Universität Potsdam
Institut für Linguistik

Relational Query Feature Structures

Diplomarbeit im Fach Computerlinguistik an der Universität Potsdam
Vorgelegt von Bryan Robert Jurish, Matrikelnr. 131495

1. Gutachter: Prof. Dr. Peter Staudacher
2. Gutachter: Dr. Thomas Hanneforth

Potsdam, Oktober 2001

Abstract

Feature Structures are formal constructs commonly used in the field of linguistics to represent interdependent and possibly recursive collections of information. *Relational Algebra* is a formalism through which queries over a set of relations commonly referred to as a *Relational Database* may be succinctly expressed. I will define in this paper a class $\mathcal{F}_{\mathcal{R}}$ of *Relational Query Feature Structures* and investigate their relationship to relational algebra in terms of expressive power. I then describe an implementation in the Perl programming language of relational query feature structures as a library of classes which generate well-formed queries of SQL, a query language accepted by many currently popular database management systems, which is derived from and equivalent to relational algebra.

Erklärung

Hiermit erkläre ich, Bryan Robert Jurish, geboren am 4. März 1974 in Hinsdale, Illinois, USA, dass ich die vorliegende Diplomarbeit "Relational Query Feature Structures" selbständig und nur unter Zuhilfenahme der angegebenen Quellen und Literatur angefertigt habe.

Potsdam, Oktober 2001

Bryan Robert Jurish

Acknowledgements

For my parents, for supporting me throughout my studies.

Additional thanks go to, in no particular order ...

... Linus Torvalds, Larry Wall, Donald E. Knuth, Jarkko Hietaniemi, Steffen Beyer, Raphael Manfredi, Joshua Chamas, Tim Bunce, and many others too numerous to mention for providing the world at large with the tools with which the empirical part of this work was built.

... Michael Chaffee, Burkhard Brehm, Cornelia Endriss, Jens Michaelis, Douglas Saddy, and my advisors Thomas Hanneforth and Peter Staudacher, for asking the important questions.

... my brother Ken Jurish, my sister Christy Furie, Knut Aufermann, Hans-Martin Gärtner, Maaret Isensee, Anja Meinke, Edmund Pohl, Olaf Westphely, and Oskar Westphely, for providing much-needed respite from all things algorithmic.

For Eva

Contents

Chapter 1

Introduction

I propose here to define and describe an implementation of an abstract and linguistically relevant formalism for the representation of queries on *relational databases*. I use the formal constructs known as *feature structures* for this purpose, defining a subclass of feature structures to represent queries, which I call *relational query feature structures*.

In Section 1.1, I present general background information on feature structures and relational databases. In Section 1.2, I focus on the use of relational databases for the storage and retrieval of linguistic information. Section 1.3 presents a more precise formulation of my proposal in light of the preceeding sections, while Section 1.4 briefly describes the organization of the rest of this document.

1.1 General Background

1.1.1 Feature Structures

Feature Structures are formal constructs commonly used in the field of linguistics to represent interdependent and possibly recursive collections of information. Their formal properties have been investigated and described by Aït-Kaci (1984, 1986), Kartunnen (1984), Pereira & Shieber (1984), Rounds & Kasper (1986), Kasper & Rounds (1986), Rounds (1996), Johnson (1988), Smolka (1992), and Carpenter (1992), among others.

Well-known grammar formalisms making use of feature structures as a vehicle for the encoding of lexical, syntactic, and/or semantic information include Functional Unification Grammar "FUG" (Kay 1984), the PATR-II formalism (Shieber 1984), Lexical Functional Grammar "LFG" (Kaplan & Bresnan 1985, Kaplan 1989), Generalized Phrase Structure Grammar "GPSG" (Gazdar, Pullum & Sag 1985), and Head-Driven Phrase Structure Grammar "HPSG" (Pollard & Sag 1987). Beyond the immediate domain of linguistics, feature structures (or close relatives thereof) have been used for terminological knowledge representation systems such as LOOM (MacGregor 1988) and even software configuration management (Zeller & Snelting 1997).

1.1.2 Relational Databases

A *database* is simply a collection of data. Dedicated database management systems (DBMSs) provide methods for the efficient storage and retrieval of the data they contain, and are usually tailored especially to the demands posed by large collections of data which may be shared between various applications. Database management systems differ with respect to the formal models used (often implicitly) to structure the data they contain, where formal database models differ with respect to the inherent constraints they impose on the type and structure of data which can be directly encoded within a database.

The *relational database model*, as introduced by Codd (1970), provides a formal specification for database systems based on the set-theoretical notion of a relation, or set of tuples, which over the last thirty years has established itself unquestionably as the dominant paradigm for database management systems. *Relational algebra* is a mathematical structure employing a small number of basic operations which may be used to derive new relations from a known set of *extensional* or database relations, and is often used as the basis for query (data-retrieval) languages for production relational database management systems.

Databases – relational ones in particular – and their query languages have been discussed by many, and at great length. Of particular interest are the original articles by Codd (1970, 1972*a*, 1972*b*), and Ullman's (1982, 1999) extensive treatment.

1.2 Linguistics and Databases

One may well ask what (relational) database management systems have to do with natural language, the domain of linguistics. Natural language applications, at least as much as any other kind, must deal in some way with the storage and retrieval of their elementary data, the sheer quantity of which may in even moderately ambitious projects grow too large to efficiently manage application-internally. This may in turn lead developers of natural language applications to consider the use of a dedicated database management system as a repository for application data, especially if some or all of that data is to be shared between multiple applications.

The most immediately apparent example of large data sets in the field of linguistics are those involved with the gathering and analysis of text corpora, such as those used by MULTEXT (Ide & Véronis 1994), or the Penn Treebank (Marcus, Santorini & Marcinkiewicz 1993). Derivable from such corpora (but also conceivably built "first-hand") are *lexical databases*, such as CELEX (Baayen, Piepenbrock & Gulikers 1995, CELEX Project 1988–1995), WordNet (Miller, Beckwith, Fellbaum, Gross, Miller & Tengi 1990, WordNet Project 1985–2001), or the MULTEXT lexicon (MULTEXT Project 1996), which, as their collective name suggests, may contain phonetic, phonological, morphological, syntactic, and/or semantic information usually associated in linguistic theories with a human speaker's lexicon.

Some other possible applications of large corpus databases are based on the approach sometimes referred to as "tree-bank grammars" (Charniak 1996) or "data-oriented processing" (Bod & Scha 1996), which attempt to provide performance models of human linguistic processing by referring directly to stored representations of past concrete language experiences, i.e. in the form of (partial) parse trees (Bod & Scha 1996, Bod, Bonnema & Scha 1996), or even mappings from partial trees to underspecified feature structures (Bod & Kaplan 1998).

Annotated corpora and lexical databases have also been used for purposes of knowledge acquisition (Martin 1995), text classification (Scott & Matwin 1998), as well as the extraction of application-specific lexica (Jurish 2000*a*), and even that of entire grammars (Chen & Vijay-Shanker 2000, XTAG Research Group 2001).

1.3 Proposal

The diversity of linguistic applications which may draw from a single data source – be it a text corpus, a lexical database, or an external knowledge base – speaks for the practice of pooling the shared data in a dedicated database management system, accessible from all applications drawing on it. Their efficiency and easy availability suggests relational databases as repositories for such pooled data. For applications drawing on multiple databases which may be similar in their informational content and yet differ regarding the way in which that information is structured, as well as for purposes of modularity, a level of abstraction is desirable which would allow users and application designers to specify their queries without direct knowledge of or reference to the manner in which any given individual source database structures its data.

With the aim of fulfilling these criteria, I propose to define here a formalism for expressing database queries, assuming the relational model to be that which is used by the underlying data source to structure the data.[1] I choose the formalism of feature structures as that on which I will base my system, for their well-known formal properties, the ease with which the various components of a given feature structure may be referenced by means of feature-paths, and of course for their linguistic salience.

The first step toward the goal of an abstract, modular, and linguistically pertinent query formalism is to define a collection of elementary queries, which I call $\mathcal{F}_\mathcal{R}$, the collection of *relational query feature structures*, and to establish their expressive power in terms of equivalence to a well-known query formalism, for which I choose the relational algebra.

After establishing the formal properties of relational query feature structures, I proceed to describe an implementation of their key properties. Database queries are generated in SQL, the "Structured Query Language", which is the *de facto* standard for relational database query languages. The system itself is implemented in Perl (Wall, Christiansen & Schwartz 1996), for purposes of portability and database independence.

[1]My choice of the relational database model is motivated primarily by the ubiquity of this model, rather than by any advantage I perceive to be inherent in that model for its use in the storage and retrieval of linguistic data.

1.4 Organization

Chapter 2 contains some necessary background definitions from the theory of relational databases and from that of feature structures.

In Chapter 3, I define the collection of relational query feature structures, and investigate their expressive power as a query formalism for relational databases.

Chapter 4 contains a decription of my implementation of relational query feature structures in the programming language Perl.

In Chapter 5, I summarize and discuss the results from the preceeding chapters.

Chapter 2

Background

In this chapter, I present a number of background definitions which I will require in order to adequately define the relational query feature structure formalism.

Section 2.1 contains some elementary set-theoretical definitions.

Section 2.2 presents some of the central definitions from relational database theory, and Section 2.3 describes the query formalism known as *relational algebra*.

Section 2.4 deals with the notion of a *type hierarchy* as a partially ordered set, and Section 2.5 contains some central definitions regarding the collection of *typed feature structures*.

2.1 Set-Theoretical Notions

My treatment of relational query feature structures makes use of the apparatus of set theory, a brief outline of some central definitions for which are given in this section. Readers familiar with the standard mechanisms of set theory may wish to skip directly to Section 2.2.

- **Set**
  A *set* is simply a collection of objects (called the *elements* of the set) in which a given element may be present at most once. We write $x \in S$

to indicate that the object x is an *element of* the set S. Typically, two basic forms of set-designation are recognized: *explicit enumeration* and *restriction*.

- **Set Enumeration**
  $\{x_1, \ldots, x_n\}$ represents the set containing all and only the objects $x_1, \ldots, x_n$.
- **Empty Set**
  The *empty set* is the set containing no objects, often written $\emptyset$. It can be defined in terms of enumeration as: $\emptyset := \{\}$.
- **Set Restriction**
  $\{x \mid P(x)\}$ represents the set containing all and only those objects x which satisfy the logical predicate P.
- **Set Identity**
  Two sets S_1 and S_2 are taken to be identical if they contain exactly the same objects:

$$S_1 = S_2 :\Leftrightarrow \forall x(x \in S_1 \Leftrightarrow x \in S_2)$$

- **Tuples**
  An *ordered n-tuple* is a construct containing exactly n components, each of which is identifiable within the tuple by a unique index i, $1 \leq i \leq n$, and is typically written between angled brackets $\langle x_1, \ldots, x_n \rangle$ or between round parentheses $(x_1, \ldots, x_n)$.

  - **1-Tuples**
    $(x) := x$
  - **Pairs (2-Tuples)**
    $(x, y) := \{\{x\}, \{x, y\}\}$
  - n**-Tuples** $(n > 2)$
    $(x_1, \ldots, x_{n-1}, x_n) := ((x_1, \ldots, x_{n-1}), x_n)$
  - **Tuple Identity**
    $(x_1, \ldots, x_n) = (y_1, \ldots, y_n) :\Leftrightarrow \forall i(1 \leq i \leq n \Rightarrow x_i = y_i)$
  - **Tuple Arity**
    $arity(x_1, \ldots, x_n) = n$

- **Tuple-Indexing** (Projection)
  $t.i = x_i :\Leftrightarrow t = (x_1, \ldots, x_n)$, if $1 \leq i \leq n$, otherwise $t.i$ is undefined.
- **Tuple Catenation**[1]
  $t^\frown u := (t.1, \ldots, t.arity(t), u.1, \ldots, u.arity(u))$

- **Relations**
  A set R is called an *n-ary relation* if it is a set of n-tuples. The term "arity" is often applied not only to tuples, but also to relations:
  $arity(R) = n :\Leftrightarrow \forall t \in R(arity(t) = n)$

  Equivalently, an n-ary relation R may be defined as a subset of the Cartesian product (see below) of n sets $S_1, \ldots, S_n$:
  $arity(R) = n \Leftrightarrow \exists S_1, \ldots, S_n(R \subseteq S_1 \times \cdots \times S_n)$

I will make use of some standard operations on sets and relations, definitions for which are given below, for sets S_1 and S_2.

- **Set Union**
  $S_1 \cup S_2 \;:=\; \{x \mid x \in S_1 \text{ or } x \in S_2\}$

- **Set Difference**
  $S_1 - S_2 \;:=\; \{x \mid x \in S_1 \text{ and } x \notin S_2\}$

- **Set Intersection**
  $$\begin{aligned} S_1 \cap S_2 &:= S_1 - (S_1 - S_2) \\ &= \{y \mid y \in S_1 \text{ and } y \in S_2\} \end{aligned}$$

- **Cartesian Product**
  $S_1 \times S_2 \;:=\; \{(x, y) \mid x \in S_1 \text{ and } y \in S_2\}$

- **Relational Product** (Expanded Product)
  If Q and R are relations of arity n and m, respectively, then the *relational product*[2] of Q and R is written $Q \otimes R$, and defined as:

[1] I use the symbol $\frown$ for the generic tuple catenation operation, reserving the frequently used symbol $\circ$ for string concatenation.

[2] Codd (1972*b*) calls the result of this operation the "expanded product" of Q and R.

$$\begin{array}{rcl} Q \otimes R & := & \{q^\frown r \mid q \in Q \text{ and } r \in R\} \\ & = & \{(q.1, \ldots, q.n, r.1, \ldots, r.n) \mid q \in Q \text{ and } r \in R\} \\ & = & \{(x_1, \ldots, x_{n+m}) \mid \begin{array}{lcl} (x_1, \ldots, x_n) & \in & Q \text{ and} \\ (x_{n+1}, \ldots, x_{n+m}) & \in & R\} \end{array} \end{array}$$

Note that while similar to the *Cartesian* product of two sets, the relational product is not necessarily identical to the Cartesian product unless the tuple constructing operation is associative, which the standard set theoretical methods of tuple construction – given above – are not.

Example 2.1 (Relation *vs.* Cartesian Product)
The following equations demonstrate that without associative n-tuples, the relational product operation is not equivalent to the more common Cartesian product operation.

$$\begin{array}{rrcl} \textbf{Assume:} & (a,(b,c)) & \neq & ((a,b),c) \\ 1. & \{(a,b)\} \times \{c\} & = & \{(x,y) \mid x \in \{(a,b)\} \text{ and } y \in \{c\}\} \\ & & = & \{((a,b),c)\} \\ 2. & \{a\} \times \{(b,c)\} & = & \{(x,y) \mid x \in \{a\} \text{ and } y \in \{(b,c)\}\} \\ & & = & \{(a,(b,c))\} \\ 3. & \{(a,b)\} \times \{(c)\} & \neq & \{(a)\} \times \{(b,c)\} \end{array}$$

Commonly used relations between sets include the *subset* relation, written "$\subseteq$", and the *proper subset* relation, written "$\subset$". Let S_1 and S_2 be sets, then:

- **Subset Relation**
  $S_1 \subseteq S_2 \;:\Leftrightarrow\; \forall x(x \in S_1 \Rightarrow x \in S_2)$

- **Proper Subset Relation**
  $S_1 \subset S_2 :\Leftrightarrow S_1 \subseteq S_2$ and $S_1 \neq S_2$

- **Powerset**
  The *powerset* of a set S, written $\mathbb{P}(S)$, is the set containing all and only the subsets of S:
  $\mathbb{P}(S) := \{X \mid X \subseteq S\}$

2.2 Relational Databases

A *Relational Database* can for present purposes be understood as a set of finite atomic relations, where an "atomic relation" is a relation whose every component is an atomic object, rather than a set.

Definition 2.1 (Extensional Database Relation)
A relation R is an *extensional database relation* if $R \subseteq D_1 \times \cdots \times D_n$ is a finite relation where each D_i is a set of atomic objects: for all D_i, $1 \leq i \leq n$, there is no $s \in D_i$ such that s is a set.[3]

Let $\mathcal{R}$ designate the collection of all extensional database relations.

Definition 2.2 (Relational Database)
A *relational database* is a finite set of extensional database relations.

Let $\mathcal{RDB}$ designate the collection of relational databases:

$\mathcal{RDB} = \{DB \in \mathbb{P}(\mathcal{R}) \mid DB \text{ finite}\}$.

An extensional database relation is specified by the set of tuples which satisfy that relation – that is to say that even a relation over an infinite domain (such as the set of all integers, $\mathbb{Z}$) may satisfy the finiteness condition on relational database relations if it only holds for some finite subset of that domain.

It is important to note that for extensional database relations, the object domain is restricted to *atomic* objects, such as numbers and strings. Crucially, no component of any tuple belonging to any relation in a relational database may itself be a relation, although techniques for normalizing higher-order relations for encoding within the relational database model exist (Codd 1970, 1972a, Ullman 1982, 1999).

2.2.1 Tuple-Set and Truth-Functional Relations

It may be noted that the definition of a database relation given above differs slightly from those to be found elsewhere in the literature: database relations

[3] The precise enumeration of allowable domains of atomic objects in production relational database systems is implementation dependent. I shall concern myself here with two common atomic object domains: that of natural numbers and that of finite character strings.

are defined as simple sets of object-tuples, rather than as total functions from object-tuples to truth values.[4] The latter notion may be recovered from the former by using the set-membership operator to construct a truth-functional relation $R' : D_1 \times \cdots \times D_n \to \{0, 1\}$ from a tuple-set relation $R \subseteq D_1 \times \cdots \times D_n$ as:

$$R'(t) := \begin{cases} 1 & \text{if } t \in R \\ 0 & \text{otherwise} \end{cases}$$

Conversely, a tuple-set relation R may be derived from a truth-functional relation R' by means of the restrictive set constructor by setting:

$$R := \{t \mid R'(t) = 1\},$$

although the tuple-set relations derived in such a fashion may not satisfy the finiteness condition for extensional database relations.

2.2.2 Tabular Notation

The atomicity constraint on extensional database relations[5] has led to the use the terms “table”, “row”, and “column”, and “cell”, to refer to a database relation, a tuple within a database relation, a component of a database relation, and a component of a tuple within a database relation, respectively. A simple database in the mathematical and tabular notations is given in Figures 2.1 and 2.2, which depict the same database $DB = \{R, S\}$ in the various notations.

$$\begin{array}{rcl} R & := & \{(a, 1), (x, 42)\} \\ S & := & \{(foo, bar), (baz, bonk)\} \\ DB & := & \{R, S\} \end{array}$$

Figure 2.1: Mathematical Notation for a Relational Database

[4]The truth-functional notion of relations is often used in formal models of semantic analysis, such as that presented by Montague (1970).

[5]In the future, I will simply write “database relation” to mean any relation satisfying the criteria for an extensional database relation.

DB

R

a	1
x	42

S

foo	bar
baz	$bonk$

Figure 2.2: Tabular Notation for a Relational Database

2.2.3 Table Names

Anticipating the requirements for developing an abstract query language for existing relational database management systems, I make explicit here a nearly universal convention of these systems: the convention of *table names*. Generally, each relation in a relational database is assigned a name which uniquely identifies it within that database – I have made use of this convention previously by using variables of the set-theoretical metalanguage (such as R and S) as placeholders for the relations which were assigned to them. The specification of a relational database query language, however, mandates a more precise formulation of this notion.

Definition 2.3 (Naming Relational Database)
A *naming relational database* is a finite surjective[6] partial function $NDB : \Sigma^* \to \mathcal{R}$, where Σ is a finite *relation name alphabet*, each relation name $w \in dom(NDB)$ is a finite string of characters from Σ, and $range(NDB) \in \mathcal{RDB}$ is a relational database.

Let $\mathcal{NRDB}$ designate the collection of *naming relational databases*.

2.3 Relational Algebra

Relational Algebra is a mathematical structure containing a small number of operations commonly used for constructing queries on a relational database. The relational algebraic operations are so defined that the result of any application of any sequence of these operations has as its result some finite

[6] A surjective function is one which assigns each element of its range (in this case, each extensional database relation from some relational database) to at least one element of its domain (in this case, at least one name).

relation satisfying Definition 2.1.

Definition 2.4 (Relational Algebra)
Formally, an instance of relational algebra is defined as a 6-tuple:

$$\langle \mathcal{C}(DB), \cup, -, \otimes, \pi, \sigma \rangle, \text{ where:}$$

- $DB \in \mathcal{RDB}$ is a relational database,
- $\cup$ is the set-theoretical union operation, restricted to relations of identical arity,
- $-$ is the set-theoretical difference operation,
- $\otimes$ is the relational product operation, also known as the "expanded product" operation,
- π is the *projection* operation, as described in Section 2.3.2,
- σ is the *selection* operation, as described in Section 2.3.3, and
- $\mathcal{C}(DB)$ is that set of finite extensional database relations induced by DB through application of the operations $\cup, -, \otimes, \pi$, and σ until closure.

Every relational database satisfying Definition 2.2 induces a relational algebra.

2.3.1 Set-theoretical Operations

The set-theoretical *union* operation $\cup$ is restricted in relational algebra in application to relations of identical arity: $R \cup S$ is thus only valid within the relational algebraic framework if $arity(R) = arity(S)$. Without this restriction, it might be that the result of an application of a relational algebraic union operation would not itself be a relation. Such restrictions are not necessary on the other set-theoretical operations *difference* $-$ and *product* $\otimes$.

Example 2.2 (Union of relations resulting in non-relation)

1. $R := \{(a, b), (c, d)\}$

2. $S := \{(1), (2)\}$

3. $arity(R) = 2$

4. $arity(S) = 1$

5. $$\begin{array}{rcl} T & := & R \cup S \\ & = & \{t \mid t \in R \text{ or } t \in S\} \\ & = & \{(a,b), (c,d), (1), (2)\} \end{array}$$

6. $\exists t, t' \in T(arity(t) \neq arity(t'))$
$$\begin{array}{rcl} t & = & (a,b) \\ t' & = & (1) \\ arity(t) & = & 2 \\ arity(t') & = & 1 \end{array}$$

7. T is not a relation. $\Box$

2.3.2 Projection

The *projection operation* π may be used to extract specific components from its argument (a relation), or to re-order its argument's components. Formally, π is defined as follows:

Definition 2.5 (Relational Algebraic Projection)
For a list of k indices $i_1, \ldots, i_k$ and a relation R of arity n, where $1 \leq i_j \leq n$, for each $i_j, 1 \leq j \leq k$, the *projection* of $i_1, \ldots, i_k$ from R is defined[7] as that k-ary relation whose j-th column is column i_j of R:

$$\begin{array}{rcl} \pi_{i_1,\ldots,i_k}(R) & := & \{(t.i_1, \ldots, t.i_k) \mid t \in R\} \\ & = & \{(x_{i_1}, \ldots, x_{i_k}) \mid (x_1, \ldots, x_n) \in R\} \end{array}$$

[7] Codd (1970) and Ullman (1982, 1999) add the additional stipulation that $k \leq n$, which I ignore here for convenience and for compatibility with conventional relational database management systems, which typically do not enforce this constraint.

2.3.3 Selection

The *selection operation* σ may be used to to extract a restricted subset of the tuples in its argument (a relation) according to whether or not these tuples satisfy a given logical formula.

