Type Inference, Type Improvement, and Type Simplification in a Language with User-Defined Polymorphic Relational Operators

by

Lajos Pál Nagy

Master of Science
in Computer Science
Technical University of Budapest
2000

A dissertation submitted
to Florida Institute of Technology
in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy
in
Computer Science

Melbourne, Florida
May, 2007

We the undersigned committee
hereby approve the attached thesis

Type Inference, Type Improvement, and Type Simplification
in a Language with User-Defined Polymorphic Relational Operators

by
Lajos Pál Nagy

Ryan Stansifer, Ph.D.
Major Advisor
Associate Professor, Computer Sciences

Philip J. Bernhard, Ph.D.
Committee Member
Associate Professor, Computer Sciences

Philip K. Chan, Ph.D.
Committee Member
Associate Professor, Computer Sciences

Jewgeni H. Dshalalow, Dr.Sci.
Committee Member
Professor, Mathematics

William D. Shoaff, Ph.D.
Associate Professor and Head
Computer Sciences

Abstract

Type Inference, Type Improvement, and Type Simplification
in a Language with User-Defined Polymorphic Relational Operators

by

Lajos Pál Nagy

Major Advisor: Ryan Stansifer, Ph.D.

The overarching goal of the current thesis is to pave the road towards a comprehensive solution to the decades old problem of integrating databases and programming languages. For this purpose, we propose a record calculus as an extension of an ML-style functional programming language core. In particular, we describe: (1) a set of polymorphic record operations that are expressive enough to define the operators of the relational algebra; (2) a type system together with a type inference algorithm, based on the theory of qualified types, to correctly capture the types of said polymorphic record operations; (3) an algorithm for checking the consistency (satisfiability of predicates) of the inferred types; (4) an algorithm for improving and simplifying types; and (5) an outline of an approach to explaining type errors in the resulting type system in an informative way.

Table of Contents

Acknowledgment

First of all, I would like to thank my parents for their patience and understanding. I know how hard it must have been to let their only son go to a far away country, on the other side of the Atlantic. I thank my sisters, Emi, Éva, and Julcsi, for their encouragement and support.

I thank all the members of my doctoral committee. I thank Dr. Chan for pushing me harder and demanding focus and clarity. Special thanks goes to my major advisor, Dr. Stansifer whose knowledge, patience, professionalism, eye for detail, and last, but certainly not least, his constant encouragement really was indispensable in helping me to write this dissertation.

I would like to thank my friend, Attila, who always lent a sympathetic ear whenever I hit obstacles in my research (and that happened a lot).

I thank Dr. Imre Paulovits, who helped me to get to the United States and provided me with a lot of advice on how to conduct world-class research.

Finally, I thank Yoshiko, who believed in me all the way, and wanted me to succeed. (She is also pretty, by the way.)

Dedication

To My Grandparents

Chapter 1
Introduction

Databases as separate entities of information systems emerged in the 1960s when it became obvious that the common task of handling shared and persistent data is best dealt with a dedicated component, a Database Management System. Up until then, each application program did its own data management using the facilities provided by the file system. Though this separation of duties proved to be extremely beneficial in the long run, it nevertheless created its own set of problems—most of them stemming from the difficulties of interfacing programming languages with databases.

1.1 The Great Chasm

Programs that access data in a database are referred to as *database applications*. Traditionally, database applications are written in some high-level programming language (often called the *host* language in this context), and use a *data sub-language* (the *embedded* language) to query and update data in the database. (This description holds true even in the context of modern object-oriented languages, although various *persistence* and *object/relational mapping* solutions blur the language boundaries somewhat.) It has long been recognized that this dualism of languages (or 'impedance mismatch', as it has become infamously known) causes several problems:

1. differences between the primitive and complex data types of the host and the embedded languages require constant translation (mapping) between two different data representations (implementing this translation is often both laborious and error-prone);

2. there are quite different approaches to the optimization of programs as opposed to the optimization of database operations, with very little work existing on how to perform them jointly;

3. most databases, nowadays almost always relational, support, in the form of primitive operations, ‘set-at-a-time’ (bulk) processing of data, while most programming languages, especially for update operations, support only ‘tuple-at-a-time’ (iterative) processing;

4. the host language is disconnected from the syntax and semantics of the embedded language which prevents compile-time checking of database operations; and

5. when dealing with concurrent access to shared data, programming languages often assume cooperation between different entities (that are considered friendly, creating only inadvertent conflicts) while databases by default assume competition among different entities (that are considered hostile, creating deliberate conflicts) which leads to widely differing views on such issues as transaction management.

Since the late 1970s, there have been several attempts at addressing (or just formulating) the ‘impedance mismatch’ problem (see [Atkinson and Buneman, 1987] for and older, but quite broad survey, and [Cook and Ibrahim, 2005] for a more recent, albeit somewhat narrower one.)

Most, if not all, solutions address this problem of language dualism, they try to solve the problem by coming up with a single, unified language, a *Database Programming Language* (DBPL), that can be used for both computation and data manipulation [Bancilhon and Buneman, 1990; Date and Darwen, 1998].

We believe in the general correctness of the unified language approach to solving the 'impedance mismatch' problem, and consider the combination of functional programming and the relational model of data the most promising candidate for such a unified language. In accordance with this, we propose a record calculus (as an extension of an implicitly-typed lambda calculus with let-bound polymorphism), that is powerful enough to express the 'standard' relational algebra operators and also to permit the definition of novel ones. Although far from a full-fledged database programming language proposal, nevertheless, we hope that by exploring a particularly complex and interesting segment of the design space, we can contribute to the design of more ambitious programming languages.

Quick Note on Terminology

The basic building block of the relational model, the relation, is defined as a set of tuples. In database theory, a *tuple* (or labelled tuple) is a finite map from *attribute names* to *attribute values*. The programming languages community has different names for the same concepts: a *record* is a finite map from *field names* (or *labels*) to *values*. In this thesis we will mostly use programming language terminology, unless the topic is directly related to database theory. Also, slightly abusing standard usage, we will call operations on relations *relational operators*.

1.1.1 Rationale for Functional Programming

Functional programming languages are important members of a broader family of *declarative languages*. These languages are characterized by their relative closeness to mathematics both in their syntax and their semantics. Functional languages are based on *lambda calculus*, which makes formal reasoning about the properties of programs much easier than in the case of imperative languages. The history of functional programming also proved several language features extremely useful, each of which we were determined to preserve while designing our system:

1. *Static Type Checking and Type Inference* The ability to statically type check programs *without* extensive explicit type annotations [Cardelli, 1997] makes for very concise programs while retaining the advantages of static type checking. Coupled with side-effect free programming it is very often true about functional programs that "if it compiles, then it is correct."

2. *Higher-Order Functions* Functions are first-class citizens in a functional language so they can be passed around and manipulated just like ordinary values. This ability to define and use higher-order functions (functions that operate on functions) allows for powerful abstraction mechanisms that greatly enhance the expressive power of the language.

3. *Polymorphism (or Generic Programming)* In a polymorphic functional language, it is possible to define functions that operate on arbitrary types *while* retaining static type checking and type inference. Polymorphism facilitates the 'Once and Only Once' principle of software engineering that puts great value on avoiding duplicating functionality.

4. *Referential Transparency* In a pure [Sabry, 1998] functional language, functions behave like their counterparts in mathematics, that is, they denote a value and invoking a function with the same list of arguments always yields the same value. In other words, a language of pure functional programs is *referentially transparent* which allows developers to use powerful equational reasoning when thinking about programs. For example, in a referentially transparent language, the expression **let** $x = f(y)$ **in** $g(x) + g(x)$ is *always* equivalent in value *and* effect to the expression $g(f(y)) + g(f(y))$ which cannot be said of languages that permit functions with side-effects.

1.1.2 Rationale for the Relational Model

In the early 1970s when Codd first introduced the relational model of data [Codd, 1970], it was not entirely obvious that relational databases would be ubiquitous thirty years later. Several factors contributed to the eventual success of the relational model:

1. The relational model, as a formal system of logic, is equivalent in expressive power to first-order predicate logic (without function symbols) restricted to non-recursive Horn clauses [van Emde Boas-Lubsen and van Emde Boas, 1998]. In addition, any theory in the relational model is always guaranteed to be finite (*safe* in relational terminology.)

2. Complex, *ad hoc* queries can be formulated using only the operators of the relational algebra, which stands in sharp contrast with the more programmatic, 'pointer-chasing' style of querying in network and object-oriented databases.

3. Unlike first-order predicate logic (where resolution is exponential), relational algebra does

have an efficient evaluation algorithm (polynomial in the size of input relations).

4. Significant amount of research has gone into the optimization of relational queries both at compile-time (query re-writing) and execution-time (indexes, scheduling).

5. Transaction semantics can be clearly defined in the relational model, while it creates serious theoretical difficulties in other data models.

1.2 Problem Statement

The algebraic approach to defining relational queries (relational algebra) is purely functional by nature, thus it seems natural to use the functional paradigm combined with the relational model of data as the basis of a database programming language:

The goal of this thesis is to define polymorphic relational operators from a small set of primitives in a functional programming language with full compile-time type checking, type inference, polymorphism, and higher-order functions.

We emphasize full compile-time type checking (catching type errors at the definition site, *if* the particular definition can be proven to be erroneous) to set our proposal apart from systems (like [Buneman and Ohori, 1996] and [Makholm and Wells, 2005]) which do only partial compile-time type checking (catching more expensive type errors only at call sites, thus potentially accepting erroneous, albeit unused definitions).

Another important difference between previous systems and our proposal is that we set out to find a handful of primitive operations in terms of which most polymorphic relational operations could be defined, as opposed to trying to canonize in the language a fixed set of primitive rela-

tional operators (see [Buneman and Ohori, 1996] and [Van den Bussche and Waller, 1999]). We consider this later approach not satisfactory and decided against it. The syntax and semantics of the language, the typing rules, and the type inference algorithm (not to mention proofs of soundness and completeness in the system), all become unduly complex when primitive relational operators need to be treated as special cases every time. Even if one is willing to accept this additional complexity, it turns out that the list of polymorphic relational operators is practically open-ended, so every time a new relational operator is introduced that cannot be expressed using the built-in ones, one has to change the language definition together with the typing rules and the type checking algorithm (and every proof concerning the type system and the type inference algorithm would have to be re-done as well).

1.2.1 Main Challenges

The following are the main challenges one has to face when designing any language with the properties laid out in the problem statement:

Value-Dependent Types The result type of a polymorphic relational operator often depends on the *value* of its operands in a non-trivial way (for example, in the case of *natural join*, the *heading* of the result is the *union* of the headings of the operands—see Section A.5 for details). As a result, the type of *natural join*, along with most other polymorphic relational operators, cannot be expressed using only the standard Milner [Damas and Milner, 1982] type system that forms the basis of all modern functional programming languages.

NP-hard Type Checking Another problem is that static type checking in a language that supports polymorphic *natural join* has been proven [Ohori and Buneman, 1988] NP-complete.

Even type checking for the seemingly weaker system of symmetric record concatenation and field selection was proven to be NP-complete [Makholm and Wells, 2005]. This is a challenge because a programming language is of little practical use unless the type checking of moderately sized programs can be done in reasonable amount of times, that is, *fast*.

First-Class Attribute (Record Label) Sets The relational operator *project* has two operands: a set of attribute names and a relation. One faces a difficulty when trying to add *project* to a functional language, because it is not clear how to represent sets of attributes *both* at the value *and* the type level. Again, the problem with *project* is that the result *type* of the operation depends on the *value* of its operands. For example, the following relational expression using is ill-typed, regardless of the type of relation r: $\pi_{\{A,B\}}(\pi_{\{C,D\}}(r))$. Notice that the individual invocations of *project* are well-typed in the Milner type system, that is, the actual arguments are of the correct types: a set of attributes and a relation. Nevertheless, because of the actual *value* of the arguments, the expression *as a whole* is ill-typed.

1.3 Contributions

The main contributions of this dissertation are:

1. A carefully chosen collection of basic record operations, complete with formal semantics, that are expressive enough to define polymorphic relational operators.

2. A polymorphic type system (with a standard type inference algorithm) that is expressive enough to capture all the type constraints necessary to guarantee compile-time checking of all basic record operations.

3. An algorithm for checking the satisfiability of type constraints generated by the type inference algorithm.

4. Algorithms for improving and simplifying inferred types, where type improvement is guaranteed to find the principal satisfiable type.

5. An outline of an algorithm for explaining type errors in the face of polymorphic relational operators that can give rise to arbitrary systems of set constraints.

1.4 Outline of Dissertation

Chapter 2 summarizes related work in the area with emphasis on various record calculi and system with polymorphic relational operators. In Chapter 3, we describe the syntax and formal semantics of the core language and the basic record operations. In Chapter 4, we describe the type system of the language and define the types of the basic record operations. Chapter 4 also contains the description of a standard type inference algorithm for the type system. In Chapter 5, we develop an algorithm for checking the satisfiability of type constraints generated by the type inference algorithm. Chapter 6 formalizes the notions of type improvement and simplification, and also describes algorithms for performing type improvement and simplification. In Chapter 7, we analyze how to best explain type-errors in the system, and propose an outline of an algorithm for doing so. Chapter 8 concludes the dissertation by summarizing the results and pointing out possible future work. Appendix A gives an overview of the relational model and algebra, and all the proofs are collected in Appendix B.

