresses that the CAMP pattern never looks at the input.

7.6 Type Inference

The results in Theorem 12 are bidirectional. The forwards direction is a typical statement of type preservation. As a statement about type checking, the backwards direction of these theorems are not very interesting. Why type check the results of a compilation when we could just type check the source? However, the bidirectionality of Theorem 12 allows us to connect not just type checking between the languages, but also type inference. Because our compilers all run in polynomial time and space, Theorem 12 ensures that type inference for one language can be used for another.

In particular, we can use this to build on related work and derive a polymorphic type inference algorithm for CAMP. Given a CAMP pattern p, we compile it to NRA using the compiler introduced in Section 4.3. We then compile the resulting NRA to NNRC using the compiler from Section 5.3. We then apply the polymorphic type inference algorithm for NNRC introduced by Van den Bussche and Vansummeren [35] and use Theorem 12 to recover the type of the original CAMP pattern. (Note that if a compilation from CAMP to NRA is well-typed, it must have a bag type, so the theorem indeed allows us to read any possible inferred type "backwards" to CAMP.)

Theorem 12 allows us to take advantage of Van den Bussche and Vansummeren's NP-completeness result for polymorphic type inference [35], proving that polymorphic type inference for CAMP is NP-complete using a standard reduction argument. Starting with an NNRC expression, the compiler introduced in Section 6.1 can, in polynomial time, compile it to a CAMP pattern that is polynomial in the size of the original expression. Thus, any polynomial time polymorphic type inference algorithm for CAMP would yield a polynomial time polymorphic type inference algorithm for NNRC. Since polymorphic type inference for

NNRC is NP-complete, this means that polymorphic type inference for CAMP (and NRA) is also NP-complete.

8 Related Work

Production rule languages hark back to Forgy's seminal work on OPS5 [19]. OPS5 already has most of the features we formalize in CAMP, notably matching against a working memory using patterns in the context of a subject value and an environment of bound variables. Production rule systems are usually implemented with (variants of) the Rete algorithm [20], which relies on its own internal representation of rules and objects. Modern production rule languages and systems include Drools [5] and JRules [24], which extend the ideas from OPS5 with aggregation and support Java objects in working memory. In contrast to OPS5, Drools, and JRules, we show how to translate rules to NRA enabling the use of algorithms for scalable evaluation of aggregates [33], as well as execution over either a relational store or a map-reduce framework [8]. Recently proposed distributed extensions for rules languages [10, 30] do not consider support for aggregation and do not address scalability issues.

Datalog is without a doubt the most studied rules language in the database area and the relationship between Datalog and the relational algebra has been studied in depth [4]. As in production systems, Datalog relies on declarative logic-based rules, but with fixpoint semantics and restrictions (e.g., actions can only insert new facts) that guarantee termination. CAMP focuses on capturing production rules semantics, including pattern matching for complex objects, with negation and aggregation. Similarly to production rules, existing Datalog extensions included complex objects [2], negation [3], and aggregation [15, 28]. The .QL project at Semmle seems closest to our approach in that it includes both a type system [17] and investigates compilation into SQL for execution [32]. To the best of our knowledge, translations of (fragments of Datalog) to NRA have not been investigated and could represent an interesting direction for future work.

Our formalization uses the NRA originally developed in [1, 6, 14, 31] and NNRC [34, 35]. Cluet and Moerkotte show how to translate nested queries on object-oriented data into NRA [14] and extensions to relational optimization; Abiteboul and Beeri explore the expressive power of NRA [1], and Ré et al. translate XQuery to NRA [31]. While some of the details of NRA differ between these papers, they all extend the relational algebra with dependent operators and in particular a form of dependent join to handle nested data and queries. Tannen et al. introduce NNRC as a language for nested relations, and demonstrate its equivalence to NRA [34]. Van den Bussche and Vansummeren present a polymorphic type inference algorithm for NNRC [35], but not for NRA.

There have been few attempts at mechanized proofs for aspects of database languages and implementation [7, 12, 26]. Malecha et al. [26] and Benzaken et al. [7] mechanize the relational algebra in Coq. In contrast to our work, neither of these handles nesting nor formalizes a type system. Cheney and Urban formalize a subset of XQuery in Isabelle [12]. While they handle nesting and formalize a type system, they do not use relational algebra.

Many programming languages feature pattern matching. Patterns without nested objects are at the center of Prolog [13]. Pattern matching is also central to functional languages with algebraic data types. Haskell, for example, extends this with support for views extending matching to other types [37]. Matchete unifies matching on algebraic types with other pattern languages such as regular expressions [23]. Stratego matches an implicit datum for program transformation [36]. While algebraic types can nest and are thus closer to production rule languages, patterns in languages like Haskell scrutinize only one datum at a time. In contrast,

JMatch offers pattern matching for Java and combines it with iteration [25]. Thorn takes this further, allowing pattern matching in many control constructs [9]. However, those control constructs exist outside the pattern, less tightly integrated with matching as in production rule languages.

9 Conclusion

This paper introduces CAMP, a calculus that captures the essence of pattern matching and aggregation in production rule languages. It presents translations from CAMP to NRA to NNRC and back to CAMP, thus demonstrating that they are all equally expressive. This is important, because it shows a way to implement production rule languages over (nested) relational stores, while taking advantage of database techniques for efficient, distributed, and incremental execution. This paper includes theorems for correctness, type preservation, and size preservation of all translations, and the accompanying mechanization includes machine-checked proofs for all theorems. This not only validates our new results for CAMP, but also adds rigor to folklore results on NRA and NNRC that had not previously been mechanized.

References

1 Serge Abiteboul and Catriel Beeri. The power of languages for the manipulation of complex values. *Journal on Very Large Data Bases (VLDB J.)*, 4(4):727–794, 1995.