I assume a basic repetoire of "built-in" logical comparisons on atomic objects $Comp$ for purposes of constructing such formulae, which for present purposes I restrict to identity and scalar comparison operators: $Comp := \{=, <, >\}$. I assume these operations are "built-in" in the sense that their extensions are clearly infinite, and thus cannot themselves be considered database relations.[8]

Formally, I define selectional formulae as functions from tuples to truth values, using the λ calculus as a notation for such functions. Since selectional formulae are required to contain only elementary comparisons, such a function must be of the form $\lambda t[x_1 \theta x_2]$, where t is the tuple variable of abstraction, $\theta \in Comp$ is an elementary comparison, and each x_i is either an indexing operation on the tuple variable t of the form $t.j$, $1 \leq j \leq arity(t)$, or some constant atomic value.

Definition 2.6 (Selectional Formula)
A *selectional formula* is a function of the form $\lambda t[x_1 \theta x_2]$, where θ is an elementary comparison operator, $\theta \in Comp$, and for $1 \leq i \leq 2$, x_i is either a constant atomic value or an indexing operation $t.j$ on the tuple-variable t of the abstraction, $j \geq 1$.

Let $\mathcal{P}_{sel}$ designate the collection of all selectional formulae.

It will prove useful to associate with each operand of a selectional formula (and by extension with the selectional formula itself) a "minimum arity".

Definition 2.7 (Operand Minimum Arity)
For an expression $\lambda t[x_1 \theta x_2]$ of the λ-calculus describing a selectional formula, the *minimum arity* of the operand x_i, $arity_{min}(x_i) \in \mathbb{N}$, where $1 \leq i \leq 2$, is defined as:

$$arity_{min}(x_i) := \begin{cases} 0 & \text{if } x_i \in Const \\ j & \text{if } x_i = t.j \end{cases}$$

[8]Such logical comparisons are nonetheless efficiently decidable, given a reasonable representation of the atomic objects for which they are defined, such as numbers and strings.

Definition 2.8 (Formula Minimum Arity)
A minimum arity may be assigned to a selectional formula $P \in \mathcal{P}_{sel}$, given its notation as an expression of the λ-calculus $\lambda t[x_1 \theta x_2]$, as:

$$arity_{min}(\lambda t[x_1 \theta x_2]) := max(arity_{min}(x_1), arity_{min}(x_2))$$

The selection operation σ itself can now be defined as:

Definition 2.9 (Relational Algebraic Selection)
For a database relation R of arity n and a selectional formula $P \in \mathcal{P}_{sel}$ such that $arity_{min}(P) \leq n$, the *selection for P in R*, $\sigma_P(R)$ is defined as:

$$\begin{aligned} \sigma_P(R) &:= R \cap \{t \mid P(t)\} \\ &= \{t \mid t \in R \text{ and } P(t)\} \end{aligned}$$

The minimum arity restriction ensures that the expansion of P is defined for all tuples in R: since the elementary comparisons which may occur in selectional formulae are defined for all arguments, the only case in which a selectional formulae $P = \lambda t[x_1 \theta x_2]$ would not be defined for a tuple t of arity n would be a case in which some x_i were an indexing expression $t.j$ such that $j > n$. Stipulating that $arity_{min}(P) \leq n$ effectively eliminates the possibility of indexing operations with undefined results.

2.3.3.1 Alternative Definitions

The definitions relating to relational algebraic selection given above differ in some respects from those used by Ullman or by Codd.

Ullman (1982, 1999) defines selectional formulae as expressions of the form $x_1 \theta x_2$, distinguishing x_i which represent indexing operations from those which represent constant values by typographical conventions. Clearly, however, selectional formulae must generalize over tuples, and our use of the λ calculus as a notation for such formulae serves only to make this abstraction explicit.

Codd (1970, 1972*b*) himself does not define selection as a operation of relational algebra at all. Rather, he defines the (equivalent) notion of a *join* between arbitrary relations, and shows that any join between a finite database relation and any infinite relation θ must result in a database relation. Constants for Codd are realized as singleton unary relations.

Contemporary relational database systems draw heavily on Codd's (1972*b*) notion of a *relational calculus*, which deals with database relations as logical predicates. The definitions given above represent a hybrid formalism drawing on both relational algebra and relational calculus, designed to model the essential aspects of contemporary database query languages.

2.3.4 Derived Operations

Various extensions to the relational algebra are possible by reducing new operators or formulae to the elementary operations listed above, for example:

- **Intersection**

$$\begin{aligned} R \cap S &:= R - (R - S) \\ &= \{t \mid t \in R\} - (\{t \mid t \in R\} - \{t \mid t \in S\}) \\ &= \{t \mid t \in R\} - \{t \mid t \in R \text{ and } t \notin S\} \\ &= \{t \mid t \in R \text{ and } \neg(t \in R \text{ and } t \notin S)\} \\ &= \{t \mid t \in R \text{ and } (t \notin R \text{ or } t \in S)\} \\ &= \{t \mid t \in R \text{ and } t \in S\} \end{aligned}$$

- **Conjunctive Selectional Formulae**

$$\begin{aligned} \sigma_{P_1 \wedge P_2}(R) &:= \sigma_{P_1}(R) \cap \sigma_{P_2}(R) \\ &= \{t \mid t \in R \text{ and } P_1(t)\} \cap \{t \mid t \in R \text{ and } P_2(t)\} \\ &= \{t \mid t \in R \text{ and } P_1(t) \text{ and } P_2(t)\} \end{aligned}$$

- **Disjunctive Selectional Formulae**

$$\begin{aligned} \sigma_{P_1 \vee P_2}(R) &:= \sigma_{P_1}(R) \cup \sigma_{P_2}(R) \\ &= \{t \mid t \in R \text{ and } P_1(t)\} \cup \{t \mid t \in R \text{ and } P_2(t)\} \\ &= \{t \mid t \in R \text{ and } (P_1(t) \text{ or } P_2(t))\} \end{aligned}$$

- **Negated Selectional Formulae**

$$\begin{aligned} \sigma_{\neg P}(R) &:= R - \sigma_P(R) \\ &= \{t \mid t \in R\} - \{t \mid t \in R \text{ and } P(t)\} \\ &= \{t \mid t \in R \text{ and } \neg(t \in R \text{ and } P(t))\} \\ &= \{t \mid t \in R \text{ and } (t \notin R \text{ or } \neg P(t))\} \\ &= \{t \mid t \in R \text{ and } \neg P(t)\} \end{aligned}$$

2.4 Type Hierarchies

My treatment of relational query feature structures defines these as a subcollection of the class of *typed feature structures*, as discussed by Carpenter (1992). Typed feature structures are defined for a *type hierarchy*, which is simply a *subsumption ordering* $\sqsubseteq$ over a set $Type$ of types. Informally, $\sigma \sqsubseteq \tau$ ("σ *subsumes* τ") just in case the type[9] $\sigma \in Type$ is at least as general as the type $\tau \in Type$. In this case, σ is said to be a *supertype* of τ, and τ is equivalently said to be a *subtype* of σ.

The type hierarchy structure $\langle Type, \sqsubseteq \rangle$ is often abbreviated $Type$.

2.4.1 Partial Orders

Type hierarchies are a special case of partially ordered sets. In this section, I give a brief overview of the terminology of partially ordered sets required for an adequate definition of type hierarchies. The definitions in this section are drawn chiefly from Davey & Priestley (1990), and to a lesser extent from Carpenter (1992).

Definition 2.10 (Partially Ordered Set)
A *partially ordered set*, or "poset" is a pair $\langle P, \leq \rangle$, where P is a set, and $\leq$ is a *partial order* on P: a binary relation on P such that the following conditions hold:

- **Reflexivity**
  $\forall x \in P(x \leq x)$
- **Antisymmetry**
  $\forall x, y \in P((x \leq y \text{ and } y \leq x) \Rightarrow x = y)$
- **Transitivity**
  $\forall x, y, z \in P((x \leq y \text{ and } y \leq z) \Rightarrow x \leq z)$

[9] In this and the following sections I use the greek letters σ and τ as variables for types. Where applicable, I will explicitly restate the variable assignment domains, to avoid confusion with the relational algebraic selection operation σ_P.

Definition 2.11 (Upper and Lower Bounds)
Given a poset $\langle P, \leq \rangle$ and a subset $S \subseteq P$, the notion of the upper and lower bounds of S with respect to the partial order $\leq$ may be defined as:

- **Upper Bounds**
  $S^u := \{x \in P \mid \forall s \in S(s \leq x)\}$

- **Lower Bounds**
  $S^\ell := \{x \in P \mid \forall s \in S(s \geq x)\}$

Definition 2.12 (LUB and GLB)
If $\langle P, \leq \rangle$ is a poset and $S \subseteq P$ is a subset of P, then:

- **Least Upper Bound (LUB)**
  The *least upper bound* (or "join") of S with respect to $\langle P, \leq \rangle$ is defined as the minimal element of S^u, if one exists:

  $$LUB(S) = x :\Leftrightarrow x \in S^u \text{ and } \forall y \in S^u(x \leq y)$$

  It is conventional to write $x \sqcup y$ to refer to the least upper bound of $\{x, y\} \subseteq P$ where it exists, and $\bigsqcup S$ to refer to the least upper bound of $S \subseteq P$ where it exists.

- **Greatest Lower Bound (GLB)**
  The *greatest lower bound* (or "meet") of S with respect to $\langle P, \leq \rangle$ is defined dually as the maximal element of S^ℓ, if one exists:

  $$GLB(S) = x :\Leftrightarrow x \in S^\ell \text{ and } \forall y \in S^\ell(x \geq y)$$

  It is conventional to write $x \sqcap y$ to refer to the greatest lower bound of $\{x, y\} \subseteq P$ where it exists, and $\bigsqcap S$ to refer to the greatest lower bound of $S \subseteq P$ where it exists.

Definition 2.13 (Consistent Subsets)
A subset $S \subseteq P$ is said to be *consistent* with respect to a poset $\langle P, \leq \rangle$ if, for every finite subset $F \subseteq S$, there exists some $z \in P$ such that $z \in F^u$.

Since type hierarchies for feature structures are themselves required to be finite, this definition may be equivalently rephrased: a consistent subset S of a finite poset $\langle P, \leq \rangle$ is a subset such that $S^u \neq \emptyset$.

Definition 2.14 (Consistently Complete Partial Orders)
A *consistently complete partial order* (CCPO), also known as a *bounded complete partial order* (BCPO), is a partial order $\leq$ on some set P such that all consistent subsets $S \subseteq P$ have a unique least upper bound:

$$\forall S \subseteq P(S^u \neq \emptyset \Rightarrow \bigsqcup S \; exists)$$

For non-empty consistently complete partially ordered sets $\langle P, \leq \rangle$, the existence of a minimal or "bottom" element (for which it is conventional to write $\perp$) can be derived as follows:

1. $\langle P, \leq \rangle \in CCPO$
2. $P \neq \emptyset$
3. $\forall S \subseteq P(S^u \neq \emptyset \Rightarrow \bigsqcup S \; exists)$
4. $\begin{array}{rcl} \emptyset^u & = & \{x \in P \mid \forall y \in \emptyset(y \leq x)\} \\ & = & P \\ & \neq & \emptyset \end{array}$
5. $\perp = \bigsqcup \emptyset \; exists$ $\square$

Also, in a consistently complete partially ordered set $\langle P, \leq \rangle$, arbitrary greatest lower bounds exist for non-empty subsets $X \subseteq P$:

1. $\langle P, \leq \rangle \in CCPO$
2. $\emptyset \neq X \subseteq P$
3. $X \subseteq (X^\ell)^u \neq \emptyset$
4. $\bigsqcap X = \bigsqcup X^\ell \; exists$ $\square$

2.4.2 Type Hierarchies as CCPOs

Type hierarchies may now be defined in terms of partially ordered sets.

Definition 2.15 (Type Hierarchy)
A *type hierarchy* (or "inheritance hierarchy") is a non-empty finite consistently complete partially ordered set $\langle Type, \sqsubseteq \rangle$.

Joins can be intuitively understood as extending the information carried by a consistent subset of types in the minimally informative fashion, and meets as generalizing the information carried by an an abitrary non-empty subset of types in the maximally informative fashion.

Techniques exist for deriving type hierarchies which satisfy this definition from arbitrary finite partial orders.[10]

2.5 Feature Structures

A *Feature Structure* $F = \langle Q, \bar{q}, \theta, \delta \rangle$ is defined for a given finite set of features $Feat$ and a given type hierarchy $\langle Type, \sqsubseteq \rangle$ as described above (Section 2.4). One can think of a feature structure as a directed graph, in which the nodes are labelled by types from $Type$, and in which the arcs are labelled by features from $Feat$. The definitions in this section are adapted from Carpenter (1992) except where otherwise noted.

Feature structures are defined in a manner reminiscent of deterministic finite state automata, as:

Definition 2.16 (Feature Structure)
$F = \langle Q, \bar{q}, \theta, \delta \rangle$ is a *Feature Structure* over a finite set of features $Feat$ and a type hierarchy $\langle Type, \sqsubseteq \rangle$ if and only if:

- Q is a finite set of **nodes**, rooted at $\bar{q}$,
- $\bar{q} \in Q$ is the **root node** of F,
- $\theta : Q \to Type$ is a total **node typing** function, and
- $\delta : Feat \times Q \to Q$ is a partial **feature value** function.

The set of all feature structures satisfying Definition 2.16 is designated by $\mathcal{F}$.

[10] See Carpenter (1992) and Davey & Priestley (1990) for some of these.

2.5.1 Subsumption and Unification

The primary relation which may hold between feature structures is that of *subsumption*. Feature-structure subsumption relies for its definition on type subsumption as described in Section 2.4. One can think of feature structure subsumption as similar to type subsumption in that a feature structure F can be said to subsume a feature structure F' (for which the notation $F \sqsubseteq F'$ is used, just as for type subsumption) if F is at least as general as F': the subsumption relation $\sqsubseteq$ on feature structures, then, provides an informational ordering on feature structures, much as the type-subsumption relation provides an informational ordering on types.

Definition 2.17 (Feature Structure Subsumption)
A feature structure $F = \langle Q, \bar{q}, \theta, \delta \rangle$ subsumes a feature structure $F' = \langle Q', \bar{q}', \theta', \delta' \rangle$ (written $F \sqsubseteq F'$) if and only if there exists a total function (or *morphism*) $h : Q \to Q'$ such that:

- $h(\bar{q}) = \bar{q}'$
- $\theta(q) \sqsubseteq \theta'(h(q))$ for all $q \in Q$
- $h(\delta(f, q)) = \delta'(f, h(q))$ for all $q \in Q$ and all $f \in Feat$ such that $\delta(f, q)$ is defined

Definition 2.18 (Alphabetic Variance)
Two feature structures F and F' are said to be *alphabetic variants* if they mutually subsume one another; that is, if $F \sqsubseteq F'$ and $F' \sqsubseteq F$. Carpenter uses the shorthand $F \sim F'$ to mean that F and F' are alphabetic variants.

The most important operation on feature structures is the binary operation of *unification*, which one may think of as synthesizing a feature structure which represents the conjunction of all the information represented by its argument feature structures.

Definition 2.19 (Feature Structure Unification)
Let $F, F' \in \mathcal{F}$ be feature structures such that $F \sim \langle Q, \bar{q}, \theta, \delta \rangle$ and $F' \sim \langle Q', \bar{q}', \theta', \delta' \rangle$, where $Q \cap Q' = \emptyset$. Let the equivalence relation $\bowtie$ be the least equivalence relation on $Q \cup Q'$ such that:

- $\bar{q} \bowtie \bar{q}'$

- $\delta(f,q) \bowtie \delta'(f,q')$ if both $\delta(f,q)$ and $\delta'(f,q')$ are defined and $q \bowtie q'$.

Let $[q]_{\bowtie}$ designate the *equivalence class* of the node $q \in Q \cup Q'$ with respect to $\bowtie$, and let $X/_{\bowtie}$ be the *quotient set* of the set $X \subseteq Q \cup Q'$ modulo $\bowtie$-equivalence:[11]

- $[q]_{\bowtie} = \{r \in Q \cup Q' \mid q \bowtie r\}$
- $X/_{\bowtie} = \{[x]_{\bowtie} \mid x \in X\}$

Then, the *unification* of F and F' (written $F \sqcup F'$) is defined to be:

$$F \sqcup F' := \langle (Q \cup Q')/_{\bowtie}, [\bar{q}]_{\bowtie}, \theta^{\bowtie}, \delta^{\bowtie} \rangle$$

where:

$$\theta^{\bowtie}([q]_{\bowtie}) = \bigsqcup \{(\theta \cup \theta')(q') \mid q \bowtie q'\}$$

and

$$\delta^{\bowtie}(f, [q]_{\bowtie}) = \begin{cases} [(\delta \cup \delta')(f,q)]_{\bowtie} & \text{if } (\delta \cup \delta')(f,q) \text{ is defined} \\ undefined & \text{otherwise} \end{cases}$$

if all the joins in the deinition of of $\theta^{\bowtie}$ exist. $F \sqcup F'$ is undefined otherwise.

It turns out that feature structure unification is nothing more or less than a least upper bound operation on the information ordering over feature structures modulo alphabetic variance provided by the subsumption relation, $\langle \mathcal{F}/_{\sim}, \sqsubseteq \rangle$. A proof of this is given in Carpenter (1992). Of more concern for the purpose of designing a system making use of feature structures is that the above constructive definition provides a method by which the unification of two given feature structures may be directly and efficiently computed. Such algorithmic concerns are discussed in more detail in Chapter 4.

2.5.2 Other Properties

It will be convenient to make use of some additional terminology in the following treatment of relational query feature structures. Precise characterizations of the notions of *feature paths*, *path values*, and *cyclicity* are

[11] Also known as the set of *partitions* of X induced by the equivalence relation $\bowtie$.

given below. Unless otherwise specified, these definitions hold for a feature structure $\langle Q, \bar{q}, \theta, \delta \rangle$ over the type hierarchy $\langle Type, \sqsubseteq \rangle$ and the features $Feat$.

Definition 2.20 (Feature Paths)
A *feature path* is a sequence of features. Let $Path = Feat^*$ be the set of all paths, where ε designates the *empty path*: the path which contains no features.

Definition 2.21 (Path Value Function δ^*)
Just as for finite state automata, the feature-value function δ may be extended to a path-value function $\delta^* : Path \times Q \to Q$ as the reflexive and transitive closure of δ:

- $\delta^*(\varepsilon, q) = q$, for all $q \in Q$
- $\delta^*(f\pi, q) = \delta^*(\pi, \delta(f, q))$, for all $f \in Feat, \pi \in Path, q \in Q$

Definition 2.22 (Reachable Nodes)
Given a feature structure $F = \langle Q, \bar{q}, \theta, \delta \rangle$ and some node $q \in Q$, the set of nodes *reachable* from q by some path in F is defined as:

$reachable_F(q) := \{\delta^*(\pi, q) \mid \pi \in Path$ and $\delta^*(\pi, q)\ defined\}$

Definition 2.23 (Rootedness)
A feature structure $F = \langle Q, \bar{q}, \theta, \delta \rangle$ is said to be *rooted at* the root node $\bar{q}$ if every node is reachable from $\bar{q}$, that is, if $Q = reachable_F(\bar{q})$.

In general, it is required that a feature structure $F = \langle Q, \bar{q}, \theta, \delta \rangle$ is indeed rooted at $\bar{q}$.

Definition 2.24 (Cyclicity)
A feature structure $F = \langle Q, \bar{q}, \theta, \delta \rangle$ is said to be *cyclic* if some node is reachable from itself *via* a non-empty path: if there exists a node $q \in Q$ and a path $\pi \in Path - \{\varepsilon\}$ such that $\delta^*(\pi, q) = q$. If no such path exists, F is said to be *acyclic*.

Let $\mathcal{AF}$ designate the collection of all acyclic feature structures.

Definition 2.25 (Path Value)
For a feature structure $F = \langle Q, \bar{q}, \theta, \delta \rangle$ and a path $\pi \in Path$ such that $\delta^*(\pi, \bar{q})$ is defined, the the *path value of* π *in* F, writte $F@\pi$, is defined as:

$F@\pi = \langle Q', \bar{q}', \theta', \delta' \rangle$, where

- $\bar{q}' = \delta^*(\pi, \bar{q})$
- $Q' = reachable_F(\bar{q}')$
- $\delta'(f, q) = \delta(f, q)$, if $q \in Q'$
- $\theta'(q) = \theta(q)$, if $q \in Q'$

If $F@\pi$ is defined for a feature structure F and a path π, $F@\pi$ is said to be a *substructure* of F. If $\pi \neq \varepsilon$ is nonempty, then $F@\pi$ is said to be a *proper substructure* of F.

Definition 2.26 (Structure Type)
A feature structure $F = \langle Q, \bar{q}, \theta, \delta \rangle$ is said to be *of type* $\theta(\bar{q})$.

2.5.3 Graphical Notation

Feature structures are often depicted graphically either in a directed graph notation similar to that used for finite state automata, or in *attribute-value matrix* (AVM) notation.

In the directed graph notation, each node $q \in Q$ of the feature structure is depicted as a node of the graph, labelled by its type: $Label(q) = \theta(q)$. Each δ-transition $(f, q, q') \in \delta$ is depicted as an edge from q to q' labelled by the feature f. The root node $\bar{q}$ is identified by an arrow.

In the AVM notation, the substructure at each node $q \in Q$ is enclosed in square brackets, where the outermost pair of brackets enclose the entire feature structure – the substructure at the root node $\bar{q}$. Each substructure $F' = \langle Q', \bar{q}', \theta', \delta' \rangle$ is labelled with its type $\theta(\bar{q}')$, and contains one row for each feature f for which $\delta(f, \bar{q}')$ is defined, in which the attribute feature f is assigned as its value the substructure $F'@f$. Additionally, substructures may be labelled with a "tag", which is usually written as a boxed natural number such as $\boxed{42}$ in Figure 2.4 below. Tag sharing between two substructures is interpreted as "structure sharing" or node identity of the substructures' root nodes. Although the AVM notation does not directly encode node identity, it encodes sufficient structural information to distinguish between feature structures up to alphabetic variance, which suffices for most purposes.

Figures 2.3 and 2.4 depict the feature structure F of Example 2.3 in directed graph and AVM notation, respectively.

Example 2.3 (Feature Structure)
$F = \langle Q, \bar{q}, \theta, \delta \rangle$ is a feature structure over the type hierarchy $\langle Type, \sqsubseteq \rangle$ and features $Feat$, where:

- $\{$**sent, noun, verb, syn, 3rd, sing**$\} \subseteq Type$
- $\{$SUBJ, PRED, AGR, PERSON, NUMBER$\} \subseteq Feat$
- $Q = \{1, 2, 3, 4, 5, 6\}$
- $\bar{q} = 1$
- $\theta(1) =$ **sent** $\quad \theta(3) =$ **noun** $\quad \theta(5) =$ **3rd**
  $\theta(2) =$ **verb** $\quad \theta(4) =$ **syn** $\quad \theta(6) =$ **sing**
- $\delta(\text{PRED}, 1) = 2 \quad \delta(\text{AGR}, 2) = 4 \quad \delta(\text{PERSON}, 4) = 5$
  $\delta(\text{SUBJ}, 1) = 3 \quad \delta(\text{AGR}, 3) = 4 \quad \delta(\text{NUMBER}, 4) = 6$

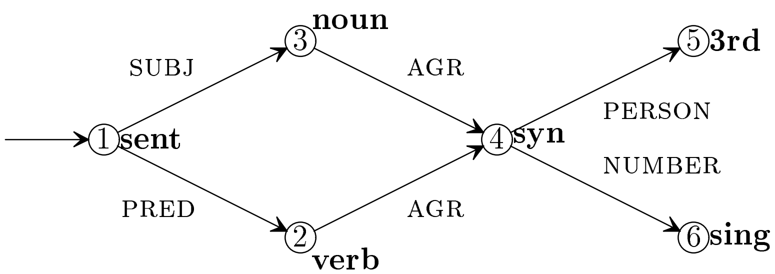

Figure 2.3: Directed graph notation for feature structures

2.5.4 Appropriateness

The use of an *appropriateness specification* allows the placement of restrictions on which features may be defined for a given type, as well as the values those features are assigned.