Chapter 2
Related Work

In this section we look at related work in the main problem areas identified earlier. To place our contribution into perspective, we first have to establish a broader context for our efforts. Since attempts at addressing interfacing issues between databases and programming languages can be dated back to the early 1960s, it is no wonder that the work done in this somewhat loosely defined area has been immense. Therefore, we do not even pretend to give a comprehensive account of all previous work, but rather settle for a general overview that will hopefully help position the current thesis in relation to other major research directions in the area. After establishing the broader context we can move on to the narrower and more specific topic of describing related work in the areas of record calculi and polymorphic relational operators, where we can take the chance to elaborate on some technologies mentioned only in passing earlier. As a technical side note, we remark that since in the programming language community 'labeled tuples' are almost exclusively referred to as 'records' we will ourselves revert to this terminology when discussing previous work in this area.

2.1 Database Programming Languages

The idea behind Database Programming Languages (DBPLs) is to make the manipulation of persistent data (definition/storage/retrieval) an integral part of the programming language [Ban-

cilhon and Buneman, 1990]. (An early survey of the area can be found in [Atkinson and Buneman, 1987]. For a more recent problem statement and analysis, see [Cook and Ibrahim, 2005].) It is important to mention here that the term Database Programming Language is not generally accepted (or recognized) as the one that correctly describes solution attempts aiming at the integration of databases and programming languages, and, as a result, not all research efforts place themselves into the DBPL camp or realize that they belong there. As for the programing paradigm and the underlying persistence mechanism to use in designing an integrated language, there never has been any real consensus among the researchers and practitioners of the DBPL field. Nevertheless, it is possible to distinguish two chief research directions (with several variations for each) that dominated the field in the last two decades or so.

2.1.1 Orthogonal Persistence

One direction, often called the *orthogonal persistence* (as in "persistence is orthogonal to type") approach, deals with persistence from the point of view of programming languages [Atkinson et al., 1990]. This movement hopes to close the gap between databases and programming languages by removing the need for a database management system (often implicitly understood to be a relational database management system) as a stand-alone system component through programming language extensions for persistence. Its advocates usually emphasize the primacy, or the very least, the needs of application development over that of database design. In an ideal language with orthogonal persistence, volatile and persistent values are indistinguishable by their types, created and used by the application in the exact same way, and the actual persistence mechanism (reading data from and writing data to secondary storage) is completely

transparent from the point of view of the programmer. Combining orthogonal persistence with Object-Oriented programing lead to the emergence of Object Database Management Systems (ODBMSs) [Dittrich, 1991]. ODBMSs support the storage and retrieval of objects that encapsulate both data and its operations but usually are tied to a single object-oriented programming language, commonly C++ [Bartels and Robie, 1992]. Research in the ODBMS field flourished in the early 90s but interest began to wane as prototypical systems refused to measure up to relational systems and the long sought after theoretical foundations did not materialize [Kim, 1991]. The Enterprise Java Bean (EJB) technology [Ran et al., 2001], although larger in scope, can also be categorized as an attempt to add orthogonal persistence to the Java programming language, showing striking similarities to the ODBMS movement both in its general approach and its general failure to meet expectations [Johnson, 2004].

Orthogonal persistence, even after decades of research, still suffers from several unsolved, or inadequately addressed, issues that prevent it from replacing relational technology, as originally hoped. There are several reasons for this failure to meet expectations:

- lack of a formal mathematical foundation (comparable to the relational model of Codd [Codd, 1970]) for the description and manipulation of persistent data;

- lack of, or poor support for, a simple but expressive query language for writing ad hoc queries, if ad hoc queries are supported at all;

- inability, or difficulty, of sharing of data between applications, due to fact that persistent data is often tightly coupled with a particular programming language and paradigm;

- data manipulation is chiefly performed in a *navigational* (‘pointer chasing’) rather than a *declarative* (for example, relational algebra) manner, despite the fact that navigational databases (network and hierarchical) had been proven to be inferior, both in theory and practice, to declarative (relational) databases a long time ago [Date and Codd, 1975];

- difficulty in handling bulk data, or handling it efficiently, a fact which is often due to the ‘tuple-at-a-time’ processing nature of general purpose programming languages;

- theoretical and practical difficulty in utilizing both compile-time and run-time query optimization techniques which are now commonplace in relational databases (and which are proved to be indispensable in achieving acceptable performance); and finally

- lack of, or limited support (let alone formal foundations) for features like multiple users, transaction isolation, integrity constraints, distribution, security, or views, all of which features are now taken for granted in relational databases.

2.1.2 The Relational Approach

Efforts that fall under the umbrella of the *relational approach* accept the relational model and its embodiment, the relational database management system (RDBMS), as the general foundation for handling persistent data. As opposed to orthogonal persistence, languages belonging to the relational approach insist that only certain data types and run-time constructs, namely, relations, can be made persistent. Thus, the emphasis is on developing a smooth interaction between the programming language and the underlying RDBMS, both in terms of run-time performance and

in terms of harmonized type systems. In essence, the relational approach is just the next logical step from the traditional embedded data sub-language approach that has been in use from the ’70s, and which is still the most popular way of accessing data in a database (despite its well-known shortcomings). Proposals in the area tend to extend mainstream languages with relations and, occasionally, relational operators, like in Pascal/R [Schmidt, 1977], Modula/R [Reimer, 1984], or Machiavelli (based on ML) [Buneman and Ohori, 1996].

Adding support for records and relational algebra is often impossible without introducing new language constructs and modifying the target language’s type system. Some languages are more amenable to extension than others. Typically, languages with some level of support for records and record operations handle the extension with more ease (Pascal) than those without (Java). Recently, the language C# has been the target of such an extension effort under the name of Language INtegrated Query (LINQ) [Torgersen, 2006], a remarkable effort, considering that C# is an object-oriented language with no direct support for records or record operations. The functional language Haskell has such an expressive type system that it was possible to define a relational extension as a language library in the project HaskellDB [Leijen and Meijer, 1999], a comprehensive, type-safe, database access library, similar in spirit to LINQ (of which it can be considered a predecessor). Although more of an exercise in pushing Haskell’s type classes to the limit, *strongly-typed heterogeneous collections* (the HList library) [Kiselyov et al., 2004] proved to be capable of expressing all practically conceivable polymorphic record and relational operations in a statically type-safe manner.

There is one language (or, more precisely, a list of language prescriptions and proscriptions) proposed by Date and Darwen in the *Third Manifesto* [Date and Darwen, 1998] that deserves

extra attention because it was designed from the ground up to alleviate some of the perceived problems of database programming languages plagued by the 'SQL Legacy', that is, the general shortcomings of SQL both as a language in itself and also as an implementation of relational algebra [Date, 1990; Date and Darwen, 1992b, 1995; Date et al., 1998]. The proposal describes a strongly-typed relational algebra language, correcting many of SQL's mistakes on the way, where attributes in relations can be of arbitrarily complex, including user-defined, types. It also emphasizes the importance of a unified language, that is, supporting general computation and database access in a single language. There exist several implementations, including a full-fledged commercial one, that are based on the principles put forward in the Third Manifesto, a rarity in the realm of DBPLs.

2.2 Deductive Databases

The field of Deductive Databases (DDB) tries to combine logic programming, like Prolog, with relational technology [Minker, 1997]. Strictly speaking, a Deductive Database can be categorized as a Database Programming Language, but the area has such markedly different points of interest and body of research that it is best discussed in its own section. Reiter [Reiter, 1982] was the first to describe relational theory in terms of a logic system, pointing out that it can be described as a sub-theory of first-order predicate calculus, more precisely, as a system of function- and recursion-free Horn clauses. The realization that predicate logic, with certain restrictions, can be used for programming [Kowalski, 1974] lead to the development of logic programming languages and most prominently to the development of Prolog that is also based on Horn clauses. It seemed natural to combine the elegance and declarative nature of logic programming with the

robustness and high performance of relational databases. In the combined system, predicate logic would serve as the unifying 'lingua franca,' used simultaneously for application programming, expressing database queries, and specifying integrity constraints. After all, when seen from a logic programming point of view, a relational database is nothing but a logic theory, where relations correspond to predicates, and tuples belonging to a relation correspond to unit clauses of the predicates. Relational algebra queries can be directly translated to goal predicates, where the result of the relational query corresponds to those variable substitutions that make the goal predicate true under the logic theory represented by the database. This new approach was christened Deductive Database since it is based on deduction (logic inference) and database technology.

2.2.1 A Simple Deductive Database

To demonstrate the feasibility of using logic as a database programming language and to compare it with relational theory, we will present a simple database together with some sample queries in this section . The relational database consist of only two relations, *Parent* and *Female* (this example is taken from [Colomb, 1998]):

PARENT

Older	*Younger*
jane	lois
jim	jane

FEMALE

Name
jane
lois

The same database expressed as a logical theory with unit clauses (using Prolog syntax):

```
parent(jane,lois).
parent(jim,jane).
female(jane).
female(lois).
```

We can now define a view (in effect, a stored query) for the mother-child relation (using Tutorial D syntax [Date and Darwen, 1998]):

```
MOTHER := (PARENT rename {Older as Mother, Younger as Child) join (FEMALE rename {Name as Mother})
```

The same view defined using logic:

```
mother(Mother, Child) :-
    female(Mother), parent(Mother, Child).
```

However, DDBs are strictly more powerful than relational algebra, since they support recursion, as demonstrated by the following *ancestor* predicate:

```
ancestor(Older, Younger) :-
    parent(Older, Younger).
ancestor(Older, Younger) :-
    parent(Older, Intermediate),
    ancestor(Intermediate, Younger).
```

The *ancestor* predicate cannot be defined using standard relational algebra, and although the SQL standard has embraced recursive queries lately [Melton and Simon, 2001], it still remains something of an afterthought that does not nicely fit with the rest of the language. Contrast this with the fact that DDBs have always put great emphasis on maintaining certain desirable properties of database queries, like finiteness, and unequivocal semantics, which amounts to allowing only safe forms of recursion (ones that cannot lead to infinite results or undecidability).

2.2.2 Deductive Databases versus Functional Database Programming

Since logic programs and databases are indistinguishable in a DDB, which is a very appealing feature, most theoretical and practical results can be applied to both, which has made research in the area a very fruitful one. If we add to this the fact that DDB technology can directly leverage the decades of research done in various areas of logic, the question arises, why choose functional programming, as opposed to logic programming, as the basis for a unified language for database programming. Despite being theoretically elegant and producing immensely exciting results over the last three decades, the field of Deductive Databases still suffer from some serious deficiencies, both theoretical and practical, which has prevented, among other things, the development of a single commercially successful DDB as of this writing.

Functional programming is based on the lambda-calculus which, besides having a well-defined and straight-forward evaluation method, easily handles higher-order functions and has no difficulty whatsoever in accommodating relational operations. On the other hand, DDBs have long suffered from the difficulties created by any attempt to effectively combine the standard top-down, 'tuple-at-a-time' execution of logic programs needed for application code that might include functions and general recursion with the bottom-up, 'set-at-a-time' evaluation of relational queries, which emphasizes run-time optimization, finite results, and performance [Ramakrishnan and Sudarshan, 1991]. In practice, the two evaluation strategies interact poorly. An additional theoretical difficulty is that different evaluation strategies can result in different semantics, and the differences among these strategies can rarely be described as simply as 'eager versus lazy' in functional programming.

Logic programming also has some difficulty supporting higher-order predicates, a feature that proved to be so enormously useful in functional programming in the form of higher-order functions. The main reason for this is that logic programming is based on the well-understood first-order predicate calculus, and higher-order predicates would lead out of this system, thus preventing the use of its elegant and effective inference procedure (resolution). Despite this, some logic programming languages [Somogyi et al., 1994], Mercury in particular, do support higher-order predicates, but in a limited fashion. Although not centrally important, one must nevertheless point out the fact that higher-order functions fit harmoniously into the syntax of functional programs, which cannot be said of higher-order predicates in logic programs.

Finally, when attributes in relations must be referenced by position, which in fact is the case if they are represented as logic predicates, one loses an important element of data independence, that is, the ability to add attributes to or remove them from relations without breaking database programs that do not depend on the affected attributes. From the point of view of reducing programmer errors, attribute names make explicit which attribute one refers to, while in the positional setting of logic programming, it is easy to confuse attributes, even in a typeful logic programming language like Mercury (for example, although predicates with arity in the dozens are rare in logic programming, they are not at all that uncommon in relation databases). We close this section with an example that demonstrates the potential drawbacks (from a software engineering point of view) of having to refer to attributes by position instead of by name (the first view is defined using Prolog and the second view is defined using Tutorial D). In both cases we assume that a relation *COURSE* exists with several attributes (*CourseNo*, *Subject*, etc.) in it:

```
course_title(CourseNo, Title) :-
    course(CourseNo, Subject, Title, Term, Instructor, Credit, Campus, Room, Hours).
```

```
COURSE_TITLE := COURSE { CourseNo, Title }
```

When using logic programming, we are forced to enumerate all attributes of the relation and we also need to keep in mind which position refers to which attribute (say, that *Title* is the *third* attribute). This is not the case when using Tutorial D (that is, relational algebra).