2 Serge Abiteboul and Stephane Grumbach. COL: A logic-based language for complex objects. In *Extending Database Technology (EDBT)*, pages 271–293, 1988.

3 Serge Abiteboul and Richard Hull. Data functions, Datalog and negation. In *International Conference on Management of Data (SIGMOD)*, pages 143–153, 1988.

4 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*. Addison-Wesley, 1995.

5 Michal Bali. *Drools JBoss Rules 5.0 Developer's Guide*. Packt Publishing, 2009.

6 Catriel Beeri and Yoram Kornatzky. Algebraic optimization of object-oriented query languages. *Theoretical Computer Science*, 116(1&2):59–94, 1993.

7 Véronique Benzaken, Evelyne Contejean, and Stefania Dumbrava. A Coq formalization of the relational data model. In *European Symposium on Programming (ESOP)*, pages 189–208, 2014.

8 K. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Eltabakh, C-C. Kanne, F. Özcan, and E. Shekita. Jaql: A scripting language for large scale semistructured data analysis. In *Conference on Very Large Data Bases (VLDB)*, pages 1272–1283, 2011.

9 Bard Bloom and Martin Hirzel. Robust scripting via patterns. In *Dynamic Languages Symposium (DLS)*, pages 29–40, 2012.

10 Federico Cabitza, Marcello Sarini, and Bemardo Dal Seno. DJess – a context-sharing middleware to deploy distributed inference systems in pervasive computing domains. In *International Conference on Pervasive Services (ICPS)*, pages 229–238, 2005.

11 James Cheney, Sam Lindley, and Philip Wadler. A practical theory of language-integrated query. In *International Conference on Functional Programming (ICFP)*, pages 403–416, 2013.

12 James Cheney and Christian Urban. Mechanizing the metatheory of mini-XQuery. In *Conference on Certified Programs and Proofs (CPP)*, pages 280–295, 2011.

13 William F. Clocksin and Christopher S. Mellish. *Programming in PROLOG*. Springer, 1987.

14 Sophie Cluet and Guido Moerkotte. Nested queries in object bases. In *Workshop on Database Programming Languages (DBPL)*, pages 226–242, 1993.

15 Mariano P. Consens and Alberto O. Mendelzon. Low complexity aggregation in GraphLog and Datalog. In *International Conference on Database Theory (ICDT)*, pages 379–394, 1990.

16 Coq reference manual, version 8.4pl41. `http://coq.inria.fr/`.

17 Oege de Moor, Damien Sereni, Pavel Avgustinov, and Mathieu Verbaere. Type inference for Datalog and its application to query optimisation. In *Principles of Database Systems (PODS)*, pages 291–300, 2008.

18 Miki Enoki, Jérôme Siméon, Hiroshi Horii, and Martin Hirzel. Event processing over a distributed JSON store: Design and performance. In *Web Information System Engineering (WISE)*, pages 395–404, 2014.

19 Charles L. Forgy. OPS5 user's manual. Technical Report 2397, CMU, 1981.

20 Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

21 Timothy Griffin and Leonid Libkin. Incremental maintenance of views with duplicates. In *International Conference on Management of Data (SIGMOD)*, pages 328–339, 1995.

22 Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. Ferry: Database-supported program execution. In *International Conference on Management of Data (SIGMOD)*, pages 1063–1066, 2009.

23 Martin Hirzel, Nathaniel Nystrom, Bard Bloom, and Jan Vitek. Matchete: Paths through the pattern matching jungle. In *Practical Aspects of Declarative Languages (PADL)*, pages 150–166, 2008.

24 IBM WebSphere ILOG JRules BRMS. `http://www.ibm.com/software/integration/business-rule-management/jrules-family/`.

25 Jed Liu and Andrew C. Myers. JMatch: Iterable abstract pattern matching for Java. In *Practical Aspects of Declarative Languages (PADL)*, pages 110–127, 2003.

26 J. Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Toward a verified relational database management system. In *Principles of Programming Languages (POPL)*, 2010.

27 N. May, S. Helmer, and G. Moerkotte. Strategies for query unnesting in XML databases. *Transactions on Database Systems (TODS)*, 31(3):968–1013, 2006.

28 Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. The magic of duplicates and aggregates. In *Conference on Very Large Data Bases (VLDB)*, pages 264–277, 1990.

29 IBM Operational Decision Manager: Decision Server Insights. `http://www.ibm.com/support/knowledgecenter/SSQP76_8.7.0`.

30 Dana Petcu. Parallel Jess. In *International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 307–316, 2005.

31 Christopher Re, Jérôme Siméon, and Mary Fernandez. A complete and efficient algebraic compiler for XQuery. In *International Conference on Data Engineering (ICDE)*, 2006.

32 Damien Sereni, Pavel Avgustinov, and Oege de Moor. Adding magic to an optimising datalog compiler. In *International Conference on Management of Data (SIGMOD)*, pages 553–566, 2008.

33 Ambuj Shatdal and Jeffrey F. Naughton. Adaptive parallel aggregation algorithms. In *International Conference on Management of Data (SIGMOD)*, pages 104–114, New York, NY, USA, 1995.

34 Val Tannen, Peter Buneman, and Limsoon Wong. Naturally embedded query languages. In *International Conference on Database Theory (ICDT)*, pages 140–154, 1992.

35 Jan Van den Bussche and Stijn Vansummeren. Polymorphic type inference for the named nested relational calculus. *Transactions on Computational Logic (TOCL)*, 9(1), 2007.

36 Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. In *Rewriting Techniques and Applications (RTA)*, pages 357–362, 2001.

37 Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Principles of Programming Languages (POPL)*, pages 307–313, 1987.