Definition 2.27 (Appropriateness Specification)
An *appropriateness specification* over the inheritance hierarchy $\langle Type, \sqsubseteq \rangle$ and features $Feat$ is a partial function $Approp : Feat \times Type \rightarrow Type$ that meets the following conditions:

$$\begin{bmatrix} \textbf{sent} & \\ \text{SUBJ:} & \begin{bmatrix} \textbf{noun} & \\ \text{AGR:} & \begin{bmatrix} \boxed{42}\ \textbf{syn} & \\ \text{PERSON:} & \begin{bmatrix}\textbf{3rd}\end{bmatrix} \\ \text{NUMBER:} & \begin{bmatrix}\textbf{sing}\end{bmatrix} \end{bmatrix} \end{bmatrix} \\ \text{PRED:} & \begin{bmatrix} \textbf{verb} & \\ \text{AGR:} & \begin{bmatrix}\boxed{42}\end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Figure 2.4: AVM notation for feature structures

- **Feature Introduction**
  for every feature $f \in Feat$, there is a most general type $Intro(f) \in Type$ such that $Approp(f, Intro(f))$ is defined

- **Upward Closure**
  if $Approp(f, \sigma)$ is defined and $\sigma \sqsubseteq \tau$, then $Approp(f, \tau)$ is also defined and $Approp(f, \sigma) \sqsubseteq Approp(f, \tau)$

Definition 2.28 (Well-Typedness)
A feature structure $F = \langle Q, \bar{q}, \theta, \delta \rangle$ is said to be *well-typed* under an appropriateness specification *Approp* if whenever $\delta(f, q)$ is defined, $Approp(f, \theta(q))$ is defined such that $Approp(f, \theta(q)) \sqsubseteq \theta(\delta(f, q))$.

Let $\mathcal{TF}$ designate the collection of well-typed feature structures.

Definition 2.29 (Total Well-Typedness)
A feature structure $F = \langle Q, \bar{q}, \theta, \delta \rangle$ is said to be *totally well-typed* if and only if:

- $F \in \mathcal{TF}$ is well-typed
- for all $q \in Q$ and all $f \in Feat$, if $Approp(f, \theta(q))$ is defined, then $\delta(f, q)$ is defined.

Let $\mathcal{TTF}$ designate the collection of totally well-typed feature structures.

2.5.5 Extensionality

An *extensionality specification* allows distinctions to be drawn between abstract types and concrete ones.

Definition 2.30 (Extensionality Specifcation)
An *extensinoality specification* over the inheritance hierarchy $\langle Type, \sqsubseteq \rangle$ is a set of types $ExtType \subseteq Type$, which is upwardly closed with respect to $\langle Type, \sqsubseteq \rangle$; that is, for all $\sigma, \tau \in Type$, if $\sigma \in ExtType$ and $\sigma \sqsubseteq \tau$, then $\tau \in ExtType$.

Informally, an extensionality specification $ExtType$ is simply an enumeration of "concrete" types. The upward closure clause simply ensures that the property of being "concrete" is inherited by all subtypes of a concrete type. A type $\sigma \in Type$ is said to be *concrete* or *extensional* if $\sigma \in ExtType$, and otherwise σ is said to be *abstract* or *intensional*.

The definition of an extensionality specification given above is identical to that given by Carpenter (1992). I depart from Carpenter's treatment, however, in the use to which I put extensionality specifications. Where Carpenter's concern was with the specification of conditions under which multiple type-identical substructures with identical feature-values must be considered node-identical, I am concerned only with the differentiation between abstract and concrete types.

My notion of an "extensionally grounded" feature structure, which I also call a *concrete* feature structure, is given in Defintion 2.31. Carpenter's notion of an "extensional" feature structure is given for completeness in Definition 2.32. My concern in what follows will be with extensionally grounded feature structures, but no inconsistencies arise if Carpenter's constraints for extensional feature structures are enforced as well.

Definition 2.31 (Extensionally Grounded Feature Structure)
A feature structure $F = \langle Q, \bar{q}, \theta, \delta \rangle$ over the type hierarchy $\langle Type, \sqsubseteq \rangle$ is said to be *extensionally grounded* with respect to an extensionality specification $ExtType \subseteq Type$ if every node is assigned a concrete type: for all nodes $q \in Q$, $\theta(q) \in ExtType$.

Let $\mathcal{EGF}$ designate the collection of extensionally grounded feature structures.

Definition 2.32 (Extensional Feature Structure)
A well-typed feature structure $F = \langle Q, \bar{q}, \theta, \delta \rangle \in \mathcal{TF}$ with respect to an appropriateness specification *Approp* is said to be *extensional* with respect to an extensionality specification *ExtType* if and only if for all $q, q' \in Q$, $q = q'$ whenever there exists some $\tau \in \mathit{ExtType}$ such that $\tau \sqsubseteq \theta(q)$, $\tau \sqsubseteq \theta(q')$, and for all features $f \in \mathit{Feat}$ such that $\mathit{Approp}(f, \tau)$ is defined, $\delta(f, q) = \delta(f, q')$.

Let $\mathcal{ETF}$ designate the collection of extensional well-typed feature structures, and let $\mathcal{ETTF}$ designate the collection of extensional totally well-typed feature structures.

It should be stressed that the collection $\mathcal{EGF}$ of extensionally grounded feature structures does not stand in any kind of containment relationship to either of the collections of extensional feature structures $\mathcal{ETF}$ and $\mathcal{ETTF}$. Extensional feature structures may contain nodes which are assigned abstract types, and are thus not extensionally grounded, whereas extensionally grounded feature structures may contain multiple nodes inheriting from a single extensional type which assign identical values to all of the features appropriate for that type, and are nonetheless not node-identical.

Chapter 3

Relational Query Feature Structures

The first step toward the goal of an abstract, re-usable, and linguistically pertinent query formalism is the definition of a collection of elementary queries, which I call $\mathcal{F}_{\mathcal{R}}$, the collection of *Relational Query Feature Structures*, and which I will abbreviate "RQFS." After making explicit some notational conventions in Section 3.1, I present in Section 3.2 a formal characterization of the syntactic properties of RQFSs: the specific type hierarchy and features for which they are defined, as well as appropriateness and extensionality specifications.

Since my proposed task is the definition of a relational database query language, a syntactic definition alone will not suffice. In Section 3.3, I provide (a subcollection of) relational query feature structures with semantic interpretations, which are themselves evaluated with respect to a given relational database.

Having precisely specified the syntactic and semantic properties of relational query feature structures, I investigate their expressive power in Section 3.4, using the relational algebra as a benchmark formalism: ideally, RQFSs will be capable of expressing all and only those queries which may be gained from a relational database by application of some sequence of the relational algebraic operations; thus, relational query feature structures should be equivalent in expressive power to relational algebra.

3.1 Notational Conventions

In the following sections, I will make use of certain notational conventions, which are made explicit below.

3.1.1 Variables

In what follows, I will use F as a variable for relational query feature structures, $F \in \mathcal{F}_{\mathcal{R}}$. I use f as a variable for features, $f \in Feat_{\mathcal{R}}$. I use the lower-case Greek letters σ, τ, and ρ as variables for types, $\sigma, \tau, \rho \in Type_{\mathcal{R}}$. I add sub- and superscripts to these variables for purposes of differentiation and of clarity.

3.1.2 Typeface Conventions

Italic text is used in formulae, and for the labels of the types in the hierarchy. Feature names are written in SMALL CAPITAL LETTERS.

3.1.3 Subsumption and Covers Relations

I declare the subsumption (sub-)relations for RQFSs by means of a relation $\succ$, read "covers", and its dual, the "covered-by" relation $\prec$, which have their usual interpretation of "immediately follows" and "immediately precedes", respectively, in the subsumption ordering of types.

$$x \prec y \Leftrightarrow x \sqsubseteq y \text{ and } y \not\sqsubseteq x \text{ and } \forall z (z \sqsubseteq y \Rightarrow z \sqsubseteq x)$$

Given the covers relation for a hierarchy, the subsumption relation can then be defined as the reflexive and transitive closure of the covers relation:

$$\begin{array}{rcl} \sqsubseteq & := & \prec^{\star} \\ & = & \{(\sigma, \tau) \mid \sigma = \tau \text{ or } \exists \rho (\sigma \prec \rho \text{ and } \rho \sqsubseteq \tau)\} \end{array}$$

I write $\sigma \succ \{\tau_1, \ldots, \tau_n\}$ to mean that the type σ covers each type τ_i, $\sigma \succ \tau_i$, for $1 \leq i \leq n$, omitting the set delimiters $\{\}$ in tables for ease of reading. Similarly, I write $\Sigma \succ \tau$, where Σ is a set, to mean that for all $\sigma \in \Sigma$, $\sigma \succ \tau$.

3.1.4 Immediate Appropriateness

Similarly to defining the subsumption relations by means of the corresponding covers relation, an appropriateness specification may be defined by means of an "immediate appropriateness specification," which I call *Appr*, which replaces the upward closure clause of Definition 2.27 with an "upward compatibility" clause:

Definition 3.1 (Immediate Appropriateness Specification)
An *immediate appropriateness specification* over the inheritance hierarchy $\langle Type, \sqsubseteq \rangle$ and features $Feat$ is a partial function $Appr : Feat \times Type \rightarrow Type$ that meets the following conditions:

- **Feature Introduction**
  for every feature $f \in Feat$, there is a most general type $Intro(f) \in Type$ such that $Appr(f, Intro(f))$ is defined
- **Upward Compatibility**
  if $Appr(f, \sigma)$ and $Appr(f, \sigma')$ are both defined for $\sigma, \sigma' \in Type$ such that $\sigma \sqsubseteq \sigma'$, then $Appr(f, \sigma) \sqsubseteq Appr(f, \sigma')$.

Given such a specification *Appr*, the appropriateness value $Approp(f, \sigma)$ can be computed for any $\sigma \in Type$ to meet the upward closure condition on appropriateness specifications in terms of a set $Appr^{\sigma}_{f}$ such that:

$$\begin{array}{rcl} Appr^{\sigma}_{f} & := & dom(Appr(f)) \cap \{\sigma\}^{\ell} \\ & = & \{\sigma' \mid \sigma' \sqsubseteq \sigma \text{ and } Appr(f, \sigma') \text{ defined}\} \end{array}$$

by setting:

$$Approp(f, \sigma) := \begin{cases} \bigsqcup \bigcup_{\sigma' \in Appr^{\sigma}_{f}} \{Appr(f, \sigma')\} & \text{if } Appr^{\sigma}_{f} \neq \emptyset \\ undefined & \text{otherwise} \end{cases}$$

Intuitively, the use of such a specification *Appr* allows me to omit the explicit specification of appropriateness conditions when these are simply inherited from a supertype. Phrased more ambitiously, the construction of an appropriateness specification *Approp* from an immediate appropriateness specification *Appr* provides a precise model of the inheritance of constraints on feature values.

3.1.5 Graphical Depictions of Type Hierarchies

In graphical depictions of RQFS subhierarchies, each type is depicted as a node in a Hasse Diagram, a directed graph in which the covers relation is indicated by edges from subtypes to supertypes. I draw hierarchies with the subtypes positioned vertically above their supertypes, and omit arrows on the edges. Nodes are of the following form:

$$\begin{array}{cc} \multicolumn{2}{c}{\mathsf{TypeName}} \\ \text{FEATURE}_1: & Value_1 \\ \vdots & \vdots \\ \text{FEATURE}_n: & Value_n \end{array}$$

where extensional types *TypeName* are boxed, as in $\boxed{\mathsf{TypeName}}$, and the immediate appropriateness specification with respect to $TypeName$ is given by: $Appr(\text{FEATURE}_i, TypeName) = Value_i$, for $1 \leq i \leq n$.

3.2 Syntax

The collection $\mathcal{F}_{\mathcal{R}}$ of relational query feature structures is properly contained by the collection $\mathcal{F}$ of all feature structures (Definition 2.16), and is defined for the specific type hierarchy $Type_{\mathcal{R}}$ and for the features $Feat_{\mathcal{R}}$, as described below. Also associated with relational query feature structures are an appropropriateness specification $Approp_{\mathcal{R}}$ (Definition 2.27) and an extensionality specification $ExtType_{\mathcal{R}}$ (Definition 2.30).

Rather than present each of these definitions alone and in its entirety, I have chosen to partition the type hierarchy by intuitive criteria, presenting each subhierarchy together with the features introduced in it and with the relevant subspecifications of appropriatess and extensionality.

Section 3.2.1 deals with the subhierarchy $Type_{id}$ of *identifier types*, Section 3.2.2 deals with the subhierarchy $Type_{deriv}$ of *derived types*, and Section 3.2.3 deals with the subhierarchy $Type_{exp}$ of *expression types*. Note that these subhierarchies are motivated primarily by intuition, and that I allow subsumption relations to hold between types in different subhierarchies. When such a subsumption relation exists, the relevant portion of the subhierarchy containing the subsuming (more general) type is for purposes of clarity included in the declarations for the subhierarchy containing the subsumed (more specific) type.

3.2.1 Identifier Types

The subhierarchy $Type_{id}$ of *identifier types* contains those types which intuitively correspond the atomic objects of relational databases, such as *string* and *natnum*, which are used to represent character strings and natural numbers, respectively. Additionally, $Type_{id}$ contains a type *Attr* which intuitively corresponds to the attribute indices of relational algebra, distinguishing these from natural number constants, as well as a type *literal*, which performs a similar differentiating function for character strings.

Definition 3.2 (Identifier Subtypes)
The subhierarchy of identifier subtypes is designated $Type_{id}$, and defined as:

$$Type_{id} \quad := \quad \Sigma \cup \left\{ \begin{array}{l} \textit{CompArg, EDBValue, string, natnum, nonzero,} \\ \textit{zero, Attribute, literal, char} \end{array} \right\}$$

where Σ is some database-dependent finite terminal alphabet, such as the set of 127 characters defined by the US-ASCII 7-bit encoding standard, or the superset of ASCII characters defined by the more extensive 16-bit Unicode standard.

Definition 3.3 (Extensional Identifier Subtypes)
The extensionality subspecification $ExtType_{id} \subseteq Type_{id}$ for identifier types is the following subset of identifier types:

$$ExtType_{id} \quad := \quad \Sigma \cup \{\, \textit{literal, Attribute, zero, nonzero, string} \,\}$$

where Σ as before is a finite database-dependent terminal alphabet.

Definition 3.4 (Identifier Features)
The subset of features $Feat_{id} \subseteq Feat_{\mathcal{R}}$ for identifier types contains the following features:

$$Feat_{id} \quad := \quad \{\, \textsc{attr},\ \textsc{chars},\ \textsc{pred},\ \textsc{str} \,\}$$

The subsumption ordering for the subhierarchy $Type_{id}$ of identifier types is given by means of the covers relation in Table 3.1, and the appropriateness

Subtype	$\succ$	Supertype(s)
$CompArg$	$\succ$	$\bot$
$EDBValue$	$\succ$	$CompArg$
$string$	$\succ$	$EDBValue$
$natnum$	$\succ$	$EDBValue$
$nonzero$	$\succ$	$natnum$
$zero$	$\succ$	$natnum$
$Attribute$	$\succ$	$CompArg$
$literal$	$\succ$	$\bot$
$char$	$\succ$	$\bot$
Σ	$\succ$	$char$

Table 3.1: Covers relation for identifier types

Appr(	Feature	,	Type	) =	Value
Appr(	CHARS	,	*string*	) =	$list(char)$
Appr(	PRED	,	*nonzero*	) =	*natnum*
Appr(	ATTR	,	*Attribute*	) =	*nonzero*
Appr(	STR	,	*literal*	) =	*string*

Table 3.2: Appropriateness for identifier types

subspecification $Approp_{id}$ for identifier types is given by means of the immediate appropriateness specification *Appr* in Table 3.2. The subhierachy $Type_{id}$ is graphically depicted in Figure 3.1.

3.2.1.1 Infinite Types

I have provided in the subhierarchy of identifier types an explicit encoding for the infinite subtypes *natnum* and *string* within the confines of a finite type hierarchy. While the inclusion of the infinite sets of natural numbers $\mathbb{N}$ and of character strings Σ^* would not jeopardize the status of our hierarchy as a CCPO,[1] I follow Carpenter's (1992) requirement that type hierarchies

[1] Even such an infinite hierarchy would remain a CCPO since the subhierarchies for any of the infinite subtypes mentioned above are themselves simple antichains sharing a single

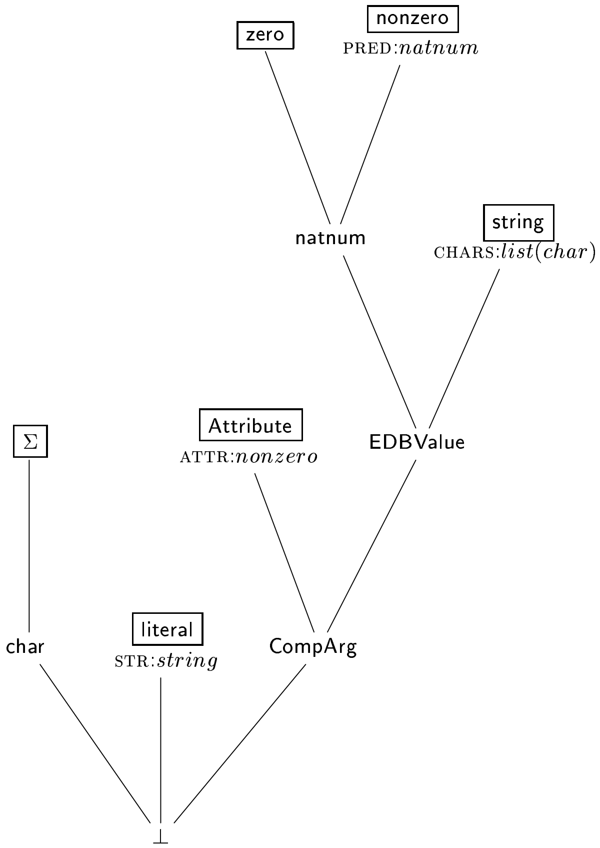

Figure 3.1: Identifier subhierarchy for $\mathcal{F}_{\mathcal{R}}$ feature structures

Natural Number	**Feature Structure Representation**
0	$\begin{bmatrix} \textit{zero} \end{bmatrix}$
1	$\begin{bmatrix} \textit{nonzero} & \\ \text{PRED:} & \begin{bmatrix} \textit{zero} \end{bmatrix} \end{bmatrix}$
ω	$\begin{bmatrix} \boxed{1}\ \textit{nonzero} & \\ \text{PRED:} & \begin{bmatrix} \boxed{1} \end{bmatrix} \end{bmatrix}$

Table 3.3: Encoding of natural numbers with a finite type hierarchy

be finite, since this is indeed possible and since it will make some of my later tasks more straightforward.

Strings are encoded in a manner reminiscent of PROLOG and LISP, as lists of characters from a finite terminal alphabet, which I assume contains the usual letters, digits, and punctuation symbols. The type *literal* includes (as the value of the feature "STR") such a character string, and serves to distinguish extensional database values (which are encoded by feature structures of types subsumed by $EDBValue$) from database system metavalues, such as table names.

Natural numbers are given an encoding similar to the "bottlecap" notation commonly used in set theory, and the type *Attribute*, which includes a natural number as the value of the feature "ATTR", is used to distinguish attribute indices from numeric constants. Some example numbers in decimal notation and the corresponding feature structure encodings are given in Table 3.3.

If it should later become necessary to include other infinite subtypes in the hierarchy, such as $integer$, $rational$, $float$, or $date$, these could be encoded as their own extensional types, using natural-number-valued features. Integers, for example could be encoded by a type *integer* with two

immediate supertype, such as those generated by the "Lifting" construction described by Davey & Priestley (1990), and antichains contain by definition no consistent subsets.

features: a natural-number-valued feature VALUE, and a bit-valued-feature SIGN, whose value would encode the sign of the integer (positive or negative), together with the requisite 3-element subhierarchy for encoding signs: $signbit \sqsubseteq \{positive, negative\}$.

3.2.2 Derived Types

The subhierarchy $Type_{deriv}$ of *derived types* contains complex types such as pairs and lists, both typed and untyped.

Definition 3.5 (Derived Subtypes)
The subhierarchy of derived subtypes is designated $Type_{deriv}$, and defined as:

$$Type_{deriv} \;:=\; \left\{ \begin{array}{l} list(\bot),\ list(char),\ list(Attribute),\ elist,\ pair(\bot), \\ nelist(\bot),\quad nelist(char),\quad nelist(Attribute), \\ pair(QueryExpr),\ pair(CompArg) \end{array} \right\}$$

Definition 3.6 (Extensional Derived Subtypes)
The extensionality subspecification $ExtType_{deriv} \subseteq Type_{deriv}$ for derived types is the following subset of derived types:

$$ExtType_{deriv} \;:=\; \left\{ \begin{array}{l} pair(CompArg),\quad pair(QueryExpr), \\ nelist(Attribute),\ nelist(char),\ elist \end{array} \right\}$$

Definition 3.7 (Derived Features)
The subset of features $Feat_{deriv} \subseteq Feat_{\mathcal{R}}$ for derived types contains the following features:

$$Feat_{deriv} \;:=\; \{\, \text{CAR},\ \text{CDR} \,\}$$

The subsumption ordering for the subhierarchy $Type_{deriv}$ of derived types is given by means of the covers relation in Table 3.4, and the appropriateness subspecification $Approp_{deriv}$ for derived types is given by means of the immediate appropriateness specification $Appr$ in Table 3.5. The subhierachy $Type_{deriv}$ is graphically depicted in Figure 3.2.