2.3 Record Calculi and Systems with Polymorphic Relational Operators

Historically, the research community has spent serious amounts of effort on designing record calculi. Each record calculus proposed in the literature differs in the trade-offs it makes in terms of the level of polymorphism, the basic operations, and the complexity of the resulting type system. Polymorphic relational operators put an additional strain on the type system of statically typed languages because the result type of most of the relational operations depends on the types of their operands in a non-trivial way. Also, the complexity of typing polymorphic relational operators is known to be NP, irrespective of the actual type system being used [Vansummeren, 2005]. Since relations are just sets of records, it is natural that work done in the area of record calculi is inherently connected with work done on language extensions involving polymorphic relational operators. In the following, we present a detailed overview of related work that is relevant to the subject of the current thesis. In our presentation, we try to progress from less expressive systems to more expressive ones. Naturally, not all systems are strict supersets of, or even meaningfully comparable to, other systems, so our ordering is at times somewhat arbitrary. As a technical note on our usage of terminology: in order to differentiate records from simple

lookup tables (also associative arrays, hash maps, dictionaries, etc.) we insist that all record operations are checked for type correctness at compile time: that is, we require static typing of record operations, otherwise the language construct in question does not qualify as a record in our view.

2.3.1 Record Support in Mainstream Languages

In most mainstream languages that provide records (for example, C, C++, Java, C#, ML), operations on records are usually limited to a single one: field selection. In addition, field selection is not polymorphic, that is, the compiler needs to know the type of each record expression at compile time in order to decide whether field selections are type correct or not. Some of these mainstream systems are further limited by the lack of light-weight records, that is, the ability to construct records 'on-the-fly' using record literals. For example, in C [Kernighan and Ritchie, 1978], each record type (called 'struct' in C) needs to be declared before it can be used (Pascal behaves similarly), and even with pre-declared records, support for programmer-friendly literals (initializers in C parlance) has only recently made it to the standard [Meyers, 2000].

The following example, where we try to select the field `color` from the record argument `r`, illustrates Standard ML's [Milner et al., 1990] inability to type polymorphic field selection:

```
- fun f r = #color r;
stdIn: ... Error: unresolved flex record
   (can't tell what fields there are besides #color)
```

In object-oriented languages (Java, C++, C#) objects can serve as records and field selection can be made somewhat polymorphic through the use of inheritance or interfaces. It is important

to note, though, that the polymorphism of field selection does not come from the name of the field, but rather it depends on the type of the object being used. Just like in C, record (in this case, object) types need to be declared before being used. The following erroneous Java code segment demonstrates that the presence and absence of record fields are determined by the nominal type of the record (object) and not by their structural presence, thus limiting the polymorphism of record operations:

```
class A {
  public int x;
  public void test() {
    A a = new A();
    Object o = a;
    a.x = 2;         // Accepted by the compiler.  The type of ‘a’
                     // indicates the presence of the field ‘x’.
    o.x = 3;         // Rejected by the compiler.  Despite the fact that
                     // the field ‘x’ is present in the object ‘o’.
  }
}
```

2.3.2 Standard Record Subtyping

A popular way of handling polymorphic record operations is to define subtyping between records [Pierce, 2002]. The intuition is that a function that expects a record with certain fields should accept any record that has those fields plus, possibly, some others. Polymorphic field selection thus involves a subtyping constraint:

$$(r.l) :: \forall\alpha.\forall\rho \leq (\!|l : \alpha|\!).Rec\ \rho \rightarrow \alpha$$

This approach has several undesirable properties. First, subtyping constraints fail to retain information on other fields in the record. Second, adding subtyping to a polymorphic type system is known to complicate type inference and interacts poorly with other useful language features, like overloading [Pierce, 2002]. Finally, other useful record operations, like extension or concatenation, cannot be described in terms of record subtyping only.

2.3.3 Rows and Unchecked Row Extension

Wand introduced the concept of *rows* as basis for the recursive definition of record types in [Wand, 1987]. A row in Wand's system is either the empty row $(\!|\!|)$, or an extension of a row $(\!|l : \tau \mid \rho|\!|)$ with the label and type pair $(l : \tau)$. The type of field selection in Wand's system is:

$$(\_.l) :: \forall\alpha.\forall\rho.Rec\ (\!|l : \alpha \mid \rho|\!) \rightarrow \alpha$$

Also, it is now possible to express record extension:

$$\langle l = \_ \mid \_\rangle :: \forall\alpha.\forall\rho.\alpha \rightarrow Rec\ \rho \rightarrow Rec\ (\!|l : \alpha \mid \rho|\!)$$

The problem with Wand's system is that record operations are *unchecked*. For example, it is possible to extend *any* record with *any* label. One consequence of this is that some programs do not have principal types [Wand, 1988].

2.3.4 Present/Absent Flags

Rémy also used rows to handle labels not pertinent to the current operation but he also introduced *flags* to keep track of what labels need to be present in (or absent from) a given row [Rémy,

1989]. In his system the type of record extension would be:

$$\langle l = \_ \mid \_ \rangle :: \forall\alpha.\forall\rho.\alpha \rightarrow Rec\ (\!|l : abs\rho \rightarrow Rec\ (\!|l : pre(\alpha) \mid \rho|\!)$$

The type-checker is now able to deny access to undefined fields or deny the extension of a record with a label that it already has. Rémy further developed his system in [Rémy, 1992] to include both *symmetric* (records are disjoint) and *asymmetric* (records might overlap) record concatenation. However, his method was to translate programs with record concatenation to programs with record extension which limited the expressiveness of his system. For example, the following expression (taken from [Rémy, 1992]) cannot be typed in Rémy's system $\Pi^{\|}$ because of ML's restrictions on polymorphism:

let *reverse r s* = **if** *true* **then** $\langle r \parallel s \rangle$ **else** $\langle s \parallel r \rangle$

in *reverse* $\langle a = 1 \rangle$ $\langle b = 2 \rangle$

In other words, under certain conditions, symmetric record concatenation was no longer commutative, a serious limitation of the system in our view.

2.3.5 Generalized Rows with Subtyping

Pottier [Pottier, 2003] described a conditional type system with subtyping constraints and generalized rows, complete with a polynomial time constraint solver, with the restriction that rows had to be ground. The paper suggested the following constrained type schemes for some polymorphic record operations (where the empty record has type $\{\partial\mathsf{Abs}\}$, α ranges over variables of sort *Type* and kind *type*, φ ranges over variables of sort *Row* and kind *field*, $\leq$ denotes a subtyping

constraint, and the dot (.) separates type constraints from the type):

$$\forall\alpha\varphi[\{\ell\} : \varphi \leq \partial(\mathsf{Pre}\ \alpha)].\{\varphi\} \rightarrow \alpha \qquad \textit{(field selection for label } \ell\textit{)}$$

$$\forall\alpha\varphi_1\varphi_2[\{\ell\} : \partial(\mathsf{Pre}\ \alpha) \leq \varphi_2 \wedge (\mathcal{L}\backslash\ell) : \varphi_1 \leq \varphi_2].\{\varphi_1\} \rightarrow \alpha \rightarrow \{\varphi_2\} \qquad \textit{(non-strict extension with label } \ell\textit{)}$$

$$\forall\varphi_1\varphi_2\varphi_3[\mathcal{L} : \mathsf{Abs} \leq \varphi_1?\varphi_2 \leq \varphi_3 \wedge \mathcal{L} : \mathsf{Abs} \leq \varphi_2?\varphi_1 \leq \varphi_3 \wedge \mathcal{L} : \mathsf{Pre} \leq \varphi_1?\varphi_2 \leq \mathsf{Abs}].\ \{\varphi_1\} \rightarrow \{\varphi_2\} \rightarrow \{\varphi_3\} \qquad \textit{(symmetric concatenation)}$$

The type scheme above for record extension does not prevent the extension of a record with a field that it already has, hence it is called ‘non-strict’ extension. Pottier did not discuss the applicability of his system for describing polymorphic relational operators.

2.3.6 Record Concatenation with Disjointness Predicates

Harper and Pierce examined a second-order system $\lambda^{\|}$ with symmetric record concatenation in [Harper and Pierce, 1991]. Rows were constructed using concatenation instead of the usual extension. A row was either the empty row, a singleton label-value pair, or the concatenation of two rows. In their system the type predicate $r_1 \# r_2$ meant that the rows r_1 and r_2 are disjoint. The type of label selection in their system was:

$$(\_.l) :: \forall\alpha.\forall\rho.(\rho\#(\!|l : \alpha|\!)) \Rightarrow \mathit{Rec}\ ((\!|l : \alpha|\!) \parallel \rho) \rightarrow \alpha$$

Being second-order means that $\lambda^{\|}$ includes both explicit type abstraction and type application. Record types need to be passed as arguments to functions with polymorphic record operations. Also, the system did not include a type inference algorithm.

Makholm and Wells in [Makholm and Wells, 2005] described their system for mixin modules based on the language **bowtie** with symmetric record concatenation. They concluded that full polymorphic type checking is NP-complete but came up with the result that type-checking becomes polynomial when they ignore expressions that are either '*dead*' (their result will never be needed) or '*sleeping*' (their result will only be used if put into a larger context, like unused function definitions.)

2.3.7 Kinded Record Types and Machiavelli

Ohori [Ohori, 1995] described a type system with polymorphic field selection together with type inference and an efficient compilation method for polymorphic record operations using numerical label offsets. The basic idea is to assign records to different 'kinds' (types of types) based on the set of fields they are expected to contain. For example, the type of field selection in this system would have the type:

$$(\_.l) :: \forall\alpha.\forall r^{\langle\!|l:\alpha|\!\rangle}.Rec\ r \rightarrow \alpha$$

This kinded type system of records formed the basis of the language Machiavelli [Buneman and Ohori, 1996] which was itself an extension of ML. Machiavelli was designed as a true database programming language with direct support for polymorphic relational operators in the language. The extensions of interest to us included: (1) sets and set operations, (2) relations

as sets of records, and (3) relational operators *join* and *project* as primitives of the language. Machiavelli also supported extensible variants, recursive record types, and a generalized version of the *join* operator that could operate on data types other than relations. The following example shows a function that selects young employees from a polymorphic relational variable ('**select** ... **from** ... **where**' is Machiavelli's syntax for set comprehension):

```
-> fun young r = select [Name=x.Name] from x <- r where x.Age < 25;
>> val young = fn: {a:: [Name: b, Age: int]} -> {[Name: b]}
```

Machiavelli inferred the most general type for the polymorphic function `young` restricting the input parameter to relations that have fields `Name` and `Age`. The return type of the function is a set of records with a single `Name` field.

Machiavelli also defines the relational operator *join* as a primitive function of the language and assigns it a special conditional type:

```
>> val join = fn : (a * b) -> c where { c = jointype(a, b) }
-> join ({[SSN=123, Name="Smith"]},
         {[SSN=123, Car=Porsche], [SSN=123, Car=bmw]});
>> val it = {[SSN=123, Name="Smith", Car=Porsche],
             [SSN=123, Name="Smith", Car=bmw]}
     : {[SSN : int, Name : string, Car : carmake]}
```

The condition `c = jointype(a, b)` requires the type variable `c` to be in a certain relation with the type variables `a` and `b`. The result of the *join* operation is a relation whose heading is the *union* of the headings of the relations being joined. This is the constraint that Machiavelli captures with its conditional typing. However, if either `a` or `b` is unbound at compile time then Machiavelli cannot compute the value of `c` so it might accept programs that contain type errors.

In other words, Machiavelli does not check the satisfiability of type conditions. Interestingly, this is not a serious limitation in practice since whenever an expression evaluation involves *join* that means that the type variables are required to be bound so the compiler can check the type conditions. In other words, Machiavelli restricts the type-checking of polymorphic relational operators in a similar way as **bowtie** did with record concatenation in Section 2.3.6: Machiavelli does not check 'dead' or 'sleeping' code. For example, the following function is accepted by Machiavelli although it is ill-typed:

```
-> fun tricky (r,s) =
     union (select x.Name from x <- join(r,s)
                          where x.Salary > 100000,
            select y.Name from y <- s
                          where y.Salary = "High");
```

The problem with `tricky` is that if `s` has a `Salary` attribute of type `string` then the join of `r` and `s` cannot have a `Salary` attribute of type `int`. Machiavelli could not catch this error because `r` and `s` are polymorphic, so the `jointype` of the relations `r` and `s` cannot be computed at compile time.

Machiavelli also introduces a generalized *project* operator that takes an arbitrary expression and a ground type (for which equality is well-defined) and projects the expression on the given type. For example, the following expression projects a record on one of its labels:

```
-> project([SSN=123, Name="Smith", Car=Porsche], [Name : String]);
>> val it = [Name="Smith"] : [Name : string]
```

Although it might seem like, the function *project* does not itself introduce first-class labels because *project* is part of Machiavelli's *syntax*. The best way to think of *project* in Machiavelli

is like a type cast operator whose correctness is checked at compile time. Needless to say, this means that there has to be a special typing rule in the type system for dealing with *project*.

Although Machiavelli introduced the relational operators *join* and *project* as primitives of the language it suffers from the same problems as any language that try to fix the list of polymorphic relational operators: there always will be relational operators that cannot be expressed using the pre-defined ones. In the case of Machiavelli, neither the relational operator *great divide* [Date and Darwen, 1992a] nor *compose* [Codd, 1970] can be expressed using the primitives.

2.3.8 Qualified Types and Rows

A system for record extension based on rows is presented by Gaster and Jones in [Gaster and Jones, 1996]. The system presented in the paper is an adaptation of Jones's theory of *qualified types* [Jones, 1992] which is a comprehensive system of using constraints (predicates) on types to restrict the applicability of polymorphic functions. Gaster and Jones's system can infer the type for expressions with polymorphic field selection, field deletion, and record extension, and also check the satisfiability of the arising type constraints in polynomial time, but it cannot express polymorphic relational operators. Similarly to Ohori's system [Ohori, 1995], the authors described an effective compilation method that calculated label offsets from the *lacks* predicates that appear in the types of expressions with polymorphic record operations. The type system of the current thesis is a direct extension of that of the system presented in [Gaster and Jones, 1996].

*2.3.9 The Language **D***

Date and Darwen presented a comprehensive language proposal in [Date and Darwen, 1998] that they tentatively named **D**. The proposal mainly consisted of prescriptions and proscriptions for various language features that the authors deemed desirable for a modern database programming language based on relational algebra. In contrast with the orthogonal persistence approach, only relational variables (representing relations in an external relational database) can be made persistent. For demonstration purposes, the authors introduced the language *Tutorial D* which embodied the language principles of they put forward. The design of Tutorial D aims at fixing some of the chief mistakes committed by SQL, but it also includes some novel contributions.

Tutorial D is an explicitly-typed language with full compile-time type checking. Attributes in relations can be of arbitrary types, which means that besides primitive types, user-defined types can also be stored in the database. The language does not support variants or recursive tuple types. The following Tutorial D code segment defines a relational variable (or *relvar*) *person*:

```
VAR person REAL Relation { id         Integer,
                           name       String,
                           address    Tuple { street     String,
                                              city       String,
                                              state      StateCode,
                                              zip        ZipCode },
                           location   GPSCoord }
    KEY { id };
```

The above definition demonstrates several features of Tutorial D: (1) explicit type signatures;

(2) arbitrary user-defined types in relvars (the attribute *location* is of type *GPSCoord* which is, presumably, a user-defined abstract data type); and (3) nested tuple types (the type of attribute *address* is a tuple type with attributes *street*, *city*, etc.).

Relational operators are part of the language definition with special syntax and typing rules. Due to the explicitly typed nature of the language, the type correctness of relational operator applications can always be checked in polynomial type during compilation (in other words, the language does not support polymorphic type inference). Also, as opposed to SQL, relational queries can be arbitrarily nested. An important contribution of Tutorial D was that it invigorated interest in relational algebra with its concise and elegant syntax and semantics, an interest that began to wane as the majority came to identify relational algebra with its most wide-spread, and less than flawless, implementation, SQL. As such, the language Tutorial D, and the language design principles advocated by the Third Manifesto, were a major inspiration for the current thesis. To convey an impression of the novelty of its approach, we list some examples that show Tutorial D expressions along with their SQL counterpart. Throughout the examples, we will use Date's familiar suppliers-parts database [Date, 1999]:

```
S { Sname , City }                      SELECT Sname, City FROM S

S JOIN SP                               SELECT S.S#, S.Sname, S.Status, S.City, SP.P#, SP.Qty
                                        FROM S JOIN SP ON S.S# = SP.S#

S WHERE City = "London" { Sname }       SELECT Sname FROM S WHERE City = "London"

EXTEND SP ADD (Qty + 10) AS AltQty      SELECT S#, P#, Qty, Qty + 10 AS AltQty FROM SP
```

2.3.10 Type Inference for the Relational Algebra

Bussche and Waller in [Van den Bussche and Waller, 1999] directly address the problem of type inference for polymorphic relational expressions. However, the target of type inference is 'pure' relational algebra, that is, an 'out-of-context' version which is not embedded in any programming language. The type of a relational variable is simply a set of attribute names, and no further types are assigned to the attributes themselves. Type inference is aimed at deriving type formulas (including boolean formulas on attributes) that describe all the possible schemas under which a given relational expression is well-typed. If the type formula is unsatisfiable, then the relational expression contains a type error and there is no schema under which it is well-typed.

The following example is taken from the same paper and shows a relational expression

$$e = \sigma_{A<5}(r \bowtie s) \bowtie ((r \times u) - v)$$

together with the inferred type formula that captures the constraints on relational variables in expression e:

$$\begin{array}{l} v : a_1a_2a_3a_4 \\ r : a_1a_3 \\ u : a_2a_4 \\ s : a_3a_4a_5 \\ A : (r \vee s) \wedge (r \Rightarrow v) \wedge \neg(r \wedge u) \end{array} \quad \mapsto \quad \begin{array}{l} e : a_1a_2a_3a_4a_5 \\ A : \textbf{true} \end{array}$$

The interpretation of the above type formula is that if each a_i is assigned some set of attributes (that does not include the attribute A and is disjoint from all other a_j's) and the constraint on

the attribute A is true (where $A : r \vee s$ is an abbreviation for $A \in r \vee A \in s$) then the type of expression e is the union $a_1 \cup a_2 \cup a_3 \cup a_4 \cup a_5$ and A must be in e.

Bussche and Wadler presented a type inference algorithm that could derive type formulas for arbitrary relational algebra expressions. However, their system was not directly aimed at solving the polymorphic type inference problem of relational algebra expressions in the context of a functional language. They ignore the types of attributes and they do not introduce tuples and tuple operations into their language. In a follow-up paper [Van den Bussche and Vansummeren, 2005], they extended their previous system with attribute selection and set comprehension and used rows to describe attribute types in tuples. The paper described a type inference algorithm that generated type formulas, but no algorithm was provided to check the satisfiability of type formulas. Their new system, like their earlier one, lacked the ability to define new relational operators.

Inspired by the work of Bussche and Wadler, Nagy and Stansifer [Nagy and Stansifer, 2005] described a functional language with polymorphic relational operators (experience with said system influenced to design of the system presented in the current thesis). Their approach was based on type formulas introduced by Bussche and Wadler but they incorporated constraints on attribute types into their system. The problem with their system is that the constraint solving phase was only hinted at and it is unclear how row unification (unifying the types of matching attributes) is supposed to be carried out in that system. As an additional problem, the type inference algorithm is quite complex and does not lend itself easily to correctness proofs.

2.3.11 HaskellDB: Strongly Typed Database Access for Haskell

Leijen and Meijer presented HaskellDB in [Leijen and Meijer, 1999] as an exercise in designing domain specific language extensions using Haskell's powerful abstraction mechanisms and expressive type system. The original design of HaskellDB relied on a Hugs [Jones and Peterson, 1999] (a Haskell implementation) specific language extension (sometimes referred to as Trex) that supported a system of extensible records based on the work of Gaster and Jones [Gaster and Jones, 1996]. From later versions of HaskellDB, this dependence on a Hugs extension has been removed to make it more standards compliant (at the cost of losing some of its original elegance).

The main idea behind HaskellDB is that instead of having the programmer build SQL queries using the traditional method of string concatenation, thus losing all effective chance at ensuring syntactic and semantic correctness at compile time, the programmer is given facilities for building queries in a type-aware *abstract syntax tree* format. To make the construction of syntactically incorrect queries impossible, the authors used Haskell's algebraic data types to describe the abstract syntax of relational algebra (actually, SQL, but due to the high level of abstraction the difference mattered little). HaskellDB further improved this idea by embellishing algebraic data types with phantom types and thus preventing the construction of semantically incorrect queries. Next, we give a hint as to how it was achieved using excerpts from the HaskellDB code base:

```
data PrimExpr =   -- Data type for primitive expressions.
    BinExpr BinOp PrimExpr PrimExpr  | UnExpr UnOp PrimExpr  | ConstExpr String


data BinOp =   -- Data type for binary operations.
    OpEq | OpAnd | OpPlus | ...
```

Writing queries directly in abstract syntax is a bit inconvenient, but thanks to Haskell, it is possible to provide combinators that correspond to the usual SQL operators:

```
constant :: Show a -> a -> PrimExpr
(.+.)    :: PrimExpr -> PrimExpr -> PrimExpr
(.AND.)  :: PrimExpr -> PrimExpr -> PrimExpr
(.==.)   :: PrimExpr -> PrimExpr -> PrimExpr
```

Using the above definitions, it is still possible to build semantically incorrect expressions, like the following:

```
constant "3" .+. constant "b"
```

Phantom types take care of this problem by including the type of the expression that the abstract syntax tree is supposed to represent in the type of the expression node (notice how the type variable `a` in `Expr a` does not appear on the right hand side of the definition, hence the 'phantom-type' name):

```
data Expr a = Expr PrimExpr


constant :: Show a -> a -> Expr a
(.+.)    :: Expr Int -> Expr Int -> Expr Int
(.AND.)  :: Expr Bool -> Expr Bool -> Expr Bool
(.==.)   :: Eq a -> Expr a -> Expr a -> Expr Bool
```

To handle arbitrary relational expressions, a comprehension based monad was introduced by the authors which represents the computation expressed by the query. Similar to most SQL implementations that also use comprehension, HaskellDB cannot concisely express natural join but rather relies on the programmer to explicitly enumerate each and every attribute that has to

appear in the result. The way to construct queries using this monadic combinator turned out to be surprisingly intuitive, especially if one is familiar with Haskell's list comprehensions, as demonstrated by the following example (we also show the same query in Tutorial D syntax):

```
do { r <- table s
   ; p <- table sp
   ; restrict (r!city .==. constant "London")
   ; restrict (r!s# .==. p!s#)
   ; project (sname = r!sname, p# = p!p#, qty = p!qty)
   }
```

```
S JOIN SP WHERE City = "London" { Sname, P#, Qty}
```

The use of monadic combinators made it possible to treat relational queries as first class values with the additional benefit of the ability to serialize access to the external database, a basic feature of monads [Wadler, 1993]. The fact that queries were represented by their abstract syntax tree also made it possible to apply traditional query optimization techniques during runtime. To ensure full static type checking, HaskellDB required the definition of the database schema to be available at compile time in the form of a separate Haskell module (which was typically generated by tools that could extract schema information from the database).

2.3.12 Heterogeneous Collections for Haskell

Kiselyov et al. [Kiselyov et al., 2004] presented for Haskell an encoding of collections (more particularly, lists) whose elements were not restricted to the same type. The encoding heavily relied on Haskell's extensible class system which is itself based on the theory of Qualified Types [Jones, 1992]. To demonstrate the expressive power of their system, the authors presented

strongly typed encodings, using Haskell's class system to represent type constraints, of an unprecedented variety of polymorphic record operations, including, among others: (1) extension, (2) field deletion, (3) symmetric concatenation, (4) the ability to ask for the set of labels (the *heading*) of a record and to perform set operations on headings, (5) to combine a list of labels and a list of values into a record, and (6) to project a record on a set of labels. In addition, field labels are first-class citizens in their system (due to the fact that they are encoded as singleton types). The following example demonstrates the construction of a record describing the cow Angus, where we first construct the record labels:

```
data Cow = Cow       -- Type used as label name space.

-- Definition of record labels.
key = firstLabel Cow "key"
name = nextLabel key "name"
breed = nextLabel name "breed"
price = nextLabel breed "price"

-- Definition of a record.
angus = key .=. (42::Integer)
.*. name .=. "Angus"
.*. breed .=. Cow
.*. emptyRecord
```

Just to give a taste of the level of Haskell type magic that goes on in the background, we show the definitions, starting from that of heterogeneous lists, that lead up to the `(.*.)` combinator:

```
data HNil = HNil deriving (Eq,Show,Read)
data HCons e l = HCons e l deriving (Eq,Show,Read)

class HList l
instance HList HNil
instance HList l => HList (HCons e l)

(.*.) :: HList l => e -> l -> HCons e l
(.*.) = HCons
```

Actually, the encoding of records uses type-level naturals annotated with a string for the label name. Name spaces (represented by singleton types like `Cow` above) are used to prevent conflicts between naturals when used as record labels. As the encoding heavily relies on Haskell's class system whose error reporting capability is far from perfect, special tricks had to be used in order to improve error messages. For this purpose, the vacuous Haskell class `Fail` was introduced and later used to provide instances of special error-reporting classes. Erroneous situations were represented by requiring the compiler to derive an instance for an error-reporting class (which it could not, since the only class providing it, `Fail`, was vacuous), thus forcing the compiler to provide a more useful failure indication. The following example gives a hint as to the nature of this method:

```
instance Fail (TypeNotFound e) => HOccurs e HNil
  where hOccurs = undefined

class Fail x -- no methods, no instances!
data TypeNotFound e -- no values, no operations!
```

Now, if we would like to ask for a list of integers from a heterogeneous list that only contains a single boolean value, the Haskell interpreter would give a more useful error message (similar error messages were defined for erroneous record operations):

```
ghci-or-hugs> hOccurs (HCons True HNil) :: Int
No instance for (Fail (TypeNotFound Int))
```

Naturally, as record encodings relied completely on the capabilities of available Haskell compilers, no special purpose constraint satisfaction algorithm was provided for checking type correctness of polymorphic record operations in general. This, however, did not prove to be a problem, since whenever such operations are applied to actual arguments, type variables are instantiated to actual types, thus the compiler can check the satisfiability of type constraints.