I have explicitly enumerated the typed-pair and -list types which I will require, naming them $pair(\sigma)$ or $list(\sigma)$, where σ is the type of elements "contained" by the derived type. Since only finitely many typed-pair and -list types are required, and they can be explicitly enumerated in such a fashion, the symbols *pair* and *list* need not be understood as *parametric type*

Subtype	$\succ$	**Supertype(s)**
$list(\bot)$	$\succ$	$\bot$
$list(char)$	$\succ$	$list(\bot)$
$list(Attribute)$	$\succ$	$list(\bot)$
$elist$	$\succ$	$list(Attribute), list(char)$
$pair(\bot)$	$\succ$	$\bot$
$nelist(\bot)$	$\succ$	$pair(\bot), list(\bot)$
$nelist(char)$	$\succ$	$nelist(\bot), list(char)$
$nelist(Attribute)$	$\succ$	$nelist(\bot), list(Attribute)$
$pair(QueryExpr)$	$\succ$	$pair(\bot)$
$pair(CompArg)$	$\succ$	$pair(\bot)$

Table 3.4: Covers relation for derived types

$Appr($	**Feature**	,	**Type**	$) =$	**Value**
$Appr($	CAR	,	$pair(\bot)$	$) =$	Q
$Appr($	CDR	,	$pair(\bot)$	$) =$	Q
$Appr($	CDR	,	$nelist(\bot)$	$) =$	$list(\bot)$
$Appr($	CAR	,	$nelist(char)$	$) =$	$char$
$Appr($	CDR	,	$nelist(char)$	$) =$	$list(char)$
$Appr($	CAR	,	$nelist(Attribute)$	$) =$	$Attribute$
$Appr($	CDR	,	$nelist(Attribute)$	$) =$	$list(Attribute)$
$Appr($	CAR	,	$pair(QueryExpr)$	$) =$	$QueryExpr$
$Appr($	CDR	,	$pair(QueryExpr)$	$) =$	$QueryExpr$
$Appr($	CAR	,	$pair(CompArg)$	$) =$	$CompArg$
$Appr($	CDR	,	$pair(CompArg)$	$) =$	$CompArg$

Table 3.5: Appropriateness for derived types

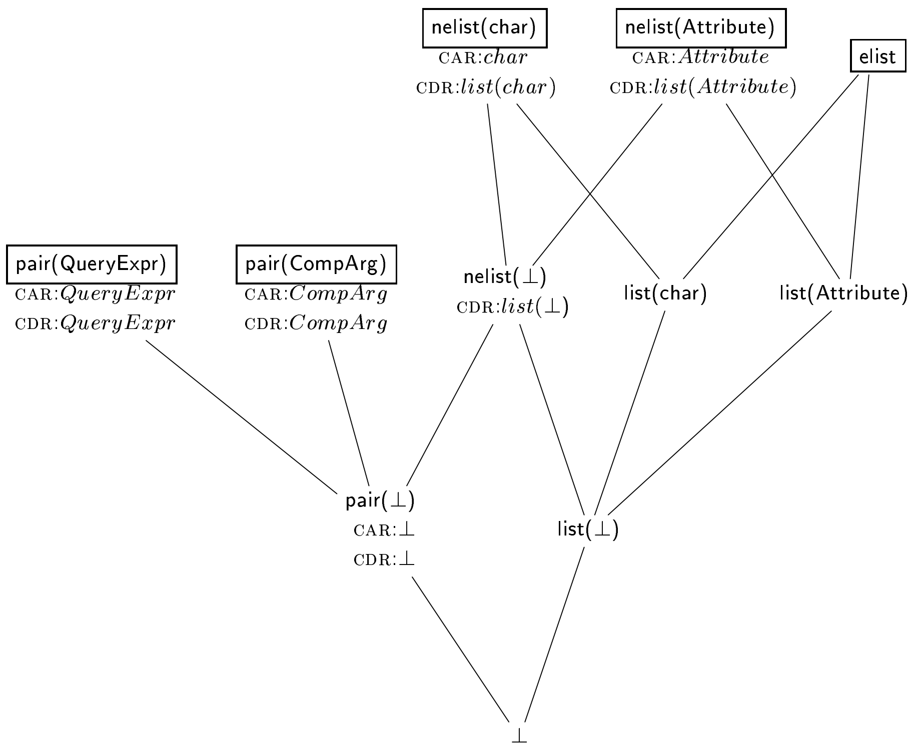

Figure 3.2: Derived subhierarchy for $\mathcal{F}_{\mathcal{R}}$ feature structures

constructors, functors which range (possibly recursively) over all types to generate new types, which are known to behave in an unruly fashion with respect to unification, as can be observed by a consideration of the cyclic list structure in Example 3.1, which is from Carpenter (1992).

Example 3.1 (Cyclic Typed Lists)
Consider the value for α in the following typed list structure:

$$\begin{bmatrix} \boxed{1}\ \mathit{list}(\alpha) & \\ \text{HEAD:} & \begin{bmatrix}\boxed{1}\end{bmatrix} \\ \text{TAIL:} & \begin{bmatrix}\boxed{1}\end{bmatrix} \end{bmatrix}$$

Carpenter argues as follows against allowing such structures, and by extension against allowing typed lists as such in his feature structures:

> In finding a minimal type for which this is well-typed, we would most likely attempt to unify α and $list(\alpha)$, leading either to an infinite regress or a cyclic parametric type.
>
> Carpenter 1992, page 79

The crux of this problem seems to be that in the feature structure in Example 3.1, α is read *as a variable denoting some type*, which is characteristic of metalanguage expressions. The possibility of including metalanguage type variables within expressions of the object language leads directly to problems such as the structure in Example 3.1.

What distinguishes my pseudo-functors *list* and *pair* from parametric type constructors such as *list* in Example 3.1 is that I restrict my use of metalanguage type variables *to the metalanguage*. That is, I will allow myself to refer to a set of types such as $pair(\sigma)$, but I do not allow type variables into relational query feature structures themselves. Thus, feature structures may be constructed with type assignments to types such as $pair(\perp)$, $list(Attribute)$, *etc.*, but not with type assignments containing a variable, such as $list(\alpha)$ above.

As a direct result, there is no way to construct a structure with a type such as $list(\alpha)$, where α is read as a variable of the object language. If α is read as a variable of the metalanguage – if the consideration pertains to a (cyclic) structure of type $list(char)$ or somesuch, the unification problem discussed by Carpenter simply disappears, since in the hierarchy defined above, there is no common upper bound for the types $char$ and $list(char)$. Indeed, there is no common upper bound for any type σ and the type which is written $list(\sigma)$.

3.2.2.1 Notational Conventions

I will make use of the usual conventions and write $\langle^{list(\sigma)} F_1, F_2, \ldots, F_n\rangle$, where each $F_i = \langle Q_i, \bar{q}_i, \theta_i, \delta_i\rangle$ is a feature structure such that $\sigma \sqsubseteq \theta_i(\bar{q}_i)$, for a (non-*list* and non-*pair*) type σ to represent the following list structure:

$$\begin{bmatrix} \mathit{nelist(\sigma)} & \\ \text{CAR:} & F_1 \\ \text{CDR:} & \begin{bmatrix} \mathit{nelist(\sigma)} & \\ \text{CAR:} & F_2 \\ \text{CDR} \mid \cdots \mid \text{CDR:} & \begin{bmatrix} \mathit{nelist(\sigma)} & \\ \text{CAR:} & F_n \\ \text{CDR:} & \begin{bmatrix} \mathit{elist} \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Note that $\langle^{list(\sigma)}\rangle$ designates that empty list, *elist*.

Similarly, I write $\langle^{pair(\sigma)} F_1, F_2\rangle$, where each F_i is a feature structure $\langle Q_i, \bar{q}_i, \theta_i, \delta_i\rangle$ such that $\sigma \sqsubseteq \theta_i(\bar{q}_i)$, to represent the following pair structure:

$$\begin{bmatrix} \mathit{pair(\sigma)} & \\ \text{CAR:} & F_1 \\ \text{CDR:} & F_2 \end{bmatrix}$$

3.2.3 Expression Types

The subhierarchy $Type_{exp}$ contains those types which intuitively correspond to the objects and operations of relational algebra, and to the logical formulae used by the selection operation. The former types are contained in the subhierarchy subsumed by the type *QueryExpr*, and the latter in the subhierarchy subsumed by the type *Comparison*.

Definition 3.8 (Expression Subtypes)
The subhierarchy of expression subtypes is designated $Type_{exp}$, and defined as:

$$Type_{exp} \;:=\; \left\{ \begin{array}{l} \textit{QueryExpr, QueryBinOp, Difference, Product,} \\ \textit{Union, EDBRelation, LookupQuery, SelectQuery,} \\ \textit{ProjectQuery, Comparison, GTComp, LTComp,} \\ \textit{EQComp} \end{array} \right\}$$

Definition 3.9 (Extensional Expression Subtypes)
The extensionality subspecification $ExtType_{exp} \subseteq Type_{exp}$ for expression types is the following subset of expression types:

$$ExtType_{exp} \;:=\; \left\{ \begin{array}{l} \textit{ProjectQuery, SelectQuery, EDBRelation, Union,} \\ \textit{Product, Difference, EQComp, LTComp, GT-} \\ \textit{Comp} \end{array} \right\}$$

Definition 3.10 (Expression Features)
The subset of features $Feat_{exp} \subseteq Feat_{\mathcal{R}}$ for expression types contains the following features:

$$Feat_{exp} \;:=\; \{\,\textsc{args},\ \textsc{attrs},\ \textsc{from},\ \textsc{qargs},\ \textsc{table},\ \textsc{where}\,\}$$

The subsumption ordering for the subhierarchy $Type_{exp}$ of expression types is given by means of the covers relation in Table 3.6, and the appropriateness subspecification $Approp_{exp}$ for expression types is given by means of the immediate appropriateness specification *Appr* in Table 3.7. The subhierachy $Type_{exp}$ is graphically depicted in Figures 3.3 and 3.4.

3.2.4 Formal Definition

The collection of relational query feature structures may now be formally defined in terms of the definitions from Sections 3.2.1–3.2.3.

Definition 3.11 (Relational Query Feature Structure)
A *relational query feature structure* $F = \langle Q, \bar{q}, \theta, \delta \rangle$ is defined for the type hierarchy $\langle Type_{\mathcal{R}}, \sqsubseteq \rangle$, and is associated with the appropriateness specification

Subtype	$\succ$	**Supertype(s)**
$QueryExpr$	$\succ$	$\bot$
$QueryBinOp$	$\succ$	$QueryExpr$
$Difference$	$\succ$	$QueryBinOp$
$Product$	$\succ$	$QueryBinOp$
$Union$	$\succ$	$QueryBinOp$
$EDBRelation$	$\succ$	$QueryExpr$
$LookupQuery$	$\succ$	$QueryExpr$
$SelectQuery$	$\succ$	$LookupQuery$
$ProjectQuery$	$\succ$	$LookupQuery$
$Comparison$	$\succ$	$\bot$
$GTComp$	$\succ$	$Comparison$
$LTComp$	$\succ$	$Comparison$
$EQComp$	$\succ$	$Comparison$

Table 3.6: Covers relation for expression types

Appr(	**Feature**	,	**Type**	) =	**Value**
Appr(	QARGS	,	$QueryBinOp$	) =	$pair(QueryExpr)$
Appr(	TABLE	,	$EDBRelation$	) =	$literal$
Appr(	FROM	,	$LookupQuery$	) =	$QueryExpr$
Appr(	WHERE	,	$SelectQuery$	) =	$Comparison$
Appr(	ATTRS	,	$ProjectQuery$	) =	$list(Attribute)$
Appr(	ARGS	,	$Comparison$	) =	$pair(CompArg)$

Table 3.7: Appropriateness for expression types

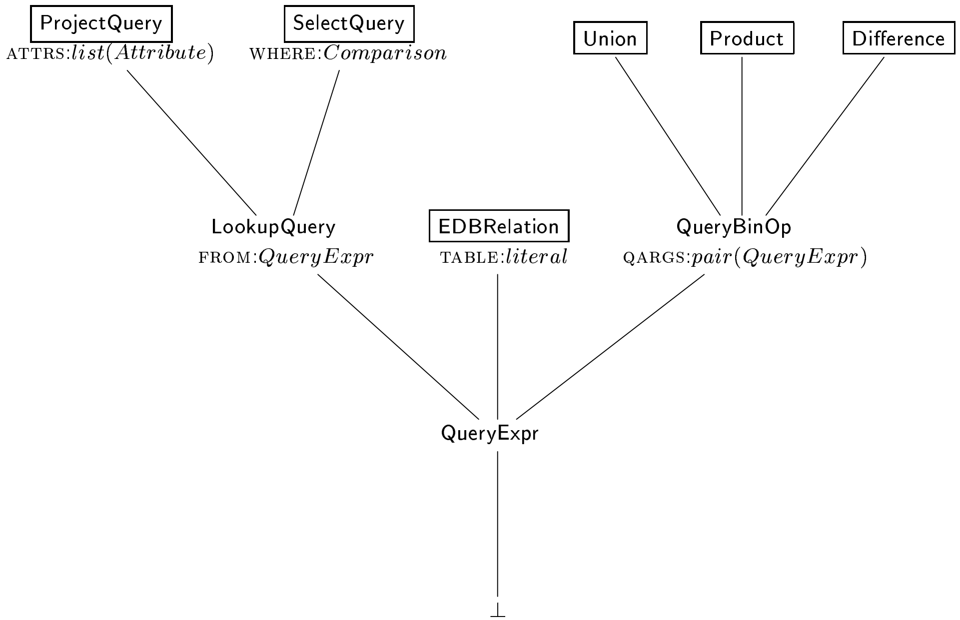

Figure 3.3: Query expression subhierarchy for $\mathcal{F}_{\mathcal{R}}$ feature structures

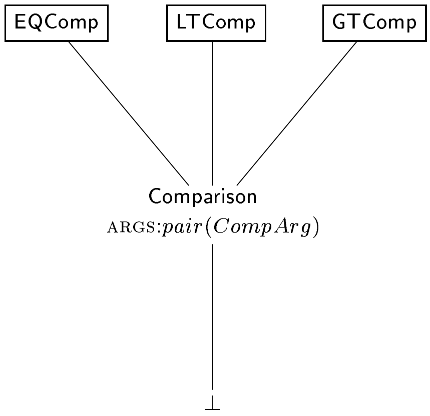

Figure 3.4: Condition expression subhierarchy for $\mathcal{F}_{\mathcal{R}}$ feature structures

$Approp_{\mathcal{R}}$ and with the extensionality specification $ExtType_{\mathcal{R}}$, where:

$$\begin{array}{llllll}
Type_{\mathcal{R}} & := & Type_{exp} & \cup\ Type_{id} & \cup\ Type_{deriv} \\
Feat_{\mathcal{R}} & := & Feat_{exp} & \cup\ Feat_{id} & \cup\ Feat_{deriv} \\
Approp_{\mathcal{R}} & := & Approp_{exp} & \cup\ Approp_{id} & \cup\ Approp_{deriv} \\
ExtType_{\mathcal{R}} & := & ExtType_{exp} & \cup\ ExtType_{id} & \cup\ ExtType_{deriv}
\end{array}$$

Let $\mathcal{F}_{\mathcal{R}}$ designate the collection of all relational query feature structures.

Let $\mathcal{TTF}_{\mathcal{R}}$ designate the collection of relational query feature structures which are totally well-typed with respect to $Approp_{\mathcal{R}}$.

Let $\mathcal{EGF}_{\mathcal{R}}$ designate the collection of relational query feature structures which are extensionally grounded with respect to $ExtType_{\mathcal{R}}$.

3.3 Semantics

Since I have proposed to use relational query feature structures as a mechanism for expressing queries over relational databases, I must assign them semantic interpretations, which I do in this section. In Section 3.3.1, I investigate the criteria which a relational query feature structure must fulfill before it can be considered eligible for semantic interpretation. In Section 3.3.2, I present definitions for some auxilliary functions which will prove useful in substructure interpretations. I present in Section 3.3.3 a formal definition of semantic interpretations for interpretable relational query feature structures with respect to a given relational database.

3.3.1 Intrepretability

Unfortunately, there exist relational query feature structures which do not intuitively correspond to any relational database query, for instance the feature structure in Example 3.2. The first task in providing RQFSs with semantic interpretations, therefore, is to delineate the subcollection $\mathcal{IF}_{\mathcal{R}} \subseteq \mathcal{F}_{\mathcal{R}}$ of relational query feature structures to which we may assign a semantic interpretation at all. I call this the subcollection $\mathcal{IF}_{\mathcal{R}}$ of *interpretable* relational query feature structures, abbreviated IRQFS.

Example 3.2 (Uninterpretable relational query feature structure) The following relational query feature structure illustrates three of the major problems encountered when attempting to interpret an arbitrary relational query feature structure $F = \langle Q, \bar{q}, \theta, \delta \rangle \in \mathcal{F}_{\mathcal{R}}$:

$$\begin{bmatrix} \textit{pair(CompArg)} & \\ \textsc{car:} & \begin{bmatrix} \textit{EDBValue} \end{bmatrix} \end{bmatrix}$$

1. **Unassigned Feature Values**
   No value whatsoever is assigned to the feature CDR.

   To deal with the problem of unassigned feature values, I will require that all interpretable relational query feature structures be *totally well-typed*, according to Definition 2.29.

2. **Root Node Uninterpretability**
   The root node is of type $pair(CompArg)$, which does not have a clear and distinct equivalent as a relational database query.

   This problem may be solved by requiring that an interpretable relational query feature structure $F = \langle Q, \bar{q}, \theta, \delta \rangle \in \mathcal{IF}_{\mathcal{R}}$ has a root node which does in fact intuitively correspond to a relational database query, by simply stipulating that $QueryExpr \sqsubseteq \theta(\bar{q})$.

3. **Abstract Types**
   The node at feature CAR is assigned the abstract type $EDBValue$,

rather than a specific and interpretable subtype such as the natural-number subtype *zero*.

The extensionality specification $ExtType_{\mathcal{R}}$ for relational query feature structures provides a mechanism for distinguishing abstract from concrete types. Making use of this mechanism, I simply require that no abstract types occur in interpretable relational query feature structures – in other words, all IRQFSs must be *extensionally grounded*, according to Definition 2.31.

The notion of an interpretable relational query feature structure may now be formally defined.

Definition 3.12 (Interpretable Relational Query Substructure)
A relational query feature structure $F = \langle Q, \bar{q}, \theta, \delta \rangle \in \mathcal{F}_{\mathcal{R}}$ is said to be an *interpretable substructure* just in case:

- $F \in \mathcal{TTF}_{\mathcal{R}}$ is totally well-typed with respect to $Approp_{\mathcal{R}}$,
- $F \in \mathcal{EGF}_{\mathcal{R}}$ is extensionally grounded with respect to $ExtType_{\mathcal{R}}$, and
- $F \in \mathcal{AF}$ contains no cycles.

Let $\mathcal{IS}^{\sigma}_{\mathcal{R}}$ be the collection of interpretable relational query substructures of type $\sigma \in Type_{\mathcal{R}}$:

$$\mathcal{IS}^{\sigma}_{\mathcal{R}} = \{\langle Q, \bar{q}, \theta, \delta \rangle \in (\mathcal{TTF}_{\mathcal{R}} \cap \mathcal{EGF}_{\mathcal{R}} \cap \mathcal{AF}) \mid \sigma \sqsubseteq \theta(\bar{q})\}$$

Thus, the collection of all interpretable relational query substructures is designated $\mathcal{IS}^{\perp}_{\mathcal{R}}$.

Definition 3.13 (Interpretable RQFS)
An interprertable relational query feature structure is any interpretable relational query substructure $F \in \mathcal{IS}^{QueryExpr}_{\mathcal{R}}$; that is, any totally well-typed, extensionally grounded, and acyclic relational query feature structure whose root node is of some type subsumed by $QueryExpr$. The collection $\mathcal{IF}_{\mathcal{R}}$ of interpretable relational query feature structures can thus be defined as:

$$\mathcal{IF}_{\mathcal{R}} = \mathcal{IS}^{QueryExpr}_{\mathcal{R}}$$

3.3.2 Auxilliary Interpretation Functions

3.3.2.1 Functions for Derived Types

In this section, I define a number of functions ranging over our derived types, for which I will find use in later definitions.

Let $\mathcal{F}^{\ell}_{\mathcal{R}} = \mathcal{F}_{\mathcal{R}} \cap \{\langle Q, \bar{q}, \theta, \delta \rangle \mid list(\bot) \sqsubseteq \theta(\bar{q})\}$ be the collection of *list structures*.

Definition 3.14 (List Length *length*)
The partial function $length : \mathcal{F}^{\ell}_{\mathcal{R}} \to \mathbb{N}$ is computed for an arbitrary list structure $L = \langle Q, \bar{q}, \theta, \delta \rangle \in \mathcal{F}^{\ell}_{\mathcal{R}}$ so that $length(L)$, the *length of* L is the number of elements L contains.

$$length(L) := \begin{cases} 0 & \text{if } \theta(\bar{q}) = elist \\ 1 + length(L@\textsc{cdr}) & \text{if } length(L@\textsc{cdr}) \text{ is defined} \\ undefined & \text{otherwise} \end{cases}$$

Note that the second clause of the definition will only apply to list structures L for which $L@\textsc{cdr}$ is itself defined.

Lemma 3.1 (List Length Computability)
For all interpretable list structures[2] $L \in \mathcal{IS}^{list(bot)}_{\mathcal{R}}$ $L = \langle Q, \bar{q}, \theta, \delta \rangle \in \mathcal{F}^{\ell}_{\mathcal{R}}$, $length(L)$ is defined.

Proof: Since L is extensionally grounded, we know that either $\theta(\bar{q}) = elist$ or $nelist(\bot) \sqsubseteq \theta(\bar{q})$. In the former case, the computability of $length(L)$ is trivial. In the latter case, since L is totally well-typed, we know that $L@\textsc{cdr}$ is defined. Indeed, $L@\textsc{cdr}$ is itself an extensionally grounded totally well-typed acyclic list structure. Since L is acyclic, and since feature structures are finite, we can compute $length(L)$ simply by following a finite number of CDR features until some substructure $L' = \langle Q', \bar{q}', \theta', \delta' \rangle$ is reached such that $\theta(\bar{q}') = elist$. □

[2] While the conditions on L listed above are sufficient for the computability of $length(L)$, they could be loosened considerably – the conditions of extensional groundedness, path existence (which were phrased in terms of total well-typedness), and acyclicity need only apply to a subset of all paths defined for a list structure L (namely, $Paths(L) \cap \textsc{cdr}^*$) in order for $length(L)$ to be computable. More relaxed conditions are possible for many of the functions in this section, but as I am concerned here only with interpretable relational query feature structures, I will restrict my attention to these where proofs are required.

Definition 3.15 (List Indexing *nth*)
The partial function $nth : \mathbb{N} \times \mathcal{F}^{\ell}_{\mathcal{R}} \to \mathcal{F}_{\mathcal{R}}$ is computed for an arbitrary list structure $L \in \mathcal{F}^{\ell}_{\mathcal{R}}$ and a natural number index $i \geq 1$ so that $nth(i, L)$, the *i-th element of* L, is defined as:

$$nth(i, L) := \begin{cases} L@\textsc{car} & \text{if } i = 1 \text{ and } \delta(\textsc{car}, \bar{q}) \text{ is defined} \\ nth(i-1, L@\textsc{cdr}) & \text{if } i > 1 \text{ and } \delta(\textsc{cdr}, \bar{q}) \text{ is defined} \\ undefined & \text{otherwise} \end{cases}$$

Lemma 3.2 (List Indexing Computability)
For all interpretable list structures $L \in \mathcal{IS}^{list(\perp)}_{\mathcal{R}}$ and all $i \in \mathbb{N}$ such that $i \geq 1$, $nth(i, L)$ is defined just in case $1 \leq i \leq length(L)$.

Proof: First, assume that $1 \leq i \leq length(L)$. If $i = 1$, then $nth(i, L) = L@\textsc{car}$ is defined, since L is totally well-typed. Otherwise, $L@\textsc{cdr}^{i-1}$ is defined and is itself an extensionally grounded totally well-typed acyclic list structure, for $1 < i \leq length(L)$, by Lemma 3.1. Therefore, $nth(i, L) = L@\textsc{cdr}^{i-1}\textsc{car}$ is defined.

To prove the converse, assume that $nth(i, L)$ is defined. It must be the case that $i \geq 1$ by Definition 3.15. By Lemma 3.1, $length(L)$ is computable. If it were the case that $i > length(L)$, it would have to be that $nelist(\perp) \sqsubseteq L@\textsc{cdr}^{length(L)}$, which contradicts Definition 3.14. Therefore, $1 \leq i \leq length(L)$. □

3.3.2.2 Functions for Identifier Types

The following functions provide semantic interpretations for the various *Identifier* substructures which may be found in an interpretable relational query feature structure.

Definition 3.16 (Numeric Interpretation Function I_{num})
If $F = \langle Q, \bar{q}, \theta, \delta \rangle \in \mathcal{IS}^{natum}_{\mathcal{R}}$, then $I_{num}(F)$, the interpretation of F as a natural number, is defined as follows:

$$I_{num}(F) := \begin{cases} 0 & \text{if } \theta(\bar{q}) = zero \\ I_{num}(F@\textsc{pred}) + 1 & \text{if } \theta(\bar{q}) = nonzero \end{cases}$$

Lemma 3.3 (Numeric Interpretability)
The function $I_{num} : \mathcal{IS}^{natnum}_{\mathcal{R}} \to \mathbb{N}$ is total.

Proof: If $F \in \mathcal{IS}_{\mathcal{R}}^{natnum}$, then it must be that $\theta(\bar{q}) \in \{zero, nonzero\}$, since these are the only extensional extensional types which are subsumed by *natnum*. The proof proceeds by induction on the number of proper *natnum* substructures in F.

- **Basis:** *Zero proper natnum-substructures in F*
  Then, it must be that $\theta(\bar{q}) = zero$. If it were possible that $\theta(\bar{q}) = nonzero$, then since F is totally well-typed, the feature PRED would be required to take a value of type *natnum*. Since F is acyclic, such a value would have to be a proper substructure of F: $\delta(\text{PRED}, \bar{q}) \neq \bar{q}$. Therefore, $\theta(\bar{q}) = zero$, and $I_{num}(F)$ is directly defined.

- **Induction:** *n+1 proper natnum-substructures in F*
  The inductive assumption is that $I_{num}(F') \in \mathbb{N}$ is defined for all $F' \in \mathcal{IS}_{\mathcal{R}}^{natnum}$ containing n or fewer *natnum* substructures.

  It must be that $\theta(\bar{q}) = nonzero$, since if $\theta(\bar{q})$ were to be *zero*, F could have no proper substructures of any type, since there are no features appropriate for *zero*, and F is totally well-typed.

  We know also that $F'' = F@\text{PRED}$ is defined and is a proper *natnum* substructure of F, since F is totally well-typed and acyclic. Further, $F'' \in \mathcal{IS}_{\mathcal{R}}^{natnum}$ has n or fewer proper *natnum* substructures, since F is acyclic and extensionally grounded. Indeed, F'' has exactly n proper substructures of type *natnum*, since the only proper substructures possible for any feature structure in $\mathcal{IS}_{\mathcal{R}}^{natnum}$ are themselves of type *natnum*. By inductive hypothesis, $I_{num}(F'') \in \mathbb{N}$ is defined, and therefore, $I_{num}(F)$ is defined as $I_{num}(F'') + 1$.

Definition 3.17 (String Interpretation Function I_{str})
If $F = \langle Q, \bar{q}, \theta, \delta \rangle \in \mathcal{IS}_{\mathcal{R}}^{list(char)}$, then $I_{str}(F)$, the interpretation of F as a character-string value, is defined as follows, where ε is the *empty string* and $\circ$ is the binary string concatenation operation, given by $w_1 \circ w_2 = w_1 w_2$.

$$I_{str}(F) := \begin{cases} \varepsilon & \text{if } \theta(\bar{q}) = elist \\ \theta(\delta(\bar{q}, \text{CAR})) \circ I_{str}(F@\text{CDR}) & \text{if } \theta(\bar{q}) = nelist(char) \end{cases}$$

Lemma 3.4 (String Interpretability)
The function $I_{str} : \mathcal{IS}_{\mathcal{R}}^{list(char)} \to \Sigma^*$ is total.

Proof: To show that I_{str} is defined for an arbitrary $F = \langle Q, \bar{q}, \theta, \delta \rangle \in \mathcal{IS}_{\mathcal{R}}^{list(char)}$, proceed by induction on the length of F, which is computable for all elements of $\mathcal{IS}_{\mathcal{R}}^{list(char)}$ by Lemma 3.1.

- **Basis:** $length(F) = 0$
  Then, it must be that $\theta(\bar{q}) = elist$, so $I_{str}(F)$ is directly defined as the empty string ε.
- **Induction:** $length(F) = n + 1$
  Then, it must be that $\theta(\bar{q}) = nelist(char)$. Since F is totally well-typed, it must be that $F' = F@\text{CDR} = \langle Q', \bar{q}', \theta', \delta' \rangle$ is defined and that $list(char) \sqsubseteq \theta'(\bar{q}')$. By Definition 3.14, $length(F') = n$. By the inductive hypothesis, we can assume that $s = I_{str}(F') \in \Sigma^*$ is defined.

  By analagous reasoning, $F^c = F@\text{CAR} = \langle Q^c, \bar{q}^c, \theta^c, \delta^c \rangle$ is defined, and $char \sqsubseteq \theta^c(\bar{q}^c)$. Since F is extensionally grounded, it must be that F^c is extensionally grounded, so that $\theta^c(\bar{q}^c) \in \Sigma$, and therefore $I_{str}(F)$ is defined as $\theta^c(\bar{q}^c) \circ s$, which can also be characterized as:

  $I_{str}(F) = \theta(\delta^*(\text{CDR}^0\text{CAR}, \bar{q})) \circ \cdots \circ \theta(\delta^*(\text{CDR}^{n+1}\text{CAR}, \bar{q})) \circ \varepsilon$

$\square$

Definition 3.18 (Value Interpretation Function I_{val})
If $F = \langle Q, \bar{q}, \theta, \delta \rangle \in \mathcal{IS}_{\mathcal{R}}^{EDBValue}$, then $I_{val}(F)$, the interpretation of F as an atomic value, is defined as follows:

$$I_{val}(F) := \begin{cases} I_{num}(F) & \text{if } natnum \sqsubseteq \theta(\bar{q}) \\ I_{str}(F@\text{CHARS}) & \text{if } string \sqsubseteq \theta(\bar{q}) \end{cases}$$

Lemma 3.5 (Value Interpretability)
The function $I_{val} : \mathcal{IS}_{\mathcal{R}}^{EDBValue} \to \Sigma^* \cup \mathbb{N}$ is total.

Proof: If $F \in \mathcal{IS}_{\mathcal{R}}^{EDBValue}$, then it must be the case either that $natnum \sqsubseteq \theta(\bar{q})$ or that $string \sqsubseteq \theta(\bar{q})$, since these two subtypes of $EDBValue$ hold its only extensional subtypes, have no common upper bounds, and F is extensionally grounded.

If $natnum \sqsubseteq \theta(\bar{q})$, then $I_{val}(F) = I_{num}(F)$ is defined, by Lemma 3.3. Otherwise, if $string \sqsubseteq \theta(\bar{q})$, then it must be that $F' = \langle Q', \bar{q}', \theta', \delta' \rangle = F@\text{CHARS}$ is defined and that $list(char) \sqsubseteq \theta'(\bar{q}')$, since F is totally well-typed, so that $I_{val}(F) = I_{str}(F')$ is defined by Lemma 3.4. $\square$

Definition 3.19 (Argument Interpretation Function I_{arg})
If $F = \langle Q, \bar{q}, \theta, \delta \rangle \in \mathcal{IS}_{\mathcal{R}}^{CompArg}$, then $I_{arg}(F)$, the interpretation of F as a comparison argument, is defined as follows:

$$I_{arg}(F) := \begin{cases} \lambda t[I_{val}(F)] & \text{if } EDBValue \sqsubseteq \theta(\bar{q}) \\ \lambda t[t.I_{num}(F@\textsc{attr})] & \text{if } \theta(\bar{q}) = Attribute \end{cases}$$

Lemma 3.6 (Argument Interpretability)
The function $I_{arg} : \mathcal{IS}_{\mathcal{R}}^{CompArg} \rightarrow ((\Sigma^* \cup \mathbb{N})^* \rightarrow (\Sigma^* \cup \mathbb{N}))$ is total.

Proof: If $F = \langle Q, \bar{q}, \theta, \delta \rangle \in \mathcal{IS}_{\mathcal{R}}^{CompArg}$, then it must be the case that either $EDBValue \sqsubseteq \theta(\bar{q})$ or that $\theta(\bar{q}) = Attribute$. In the former case, we know from Lemma 3.5 that $I_{val}(F)$ is defined, and therefore that $I_{arg}(F) = \lambda t[I_{val}(F)]$ is defined as a constant function from tuples t of atomic objects to the interpretation of F as an atomic value.

If $\theta(\bar{q}) = Attribute$, then it must be that $F' = \langle Q', \bar{q}', \theta', \delta' \rangle = F@\textsc{attr}$ is defined and that $nonzero \sqsubseteq \theta'(\bar{q}')$, since F is totally well-typed. By Lemma 3.3, $k = I_{num}(F') \in \mathbb{N}$ must be defined, so that $I_{arg}(F)$ is defined as a function from tuples t of atomic objects to their k^{th} components. □

Lemma 3.7 (Argument Minimum Arity)
For every feature structure $F \in \mathcal{IS}_{\mathcal{R}}^{CompArg}$, there is a natural number $k \in \mathbb{N}$ such that the interpretation $I_{arg}(F)$ of F is a total function on atomic $k+n$-tuples, for all $n \in \mathbb{N}$. I will refer to k as the *minimum arity* of F, and write $arity_{min}(F) = k$, just as for the selectional formulae of relational algebra.

Proof: By Lemma 3.6, we know that $I_{arg}(F) \in ((\Sigma^* \cup \mathbb{N})^* \rightarrow (\Sigma^* \cup \mathbb{N}))$ is defined as a function from atomic object tuples to atomic objects. If $EDBValue \sqsubseteq \theta(\bar{q})$, then $I_{arg}(F)$ is the constant function $\lambda t[I_{val}(F)]$, according to Definition 3.19. The arity of the argument tuples t plays no role whatsoever in the evaluation of a constant function such as $I_{arg}(F)$, therefore $k = 0$.

Otherwise, it must be that $\theta(\bar{q}) = Attribute$, and $I_{arg}(F) = \lambda t[t.i]$, where $i = I_{num}(F@\textsc{attr})$. Since tuple-indexing expressions such as $t.i$ are defined just in case $1 \leq i \leq arity(t)$, $[I_{arg}(F)](t)$ is defined just in case $1 \leq I_{num}(F@\textsc{attr}) \leq arity(t)$.

By Definition 3.16, $1 \leq I_{num}(F@\textsc{attr})$, since F is totally well-typed and the feature $\textsc{attr}$ takes a value of type $nonzero$. Therefore, $I_{arg}(F)$ is defined for all tuples t with $I_{num}(F@\textsc{attr}) \leq arity(t)$, so $k = I_{num}(F@\textsc{attr})$. □

3.3.2.3 Functions for Expression Types

Definition 3.20 (Comparison Interpretation Function I_{cmp})
If $F = \langle Q, \bar{q}, \theta, \delta \rangle \in \mathcal{IS}_{\mathcal{R}}^{Comparison}$, then $I_{cmp}(F)$, the interpretation of F as an elementary comparison function, is defined as follows:

$$I_{cmp}(F) := \begin{cases} \lambda t[[I_{arg}(F@\text{ARGS:CAR})](t) = [I_{arg}(F@\text{ARGS:CDR})](t)] & \text{if } EQComp \sqsubseteq \theta(\bar{q}) \\ \lambda t[[I_{arg}(F@\text{ARGS:CAR})](t) < [I_{arg}(F@\text{ARGS:CDR})](t)] & \text{if } LTComp \sqsubseteq \theta(\bar{q}) \\ \lambda t[[I_{arg}(F@\text{ARGS:CAR})](t) > [I_{arg}(F@\text{ARGS:CDR})](t)] & \text{if } GTComp \sqsubseteq \theta(\bar{q}) \end{cases}$$

Lemma 3.8 (Comparison Interpretability)
The function $I_{cmp} : \mathcal{IS}_{\mathcal{R}}^{Comparison} \rightarrow ((\Sigma^* \cup \mathbb{N})^* \rightarrow \{0, 1\})$ is total.

Proof: Let $A = \Sigma^* \cup \mathbb{N}$ represent the collection of all atomic objects. If $F = \langle Q, \bar{q}, \theta, \delta \rangle \in \mathcal{IS}_{\mathcal{R}}^{Comparison}$, then it must be the case that $\theta(\bar{q}) \in \{EQComp, LTComp, GTComp\}$, since these are the only extensional types subsumed by *Comparison*. In all cases, the interpretation of the comparison arguments $f_1 = I_{arg}(F@\text{ARGS:CAR})$ and $f_2 = I_{arg}(F@\text{ARGS:CDR})$ are defined as functions from atomic object tuples to atomic objects, by total well-typedness, extensional groundedness, and Lemma 3.6: $f_1, f_2 \in A^* \rightarrow A$.

Assuming that the elementary comparison operations $C \in Comp = \{=, <, >\}$ are defined for all pairs of atomic objects $C : (A \times A) \xrightarrow{total} \{0, 1\}$, $I_{cmp}(F) = \lambda t[f_1(t)\ C\ f_2(t)]$ is defined as a (partial) function from atomic object tuples to truth values, $I_{cmp}(F) \in A^* \rightarrow \{0, 1\}$. □

Lemma 3.9 (Comparison Minimum Arity)
For every feature structure $F \in \mathcal{IS}_{\mathcal{R}}^{Comparison}$, there is a natural number $k \in \mathbb{N}$ such that the interpretation $I_{cmp}(F)$ of F is a total function on atomic $k + n$-tuples, for all $n \in \mathbb{N}$. I will refer to k as the *minimum arity* of F, and write $arity_{min}(F) = k$.

Proof: By Lemma 3.8, we know that the *CompArg* substructures $F_1 = F@\text{ARGS:CAR}$ and $F_2 = F@\text{ARGS:CDR}$ are defined and have as their interpretations functions $f_1, f_2 \in A^* \rightarrow A$. By Lemma 3.9, we know that there exist natural numbers $k_1 = arity_{min}(F_1)$ and $k_2 = arity_{min}(F_2)$ such that $f_1 \in A^{k_1+n} \xrightarrow{total} A$ and $f_2 \in A^{k_2+n} \xrightarrow{total} A$. Only if both $f_1(t)$ and $f_2(t)$ are defined for an arbitrary atomic object tuple t will $[I_{cmp}(F)](t)$ be defined, so it must be that $arity_{min}(F) \geq max(arity_{min}(F_1), arity_{min}(F_2))$. The arity of the argument tuple t plays no other role in the evaluation of $[I_{cmp}(F)](t)$, therefore $arity_{min}(F) = max(arity_{min}(F_1), arity_{min}(F_2))$. $\square$

3.3.3 Interpretation

In this section, I provide a formal characterization of semantic interpretations for interpretable relational query feature structures themselves. As a relational database query formalism, IRQFSs should have as their semantic value or extension some finite set of n-tuples satisfying Definition 2.1. As a query *language*, IRQFSs must be interpreted with respect to a given relational database DB. In accordance with the conventions common to currently available relational database management systems, I have defined IRQFSs to operate as a query language for *naming* relational databases – partial functions satisfyning Definition 2.3. I ignore here the other major convention common to existing RDBMSs, that of *named attributes*, for purposes of simplicity.

Definition 3.21 (Database Interpretation)
If $F = \langle Q, \bar{q}, \theta, \delta \rangle \in \mathcal{IF}_{\mathcal{R}}$ is an interpretable relational query feature structure and DB is a naming relational database, then $[\![F]\!]_{DB}$, the *database interpretation* of F with respect to DB is defined for the following cases:

- **Case:** $\theta(\bar{q}) = EDBRelation$
  $[\![F]\!]_{DB} = DB(I_{str}(F@\text{TABLE:STR:CHARS}))$

- **Case:** $\theta(\bar{q}) = Union$
  If $S_1 = [\![F@\text{QARGS:CAR}]\!]_{DB}$ and $S_2 = [\![F@\text{QARGS:CDR}]\!]_{DB}$ are both defined and such that $arity(S_1) = arity(S_2)$, then

  $[\![F]\!]_{DB} = S_1 \cup S_2$

- **Case:** $\theta(\bar{q}) = Product$
  If $S_1 = [\![F@\text{QARGS:CAR}]\!]_{DB}$ and $S_2 = [\![F@\text{QARGS:CDR}]\!]_{DB}$ are both defined, then

  $[\![F]\!]_{DB} = S_1 \otimes S_2$

- **Case:** $\theta(\bar{q}) = Difference$
  If $S_1 = [\![F@\text{QARGS:CAR}]\!]_{DB}$ and $S_2 = [\![F@\text{QARGS:CDR}]\!]_{DB}$ are both defined, then

$[\![F]\!]_{DB} = S_1 - S_2$

- **Case:** $\theta(\bar{q}) = Projection$
  Let $k = length(F@\textsc{attrs})$, and for $1 \leq j \leq k$, let $i_j = I_{num}(nth(j, F@\textsc{attrs})@\textsc{attr})$. If $S = [\![F@\textsc{from}]\!]_{DB}$ is defined, and if each i_j is such that $1 \leq i_j \leq arity(S)$, then

  $[\![F]\!]_{DB} = \{(t.i_1, \ldots, t.i_k) \mid t \in S\}$

- **Case:** $\theta(\bar{q}) = Selection$
  Let $F_P = F@\textsc{where}$ and let $P = I_{cmp}(F_P)$. If $S = [\![F@\textsc{from}]\!]_{DB}$ is defined, and if $arity_{min}(F_P) \leq arity(S)$, then

  $[\![F]\!]_{DB} = \{t \mid t \in S \text{ and } P(t)\}$

- $[\![F]\!]_{DB}$ is otherwise $undefined$.

F is said to be *database-interpretable* with respect to DB or simply DB-interpretable just in case $[\![F]\!]_{DB}$ is defined.

Let $\mathcal{DIF}_{\mathcal{R}}^{DB}$ designate the collection of relational query feature structures which are *database-interpretable* with respect to DB.

3.4 Expressiveness

In this section, I show that database-interpretable relational query feature structures are sound and complete with relational algebra, establishing that their expressive power as a query formalism is equivalent to that of relational algebra.

3.4.1 From Feature Structures to Algebra

In this section, I show that database interpretable relational query feature structures are sound with respect to relational algebra – that is, that every database interpretable relational query feature structure has as its interpretation a (finite, flat) relation achievable by some application of the relational algebraic operations to the relations in the database with respect to which the feature structure's interpretation is defined. Formally, this means

I must show for every database interpretable relational query feature structure $F \in \mathcal{DIF}_{\mathcal{R}}^{DB}$ that F's interpretation is an element of the carrier of that instance of relational algebra induced by the relational database $range(DB)$: $[\![F]\!]_{DB} \in \mathcal{C}(range(DB))$, where $\mathcal{C}$ is the relational-algebraic closure operation.

Theorem 3.1 (Relational Algebraic Soundness)
If $F = \langle Q, \bar{q}, \theta, \delta \rangle \in \mathcal{DIF}_{\mathcal{R}}^{DB}$ is a database-interpretable relational query feature structure with respect to the naming relational database DB, then there is an instance of relational algebra $\langle \mathcal{C}(range(DB)), \cup, -, \otimes, \pi, \sigma \rangle$ that contains the database interpretation of F with respect to DB: $[\![F]\!]_{DB} \in \mathcal{C}(range(DB))$.

Proof: Since DB is a naming relational database, it must be that the range of DB is a relational database satisfying Definition 2.2: $range(DB) \in \mathcal{RDB}$, by Definition 2.3. Every relational database $RDB \in \mathcal{RDB}$ induces a relational algebra $\langle \mathcal{C}(RDB), \cup, -, \otimes, \pi, \sigma \rangle$, by Definition 2.4; therefore the structure $\langle \mathcal{C}(range(DB)), \cup, -, \otimes, \pi, \sigma \rangle$ is an instance of relational algebra.

To show that $[\![F]\!]_{DB} \in \mathcal{C}(range(DB))$, I proceed by induction on the number of proper substructures of type $QueryExpr$ in F.

- **Basis:** *Zero proper substructures of type QueryExpr in F*
  Then, it must be that $\theta(\bar{q}) = EDBRelation$, since $F \in \mathcal{DIF}_{\mathcal{R}}^{DB}$ is totally well-typed and extensionally grounded, and all other types subsumed by $QueryExpr$ have features which take substructures of type $QueryExpr$ as their values. By Definition 3.21, $[\![F]\!]_{DB} = DB(I_{str}(F@$ TABLE:STR:CHARS$))$ is defined and therefore $[\![F]\!]_{DB} \in range(DB)$. Since the relational algebraic closure contains its inducers[3], $range(DB) \subseteq \mathcal{C}(range(DB))$, it must be that $[\![F]\!]_{DB} \in \mathcal{C}(range(DB))$.

- **Induction:** *n+1 proper substructures of type QueryExpr in F*
  Assume that for all $F' \in \mathcal{DIF}_{\mathcal{R}}^{DB}$ containing n or fewer proper substructures of type $QueryExpr$, the database interpretation of F' is expressable by means of the relational algebraic operations: $[\![F']\!]_{DB} \in \mathcal{C}(range(DB))$.

  Since $F \in \mathcal{DIF}_{\mathcal{R}}^{DB}$ is database interpretable with respect to DB, all proper substructures of type $QueryExpr$ in F must also be database

[3]For all $R \in DB$, $R = (R \cup R) \in \mathcal{C}(DB)$.

interpretable with respect to DB, by Definition 3.21. Since $F \in \mathcal{DIF}_{\mathcal{R}}^{DB}$ is acyclic, every proper substructure of F contains strictly fewer nodes than F itself; in particular, every proper substructure of type $QueryExpr$ must itself contain n or fewer proper subsutrctures of type $QueryExpr$. Therefore, the inductive hypothesis allows us to assert that for all proper substructures F' of type $QueryExpr$ in F, the substructure's interpretation is expressable by means of the relational algebraic operations: $[\![F']\!]_{DB} \in \mathcal{C}(range(DB))$.

This assumption is used below to construct $[\![F]\!]_{DB}$ by means of applications of the relational algebraic operations to the database interpretations of the proper $QueryExpr$ substructures of $F = \langle Q, \bar{q}, \theta, \delta \rangle$, in a fashion depending on the type of the root node of F.