Chapter 3
Language Syntax and Semantics

In this chapter, we describe the syntax and the semantics of the language. We begin with the syntax of the core language and of the basic record operations, which we follow with a section on the considerations that shaped our design choices. Next, we provide a detailed description of the basic record operations, together with usage examples. We continue with the formal definition of language semantics through evaluation rules (which we include chiefly for the sake of completeness). After introducing sets and relations, we conclude the chapter with the development of relational algebra, from basic record operations and sets, complete with examples.

The term language is an extension of core-ML, that is, an implicitly-typed λ-calculus with let-bound polymorphism. Figure 3.1 defines the syntax of terms where the standard core and the record extensions are separated by a line (both variables x and labels l draw their values from a countable set of names $\mathcal{L}$). For the purposes of defining formal semantics, we also define *values* in Figure 3.2 as a subset of terms.

3.1 Language Design Considerations

Our goal was to design a language that can not only express polymorphic relational operators, but which would also allow the definition of user-defined ones. The decision had to be made what *primitives*, or basic operations, need to be present in the language in order to achieve this

t ::=		*term*
	x	*variable*
	c	*constant*
	$\lambda x.t$	*abstraction*
	$t\ t$	*application*
	let $x = t$ **in** t	*let*
	$\langle l_1 = t, ..., l_n = t \rangle$	*record literal*
	$\langle l = t \mid t \rangle$	*record extension*
	$t \cdot l$	*field selection*
	$\langle t \parallel t \rangle$	*record concatenation*
	$\langle t \backslash t \rangle$	*record difference*
	$t!l$	*field deletion*
	$t \cdot [t]$	*record projection*

Figure 3.1: Syntax of Terms

v ::=		*value*
	c	*constant value*
	$\lambda x.t$	*abstraction value*
	$\langle l_1 = v, ..., l_n = v \rangle$	*record value*

Figure 3.2: Value Terms

goal. Since relations are sets of records (tuples are called records in the programming language community), it was clear from the beginning that the language would have to based on some form of record calculus. The particular set of basic record operations we eventually chose was arrived at through experimentation, where we strived for a minimal set of operations that could express all original relational operators and was powerful enough to define new ones. As it will be demonstrated later, the chosen basic operations are *sufficiently* expressive to describe user-defined polymorphic relational operators. The question arises whether each is also *necessary*, that is, whether the set of basic operations is minimal. Curiously, the answer is *no*. Not all basic operations are strictly necessary and there is a proper subset of the basic operations that have the same expressive power as the original set. Why include the additional operators then? The short answer is: for software engineering considerations. Some operations, which are expressible in terms of more basic ones, nevertheless, allow us to put additional constraints on their operands. For example, although *field deletion* can be expressed using *record difference*, a separate *field deletion* operation allows us the require the presence of the field to be deleted, something that can help catch typos in record labels (otherwise, an attempt to delete a non-existent field would always succeed, instead of resulting in an error).

3.1.1 The Notion of Record and the Omission of Variants

A *record* is a collection of values (known as *fields*), possibly of different types, each of which is associated with a distinct *label* drawn from the *heading* of the record. A record is thus a *product* of values of possibly different types. The dual of product is *sum*, and the dual of record is called *variant* (or *tagged union*). A variant is one particular value, tagged by a label, from some fixed

set of possibly different types. As a result of this theoretical duality, records and variants are often introduced side by side and treated similarly in some systems, for example in [Buneman and Ohori, 1996; Gaster and Jones, 1996; Leijen, 2004]. During the preliminary design phase, a conscious decision was made to *omit* variants from the system, because variants, as opposed to records, play a marginal role in relational algebra, our main topic of interest. It remains as an interesting future work to consider the implications of adding variants to the current system.

3.1.2 First-class Labels

In a language with first-class record labels we have the ability to treat record labels as ordinary values that can be passed in as function arguments, stored in data structures, or serve as return values from functions. A system with first-class labels was described in [Leijen, 2004]. Unfortunately, the ability to pass around record labels as ordinary values, does not, in itself, allow us to define polymorphic relational operators. Hence, after due consideration, it was decided to omit first-class labels from the current system, mainly because the costs it would incur (chiefly, loosing the ability to derive principle types) seemed to outweigh its possible advantages (intersection types, type selective functions, etc). Thus, in the current system record labels are part of the *syntax* of the language.

3.2 Basic Record Operations

In this section we introduce the basic record operations. A word of caution before we proceed. It is important to realize that the particular syntax chosen for the basic operations is to a large extent *irrelevant* from a theoretical point of view. Nevertheless, syntax *does* play an important role in

Operation	**Shorthand**	**Expansion**
Heading Literal	$\langle l_1, ..., l_n \rangle$	$\langle l_1 = (), ..., l_n = () \rangle$
Record Restriction	$(e_1![e_2])$	$\langle e_1 \backslash (e_1 \cdot [e_2]) \rangle$

Table 3.1: Notational Shorthands

the practical usefulness of any programming language so we strived for forms that respect good language design principles and aesthetics. The decisions for operator associativity, precedence, and syntactic form were governed by a search for economy of expression, readability, and, above all, the somewhat elusive notion of 'conceptual integrity' (as advocated in [Brooks, 1995]).

We defer the discussion of types to the next chapter so that we can concentrate on the meaning of and the rationale for each basic operation here. Using the basic operations, it is possible to define other useful record operations that are common enough to merit their own notational shorthands. A summary of these derived operations is presented in Table 3.1. A detailed description of the operations themselves follow:

- *Record Extension* Record extension '$e = \langle l = e_1 \mid e_2 \rangle$' is used to extend records with new fields. Record extension will also play an important role in type-checking record literals, since, at the conceptual level, record extension is used to recursively build records, starting from the empty record. In order to be well-defined, the expression e_2 must evaluate to a record that *lacks* the label l. The result of record extension e is a record that defines the same mapping as the record e_2, and in addition it maps the label l to e_1. Examples of

behavior (the symbol '$\rightsquigarrow$' denotes evaluation):

$$\langle a = 1 \mid \langle\rangle\rangle \quad \rightsquigarrow \quad \langle a = 1\rangle$$

$$\langle b = 2 \mid \langle a = 1\rangle\rangle \quad \rightsquigarrow \quad \langle b = 2, a = 1\rangle$$

$$\langle b = 2 \mid \langle c = \text{"}A\text{"}, a = 1\rangle\rangle \quad \rightsquigarrow \quad \langle b = 2, c = \text{"}A\text{"}, a = 1\rangle$$

$$\langle b = 2 \mid \langle b = 1\rangle\rangle \quad \rightsquigarrow \quad \textbf{UNDEFINED}$$

$$\langle b = 2 \mid \langle a = 42, b = 1\rangle\rangle \quad \rightsquigarrow \quad \textbf{UNDEFINED}$$

- *Field Selection* Field selection '$e \cdot l$' is used to access the field l in record e. In order to be well-defined, the expression e must evaluate to a record that *has* the label l. The result of field selection is the value for the specific field in record e. In the syntax used throughout the thesis, field selection is left-associative and has higher precedence than function application, for example, '$f\ x \cdot a \cdot b$' means '$f\ ((x \cdot a) \cdot b)$' and not '$((f\ x) \cdot a) \cdot b$'. Examples of behavior:

$$\langle a = 42\rangle \cdot a \quad \rightsquigarrow \quad 42$$

$$\langle c = \text{"}A\text{"}, a = 1\rangle \cdot c \quad \rightsquigarrow \quad \text{"}A\text{"}$$

$$\langle b = 1\rangle \cdot a \quad \rightsquigarrow \quad \textbf{UNDEFINED}$$

$$\langle b = 2, c = 42\rangle \cdot a \quad \rightsquigarrow \quad \textbf{UNDEFINED}$$

- *Record Concatenation* Record concatenation '$e = \langle e_1 \parallel e_2\rangle$' is used to merge two records. This operation should more precisely be called *symmetric* record concatenation since the two records are required to be disjoint. Because the records to be merged are disjoint, the operation is commutative (hence the name, 'symmetric'). In order to be well-defined, both expressions e_1 and e_2 must evaluate to records and they must have *disjoint* headings (sets

of labels). The result of the concatenation of records e_1 and e_2 is a record that defines the same mapping as the record e_1 (e_2) when restricted to labels in e_1 (e_2). The empty record is a unit element for this operation. Examples of behavior:

$$\langle a = 1 \parallel \langle\rangle\rangle \quad \rightsquigarrow \quad \langle a = 1\rangle$$

$$\langle\langle\rangle \parallel a = 1\rangle \quad \rightsquigarrow \quad \langle a = 1\rangle$$

$$\langle\langle b = 2\rangle \parallel \langle a = 1\rangle\rangle \quad \rightsquigarrow \quad \langle b = 2, a = 1\rangle$$

$$\langle\langle b = 2\rangle \parallel \langle b = 1\rangle\rangle \quad \rightsquigarrow \quad \textbf{UNDEFINED}$$

$$\langle\langle b = 2\rangle \parallel \langle a = 42, b = 1\rangle\rangle \quad \rightsquigarrow \quad \textbf{UNDEFINED}$$

- *Record Difference* Probably the most versatile of the basic operations, record difference '$e = \langle e_1 \backslash e_2\rangle$' is used to throw away those fields from e_1 that also appear in e_2. The field *values* of the second operand play no role in the operation, only its *labels*. The only requirement for this operation to be well-defined is that both expressions e_1 and e_2 must evaluate to records. The result of the operation is a record that defines the same mapping as the record e_1 but is restricted to those labels that *do not* appear in e_2. The empty record is a right unit element for this operation. Examples of behavior:

$$\langle\langle a = 1\rangle \backslash \langle\rangle\rangle \quad \rightsquigarrow \quad \langle a = 1\rangle$$

$$\langle\langle\rangle \backslash \langle a = 1\rangle\rangle \quad \rightsquigarrow \quad \langle\rangle$$

$$\langle\langle b = 2\rangle \backslash \langle a = 1\rangle\rangle \quad \rightsquigarrow \quad \langle b = 2\rangle$$

$$\langle\langle b = 2, a = \text{"}A\text{"}\rangle \backslash \langle a = 1\rangle\rangle \quad \rightsquigarrow \quad \langle b = 2\rangle$$

$$\langle\langle b = 2, a = \text{"}A\text{"}\rangle \backslash \langle a = \text{"}Yoshiko\text{"}\rangle\rangle \quad \rightsquigarrow \quad \langle b = 2\rangle$$

$$\langle\langle b = 2, a = 1971\rangle \backslash \langle a = \text{"}Yoshiko\text{"}\rangle\rangle \quad \rightsquigarrow \quad \langle b = 2\rangle$$

- *Field Deletion* To remove a field l from a record e we use field deletion '$e!l$'. This operation is not strictly necessary since it can be expressed using record difference in the following way: $e!l \equiv \langle e \backslash \langle l = () \rangle \rangle$. The reason why it was included among basic operations is that, in contrast with record difference, it requires the presence of the field to be removed, thus it is not defined on all operands. From a software engineering point of view, it is useful if we can signal potential programmer errors as early as possible. Trying to remove a non-existent field can be the sign of a typo in the name of the field, thus it is better reported. If the programmer decides that the operation is in fact correct, then field deletion can always be re-written using the more forgiving record difference. In order to be well-defined, the expression e must evaluate to a record that *has* the label l. The result of field deletion is a record that defines the same mapping as e but is undefined for the label l. In the syntax used throughout the thesis, field deletion is left-associative and has higher precedence than function application (but the same as field selection), for example, '$f\ x!a!b$' means '$f\ ((x!a)!b)$' and not '$((f\ x)!a)!b$'. Examples of behavior:

  $$\langle a = 42 \rangle!a \quad \leadsto \quad \langle\rangle$$

  $$\langle c = \text{"}A\text{"}, a = 1 \rangle!c \quad \leadsto \quad \langle a = 1 \rangle$$

  $$\langle b = 3, c = \text{"}A\text{"}, a = 1 \rangle!c!b \quad \leadsto \quad \langle a = 1 \rangle$$

  $$\langle b = 1 \rangle!a \quad \leadsto \quad \textbf{UNDEFINED}$$

  $$\langle b = 2, c = 42 \rangle!a \quad \leadsto \quad \textbf{UNDEFINED}$$

- *Record Projection* This operation is analogous to the *projection* operation of relational algebra, and was introduced so that the subset relation constraint between the headings of operands could be enforced. Record projection '$e = e_1 \cdot [e_2]$' is used to project the