- *Case:* $\theta(\bar{q}) = Union$
  Then, there exist $S_1 = [\![F@\text{QARGS:CAR}]\!]_{DB}$ and $S_2 = [\![F@\text{QARGS:CDR}]\!]_{DB}$ such that $arity(S_1) = arity(S_2)$, by Definition 3.21. By inductive hypothesis, $\{S_1, S_2\} \subseteq \mathcal{C}(range(DB))$, so that $[\![F]\!]_{DB} = (S_1 \cup S_2) \in \mathcal{C}(range(DB))$, since $\mathcal{C}(range(DB))$ is closed under the relational algebraic union operation.
- *Case:* $\theta(\bar{q}) = Product$
  Then, there exist $S_1 = [\![F@\text{QARGS:CAR}]\!]_{DB}$ and $S_2 = [\![F@\text{QARGS:CDR}]\!]_{DB}$, which by inductive hypothesis are such that $\{S_1, S_2\} \subseteq \mathcal{C}(range(DB))$, so that $[\![F]\!]_{DB} = (S_1 \otimes S_2) \in \mathcal{C}(range(DB))$, since $\mathcal{C}(range(DB))$ is closed under the relational algebraic product operation.
- *Case:* $\theta(\bar{q}) = Difference$
  Then, there exist $S_1 = [\![F@\text{QARGS:CAR}]\!]_{DB}$ and $S_2 = [\![F@\text{QARGS:CDR}]\!]_{DB}$, which by inductive hypothesis are such that $\{S_1, S_2\} \subseteq \mathcal{C}(range(DB))$, so that $[\![F]\!]_{DB} = (S_1 - S_2) \in \mathcal{C}(range(DB))$, since $\mathcal{C}(range(DB))$ is closed under the relational algebraic difference operation.
- *Case:* $\theta(\bar{q}) = Projection$
  Then, there exists a natural number $k = length(F@\text{ATTRS})$ which is computable by Lemma 3.1, and for $1 \leq j \leq k$ there exist natural numbers $i_j = I_{num}(nth(j,\ F@\text{ATTRS})@\text{ATTR})$, which are computable for $1 \leq j \leq k$ by Lemmas 3.2 and 3.3. By Definition 3.21 and inductive hypothesis, there exists a relational

algebraic object $S = [\![F@\textsc{from}]\!]_{DB} \in \mathcal{C}(range(DB))$. By Definition 3.21, each i_j is such that $1 \leq i_j \leq arity(S)$, so that the database interpretation of F, $[\![F]\!]_{DB} = \{(t.i_1, \ldots, t.i_k) \mid t \in S\}$ is equivalent to a relational algebraic projection operation on the database interpretation of its immediate *QueryExpr* substructure: $[\![F]\!]_{DB} = \pi_{i_1,\ldots,i_k}(S) \in \mathcal{C}(range(DB))$.

– *Case:* $\theta(\bar{q}) = Selection$
  Then, there exists a relation $S = [\![F@\textsc{from}]\!]_{DB} \in \mathcal{C}(range(DB))$, by Definition 3.21 and inductive hypothesis. Further, there exists a *Comparison* substructure $F_P = F@\textsc{where}$ with computable minimum arity $arity_{min}(F_P)$ such that $arity_{min}(F_P) \leq arity(S)$ by Definition 3.21 and Lemma 3.9. Also by Lemma 3.9, there exists a selectional formula $P = I_{cmp}(F_P) \in \mathcal{P}_{sel}$ which is defined for all tuples in S, so that the database interpretation of F, $[\![F]\!]_{DB} = \{t \mid t \in S \text{ and } P(t)\}$ is equivalent to a relational algebraic selection operation on the database interpretation of its immediate *QueryExpr* substructure: $[\![F]\!]_{DB} = \sigma_P(S) \in \mathcal{C}(range(DB))$.

– There are no other types which could be assigned to $\bar{q}$, therefore the proof is complete. □

3.4.2 Encoding Functions

The following functions provide feature structure encodings for various constituents of relational algebraic operations where possible. In the case of selectional formulae, where a direct encoding is impractical, a proof is presented that a feature structure encoding exists.

Definition 3.22 (Numeric Encoding Function I'_{num})
If $n \in \mathbb{N}$ is a natural number, then $I'_{num}(n)$, the encoding of n as a relational query substructure, is defined as follows:

$$I'_{num}(n) := \begin{cases} \begin{bmatrix} \textit{zero} \end{bmatrix} & \text{if } n = 0 \\ \begin{bmatrix} \textit{nonzero} & \\ \textsc{pred:} & I'_{num}(n-1) \end{bmatrix} & \text{if } n > 0 \end{cases}$$

Lemma 3.10 (Numeric Encodability)
The function $I'_{num} : \mathbb{N} \to \mathcal{IS}^{natnum}_{\mathcal{R}}$ is total.

Proof: I show that $I'_{num}(n) \in \mathcal{IS}^{natnum}_{\mathcal{R}}$ is defined for arbitrary $n \in \mathbb{N}$ by induction on n.

- **Basis:** $n = 0$
  Then, $I'_{num}(k)$ is directly defined.
- **Induction:** $n > 0$
  Assume that $I'_{num}(k) \in \mathcal{IS}^{num}_{\mathcal{R}}$ is defined for all $k \in \mathbb{N}$ such that $k < n$. Specifically, $I'_{num}(n-1) \in \mathcal{IS}^{num}_{\mathcal{R}}$ will be defined, so that $I'_{num}(n)$ is defined by the second clause of Definition 3.22. □

Lemma 3.11 (Numeric Encoding Recoverability)
For all natural numbers $n \in \mathbb{N}$, $I_{num}(I'_{num}(n)) = n$.

Proof: Induction on n.

- **Basis:** $n = 0$
  Then, $I'_{num}(n) = [zero]$ by Definition 3.22, so that by Definition 3.16, $I_{num}(I'_{num}(n)) = 0 = n$.
- **Induction:** $n > 0$
  Assume that $I_{num}(I'_{num}(k)) = k$, for $k \in \mathbb{N}$ such that $k < n$. In particular, $I_{num}(I'_{num}(n-1)) = n - 1$, so that by Definitions 3.22 and 3.16,
  $$\begin{aligned} I_{num}(I'_{num}(n)) &= I_{num}(I'_{num}(n-1)) + 1 \\ &= (n-1) + 1 \\ &= n \end{aligned}$$
  □

Definition 3.23 (String Encoding Function I'_{str})
If $w \in \Sigma^*$ is a finite character string, then $I'_{str}(w)$, the encoding of w as a relational query substructure, is defined as follows:

$$I'_{str}(w) = \begin{cases} \begin{bmatrix} elist \end{bmatrix} & \text{if } w = \varepsilon \\[2ex] \begin{bmatrix} nelist(char) & \\ \text{CAR:} & \begin{bmatrix} c \end{bmatrix} \\ \text{CDR:} & I'_{str}(w') \end{bmatrix} & \text{if } w = cw', c \in \Sigma, w' \in \Sigma^* \end{cases}$$

Lemma 3.12 (String Encodability)
The function $I'_{str} : \Sigma^* \rightarrow \mathcal{IS}_{\mathcal{R}}^{list(char)}$ is total.

Proof: I show that $I'_{str}(w)$ is defined for arbitrary $w \in \Sigma^*$ by induction on $len(w)$, the length of w.

- **Basis:** $len(w) = 0$
  Then, $w = \varepsilon$, so $I'_{str}(w)$ is directly defined.

- **Induction:** $len(w) = n + 1$
  Assume that $I'_{str}(w')$ is defined for $w' \in \Sigma^*$ such that $len(w') \leq n$. Since $len(w) = n+1$, there must be a $c \in \Sigma$ and a $w' \in \Sigma^*$ with $length(w') = n$ such that $w = cw'$. By the inductive hypothesis, $I'_{str}(w') \in \mathcal{IS}_{\mathcal{R}}^{list(char)}$ is defined, so $I'_{str}(w)$ is defined by the second clause of Definition 3.23.
  $\square$

Lemma 3.13 (String Encoding Recoverability)
For all $w \in \Sigma^*$, $I_{str}(I'_{str}(w)) = w$.

Proof: Induction on $len(w)$.

- **Basis:** $len(w) = 0$
  Then, $w = \varepsilon$, so $I'_{str}(w) = [elist]$ by Definition 3.23, and therefore $I_{str}(I'_{str}(w)) = \varepsilon = w$, by Definition 3.17.

- **Induction:** $len(w) = n + 1$
  Assume that $I_{str}(I'_{str}(w')) = w'$ for all $w' \in \Sigma^*$ such that $len(w') \leq n$. Since $len(w) = n + 1$, there must be a $c \in \Sigma$ and a $w' \in \Sigma^*$ with $len(w') = n$ such that $w = cw'$. The inductive hypothesis states that $I_{str}(I'_{str}(w')) = w'$. By Definitions 3.23 and 3.17, then, $I_{str}(I'_{str}(w)) = c \circ I_{str}(I'_{str}(w')) = c \circ w' = cw' = w$. $\square$

Definition 3.24 (Value Encoding Function I'_{val})
If $v \in \Sigma^* \cup \mathbb{N}$ is a constant atomic value, then $I'_{val}(v)$, the encoding of v as a relational query substructure is defined as follows:

$$I'_{val}(v) = \begin{cases} I'_{num}(v) & \text{if } v \in \mathbb{N} \\ \begin{bmatrix} \mathit{string} & \\ \text{CHARS:} & I'_{str}(v) \end{bmatrix} & \text{if } v \in \Sigma^* \end{cases}$$

Lemma 3.14 (Value Encodability)
The function $I'_{val} : (\Sigma^* \cup \mathbb{N}) \rightarrow \mathcal{IS}_{\mathcal{R}}^{EDBValue}$ is total.

Proof: Follows from Lemmas 3.10 and 3.12, given that we can distinguish between natural numbers and character strings. □

Lemma 3.15 (Value Encoding Recoverability)
For all $v \in \Sigma^* \cup \mathbb{N}$, $I_{val}(I'_{val}(v)) = v$.

Proof: Follows from Lemms 3.11 and 3.13. □

Lemma 3.16 (Comparison Encodability)
If $P \in \mathcal{P}_{sel}$ is a selectional formula of relational algebra, then there exists an interpretable relational query substructure $F \in \mathcal{IS}_{\mathcal{R}}^{Comparison}$ such that $I_{cmp}(F) = P$ and $arity_{min}(F) = arity_{min}(P)$.

Proof: According to Definition 2.6, any selectional formula P must be describable by an expression of the λ-calculus of the form $\lambda t[x_1 C x_2]$, where $C \in Comp$ is a comparison operator and each x_i $(1 \leq i \leq 2)$ is either an atomic constant or an indexing operation $t.j$ $(j \geq 1)$ on the tuple-variable t.

First, I show that for each x_i $(1 \leq i \leq 2)$, there exists an interpretable relational query substructure F_i such that $I_{arg}(F_i) = \lambda t[x_i]$ and $arity_{min}(F_i) = arity_{min}(x_i)$. If x_i is a constant, the existence of such an F_i is guaranteed by Lemma 3.15, and the preservation of minimum arity follows from the definition in Section 2.3.3 and Lemma 3.7: $arity_{min}(F_i) = 0 = arity_{min}(x_i)$.

Otherwise, x_i must be a tuple-indexing expression on the tuple-variable t of the form $t.j$, where $j \geq 1$, and $arity_{min}(x_i) = j$, by Definitions 2.6 and 2.7. In this case, we can construct F_i as:

$$F_i = \begin{bmatrix} \mathit{Attribute} & \\ \text{ATTR:} & I'_{num}(j) \end{bmatrix}$$

which by Definition 3.19 and Lemma 3.11 is such that $I_{arg}(F_i) = \lambda t[t.j] = \lambda t[x_i]$, and by Lemmas 3.7 and 3.11 preserves minimum arity: $arity_{min}(F_i) = j = arity_{min}(x_i)$.

Given these two argument-encodings, we can find an extensional type $C' \in ExtType_{\mathcal{R}}$ such that $Comparison \sqsubseteq C'$ (for $C =$ "$=$", C' will be *EQComp*, etc.), and use it to construct F as:

$$F = \begin{bmatrix} C' & \\ & \\ \text{ARGS:} & \begin{bmatrix} pair(Argument) & \\ \text{CAR:} & F_1 \\ \text{CDR:} & F_2 \end{bmatrix} \end{bmatrix}$$

so that by Definition 3.20,

$$\begin{aligned} I_{cmp}(F) &= \lambda t[\lambda t[I_{arg}(F_1)](t)\ C\ \lambda t[I_{arg}(F_2)](t)] \\ &= \lambda t[\lambda t[x_1](t)\ C\ \lambda t[x_2](t)] \\ &= \lambda t[x_1\ C\ x_2] \\ &= P \end{aligned}$$

Since $arity_{min}(x_i) = arity_{min}(F_i)$ for $1 \leq i \leq 2$, $arity_{min}(F) = arity_{min}(P)$, by Definition 2.8 and Lemma 3.9. □

3.4.3 From Algebra to Feature Structures

In this section, I show that database interpretable relational query feature structures are complete with respect to relational algebra – that is, that every relation achievable by some application of the relational algebraic operations to the relations in a naming relational database can be characterized as the interpretation of some relational query feature structure. Formally, I must show that for every every relation $R \in \mathcal{C}(range(DB))$ in the carrier of the relational algebra induced by a naming relational database, there exists a database interpretable relational query feature structure $F \in \mathcal{DIF}_{\mathcal{R}}^{DB}$ such that $R = [\![F]\!]_{DB}$.

Theorem 3.2 (Relational Algebraic Completeness)
If $R \in \mathcal{C}(range(DB))$ is an object in the carrier of the relational algebra $\langle \mathcal{C}(range(DB)), \cup, -, \otimes, \pi, \sigma \rangle$ induced by the range of a naming relational database $DB \in \mathcal{NRDB}$, then there exists a DB-interpretable relational query feature structure $F \in \mathcal{DIF}_{\mathcal{R}}^{DB}$ whose interpretation is set-identical with R: $[\![F]\!]_{DB} = R$.

Proof: Induction on the least number of relational algebraic operations required to achieve R.

- **Basis:** *Zero operations required to achieve R*
  Then, it must be that $R \in range(DB)$, so that there is some relation

name $w \in \Sigma^*$ such that $R = DB(w)$. By Lemma 3.12, we can construct a feature structure $F_w = I'_{str}(w)$ to represent w, and therefore a feature structure

$$F = \begin{bmatrix} \textit{EDBRelation} & \\ \text{TABLE:} & \begin{bmatrix} \textit{literal} & \\ \text{STR:} & \begin{bmatrix} \textit{string} & \\ \text{CHARS:} & F_w \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

By Lemma 3.13, we know that $I_{str}(I'_{str}(w)) = w$, so that by Definition 3.21, $[\![F]\!]_{DB} = DB(I_{str}(F_w)) = DB(w) = R$.

- **Induction:** $n + 1$ *operations required to achieve* R
Assume that for each $R' \in \mathcal{C}(range(DB))$ such that R' is achievable by application of n or fewer operations, there exists a DB-interpretable feature structure $F' \in \mathcal{DIF}^{DB}_{\mathcal{R}}$ with $[\![F']\!]_{DB} = R'$. The proof proceeds by case-based reasoning on the final operation performed to achieve R: the operands of this operation must themselvesbe achievable by application of n or fewer operations, since $n+1$ is the least number of operations required to achieve R.

  - *Case:* $R = S_1 \cup S_2$
  Then, there exist by inductive hypothesis $F_1, F_2 \in \mathcal{DIF}^{DB}_{\mathcal{R}}$ such that $[\![F_1]\!]_{DB} = S_1$ and $[\![F_2]\!]_{DB} = S_2$, so that we can construct a feature structure

$$F = \begin{bmatrix} \textit{Union} & \\ \text{QARGS:} & \begin{bmatrix} \textit{pair}(\textit{QueryExpr}) & \\ \text{CAR:} & F_1 \\ \text{CDR:} & F_2 \end{bmatrix} \end{bmatrix}$$

  which by Definition 3.21 is DB-interpretable, $F \in \mathcal{DIF}^{DB}_{\mathcal{R}}$, since $arity(S_1) = arity(S_2)$, by Definition 2.4, and such that $[\![F]\!]_{DB} = [\![F_1]\!]_{DB} \cup [\![F_2]\!]_{DB} = S_1 \cup S_2 = R$.

  - *Case:* $R = S_1 \otimes S_2$
  Then, there exist by inductive hypothesis $F_1, F_2 \in \mathcal{DIF}^{DB}_{\mathcal{R}}$ such that $[\![F_1]\!]_{DB} = S_1$ and $[\![F_2]\!]_{DB} = S_2$, so that we can construct a feature structure

$$F = \begin{bmatrix} \mathit{Product} & \\ & \\ \text{QARGS:} & \begin{bmatrix} \mathit{pair(QueryExpr)} & \\ \text{CAR:} & F_1 \\ \text{CDR:} & F_2 \end{bmatrix} \end{bmatrix}$$

which by Definition 3.21 is DB-interpretable, $F \in \mathcal{DIF}_{\mathcal{R}}^{DB}$, and such that $[\![F]\!]_{DB} = [\![F_1]\!]_{DB} \otimes [\![F_2]\!]_{DB} = S_1 \otimes S_2 = R$.

– *Case:* $R = S_1 - S_2$
Then, there exist by inductive hypothesis $F_1, F_2 \in \mathcal{DIF}_{\mathcal{R}}^{DB}$ such that $[\![F_1]\!]_{DB} = S_1$ and $[\![F_2]\!]_{DB} = S_2$, so that we can construct a feature structure

$$F = \begin{bmatrix} \mathit{Difference} & \\ & \\ \text{QARGS:} & \begin{bmatrix} \mathit{pair(QueryExpr)} & \\ \text{CAR:} & F_1 \\ \text{CDR:} & F_2 \end{bmatrix} \end{bmatrix}$$

which by Definition 3.21 is DB-interpretable, $F \in \mathcal{DIF}_{\mathcal{R}}^{DB}$, and such that $[\![F]\!]_{DB} = [\![F_1]\!]_{DB} - [\![F_2]\!]_{DB} = S_1 - S_2 = R$.

– *Case:* $R = \pi_{i_1,\ldots,i_k}(S)$
Then, there exists by inductive hypothesis a DB-interpretable feature structure $F_S \in \mathcal{DIF}_{\mathcal{R}}^{DB}$ such that $[\![F_S]\!]_{DB} = S$. By Definition 2.5, each i_j is such that $1 \leq i_j \leq arity(S)$, for $1 \leq j \leq k$. By Lemma 3.10, we can construct for each i_j a feature structure $F_j = I'_{num}(i_j) \in \mathcal{IS}_{\mathcal{R}}^{natnum}$. From each F_j, then, we can construct a feature structure $F_j^A \in \mathcal{IS}_{\mathcal{R}}^{Attribute}$ as:

$$F_j^A = \begin{bmatrix} \mathit{Attribute} & \\ \text{ATTR:} & F_j \end{bmatrix}$$

From these F_j^A, we can construct a feature structure F as:

$$F = \begin{bmatrix} \mathit{Projection} & \\ \text{FROM:} & F_s \\ \text{ATTRS:} & \langle^{list(Attribute)} F_1^A, \ldots, F_k^A \rangle \end{bmatrix}$$

Since $1 \leq i_j \leq arity(S)$ for $1 \leq j \leq k$, by Definition 2.4, and since the i_j are recoverable from their feature structure encodings by Lemma 3.11, $F \in \mathcal{DIF}_{\mathcal{R}}^{DB}$ is DB-interpretable by Definition 3.21, and such that $[\![F]\!]_{DB} = \{(t.i_1, \ldots, t.i_k) \mid t \in [\![F_S]\!]_{DB}\} =$

$\{(t.i_1, \ldots, t.i_k) \mid t \in S\} = \pi_{i_1,\ldots,i_k}(S) = R.$

– *Case:* $R = \sigma_P(S)$
Then, there exists by inductive hypothesis a DB-interpretable feature structure $F_S \in \mathcal{DIF}_{\mathcal{R}}^{DB}$ such that $[\![F_S]\!]_{DB} = S$. By Definition 2.9, $P \in \mathcal{P}_{sel}$ is a selectional formula for which a feature structure equivalent F_P exists, with $I_{cmp}(F_P) = P$ and $arity_{min}(F_P) = arity_{min}(P)$ by Lemma 3.16, so we can construct a feature structure F as:

$$F = \begin{bmatrix} Selection & \\ \text{FROM:} & F_S \\ \text{WHERE:} & F_P \end{bmatrix}$$

Since by Definition 3.21 $arity_{min}(F_P) \leq arity(S)$, $F \in \mathcal{DIF}_{\mathcal{R}}^{DB}$ is DB-interpretable and is such that

$$\begin{aligned} [\![F]\!]_{DB} &= \{t \mid t \in [\![F_S]\!]_{DB} \text{ and } P(t)\} \\ &= \{t \mid t \in S \text{ and } P(t)\} \\ &= \sigma_P(S) \\ &= R \end{aligned}$$

– There are no other relational algebraic operations which could be applied to achieve R, therefore the proof is complete. □

3.4.4 Expressive Power

Corollary 3.1 (RQFS Expressiveness)
Database interpretable relational query feature structures are equivalent in expressive power to relational algebra.

Proof: The proof follows from theorems 3.1 and 3.2. □

Chapter 4

Implementation

In this chapter, I describe an implementation of the relational query feature structures (RQFSs) from Chapter 3 as a class library for the programming language Perl.

Section 4.1 provides an overview of the implementation, while Section 4.2 contains more detailed information on selected individual modules.

4.1 Overview

I have implemented relational query feature structures as a class of objects in the high-level programming language Perl which produce database queries in the language SQL. The complete source code and documentation for the library of Perl classes and modules comprising this implementation can be found on the compact disc included with this document. In what follows, I will refer to the Perl library I have implemented as "the QuD library", or simply "QuD".

Section 4.1.1 provides some background information on the Perl programming language and an argument that it is indeed a suitable host language for relational query feature structures. Section 4.1.2 briefly describes the database query language SQL, while Section 4.1.3 briefly introduces each of the major subdivisions of the QuD library.

4.1.1 Perl

My implementation of relational query feature structures is written in the Perl programming language (Wall et al. 1996). Perl was chosen for a number of reasons, which are outlined below.

1. **Availability**
   Perl is freely available software, as are many extensions to it. Because of their free availability, I was able to make use of existing Perl libraries written by other programmers, without subjecting the QuD library itself to restrictive licensing conditions often enforced by libraries of other programming languages. In particular, the `DBI` module (Bunce 2001) provides a high-level object-oriented interface to many currently available relational database management systems.

2. **Portability**
   Perl itself runs on most currently popular platforms, and the key features of the QuD library do not rely on any other platform-dependent constants.

3. **High Level of Abstraction**
   Perl as a language provides a high level of symbolic abstraction and many built-in data structures which greatly eased the task of programming a large library by allowing me to focus on the essential properties of the implementation.

4. **String Processing Capability**
   Perl provides a number of powerful built-in mechanisms for processing character strings, including native support for tokenization, regular expression pattern matching, and regular transduction. These capabilities make Perl particularly well-suited as a host language for linguistic applications.

4.1.2 SQL

For the basic interface to relational databases, I chose the common database query language SQL (ANSI X3.135-1992, ISO/IEC 9075:1992) which is derived from and equivalent to the relational algebra as defined in Section

2.3 (Ullman 1982, 1999). The freely available Perl `DBI` module provides an abstract object-oriented interface to many currently popular database management systems accepting SQL as a query language. The SQL output of relational query feature structure objects can therefore be passed to most currently available database systems through the `DBI` module.

4.1.3 Organization

The development of the QuD library closely paralleled the development of the theoretical background information in Chapter 2, and the structural relationships between the various modules reflect this pattern as well. For the most part, I attempted to implement the various constructs from Chapters 2 and 3 in an object-oriented fashion as Perl classes[1] – some modules however, were defined simply to add additional functionality to existing classes. The modules of the QuD library can be broken down into the following major categories:

- **Support Modules**
  Support modules such as `Set::Hashed` are used by many higher-level QuD classes. Third party modules such as Steffen Beyer's `Bit::Vector` (Beyer 2001) also provide low-level support for some QuD classes. Such modules are not described further here.

- **Hierarchy Modules**
  Datatype hierarchies, the theoretical definitions concerning which can be found in Section 2.4, are implemented as objects descending from an abstract base class `QuD::Hierarchy`. Rather than strictly enforce the condition that hierarchies be consistently complete partial orders as required in Section 2.4, `QuD::Hierarchy` objects are defined such as to represent arbitrary finite partial orders, although they do provide additional functionality for deterministic CCPOs.

- **Feature Structure Modules**
  Feature structures as such are implemented as Perl objects derived from

[1] Technically, a Perl *class* is simply a *package*, and a Perl *object* is a *reference* which is *blessed* into the package implementing its class – for details, see (Wall et al. 1996, Chapter 5).

the class `QuD::FStructure::Base`. Basic feature-structure operations such as path-value assignment and unification are supported. Two additional interfaces to standard Perl data structures are also provided.

- **Relational Query Feature Structure Modules**
  Relational query feature structures are implemented as classes of a native Perl inheritance hierarchy, and are additionally associated with a `QuD::Hierarchy` object. Objects of the RQFS subhierarchy inherit from the class `QuD::RQFS`, and provide an additional method for the generation of database queries as Perl strings containing a query of SQL.

4.2 Perl Modules

4.2.1 Hierarchy Modules

Datatype hierarchies such as those described in Section 2.4 are implemented as subclasses of an abstract base class `QuD::Hierarchy`, which is described in Section 4.2.1.1. Actually, the `QuD::Hierarchy` class is capable of representing any finite partial order, not just consistently complete ones. All of the subclasses described below additionally enforce a single-root condition, but this is not required directly by the abstract `QuD::Hierarchy` class itself.

Hierarchies are defined in terms of the "covers" relation, as described in Section 3.1, which is referred to here and in the documentation more intuitively as the "has-parent" relation. Its converse – the "is-covered-by" relation – is referred to as the "has-child" relation.

Hierarchies defined in terms of immediate inheritance information such as the "has-parent" and "has-child" relations are not particularly well-suited for the computation of type-join and -meet operations by means of the "pure" logical definitions of these operations as given in Section 2.4.

Alternative compuational mechansims such as those described by Aït-Kaci, Boyer, Lincoln & Nasr (1989) rely on an expensive compilation phase, and thus require their hierarchies to be static. I did not wish to enforce this constraint, and so in addition to compiling hierarchy subclasses such as `QuD::Hierarchy::CMasked` (described in Section 4.2.1.6), I defined a subclass `QuD::Hierarchy::Set` (described in Section 4.2.1.3) which uses an iterative algorithm inherited from the abstract hierarchy base class to compute least upper bounds based only on the "has-child" relation, and derived from this class two hierarchy classes which cache the results of hierarchy lookups: `QuD::Hierarchy::Caching` and `QuD::Hierarchy::LRUCaching` (described in Sections 4.2.1.4 and 4.2.1.5, respectively).

A simplified version of the iterative LUB algorithm – which to the best of my knowledge is my own original work – is given in Figure 4.1. The basic idea is that the algorithm "climbs" up the hierarchy, from parent-types to child-types, and stops when it has found the least upper bounds. This method of computation is more efficient than the "brute-force" pure logical approach for non-compiling hierarchies, but still leaves much room for improvement. It will always terminate for non-cyclic hierarchies.[2]

User-defined "hooks" into subsumption checks, as well as into the LUB and GLB operations are provided in the object-oriented style as well as in the purely functional style.

For large hierarchies[3], the module `QuD::Hierarchy::Loader` (described in Section 4.2.1.2) is provided, which allows storage and retrieval of hierarchy information in a portable binary format, which can be shared even between different hierarchy subclasses.

4.2.1.1 QuD::Hierarchy

This package defines the API (or *Application Program Interface*) for hierarchy subclasses, and provides some basic functionality in terms of class methods which it requires its descendants to define. Subsumption-tests, as well as non-deterministic[4] LUB and GLB operations are supported, and default iterative algorithms for their computation are provided, such as the LUB

[2] To catch pathological (cyclic) hierarchies, the actual implementation keeps track of the number of "child" links it has followed, and terminates if this number grows larger than the number of types in the hierarchy.

[3] I consider a hierarchy "large" if it contains at least five thousand types.

[4] The LUB and GLB operations supported by `QuD::Hierarchy` are "non-deterministic" in the following sense: for a non-CCPO hierarchy, the `lub()` method, for example, may return a *list* of pairwise incomparable types as the "least upper bounds" of two given datatypes. For a hierarchy which meets the structural criteria for a CCPO, the returned list will contain at most one element.

```
function lub-iter (t1, t2)
begin
    queued1 := {t1}; upper1 := {t1};
    queued2 := {t2}; upper2 := {t2};
    solutions := ∅;
    while (queued1 ≠ ∅ or queued2 ≠ ∅) do begin
        next := ∅;
        foreach e ∈ queued1 do begin
            upper1 := upper1 ∪ {e};
            if e ∈ upper2 then begin
                foreach s ∈ solutions do
                    if s ⊏ e then break
                    else if e ⊏ s then
                        solutions := (solutions − {s}) ∪ {e};
            end else next := next ∪ children(e);
        end ;
        queued1 := next;
        next := ∅;
        foreach e ∈ queued2 do begin
            upper2 := upper2 ∪ {e};
            if e ∈ upper1 then begin
                foreach s ∈ solutions do begin
                    if s ⊏ e then break
                    else if e ⊏ s then
                        solutions := (solutions − {s}) ∪ {e};
            end else next := next ∪ children(e);
        end ;
        queued2 := next;
    end ;
    return solutions;
end /* lub-iter */
```

Figure 4.1: An iterative algorithm for computing least upper bounds

algorithm given in Figure 4.1.

4.2.1.2 QuD::Hierarchy::Loader

The `QuD::Hierarchy::Loader` module provides persistence for hierarchy objects. It extends the `QuD::Hierarchy` package with methods for text-mode loading and saving of hierarchy data, as well as methods for binary storage and retrieval of data, using Raphael Manfredi's `Storable` module (Manfredi 2001). `QuD::Hierarchy::Loader` also provides some methods for hierarchy visualization, producing diagrams similar to the ones in Section 3.2.

4.2.1.3 QuD::Hierarchy::Set

`QuD::Hierarchy::Set` is a basic iterative hierarchy implementation, suitable for use with small hierarchies. It uses my `Set::Hashed` module to represent sets of types, which itself owes a lot to Jarkko Hietaniemi's `Set::Scalar` package (Hietaniemi 2001). It is the most transparent of all the hierarchy subclasses, but also the least efficient.

4.2.1.4 QuD::Hierarchy::Caching

`QuD::Hierarchy::Caching` is a `QuD::Hierarchy` implementation for data-type hierarchies which caches the results of all inheritance- and type-operation lookups using perl hashes, which improves performance for small- to mid-sized hierarchies. It inherits from `QuD::Hierarchy::Set`.

4.2.1.5 QuD::Hierarchy::LRUCaching

`QuD::Hierarchy::LRUCaching` is a `QuD::Hierarchy` implementation for hierarchies inheriting from `QuD::Hierarchy::Set` which caches the results of inheritance- and type-operation-lookups in a `Tie::Cache` object (Chamas 2001), which implements an LRU (*Least Recently Used*) cache. This may improve performance for large hierarchies which must repeatedly perform the same lookups, or for applications using localized areas of large hierarchies.

4.2.1.6 QuD::Hierarchy::CMasked

`QuD::Hierarchy::CMasked` is a compiling `QuD::Hierarchy` implementation. It is derived from a chain of three hierarchy subclasses not described here. It uses the `Bit::Vector` module (Beyer 2001) to represent compiled hierarchy information as a pair of $n \times n$ boolean matrices, where n is the number of types in the hierarchy. The compiled hierarchy information is just the transitive closure of the "has-parent" and "has-child" relations, respectively. This common encoding is described by Aït-Kaci et al. (1989). Subsumption lookups are very fast, but the use by this class of the pure brute-force logical definition to compute least upper bounds appears not to provide much improvement over the iterative algorithm described above, especially for large hierarchies.

4.2.2 Feature Structure Modules

Feature structures as described in Section 2.5 are implemented as perl objects. The implementation closely mirrors the formal definition of feature structures as entities similar to deterministic finite state automata.

The most complex feature-structure operation is that of *unification*, (Definition 2.19, page 22). The most computationally complex aspect of computing the unification of two feature structures is the identification of nodes – in terms of Definition 2.19, the "identification of nodes" is the construction of the equivalence relation $\bowtie$.

Aït-Kaci (1984) includes an algorithm for computing the equivalence-classes $[q]_{\bowtie}$ for two arbitrary node-disjunct finite state automata, which I adapted for the QuD implementation, and which is reproduced in a slightly modified form in Figure 4.2.

In addition to the abstract base class `QuD::FStructure::Base` described in Section 4.2.2.1, additional subclasses are provided for embedding linked lists, as well as numerically indexed arrays in FStructure objects. These subclasses are described in Sections 4.2.2.2 and 4.2.2.3, respectively. Garbage-collection routines are provided by the `Safe` subclass described in Section 4.2.2.4. Section 4.2.2.5 describes the front end to the QuD feature structure classes, and one interface to standard Perl data structures is described in Section 4.2.2.6. My implementation of appropriateness specifications is described in Section

function merge-coreference ($\langle Q_1, \bar{q}_1, \delta_1\rangle$, $\langle Q_2, \bar{q}_2, \delta_2\rangle$)
begin
    $pairs$:= $\{\langle \bar{q}_1, \bar{q}_2\rangle\}$;
    $classes$:= $\emptyset$;
    foreach $q \in (Q_1 \cup Q_2)$ **do**
        $classes$:= $classes \cup \{\{q\}\}$;
    while $pairs \neq \emptyset$ **do begin**
        $\langle q_1, q_2\rangle$:= pick_element($pairs$);
        $pairs$:= $pairs - \{\langle q_1, q_2\rangle\}$;
        $class_1$:= find_class_of($q_1, classes$);
        $class_2$:= find_class_of($q_2, classes$);
        if $class_1 \neq class_2$ **then begin**
            $class_1$:= $class_1 \cup class_2$;
            $classes$:= $classes - class_2$;
            foreach $f \in Feat$ **do begin**
                if (defined($\delta_1(f, q_1)$) **and** defined($\delta_2(f, q_2)$)) **then**
                    $pairs$:= $pairs \cup \{ \langle \delta_1(f, q_1),\ \delta_2(f, q_2)\rangle \}$;
            end
        end
    end ;
    return $classes$;
end /* merge-coreference */

Figure 4.2: An algorithm to compute node equivalence classes

4.2.2.7.

4.2.2.1 QuD::FStructure::Base

The `QuD::FStructure::Base` module provides an abstract base-class for feature-structure ("FStructure") objects. All of the elementary FStructure operations (such as `subsumes()`, `unify()`, `delta()`, `theta()`, *etc.*) are defined in this package, from which all other FStructure objects inherit.

4.2.2.2 QuD::FStructure::List

`QuD::FStructure::List` provides linked lists such as those defined in Section 3.2.2 for QuD FStructure objects by adding some basic list-processing methods (such as `list_length()` and `nth_elt()`) to the `QuD::FStructure-::Base` package.

4.2.2.3 QuD::FStructure::Array

`QuD::FStructure::Array` provides array functionality for FStructure objects. Arrays are implemented internally as Perl array references, and appear as substructures with numeric indices as features. Arrayed FStructures as defined by this module are nonetheless entirely compatible with "standard" feature structures, and even allow the use of array indices and symbolic features at one and the same node.

4.2.2.4 QuD::FStructure::Safe

`QuD::FStructure::Safe` provides garbage-collection routines in the style of Lisp for FStructure objects. Since operations such as feature-transition assignment and path-value assignment may cause nodes in a feature structure to become unreachable, FStructure objects must be periodically investigated in their entirety for "dangling" nodes – this is a time-consuming process and should not be performed after every destructive operation.[5] The `Safe`

[5]Unification itself (even destructive unification) is not affected by such dangling nodes, nor does the unification operation create any. The unification operation *will* however perpetuate dangling nodes if any exist in either of its operands.

class allows users to define a maximum number of destructive operations a given FStructure object will undergo before that structure is "swept" and any dangling nodes are removed.

4.2.2.5 QuD::FStructure

`QuD::FStructure` is the front-end class for the FStructure classes described above. It inherits from the `List`, `Array`, `Safe`, and `Base` classes. It also automatically loads the `perldata` interface methods, not described here.

4.2.2.6 QuD::FStructure::Tied

`QuD::FStructure::Tied` provides an interface to `FStructure` objects as recursive Perl hashes[6] using Perl's `Tie::Hash` module.[7]

4.2.2.7 QuD::Approp

`QuD::Approp` is the least modular package in the QuD library. It adds methods both to `QuD::Hierarchy` and to `QuD::FStructure::Base` for the declaration and checking of feature value appropriateness, as defined in Section 2.5.4.

Appropriateness is declared minimally in terms of an "immediate appropriateness specification", very similar to the one described in Section 3.1.4. Both of the constraints on appropriateness specifications are relaxed – for instance, there need not be a single most general introductory type, nor must the appropriateness specification be upward-closed. Users are allowed to override inherited appropriateness constraints at any level of the hierarchy, and even

[6] Otherwise known as "associative arrays", Perl hashes can be thought of as finite partial functions from finite character strings to any Perl "scalar" data structure, which includes references to (other) hashes.

[7] This module is really just a workaround for what I see as a failing of the QuD library: namely, that FStructure objects are not directly defined as Perl hash references, to which they bear a striking resemblance in practice. Were it possible to define FStructures directly as Perl hashes, it might be possible to discard the `Tied` module, the `Safe` module, as well as the large and unwieldy `perldata` module, and to rely on Perl's internal reference-counting memory-management scheme to ensure that dangling nodes ceased to exist as soon as all references to them were lost.

to negate a feature's appropriateness (by declaring its appropriate value to be the inconsistent type $\top$, or "top").

User "hooks" are built into the appropriateness-checking hierarchy methods `is_approp()` and `get_approp()`, which allows users to construct appropriateness specifications even for FStructure objects with a potentially infinite number of features, such as `QuD::FStructure::Array` objects.

4.2.3 Relational Query Feature Structure Modules

Relational query feature structures are implented as a Perl class hierarchy gathered under the abstract base type `QuD::RQFS`. A `QuD::Hierarchy` object is defined which parallels Perl's built-in class hierarchy, to allow for type operations on RQFS feature structures not supported directly by Perl (such as LUB and GLB operations).

The RQFS base class `QuD::RQFS` inherits from the class `QuD::FStructure`, and supports all feature-structure operations defined in `QuD::FStructure::Base`, such as subsumption and unification. An appropriateness specification is declared for RQFS objects by the conventions defined in the `QuD::Approp` module, and the "derived" types from Section 3.2.2 are defined as FStructure arrays, using closures[8] to capture type constraints at a general level, in terms of the user "hooks" provided by `QuD::Approp`.

4.2.3.1 QuD::RQFS

The `QuD::RQFS` class is just an abstract base class for relational query feature structures as perl objects. As a *package*, it defines a number of subclasses each of which represent some component of a standard SQL query. Only *map* queries are currently supported – map queries have the form:

$$\pi_{i_1,\ldots,i_k}(\sigma_P(R_1 \otimes \cdots \otimes R_n))$$

for indices $i_1, \ldots, i_k$, a selectional formula P, and extensional database relations $R_1, \ldots, R_n$.

[8] A "closure" is a construct common in functional programming languages such as LISP (Steele, Jr. 1990). Perl closures are `CODE` references which use lexically scoped variables to provide the behavior of persistent and encapsulated objects.

`QuD::RQFS` objects provide a method `to_sql()` which produces for each substructure a well-formed query of the database query language SQL corresponding to the query encoded in that substructure. Users are allowed to insert literal SQL strings anywhere in the structure, overriding the default SQL-generation process.[9]

The lack of support for query types such as union, difference, *etc.* could be remedied by the addition of more classes with the corresponding `to_sql()` methods.

4.2.3.2 QuD::RQFS::Tied

`QuD::RQFS::Tied` provides an interface to `QuD::RQFS` objects as tied perl hashes. It inherits from `QuD::RQFS` and from `QuD::FStructure::Tied`.

[9]Of course, by allowing the user such freedom, I cannot guarantee well-formedness of the resulting query – the description of generated SQL queries as "well-formed" above refers only to the queries generated by "pure" RQFS objects.

Chapter 5

Discussion

In this chapter, I present a discussion and evaluation of the results from Chapters 3 and 4 in light of the considerations from Chapter 1. Section 5.1 provides an overview of the results from Chapter 3. Section 5.2 presents an evaluation of the extent to which the stated goal of data-abstraction is achieved by relational query feature structures as I have defined them, while Section 5.3 presents an evaluation of the linguistic relevance of relational query feature structures. In Section 5.4, I speculate about possible improvements of or extensions to the relational query feature structure formalism. Section 5.5 contains some concluding remarks.

5.1 Summary of Results

In Chapter 3, a definitinon was presented of *Relational Query Feature Structures* (RQFSs) as a subcollection of the *Typed Feature Structures* defined by Carpenter (1992).

Focus was drawn to a subcollection of RQFSs which I termed "Interpretable Relational Query Feature Structures" (IRQFSs), delimiting the boundaries of this subcollection by means of common mechanisms. Constraints on feature-values were captured by means of an *Appropriateness Specification*, while constraints on self-reference were captured in terms of Carpenter's definition of *Acyclic Feature Structures*.

The hierarchy for which RQFSs were defined was given minimally by means of the *covers* relation, and the appropriateness specification by means of what I called an *immediate appropriateness specification*. While these devices were not strictly necessary, their use provided a succint notation for capturing the full generality of the inheritance hierarchy and the constraints associated with it.

I put Carpenter's notion of an *Extensionality Specification* to a use for which it does not appear originally to have been intended: enforcing a distinction between "abstract" and "concrete" types. While not strictly necessary, such a distinction was useful in that it allowed me to make use of the inheritance of feature-value constraints implied by the upward-closure clause for appropriateness specifications in order to minimize the number of features required for the definition of RQFSs, most accurately capturing the commonalities between "concrete" types (such as $Union$ and $Product$, or $EQComp$ and $LTComp$) in terms of inheritance from a common "abstract" type (such as $QueryExpr$ or $Comparison$).

For many concrete subtypes in the hierarchy, I showed that IRQFS-substructures bearing these subtypes can be construed as the "encodings" of a certain collection of atomic objects, and that these atomic objects are recoverable from their IRQFS-substructure encodings – e.g. a substructure bearing a concrete subtype of $natnum$ encodes exactly one natural number, while a substructure bearing a concrete subtype of $list(char)$ encodes exactly one finite character string.

A construction was provided which partially mapped IRQFSs themselves to finite flat relations, for an aribitrary naming relational database. IRQFSs with a defined interpretation for a given database DB were termed "Database-Interpretable Relational Query Feature Structures" (DBIRQFSs). Database-interpretable RQFSs were shown to be encodings for nothing more or less than those relations achievable by application of the relational algebraic operations.

A partial implementation of RQFSs as a class of objects in the programming language Perl (Wall et al. 1996) was described in Chapter 4. Both type hierarchies and appropriateness specifications were implemented in terms of minimal declarations, and many of the formal constraints on these structures were loosened. Atomic objects were encoded directly as native Perl scalar values, and the mapping of RQFS objects to database relations was achieved by means of translation to the common database query language SQL.

The theoretical result is perhaps unsurprising, given that the relational algebra, construed as a language, is context-free, and that feature structures, with their formal definition strongly resembling that of deterministic finite-state automata, are capable of representing the syntactic structure of any context-free language. Nonetheless, the definitions I have presented provide a constructive proof not only of the possibilty of encoding syntactic structure, but also of semantic equivalence, for which the implementation serves to provide empirical support.

5.2 Data Abstraction

In Section 1.3, I stated that an *abstract* query formalism for relational databases was a primary goal of this work. In this section, I attempt to answer the question of whether this goal has been fully attained, and if not, then to what extent I can consider it to have been accomplished. The abstractness criterion can be analyzed in terms of the following three factors:

1. **Database Independence**
   The query formalism should not directly depened for its functionality on the structure of the database which it is used to query. This criterion is discussed in Section 5.2.1.

2. **Query-Detail Independence**
   The query formalism should allow abstraction from arbitrary details of the database-query. This criterion is discussed in Section 5.2.2.

3. **Data-Structure Independence**
   The query formalism should allow abstraction from the manner in which the queried database structures its data, and ideally from the constraints inherent in the relational framework itself. This criterion is discussed in Section 5.2.3.

5.2.1 Database Independence

> *The query formalism should not directly depened for its functionality on the structure of the database which it is used to query.*

This goal has been accomplished to the extent that it is possible for any database query formalism to be truly "independent" of the database queried. The RQFS query formalism relies on the existence and structure of an external database for its very definition, as well as for its interpretation. Nonetheless, the goal of database-abstraction should be viewed as accomplished by the RQFS formalism, since no particulars of any given database or database management system are "hard-wired" into it. Rather, since the definition and interpretation of RQFSs as queries on a relational database are defined in terms of an *arbitrary* naming relational database, the formalism relegates its database-dependent components to what amounts to a formal parameter, therefore RQFSs can and should be considered "independent" of the specifics of any single database or database management system.

The RQFS implementation, relying as it does on the common database language SQL, is similarly well abstracted from the particulars of any single database implementation. Further, the Perl `DBI` module (Bunce 2001) allows SQL strings such as those generated by RQFS objects to be passed directly to many currently popular database management systems.

5.2.2 Query-Detail Independence

> *The query formalism should allow abstraction from arbitrary details of the database-query.*

Again, as a query formalism, RQFSs cannot be entirely independent of the details of a given query – were this to be so, their utility as a query formalism would be somewhat dubious. Indeed, as a relational-complete query formalism, RQFSs must necessarily allow the expression of all possible details of a given query.

The formalism as I have defined it, however, does not directly provide any mechanism for the abstraction from query-details: each database interpretable RQFS must necessarily provide all relevant details of the database query it represents.

Nonetheless, the RQFS formalism can be construed as at least partially fulfilling the criterion of query-detail independence. Feature-structure unification and the assignment of path-values provide mechanisms by means of which a library of of underspecified feature structures might be combined in the course of a computation in order to produce a database-interpretable RQFS – a fully-fledged relational database query. In this much, at least, RQFSs constitute a significant increase in abstraction over query-detail from that provided by relational algebra or its direct descendants.

In particular, the ability to abstract in this manner over relation names and tuple-components provides a level of abstraction which is missing from the ubiquitous relational database query language SQL. The implemented RQFS objects additionally allow users to specify literal SQL strings anywhere in the structure, providing greater flexibility of expression than the purely formal constructs from Chapter 3.