first record e_1 on the heading of the second record e_2. Similarly to *field deletion*, record projection is not strictly required since it can also be expressed using record difference: $e_1 \cdot [e_2] \equiv \langle e_1 \backslash \langle e_1 \backslash e_2 \rangle \rangle$. The reasons for including it are similar to those of *field deletion*, that is, software engineering considerations. In order to be well-defined, both expressions e_1 and e_2 must evaluate to records and the heading (set of labels) of e_2 must be a subset of the heading of e_1. The result of projecting record e_1 on e_2 is a record that defines the same mapping as e_1 restricted to labels that appear in e_2. It is important to note, that the field values of record e_2 play no part in the operation. Heading literals (see Table 3.1), often used in record projection, are simply records whose field values are of no importance (since we are only interested in the set of labels they define). In the syntax used throughout the thesis, record projection is left-associative and has higher precedence than function application, for example, '$f\ x \cdot [y] \cdot [z]$' means '$f\ ((x \cdot [y]) \cdot [z])$' and not '$((f\ x) \cdot [y]) \cdot [z]$'. Also, notice the deliberate similarity in notation between field selection (projecting on a single label) and record projection (projecting on a set of labels). Examples of behavior:

$$\langle a = 1 \rangle \cdot [\langle\rangle] \quad \rightsquigarrow \quad \langle\rangle$$

$$\langle b = 2, a = 1 \rangle \cdot [\langle a = 23 \rangle] \quad \rightsquigarrow \quad \langle a = 1 \rangle$$

$$\langle b = 2, a = 1 \rangle \cdot [\langle b \rangle] \quad \rightsquigarrow \quad \langle b = 2 \rangle$$

$$\langle b = 2, a = 1 \rangle \cdot [\langle a = 42, b = \text{"}B\text{"} \rangle] \quad \rightsquigarrow \quad \langle b = 2, a = 1 \rangle$$

$$\langle b = 2, a = 1 \rangle \cdot [\langle c = \text{"}C\text{"} \rangle] \quad \rightsquigarrow \quad \textbf{UNDEFINED}$$

$$\langle b = 2, a = 1 \rangle \cdot [\langle a = 3, c = \text{"}C\text{"} \rangle] \quad \rightsquigarrow \quad \textbf{UNDEFINED}$$

To showcase the expressive power of basic record operations, in the closing example we

show how *default* values for missing fields can be supplied in a statically type-safe manner:

$$default \equiv \lambda t.\textbf{let } d = \langle a = 7\rangle \textbf{ in } \langle t \parallel \langle d \backslash t\rangle\rangle$$

$$default\ \langle a = 2, c = True\rangle \quad \rightsquigarrow \quad \langle a = 2, c = True\rangle$$

$$default\ \langle b = 5\rangle \quad \rightsquigarrow \quad \langle a = 7, b = 5\rangle$$

3.3 Formal Semantics

The formal meaning of programs in the language is defined through operational semantics, that is, using evaluation rules. With regard to these evaluation rules can we say that "well-typed programs don't go wrong", that is, if a term is well-typed according the type system described in the next chapter, then the evaluation of a term will always result in a value. Figure 3.3 describes the standard evaluation rules [Pierce, 2002] for an implicitly-typed λ-calculus with an additional **let** construct. The evaluation rules follow a lazy, or more precisely 'call-by-name', evaluation strategy, that is, we always reduce the leftmost, outermost redex (reducible expression) and we never reduce inside abstractions. The substitutions used in rules *(E-AppAbs)* and *(E-Let)* are standard capture-avoiding substitutions, as defined in [Pierce, 2002]. Figures 3.4, 3.5, and 3.5 formalize the meaning of basic record operations that were described earlier (notice how record projection and field deletion is defined in terms of record difference).

We remark that in the evaluation rules we make use of our earlier definition of values and require certain terms to be actually values to enforce evaluation order, for example in *(E-Diff2)*. Also, although we could have defined it directly, we chose to define record concatenation as a series of record extensions to simplify presentation.

$$(\textit{E-App}) \qquad \frac{t_1 \rightsquigarrow t_1'}{(t_1\ t_2) \rightsquigarrow (t_1'\ t_2)}$$

$$(\textit{E-AppAbs}) \qquad ((\lambda x.t_1)\ t_2) \rightsquigarrow [x \mapsto t_2]t_1$$

$$(\textit{E-Let}) \qquad \textbf{let}\ x = t_1\ \textbf{in}\ t_2 \rightsquigarrow [x \mapsto t_1]t_2$$

Figure 3.3: Evaluation Rules for the Core Language

$$(\textit{E-Rec}) \qquad \frac{t_j \rightsquigarrow t_j'}{\langle l_1 = t_1, ..., l_j = t_j, ..., l_n = t_n\rangle \rightsquigarrow \langle l_1 = t_1, ..., l_j = t_j', ..., l_n = t_n\rangle}$$

$$(\textit{E-Ext1}) \qquad \frac{t_2 \rightsquigarrow t_2'}{\langle l = t_1 \mid t_2\rangle \rightsquigarrow \langle l = t_1 \mid t_2'\rangle}$$

$$(\textit{E-Ext2}) \qquad \frac{l_x \notin \{l_1, ..., l_n\}}{\langle l_x = t_x \mid \langle l_1 = t_1, ..., l_n = t_n\rangle\rangle \rightsquigarrow \langle l_x = t_x, l_1 = t_1, ..., l_n = t_n\rangle}$$

$$(\textit{E-Conc1}) \qquad \frac{t_1 \rightsquigarrow t_1'}{\langle t_1 \parallel t_2\rangle \rightsquigarrow \langle t_1' \parallel t_2\rangle}$$

$$(\textit{E-Conc2}) \qquad \langle\langle l_1 = t_1, ..., l_n = t_n\rangle \parallel t_x\rangle \rightsquigarrow \langle\langle l_1 = t_1 \mid ...\langle l_n = t_n \mid t_x\rangle ...\rangle$$

Figure 3.4: Evaluation Rules for Basic Record Operations

$$\text{(E-Diff1)}\quad \frac{t_1 \rightsquigarrow t_1'}{\langle t_1 \backslash t_2 \rangle \rightsquigarrow \langle t_1' \backslash t_2 \rangle}$$

$$\text{(E-Diff2)}\quad \frac{t_2 \rightsquigarrow t_2'}{\langle v_1 \backslash t_2 \rangle \rightsquigarrow \langle v_1 \backslash t_2' \rangle}$$

$$\text{(E-Diff3)}\quad \frac{l_1^x \in \{l_1^y, ..., l_m^y\}}{\begin{array}{c}\langle \langle l_1^x = t_1^x, ..., l_n^x = t_n^x \rangle \backslash \langle l_1^y = t_1^y, ..., l_m^y = t_m^y \rangle \rangle \\ \rightsquigarrow \langle \langle l_2^x = t_2^x, ..., l_n^x = t_n^x \rangle \backslash \langle l_1^y = t_1^y, ..., l_m^y = t_m^y \rangle \rangle\end{array}}$$

$$\text{(E-Diff4)}\quad \frac{l_1^x \notin \{l_1^y, ..., l_m^y\}}{\begin{array}{c}\langle \langle l_1^x = t_1^x, ..., l_n^x = t_n^x \rangle \backslash \langle l_1^y = t_1^y, ..., l_m^y = t_m^y \rangle \rangle \\ \rightsquigarrow \langle l_1^x = t_1^x \mid \langle \langle l_2^x = t_2^x, ..., l_n^x = t_n^x \rangle \backslash \langle l_1^y = t_1^y, ..., l_m^y = t_m^y \rangle \rangle \rangle\end{array}}$$

Figure 3.5: Evaluation Rules for Basic Record Operations (Continued)

(*E-Del1*) $$\frac{t_1 \rightsquigarrow t_1'}{t_1!l \rightsquigarrow t_1'!l}$$

(*E-Del2*) $$\frac{l \in \{l_1, ..., l_n\}}{\langle l_1 = t_1, ..., l_n = t_n\rangle!l \rightsquigarrow \langle\langle l_1 = t_1, ..., l_n = t_n\rangle \backslash \langle l = ()\rangle\rangle}$$

(*E-Proj1*) $$\frac{t_1 \rightsquigarrow t_1'}{t_1 \cdot [t_2] \rightsquigarrow t_1' \cdot [t_2]}$$

(*E-Proj2*) $$\frac{t_2 \rightsquigarrow t_2'}{v_1 \cdot [t_2] \rightsquigarrow v_1 \cdot [t_2']}$$

(*E-Proj3*) $$\frac{\{l_1^y, ..., l_m^y\} \subseteq \{l_1^x, ..., l_n^x\}}{\langle l_1^x = t_1^x, ..., l_n^x = t_n^x\rangle \cdot [\langle l_1^y = t_1^y, ..., l_m^y = t_m^y\rangle] \rightsquigarrow \langle\langle l_1^x = t_1^x, ..., l_n^x = t_n^x\rangle \backslash \langle\langle l_1^x = t_1^x, ..., l_n^x = t_n^x\rangle \backslash \langle l_1^y = t_1^y, ..., l_m^y = t_m^y\rangle\rangle\rangle}$$

Figure 3.6: Evaluation Rules for Basic Record Operations (Continued)

3.4 Sets and Relations

Sets and set operations are not central to the current thesis, but are required in any practical implementation and are also convenient in presenting the examples. Hence, we will assume that standard set operations, together with an empty set constant, are defined among the constants of the language. In addition to having the standard set operations defined among the constants, it is highly desirable to provide a special syntax for set comprehensions since they greatly simplify the definition of relational operators. The syntactic extensions for sets and set comprehensions are described in Figure 3.7. Although we will not provide formal evaluation rules for the standard set operations (we assume that they are provided by some external language), we will nevertheless give the translation rules for set comprehension to clarify its meaning. The translation rules (based on the translation rules for list comprehension in Haskell [Jones and et al., 2003]) are presented in Figure 3.8 (the set operation $c_{unionMap}$ is analogous to the list operation *concatMap* in Haskell's standard prelude).

A relation is simply a set of records. However, due to the way sets are typically implemented, there is a technical issue that must be addressed when introducing relations to a programming language. The technical issue, which is somewhat marginal to the current thesis but which nevertheless is one that still merits some attention, is the question of computable equality between values. For example, it is well-known that the problem of comparing two function values for equality is undecidable (a reason why programming language implementations of sets normally cannot handle sets of functions). As a result, only values of types on which equality is defined and computable are allowed as members in sets. Some set implementations go even further

q	::=		*qualifier*
		$x \leftarrow t$	*generator*
		t	*guard*
t	::=		*term*
		...	...
		$\{t, ..., t\}$	*set literal*
		$\{t \mid q, ..., q\}$	*set comprehension*

Figure 3.7: Extra Terms for Sets and Set Comprehension

(*SetComp1*) $\{t_1 \mid True\} \rightsquigarrow \{t_1\}$

(*SetComp2*) $\{t_1 \mid q_1\} \rightsquigarrow \{t_1 \mid q_1, True\}$

(*SetComp3*) $\{t_1 \mid x \leftarrow t_2, q_1, ..., q_n\} \rightsquigarrow c_{unionMap}\ (\lambda x.\{t_1 \mid q_1, ..., q_n\})\ t_2$

(*SetComp4*) $\{t_1 \mid t_2, q_1, ..., q_n\} \rightsquigarrow$ **if** t_2 **then** $\{t_1 \mid q_1, ..., q_n\}$ **else** $\{\}$

Figure 3.8: Translation Rules for Set Comprehension

when, for reasons of efficiency, they require that only values of types with an *ordering* defined on them are to be used as members of sets. It follows that record values in a relation must be comparable to each other, at least for equality. For example, in [Buneman and Ohori, 1996] the authors achieved this by restricting field types to *description* types of ML (types for which equality is defined) and then defining equality between records recursively as equality on fields.

While acknowledging the problem, we would rather not provide a complete solution for it inside the framework of the current thesis. This is simply because doing so would require the bringing up of a large scaffolding, in the form of various advanced programming language features, that would be completely extraneous to our presentation. All the same, we envision that in a practical implementation, something along the lines of Haskell's type class system [Jones and et al., 2003] could be used to define records as members of the standard *Ord* type class (types with ordering defined on them), with the understanding, of course, that only records whose field types are themselves in *Ord* would be members of *Ord*. Using type classes would also have the advantage that records with function values in their fields would not be outlawed, unless, of course, we tried to use them in relations.

3.4.1 Relation Headings

The definition of certain relational operators requires access to the headings of their operands. Since the heading of a relation is simply the heading of one of its records, it might seem an easy task to get the heading of a relation. The problem is caused by empty relations, since there is no member record to provide the heading of the relation. The solution is to provide access to the *type* of the relation through a controlled form of type reflection. The operation *relation heading* accesses this type and returns a record whose heading is the same as that of the relation in question. We will use the notation r^* to denote the heading of relation r. Readers with a keen eye for detail must have noticed that this kind of type reflection implies that relation values must carry with them their type information during execution. In fact, this is what we propose, using, again, Haskell's type classes to provide the necessary scaffolding.