5.2.3 Data-Structure Independence

> *The query formalism should allow abstraction from the manner in which the queried database structures its data, and ideally from the constraints inherent in the relational model itself.*

Unfortunately, this criterion cannot be considered to have been adequately accomplished by RQFSs as I have defined them. On the one hand, RQFSs must make some reference to a database's logical structure if they are to function as a query mechanism. On the other hand, RQFSs have been designed from the start to parallel the operations of relational algebra, and thus inherit all the limitations of the relational algebra, and of the relational database model itself. I have not provided any mechanism here to abstract over these constraints, either in the formal definitions from Chapter 3 or in the Perl library described in Chapter 4.

5.3 Linguistic Salience

In Section 1.3, I stated that the query formalism developed in this work should be *linguistically salient*. The extent to which this goal has been attained may be analyzed in terms of the following factors:

1. **System Input**
   The query formalism should accept as its input data structures suited

for use in natural-language computations. This factor is discussed in Section 5.3.1.

2. **System Output**
The query formalism should produce as its output data structures suited for use in natural-language computations. This factor is discussed in Section 5.3.2.

5.3.1 System Input

The query formalism should accept as its input data structures suited for use in natural-language computations.

The definition of relational query feature structures takes as a parameter an arbitrary relational database. In addition, the interpretation of RQFSs takes as its input an interpretable relational query feature structure. Answering the question of whether the goal of linguistically relevant system input has been attained can then be divided into the questions of whether, on the one hand, relational databases are suited for the storage of linguistic data, and on the other hand, whether interpretable relational query feature structures constitute a linguistically relevant data structure in themselves.

The answer to both of the above questions, for the system as it stands, must be "no". Relational databases are not particularly well-suited to the storage of linguistic data, which is often characterized by data structures nested to arbitrary depths, which as has been noted cannot be encoded directly in the "flat" relations of a relational database, as mentioned by Véronis & Ide (1992). Although techniques exist for simulating nested data structures within the relational framework (Codd 1970, 1972*a*, Ullman 1982, 1999), the ability – desirable from the standpoint of natural-language processing applications – to recover such normalized data into structures nested to an arbitrary depth amounts to a transitive closure operation, which cannot be directly computed by the relational algebra (Ullman 1982, 1999) and thus cannot be directly computed by RQFSs.[1]

[1] It should be noted that for any fixed maximum depth of nesting n, the relational framework – and by extension my RQFSs – is fully capable of storing and extracting an $(n+k)$-ary flat relation representing that data, where the natural number k depends on the specifics of the data structure in question.

Despite the fact that feature structures in general have often been used for natural-language processing, relational query feature structures are not particularly well-suited to the task of acting as linguistically relevant structures. This follows directly from their design as a query formalism for relational databases and the inappropriateness of the relational model for representing linguistic data.

These shortcomings aside, RQFSs are eminently well-suited to query any natural-language data which can be represented within the flat relational model, which as noted in Section 1.3 was chosen not so much for its linguistic appropriateness, but rather for its ubiquity.

5.3.2 System Output

> *The query formalism should produce as its output data structures suited for use in natural-language computations.*

The chief consideration in the evaluation of this factor is, as in Section 5.3.1 above, the suitedness of the relational framework in particular to linguistically relevant data-structures. As discussed above, the structural constraint of flat, finite relations combined with the inability to directly compute a relation's transitive closure leads to the inevitable conclusion that the RQFS query formalism is not particularly well-suited to many linguistically relevant data structures, since these constraints apply not only to the underlying relational database, but also to the system's output.

5.4 Improvements

In Sections 5.2 and 5.3, I identified several shortcomings of relational query feature structures as an abstract and linguistically useful query mechanism. In this section, I outline the structure of an expanded query system using RQFSs as a low-level database interface module, arguing that such an expanded system would be able to compensate for the shortcomings of RQFSs as they stand. Section 5.4.1 deals with the notion of a "query template", while Section 5.4.2 describes a more systematic approach of "query descriptions".

5.4.1 Query Templates

By a *Query Template*, I simply mean an underspecified relational query substructure. Such structures can be seamlessly integrated into any feature structure whose hierarchy, appropriateness, and extensionality specifications are supersets of those defined in Section 3.2. This would allow linguistically relevant feature structures – such as representations of lexical entries or of the various stages of a parsing process – to be directly associated with a partially specified relational query feature structure.

The mechanisms of feature-structure unification and path-value assignment, as mentioned in Section 5.2.2 allow the construction of a database-interpretable relational query feature structure from just such partially specified substructures. The embedding application would have to be designed such that upon successful completion of a given computation, e.g. lexical identification or parse acceptance, the partial substructures associated with the application-internal data had been unified (the "logical" approach) or combined by means of path-value assignment (the "imperative" approach) or translation (the "functional" approach) to produce a full-fledged database-interpretable relational query feature structure, which could then be evaluated with respect to the application's database.

The use of such query templates in lexical analysis and parsing applications can be likened to well-known and established techniques from compiler construction (Aho & Ullman 1972, Levine, Mason & Brown 1990), which in the case of lexical analyzers such as those generated by `lex` associate with each token-identification rule[2] a function which accepts as a parameter the literal text of the token and returns an application-internal identifier. In the case of parsing applications, the incremental generation of RQFSs by means of query templates bears a strong resemblance to code-generation functions typically associated with each rule of a context-free grammar by parsers such as those generated by `yacc`. Such functions take as a parameter a list of values of the rule's right-hand-side symbols, and return a value for the rule's left-hand-side symbol.

Query templates provide a solution to the shortcomings of RQFSs in the area of query-detail abstraction discussed in Section 5.2.2. A query template system based directly on RQFSs would provide a more principled approach

[2] "Token identification" rules are also known as "lexical insertion" rules.

to query generation than existing systems based directly on SQL, such as UGH (Jurish 2000*b*).

Additionally, the possibilty of integrating query templates into application-specific feature structures would allow application developers much more freedom in the manner in which their elementary query data is structured, which goes some way toward compensating for the shortcomings of RQFSs as linguistically salient input data structures, as discussed in Section 5.3.1.

5.4.2 Query Descriptions

A more powerful mechanism than that of query templates is needed to overcome the shortcomings of RQFSs in the areas of data-structure independence and linguistically salient output, as discussed in Sections 5.2.3 and 5.3.2, respectively.

Required are two separate rule-sets, the first of which – the "query input schema" – would be responsible for constructing (series of) database-interpretable relational query feature structures from arbitrary feature structures. The other rule-set – the "query output schema" – would be responsible for (iteratively) processing the flat finite relations constituting the output of a relational database query, transparently re-creating nested structures from their flat encodings. Such schemata would themselves necessarily be tailored to the specifics of a particular database, but would effectively encapsulate database-specific information so as to allow a great deal of data-structure independence.

Feature-structure description logics (Kartunnen 1984, Aït-Kaci 1984, Kasper & Rounds 1986, Rounds & Kasper 1986, Aït-Kaci 1986, Johnson 1988, Carpenter 1992, Smolka 1992, Rounds 1996) might provide a vehicle for the expression of query schemata in terms of arbitrary feature structures. Indeed, the examples of query-template applications in the preceeding section (unification-based parsing and tokenization) can be reduced to applications of just such feature-structure description logics.[3]

[3] The prefix "QuD" on the module-names of the implemented library was meant to stand for "Query Descriptions" – I had hoped to be able to implement a more abstract query mechanism than that provided by pure RQFSs, but due to time constraints, an implementation of "query descriptions" such as described here was not possible.

Schema-evaluation strategies such as those used by DATALOG (Ceri, Gottlob & Tanca 1990, Ullman 1999) might provide a mechanism by means of which two of the critical limitations inherent in the relational framework – the imposition of a flat data structure and the inability to directly compute a relation's transitive closure – might be overcome.

5.5 Conclusions

My project to provide an abstract and linguistically useful query system for relational databases has met with at least partial success. As shown in Chapter 3, relational query feature structures as I have defined them are equivalent to relational algebra. The expressive power of the RQFS query formalism has therefore been established in terms of a well-known query formalism.

Limitations inherent in the relational algebra and the relational data model itself – such as the inability to store or retrieve data nested to an arbitrary depth with a single query – thus extend to the RQFS formalism and its implementation as described in Chapter 4, preventing me from fully attaining the stated goals of linguistic utility and abstraction from data-structure. Despite these limitations, the RQFS query system provides a structured and fully parameterized alternative to relational algebra and its direct descendants.

Existing mechanisms from the general theory of feature-structures – such as feature-structure unification and feature-structure description logics – could in cooperation with and application to relational query feature structures provide a more flexible approach to the problem of query generation. Existing mechanisms from database theory – such as the predicative query paradigm employed by DATALOG – might be adapted to compensate for the limitations inherent in the relational algebra.

Overall, relational query feature structures as I have defined and implemented them constitute a small but significant first step towards the abstract and linguistically useful systematic approach to the formulation of relational database queries which I had sought.

Appendix A

Zusammenfassung

In dieser Arbeit definiere ich eine Klasse von Merkmalstrukturen, die ich *Relational Query Feature Structures* nenne – Relationale Abfrage-Merkmalstrukturen, abgekürzt "RQFS", die in der Lage sind, Abfragen auf *Relationale Datenbanken* auszudrücken, und beschreibe eine Implementierung solcher Merkmalstrukturen in der Programmiersprache Perl. Ziel dieses Projektes ist es, einen modularen und Datenbank-unabhängigen Abfragemechanismus zu entwickeln, der für das Einsetzen insbesondere in linguistischen Anwendungen geeignet ist.

Kapitel 2 befasst sich mit den grundlegenden Theorien, auf denen die spätere Definition von Relationalen Abfrage-Merkmalstrukturen basiert. Die Theorie relationaler Datenbanken und des Abfrageformalismus *relationaler Algebra* werden in Abschnitten 2.2 und 2.3 jeweils präsentiert, basierend auf Arbeiten von Codd (1970, 1972*a*, 1972*b*) und Ullman (1982, 1999). Abschnitt 2.4 beschreibt formal das Konzept einer *Typenhierarchie*[1] nach Davey & Priestley (1990) als eine besondere Art partiell geordneter Menge. Abschnitt 2.5 enthält formale Definitionen bezüglich der Klasse *typisierter Merkmalstrukturen*, basierend hauptsächlich auf Carpenter (1992).

Gegenstand von Kapitel 3 ist die Definition und Beschreibung von relationalen Abfrage-Merkmalstrukturen an sich. Nach Festlegung einiger Konventionen in Abschnitt 3.1 definiere ich zuerst die syntaktischen Eigenschaften relationaler Abfrage-Merkmalstrukturen in Abschnitt 3.2. In Abschnitt 3.3

[1] Typenhierarchien werden auch *Vererbungshierarchien* genannt.

belege ich den Begriff der semantischen *Interpretierung* einer relationalen Abfrage-Merkmalstruktur bezüglich einer beliebigen relationalen Datenbank. In Abschnitt 3.4 zeige ich, dass relationale Abfrage-Merkmalstrukturen in ihrer Ausdrucksmächtigkeit äquivalent zu relationaler Algebra sind.

In Kapitel 4 beschreibe ich eine Implementierung von relationalen Abfrage-Merkmalstrukturen in der Programmiersprache Perl (Wall et al. 1996). Die Quelltexte und ausführliche Dokumentation zu der Implementierung befinden sich auf einer zusätzlichen CD (anbei).

Kapitel 5 enthält eine Diskussion der Resultate aus Kapiteln 3 und 4, die in Abschnitt 5.1 zusammengefasst sind. In Abschnitt 5.2 analysiere ich die Resultate bezüglich der oben genannten Ziele der Modularität und Datenbankunabhängigkeit. Abschnitt 5.3 enthält eine Evaluierung der linguistischen Relevanz relationaler Abfrage-Merkmalstrukturen. Leider kann ich nicht alle Ziele des Projekts als vollkommen erreicht betrachten, und skizziere deswegen kurz in Abschnitt 5.4 zwei mögliche Erweiterungen des relationalen Abfrage-Merkmalstruktur-Formalismus: *Query Templates* – Abfrageschablonen – auf der einen, und *Query Descriptions* – Abfragebeschreibungen – auf der anderen Seite.

Abschnitt 5.5 fasst die Arbeit noch einmal kurz zusammen, und enthält einige allgemeine Bemerkungen zu den Gesamtergebnissen, die ich als partiellen Erfolg bezeichne.

Bibliography

Aho, A. V. & Ullman, J. D. (1972), *The Theory of Parsing, Translation and Compiling, Volume 1: Parsing*, Prentice-Hall, Englewood Cliffs, N.J.

Aït-Kaci, H. (1984), A lattice theoretic approach to computation based on a calculus of partially ordered type structures, PhD thesis, University of PA.

Aït-Kaci, H. (1986), 'An algebraic semantics approach to the effective resolution of type equations', *Theoretical Computer Science* **45**, 293–351.

Aït-Kaci, H., Boyer, R., Lincoln, P. & Nasr, R. (1989), 'Efficient implementation of lattice operations', *ACM Transactions on Programming Languages and Systems, TOPLAS* **11**(1), 115–146.
*http://www.csl.sri.com/~lincoln/papers/toplas89.ps.gz

ANSI X3.135-1992 (1992), *Database Language – SQL*, ANSI, New York.

Baayen, R. H., Piepenbrock, R. & Gulikers, L. (1995), CELEX2. ISBN: 1-58563-085-3.

Beyer, S. (2001), *Bit::Vector package documentation.*
*http://www.cpan.org/authors/id/STBEY/Bit-Vector-6.1.tar.gz

Bod, R., Bonnema, R. & Scha, R. (1996), A data-oriented approach to semantic interpretation, *in* 'Proceedings Workshop on Corpus-Oriented Semantic Analysis, ECAI-96', Budapest.
*http://earth.let.uva.nl/~rens/ecai.ps

Bod, R. & Kaplan, R. (1998), A probabilistic corpus-driven model for lexical-functional analysis, *in* C. Boitet & P. Whitelock, eds, 'Proceedings of

the Thirty-Sixth Annual Meeting of the Association for Computational Linguistics and Seventeenth International Conference on Computational Linguistics', Morgan Kaufmann Publishers, San Francisco, California.
*http://earth.let.uva.nl/~rens/lfgdop.ps

Bod, R. & Scha, R. (1996), 'Data-oriented language processing: an overview'.
*http://www.wins.uva.nl/research/illc/Publications/ResearchReports-/LP-1996-13.text.ps.gz

Bunce, T. (2001), *DBI package documentation.*
*http://www.cpan.org/authors/id/TIMB/DBI-1.20.tar.gz

Carpenter, B. (1992), *The Logic of Typed Feature Structures*, Vol. 32 of *Cambridge Tracts in Theoretical Computer Science*, Cambridge University Press, Cambridge.

CELEX Project (1988–1995), *The* CELEX *Lexical Database*, Centre for Lexical Information, Max Planck Institute for Psycholinguistics, Nijmegen, the Netherlands.
*http://www.kun.nl/celex

Ceri, S., Gottlob, G. & Tanca, L. (1990), *Logic Programming and Databases*, Surveys in Computer Science, Springer-Verlag, Berlin.

Chamas, J. (2001), *Tie::Cache package documentation.*
*http://www.cpan.org/authors/id/C/CH/CHAMAS/Tie-Cache-0.15.tar.gz

Charniak, E. (1996), Tree-bank grammars, *in* 'Proceedings of AAAI/IAAI'96, Vol. 2', Portland, Oregon, pp. 1031–1036.
*ftp://ftp.cs.brown.edu/pub/techreports/96/cs96-02.ps.Z

Chen, J. & Vijay-Shanker, K. (2000), Automated extraction of TAGs from the Penn treebank, *in* 'Proceedings of IWPT6'.
*http://www.eecis.udel.edu/~jchen/autoextract.ps

Codd, E. F. (1970), 'A relational model for large shared databanks', *Communications of the ACM* **13**(6), 377–387.

Codd, E. F. (1972*a*), Further normalization of the data base relational model, *in* R. Rustin, ed., 'Data Base Systems', Prentice-Hall, Englewood Cliffs, New Jersey, pp. 33–64.

Codd, E. F. (1972*b*), Relational completeness of data base sublanguages, *in* R. Rustin, ed., 'Data Base Systems', Prentice-Hall, Englewoord Cliffs, New Jersey, pp. 65–98.

Davey, B. A. & Priestley, H. A. (1990), *Introduction to Lattices and Order*, Cambridge University Press, Cambridge.

Gazdar, G., Pullum, G. K. & Sag, I. A. (1985), *Generalized Phrase Structure Grammar*, Blackwell, Oxford, England.

Hietaniemi, J. (2001), *Set::Scalar package documentation.*
*http://www.cpan.org/authors/id/JHI/Set-Scalar-1.12.tar.gz

Ide, N. & Véronis, J. (1994), MULTEXT: multilingual text tools and corpora, *in* 'Proceedings of the 15th. International Conference on Computational Linguistics (COLING 94)', Vol. I, Kyoto, Japan, pp. 588–592.
*http://www.cs.vassar.edu/~ide/papers/MULTEXT.ps

ISO/IEC 9075:1992 (1992), *Database Language – SQL*, ISO.

Johnson, M. (1988), *Attribute value logic and the theory of grammar*, Vol. 16 of *CSLI lecture notes*, CSLI, Stanford, California.

Jurish, B. (2000*a*), *Systemkonstruktion Seminar, interne Dokumentation: Lexikon*, Universität Potsdam, Institut für Linguistik, Potsdam.
*http://www.ling.uni-potsdam.de/~moocow/classwork/sk/lexdoc-html/

Jurish, B. (2000*b*), *UGH Documentation, Version 1.6.4*, Universität Potsdam, Institut für Linguistik, Potsdam.
*http://www.ling.uni-potsdam.de/~moocow/projects/ugh

Kaplan, R. & Bresnan, J. (1985), Lexical functional grammar: a formal system for grammatical representation, *in* J. Bresnan, ed., 'The Mental Representation of Grammatical Relations', MIT Press, Cambridge, Massachusetts, chapter 4, pp. 173–281.

Kaplan, R. M. (1989), 'The formal architecture of lexical functional grammar', *Journal of Information Scienece and Engineering* **5**, 305–322.
*http://www-lfg.stanford.edu/lfg/lfg-introductions/formal-architecture.ps

Kartunnen, L. (1984), Features and values, *in* 'Proceedings of the Tenth International Conference on Computational Linguistics', Stanford University, Stanford, California, pp. 28–33.

Kasper, R. T. & Rounds, W. C. (1986), A logical semantics for feature structures, *in* 'Proceedings of the 24th Annual Conference of the Association for Computational Linguistics', New York, pp. 235–242.

Kay, M. (1984), Functional unification grammar: a formalism for machine translation, *in* 'Proceedings of the Tenth International Conference on Computational Linguistics', Stanford University, Stanford, California, pp. 75–78.

Levine, J. R., Mason, T. & Brown, D. (1990), *lex & yacc*, O'Reilly & Associates, Köln.

MacGregor, R. M. (1988), A deductive pattern matcher, *in* 'Proceedings of the Seventh National Conference on Artificial Intelligence, (AAAI 88)', pp. 403–408.

Manfredi, R. (2001), *Storable package documentation.*
*http://www.cpan.org/authors/id/RAM/Storable-1.0.13.tar.gz

Marcus, M., Santorini, B. & Marcinkiewicz, M. (1993), 'Building a large annotated corpus of english: the penn treebank', *Computational Linguistics* **19**(2), 313–330.
*ftp://ftp.ciril.fr/grace/pub1/upenn.treebank/overview.ps.gz

Martin, P. (1995), Using the WordNet concept catalog and a relation hierarchy for knowledge acquisition, *in* 'Proceedings Peirce'95 – 4th Peirce Workshop, California, August 18, 1995'.
*http://www-sop.inria.fr/acacia/Publications/1995/peirce95phm.ps.Z

Miller, G. A., Beckwith, R., Fellbaum, C., Gross, D., Miller, K. & Tengi, R. (1990), *5 papers on WordNet*, Cognitive Science Laboratory, Princeton University, Princeton, New Jersey.
*ftp://ftp.cogsci.princeton.edu/pub/wordnet/5papers.ps

Montague, R. (1970), The proper treatment of quantification in ordinary english, *in* J. Hintikka, J. Moravcsik & P. Suppes, eds, 'Approaches to Natural Language: Proceedings of the 1970 Stanford Workshop on Grammar and Semantics', D. Reidel, Dordrecht, pp. 221–42.

MULTEXT Project (1996), MULTEXT*: multilingual text tools and corpora*, Aix-en-Provence, France. Jean Véronis (coordinator).
*http://www.lpl.univ-aix.fr/projects/multext

Pereira, F. C. N. & Shieber, S. M. (1984), The semantics of grammar formalisms seen as computer languages, *in* 'Proceedings of the Tenth International Conference on Computational Linguistics', Stanford University, Stanford, California, pp. 123–129.

Pollard, C. & Sag, I. A. (1987), *Information-Based Syntax and Semantics*, number 13 *in* 'CSLI Lecture Notes', CSLI, Stanford, California.

Rounds, W. C. (1996), Feature logics, *in* J. van Benthem & A. ter Meulen, eds, 'Handbook of Logic and Language', Elsevier, Amsterdam, pp. 475–533.
*ftp://ftp.eecs.umich.edu/people/rounds/feature-logic.ps.gz

Rounds, W. C. & Kasper, R. T. (1986), A complete logical calculus for record structures representing linguistic information, *in* 'Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science', Cambridge, Massachusetts, pp. 38–43.

Scott, S. & Matwin, S. (1998), Text classification using WordNet hypernyms, *in* S. Harabagiu, ed., 'Use of WordNet in Natural Language Processing Systems: Proceedings of the Conference', Association for Computational Linguistics, Somerset, New Jersey, pp. 38–44.
*http://www.ai.sri.com/~harabagi/coling-acl98/acl_work/sscot.ps.gz

Shieber, S. M. (1984), The design of a computer language for linguistic information, *in* 'Proceedings of the Tenth International Conference on Computational Linguistics', Stanford University, Stanford, California, pp. 362–366.

Smolka, G. (1992), 'Feature constraint logics for unification grammars', *Journal of Logic Programming* **12**, 51–87.

Steele, Jr., G. L. (1990), *Common* LISP*: the Language*, Digital Press.

Ullman, J. D. (1982), *Principles of Database Systems*, Computer Science Press, Rockville, Maryland.

Ullman, J. D. (1999), *Principles of Database and Knowledge-Base Systems, Volume I: Classical Database Systems*, Vol. 14 of *Principles of Computer Science Series*, Computer Science Press, Rockville, Maryland.

Véronis, J. & Ide, N. (1992), A feature-based model for lexical databases, *in* 'Proceedings of the 14th International Conference on Computational Linguistics – COLING'92', August 23-28 1992, Nantels, France.
*http://www.up.univ-mrs.fr/~veronis/pdf/1992coling.pdf

Wall, L., Christiansen, T. & Schwartz, R. L. (1996), *Programming Perl, Second Edition*, O'Reilly & Associates, Köln.

WordNet Project (1985–2001), *WordNet – a Lexical Database for English*, Cognitive Science Laboratory, Princeton University, Princeton, New Jersey.
*http://www.cogsci.princeton.edu/~wn

XTAG Research Group (2001), A lexicalized tree adjoining grammar for english, Technical Report IRCS-01-03, IRCS, University of Pennsylvania.

Zeller, A. & Snelting, G. (1997), 'Unified versioning through feature logic', *ACM Transactions on Software Engineering and Methodology* **6**(4), 398–441.
*ftp://ftp.ips.cs.tu-bs.de/pub/local/softech/papers/tr-96-01.ps.gz

This page is a figment of LaTeX's imagination.