3.5 Defining Relational Algebra Operators

In this section we demonstrate the expressive power of the chosen set of basic record operations by providing definitions for standard (and also for some non-standard) relational operators. The standard set operations of relational algebra (*union*, *intersection*, and *difference*) are considered to be defined among the constants of the language so we omit their definitions (with the exception of *Cartesian product* which we define explicitly). In order to help understanding, some of the more involved definitions are introduced with some brief explanatory remarks.

- *Cartesian Product* The definition of this operator is straightforward. Notice though, that the application of the record concatenation operator requires that the relational operands be disjoint:

$$times \equiv \lambda r.\lambda s.\{\langle ts \parallel tr\rangle \mid ts \leftarrow s,\ tr \leftarrow r\}$$

- *Restriction* The function parameter f, representing the restriction criteria, must return a boolean value when applied to a record from the relation r:

$$restrict \equiv \lambda f.\lambda r.\{tr \mid tr \leftarrow r, f\ tr\}$$

- *Projection* The record parameter 'h' represents the heading on which we want project the relation r. The application of record projection guarantees that h is actually a subset of the heading of r:

$$project \equiv \lambda h.\lambda r.\{tr \cdot [h] \mid tr \leftarrow r\}$$

- *Join* The local variable h represents the intersection of the headings of relations r and s. This common set of attributes is then used to match pairs of records from both relations. By the way, this comparison for equality also implies that both relations should have the same types for their common attributes:

$$join \equiv \lambda r.\lambda s.\textbf{let } h = \langle r^* \backslash \langle r^* \backslash s^* \rangle \rangle \textbf{ in}$$
$$\{\langle tr \parallel \langle ts \backslash tr \rangle \rangle \mid ts \leftarrow s,\ tr \leftarrow r,\ tr \cdot [h] == ts \cdot [h]\}$$

- *Division* The local variable h represents the unique set of attributes of relation r (its computation also enforces the constraint that the heading of s must be a subset of r). Otherwise, the computation of division is based directly on its text book definition:

$$divide \equiv \lambda r.\lambda s.\textbf{let } h = r^*![s^*] \textbf{ in}$$
$$\textbf{let } w_1 = project\ h\ r \textbf{ in}$$
$$\textbf{let } w_2 = times\ w_1\ s \textbf{ in}$$
$$\textbf{let } w_3 = project\ h\ (minus\ w_2\ r) \textbf{ in } minus\ w_1\ w_3$$

- *All But ...* Using the terminology of [Date and Darwen, 1998], we will name the operation of projecting *away* a set of attributes from a relation as *allbut*. The set of attributes projected away must form a subset of the input relation:

$$allbut \equiv \lambda h.\lambda r.\{tr![h] \mid tr \leftarrow r\}$$

- *Composition* The definition of this operator is straightforward. We simply project away the common attributes from the result of the join:

$$compose \equiv \lambda r.\lambda s.\textbf{let } h = \langle r^* \backslash \langle r^* \backslash s^* \rangle \rangle \textbf{ in } allbut\ h\ (join\ r\ s)$$

- *Small Divide* The definition of this ternary operator is lifted almost verbatim from [Date and Darwen, 1998]:

$$
\begin{aligned}
&\mathit{small} \equiv \lambda q.\lambda r.\lambda s. \\
&\quad \textbf{let } w_1 = \mathit{times}\ q\ r \textbf{ in} \\
&\quad \textbf{let } w_2 = \mathit{minus}\ w_1\ s \textbf{ in } \mathit{minus}\ q\ (\mathit{project}\ q^*\ w_2) \textbf{ in}
\end{aligned}
$$

- *Aggregation* In order to calculate various summaries over relations, we introduce the *groupby* operator that partitions a given relation r by grouping records that match on the supplied heading h. A user-supplied aggregation function g is then applied to each partition to calculate summaries (typical aggregation functions are *count*, *sum*, *average*, *minimum*, and *maximum*):

$$
\begin{aligned}
&\mathit{groupby} \equiv \lambda g.\lambda h.\lambda r. \\
&\quad \textbf{let } p = \lambda v.\ \{u![h] \mid u \leftarrow r,\ u \cdot [h] == v\} \\
&\quad \textbf{in } \{g\ t\ (p\ t) \mid t \leftarrow \mathit{project}\ h\ r\}
\end{aligned}
$$

3.5.1 Sample Relational Algebra Queries

Having defined relational operators, we put them to use in this section to form complex queries over a toy relational database. We show the queries together with their results. The sample database consists of three relations: the relation *emps* for describing employees, the relation *depts* for describing departments, and the relation *projs* for describing projects. Their definitions

are as follows:

$$
\begin{aligned}
&\mathit{depts} \equiv \{ \\
&\quad \langle \mathit{dname} = \text{"}\mathit{CSE}\text{"},\ \mathit{deptno} = 1 \rangle, \\
&\quad \langle \mathit{dname} = \text{"}\mathit{PHY}\text{"},\ \mathit{deptno} = 3 \rangle, \\
&\} \\
&\mathit{emps} \equiv \{ \\
&\quad \langle \mathit{ename} = \text{"}\mathit{Smith}\text{"},\ \mathit{age} = 34,\ \mathit{deptno} = 1,\ \mathit{empno} = 1 \rangle, \\
&\quad \langle \mathit{ename} = \text{"}\mathit{Jones}\text{"},\ \mathit{age} = 28,\ \mathit{deptno} = 3,\ \mathit{empno} = 2 \rangle, \\
&\quad \langle \mathit{ename} = \text{"}\mathit{Adams}\text{"},\ \mathit{age} = 42,\ \mathit{deptno} = 3,\ \mathit{empno} = 3 \rangle \\
&\} \\
&\mathit{projs} \equiv \{ \\
&\quad \langle \mathit{pname} = \text{"}\mathit{Laser}\text{"},\ \mathit{empno} = 1 \rangle, \\
&\quad \langle \mathit{pname} = \text{"}\mathit{Robot}\text{"},\ \mathit{empno} = 3 \rangle, \\
&\quad \langle \mathit{pname} = \text{"}\mathit{Robot}\text{"},\ \mathit{empno} = 1 \rangle \\
&\}
\end{aligned}
$$

- Find the names of employees who are under 40:

$$
\begin{aligned}
&\mathit{q1} \equiv \textbf{let}\ c = \lambda t.t \cdot \mathit{age} < 40 \\
&\qquad \textbf{in}\ \mathit{project}\ \langle \mathit{ename} \rangle\ (\mathit{restrict}\ c\ \mathit{emps}) \\
&\mathit{q1} \quad \leadsto \quad \{\langle \mathit{ename} = \text{"}\mathit{Jones}\text{"} \rangle,\ \langle \mathit{ename} = \text{"}\mathit{Smith}\text{"} \rangle\}
\end{aligned}
$$

- Find the names of employees who work in the Physics department:

$$q2 \equiv \textbf{let}\ c = \lambda t.t \cdot dname == \text{"}PHY\text{"}$$
$$\textbf{in}\ project\ \langle ename \rangle\ (restrict\ c\ (join\ emps\ depts))$$
$$q2 \quad \leadsto \quad \{\langle ename = \text{"}Jones\text{"} \rangle,\ \langle ename = \text{"}Adams\text{"} \rangle\}$$

- Find the names of employees who work on all projects:

$$q3 \equiv \textbf{let}\ empnos = divide\ projs\ (project\ \langle pname \rangle\ projs)$$
$$\textbf{in}\ project\ \langle ename \rangle\ (join\ emps\ empnos)$$
$$q3 \quad \leadsto \quad \{\langle ename = \text{"}Smith\text{"} \rangle\}$$

- Count the number of employees per department (we assume that the function *size* tells the size of a given set):

$$q4 \equiv \textbf{let}\ cnt = \lambda t.\lambda s.\langle count = size\ s \mid t \rangle$$
$$\textbf{in}\ project\ \langle dname,\ count \rangle\ (join\ depts\ (groupby\ cnt\ \langle deptno \rangle\ emps))$$
$$q4 \quad \leadsto \quad \{\langle dname = \text{"}PHY\text{"},\ count = 2 \rangle,\ \langle dname = \text{"}CSE\text{"},\ count = 1 \rangle\}$$

- Find the names of employees who are not assigned to any projects at all:

$$q5 \equiv \textbf{let}\ empnos = minus\ (project\ \langle empno \rangle\ emps)\ (project\ \langle empno \rangle\ projs)$$
$$\textbf{in}\ project\ \langle ename \rangle\ (join\ emps\ empnos)$$
$$q5 \quad \leadsto \quad \{\langle ename = \text{"}Jones\text{"} \rangle\}$$

Chapter 4

Type System

In this chapter, we present the type system of the language and provide types for the basic record operations. The type system we develop here is an application of the theory of qualified types by Jones [Jones, 1992] and is a direct extension of a system for extensible records and variants [Gaster and Jones, 1996] using *rows* [Wand, 1987] to describe record types. The theory of qualified types extends the standard Milner type system [Damas and Milner, 1982] of polymorphic types with *predicates* on types to *constrain* (qualify) the possible instantiations of type variables. It also comes with a nice formal development, typing rules, type inference algorithm, and proofs of soundness for the main components. The presentation of the type system follows similar sections from [Gaster and Jones, 1996; Leijen, 2004].

4.1 Kinds

We will use the kind system to distinguish between different kinds of type constructors and to ensure that types are well-formed. The set of kinds is defined with the following grammar:

$$
\begin{array}{lll}
\kappa ::= * & & \textit{the kind of all types} \\
\mid \mathsf{row} & & \textit{the kind of all rows} \\
\mid \kappa \to \kappa & & \textit{arrow kinds}
\end{array}
$$

Terms of the language have types of kind $*$. Arrow kinds are used in type constructors like polymorphic sets or lists.

4.2 Types

We define the set C^κ of type constructors of kind κ as:

$$\begin{array}{rcll} C^\kappa & ::= & \chi^\kappa & \textit{constants} \\ & | & \alpha^\kappa & \textit{variables} \\ & | & C^{\kappa' \to \kappa}\ C^{\kappa'} & \textit{applications} \end{array}$$

Types are now defined simply as $\tau ::= C^*$. We assume the following initial set of type constructors is defined (presented in distfix notation):

$$\begin{array}{rcll} \textit{Int} & ::: & * & \textit{integers} \\ \textit{Bool} & ::: & * & \textit{booleans} \\ \textit{Unit} & ::: & * & \textit{unit} \\ \{\_\} & ::: & * \to * & \textit{set type} \\ \to & ::: & * \to * \to * & \textit{function space} \\ (\!|\,|\!) & ::: & \mathsf{row} & \textit{empty row} \\ (\!|\, l : \_ \mid \_ \,|\!) & ::: & * \to \mathsf{row} \to \mathsf{row} & \textit{row extension} \\ \textit{Rec} & ::: & \mathsf{row} \to * & \textit{record type} \end{array}$$

Types are well-formed only when type constructors are fully applied (we forbid partial application of type constructors) to arguments of correct kinds. For example:

- The result of applying the type constructor $\to$ of kind $* \to * \to *$ to types *Int* of kind $*$ and *Bool* of kind $*$ is $\textit{Int} \to \textit{Bool}$. In general, function types are constructed by applying

the function type constructor $\rightarrow$ to arguments τ_1 and τ_2, both of kind $*$. Also, instead of writing $\rightarrow\ \tau_1\ \tau_2$ for the result of the application, it is customary to use the infix notation $\tau_1 \rightarrow \tau_2$ (where the $\rightarrow$ operator associates to the right).

- The result of applying the type constructor *Rec* of kind $\mathsf{row} \rightarrow *$ to the empty row $(\!|\,|\!)$ of kind row is the record type *Rec* $(\!|\,|\!)$ of kind $*$.

- The result of applying the type constructor *row extension* to a type τ of kind $*$ and a row r of kind row is the row $(\!|l : \tau \mid r|\!)$. Since there are no first-class labels in the language, we are actually talking about a family of type constructors, one for each possible label. Notice, that the syntactic construction of rows permits the building of ill-formed rows (rows with duplicate labels), something that we will have to prevent using type predicates.

4.3 Type-level Operations and Relations

In this section we introduce some type-level operations and relations on rows that are needed in the rest of the thesis, some of which has already appeared elsewhere [Gaster and Jones, 1996; Leijen, 2004]. An important property of the operations and relations presented in this section is that they all are simple—they have polynomial complexity. In the following definitions, as in the rest of the thesis, the meta variable *r* stands for type expressions of kind *row*.

The syntactic order of labels is irrelevant in deciding equality between rows, formally:

$$(\!|l_1 : \tau_1 \mid (\!|l_2 : \tau_2 \mid r|\!)|\!) \quad = \quad (\!|l_2 : \tau_2 \mid (\!|l_1 : \tau_1 \mid r|\!)|\!)$$

Label deletion removes a label l from a row:

$$\begin{aligned} (\!|l : \tau \mid r|\!) - l &= r \\ (\!|l_1 : \tau_1 \mid r|\!) - l_2 &= (\!|l_1 : \tau_1 \mid r - l_2|\!) \end{aligned}$$

Row concatenation is defined as:

$$\begin{aligned} r \parallel (\!|\,|\!) &= r \\ r_1 \parallel (\!|l : \tau \mid r_2|\!) &= (\!|l : \tau \mid r_1|\!) \parallel r_2 \end{aligned}$$

Difference of rows is defined as:

$$\begin{aligned} r \backslash (\!|\,|\!) &= r \\ r_1 \backslash (\!|l : \tau \mid r_2|\!) &= (r_1 - l) \backslash r_2 \end{aligned}$$

Intersection is thus simply $r_1 \cap r_2 \equiv r_1 \backslash (r_1 \backslash r_2)$. Notice though, that due to the way row difference is defined, row intersection *is not* commutative: the field types of the resulting row are ‘coming from’ the first operand.

Much like [Gaster and Jones, 1996] we define a *membership* relation $(l : \tau)$ **in** r which holds if the label l with type τ appears in row r:

$$(l : \tau) \textbf{ in } (\!|l : \tau \mid r|\!) \qquad \frac{(l_1 : \tau_1) \textbf{ in } r \quad l_1 \neq l_2}{(l_1 : \tau_1) \textbf{ in } (\!|l_2 : \tau_2 \mid r|\!)}$$

π ::=		*predicate*
	$r \simeq r$	*row equality*
	r **lacks** l	*lacks (for each label l)*
	r **has** $(l : \tau)$	*has (for each label l)*
	$r\#r$	*disjoint*
	$r \leq r$	*subset*

Figure 4.1: Syntax of Type Predicates

4.4 Type Predicates and Qualified Types

Basic record operations are well-defined only if the types of the operands satisfy certain constraints. Following the theory of qualified types [Jones, 1992] we will use *predicates* on types to capture these constraints. The syntax of predicates is presented in Figure 4.1.

The predicate *row equality* is separated by a line from the rest for a good reason: all other predicates (serving only as notational shorthands) can be expressed in terms of *row equality* . Why include these 'derived' predicates then? The answer is that they make for more readable (programmer-friendly) predicates, and, more importantly, they are of great help in type simplification, as we will see in Chapter 6. An explanation of the meaning of predicates follows:

- The *row equality* predicate requires two rows to be equal, up to permutation of labels.
- The *lacks* predicate requires the absence of a label from a row.
- The *has* predicate requires the presence of a label with a specific type in a row.

Predicate	**Shorthand**	**Expansion**
lacks	r **lacks** l	$r \simeq r - l$
has	r **has** $(l : \tau)$	$(\!\|l : \tau \mid (\!\|\|\!)\|\!) \simeq r \backslash (r - l)$
disjoint	$r_1 \# r_2$	$r_1 \simeq r_1 \backslash r_2$
subset	$r_1 \leq r_2$	$(\!\|\|\!)\| \simeq r_1 \backslash r_2$

Table 4.1: Expansion of Derived Predicates

- The *disjoint* predicate requires two rows to have disjoint headings.

- The *subset* predicate requires the heading of one row to be a subset of the heading of some other row.

The formal semantics of predicates is defined by the entailment relation in Figure 4.2. We only have to formalize the *row equality* predicate, since all other predicates are just notational shorthands that expand to *row equality* predicates, as defined in Table 4.1. Notice how we make use of type-level operations and relations on rows, introduced in Section 4.3, to define expansions for derived predicates and also to formalize *row equality* in the entailment relation. The definition of *row equality* is based on that in [Gaster and Jones, 1996].

A derivation of predicate π from a finite set of predicates P, written as $P \Vdash \pi$, proves that when all predicates in P hold, then the predicate π must hold as well. The entailment relation naturally extends to finite sets of predicates, that is, when all predicates in a finite set P hold, then all predicates in the finite set Q must hold as well, if there is a derivation from P for each

$$(\textit{taut}) \quad \frac{\pi \in P}{P \Vdash \pi}$$

$$(\textit{rowEqEmpty}) \quad P \Vdash (\!|\,|\!) \simeq (\!|\,|\!)$$

$$(\textit{rowEqVar}) \quad P \Vdash \rho \simeq \rho$$

$$(\textit{rowEqHead}) \quad \frac{(l : \tau)\ \mathbf{in}\ r_1 \quad P \Vdash r_2 \simeq (r_1 - l)}{P \Vdash r_1 \simeq (\!|l : \tau \mid r_2|\!)}$$

Figure 4.2: Entailment Relation

predicate in Q. Formally: $P \Vdash Q \equiv \forall \pi \in Q.P \Vdash \pi$.

The theory of qualified types allows us to use any set of predicates, *as long as* we can prove that the entailment relation is monotone, transitive, and closed under substitution:

Theorem 4.4.1. *The entailment relation in Figure 4.2 is:*

1. *monotone:* $Q \subseteq P \Rightarrow P \Vdash Q$,

2. *transitive:* $P \Vdash T \wedge T \Vdash Q \Rightarrow P \Vdash Q$, *and*

3. *closed under substitution:* $P \Vdash Q \Rightarrow sP \Vdash sQ$.

Now we are in the position to introduce *qualified types*, types that are qualified by a set of type predicates. Like in [Gaster and Jones, 1996], we distinguish between types, τ, described in Section 4.2, and type schemes, σ (types with universally quantified type variables). Formally:

$$\varphi \quad ::= \quad \tau \quad | \quad \pi \Rightarrow \varphi \qquad \textit{qualified types}$$

$$\sigma \quad ::= \quad \varphi \quad | \quad \forall \alpha.\sigma \qquad \textit{type schemes}$$

Since the order of predicates and quantified type variables does not matter we introduce the following abbreviations:

$$\pi_1 \Rightarrow ... \Rightarrow \pi_n \Rightarrow \tau \quad \equiv \quad \overline{\pi} \Rightarrow \tau \;\equiv\; P \Rightarrow \tau$$

$$\forall \alpha_1 .... \forall \alpha_n . \varphi \qquad \equiv \quad \forall \overline{\alpha}.\varphi$$

Universal quantification, hence polymorphism, is restricted by constraints on types, captured by the set of predicates P.

Like in [Leijen, 2004], for qualified types to be well-formed, we require that only row variables or the empty row appear in types and not arbitrary row expressions. This is an important technical necessity, because the unification algorithm, used during type inference, does not know how to unify arbitrary row expressions (like row concatenation). Row expressions thus appear only in type predicates. For example, instead of writing something like $Rec\ (\rho_1 \parallel \rho_2)$, we have to write $\rho_3 \simeq (\rho_1 \parallel \rho_2) \Rightarrow Rec\ \rho_3$.

Operation	**Surface Syntax**	**Core Syntax**
Empty Record	$\langle\rangle$	$c_{\langle\rangle}$
Record Extension	$\langle l = e_1 \mid e_2\rangle$	$((c_{l=}\ e_1)\ e_2)$
Field Selection	$(e \cdot l)$	$(c_{.l}\ e)$
Record Concatenation	$\langle e_1 \parallel e_2\rangle$	$((c_{\parallel}\ e_1)\ e_2)$
Record Difference	$\langle e_1 \backslash e_2\rangle$	$((c_{\backslash}\ e_1)\ e_2)$
Field Deletion	$(e!l)$	$(c_{!l}\ e)$
Record Projection	$(e_1 \cdot [e_2])$	$((c_{[]}\ e_1)\ e_2)$

Table 4.2: Translations of Record Operations to Core Syntax

4.5 Typing Basic Record Operations

The theory of qualified types assumes that the term language is core-ML, that is, an implicitly-typed lambda calculus with let-bound polymorphism. Thus, the special syntactic constructs for records and record operations have to be translated to core syntax before type-checking can take place. The translation rules are presented in Table 4.2, with the understanding that record literals are treated as series of record extensions, that is, where each record literal $\langle l_1 = e_1, ..., l_n = e_n\rangle$ is treated as the equivalent expression $\langle l_1 = e_1 \mid ...\langle l_n = e_n \mid \langle\rangle\rangle...\rangle$.

There is one constant function corresponding to each record operation in the translation, and it is the types of these constants that will capture the meaning of record operations for type-checking purposes. Notice, that during the translation, record labels disappear from the syntax of the language as they get incorporated into the names of the constants corresponding to the

Operation (Constant)	**Predicates**	**Type**
Empty Record $(c_{\langle\rangle})$		$Rec\ (\!\|\|\!)$
Record Extension $(c_{l=})$	$\rho_2 \simeq (\!\|l : \alpha \mid \rho_1\|\!),\ \rho_1$ **lacks** l	$\alpha \to Rec\ \rho_1 \to Rec\ \rho_2$
Field Selection $(c_{.l})$	ρ_1 **has** $(l : \alpha)$	$Rec\ \rho_1 \to \alpha$
Record Concatenation $(c_{\|\|}$	$\rho_3 \simeq \rho_1 \,\|\|\, \rho_2,\ \rho_1 \# \rho_2$	$Rec\ \rho_1 \to Rec\ \rho_2 \to Rec\ \rho_3$
Record Difference $(c_{\backslash})$	$\rho_3 \simeq \rho_1 \backslash \rho_2$	$Rec\ \rho_1 \to Rec\ \rho_2 \to Rec\ \rho_3$
Field Deletion $(c_{!l})$	ρ_1 **has** $(l : \alpha),\ \rho_2 \simeq \rho_1 - l$	$Rec\ \rho_1 \to Rec\ \rho_2$
Record Projection $(c_{[]})$	$\rho_3 \simeq \rho_1 \cap \rho_2,\ \rho_2 \leq \rho 1$	$Rec\ \rho_1 \to Rec\ \rho_2 \to Rec\ \rho_3$

Table 4.3: Qualified Types for the Basic Record Operations

record operations. This clarifies what we meant when we said that record operations referring to record labels form a *family* of operations (one for each label).

The qualified type schemes (predicates together with types) for the constants representing the basic record operations, introduced by the translation to core syntax, are presented in Table 4.3 (all type variables in the predicates and in the types are assumed to be universally quantified.) Notice how, in the type predicates for basic operations, the construction of rows at the type level mirrors the construction of records at the value level. This property of the predicates will play an important part in checking the satisfiability of predicates in Chapter 5.

4.6 Typing Rules and Type Inference

In this section we present the typing rules and the type inference algorithm *W* for the theory of qualified types as developed by Jones [Jones, 1992]. A type assignment A is a finite map from term variables x to type schemes σ:

$$A ::= \{x_1 : \sigma_1, ..., x_n : \sigma_n\}$$

The notation A_x stands for the type assignment A with the assumption on x removed. We abbreviate $A_x \cup \{x : \sigma\}$ as $A_x, x : \sigma$. Figure 4.3 shows the standard typing rules for the theory of qualified types (*ftv* returns the set of free type variables). The typing judgement $P \mid A \vdash e : \sigma$ asserts that the expression e has type scheme σ when the predicates P are satisfied and types of the free variables in e are given by the type assignment A. Notice that the typing rules treat sets of predicates as 'black boxes', that is, the type inference algorithm is neutral towards the concrete system of predicates being used. This 'pluggability' of the system is its greatest advantage since the same set of typing rules (and type inference algorithm) can be used without modification in different contexts.

4.6.1 Substitution and Unification

A substitution is a map from type variables to type constructors, and it is the identity function for all but a finite set of variables. We restrict ourselves to *kind preserving* substitutions: that is, to substitutions that map type variables to type constructors of the same kind. For a substitution that maps a type variable α to a type constructor C we will write $[\alpha \mapsto C]$, for the identity substitution we will write *id* and for the composition of substitutions s and t we will write st.

$$(const) \qquad P \mid A \vdash c : \sigma_c$$

$$(var) \qquad \frac{(x : \sigma) \in A}{P \mid A \vdash x : \sigma}$$

$$(\rightarrow E) \qquad \frac{P \mid A \vdash e : \tau' \rightarrow \tau \quad P \mid A \vdash e' : \tau'}{P \mid A \vdash e\ e' : \tau}$$

$$(\rightarrow I) \qquad \frac{P \mid A_x, x : \tau' \vdash e : \tau}{P \mid A \vdash \lambda x.e : \tau' \rightarrow \tau}$$

$$(\Rightarrow E) \qquad \frac{P \mid A \vdash e : \pi \Rightarrow \varphi \quad P \Vdash \pi}{P \mid A \vdash e : \varphi}$$

$$(\Rightarrow I) \qquad \frac{P \cup \{\pi\} \mid A \vdash e : \varphi}{P \mid A \vdash e : \pi \Rightarrow \varphi}$$

$$(\forall E) \qquad \frac{P \mid A \vdash e : \forall\alpha.\sigma}{P \mid A \vdash e : [\alpha \mapsto \tau]\sigma}$$

$$(\forall I) \qquad \frac{P \mid A \vdash e : \sigma \quad \alpha \notin ftv(A) \cup ftv(P)}{P \mid A \vdash e : \forall\alpha.\sigma}$$

$$(let) \qquad \frac{P \mid A \vdash e : \sigma \quad Q \mid A_x, x : \sigma \vdash e' : \tau}{P \cup Q \mid A \vdash \textbf{let}\ x = e\ \textbf{in}\ e' : \tau}$$

Figure 4.3: Typing Rules

A substitution s is called a *unifier* of type constructors C_1 and C_2 if $sC_1 = sC_2$. A unifier s is a *most general unifier* if for all unifiers u of constructors C_1 and C_2 there exists a substitution

$$C \overset{id}{\sim} C \qquad \frac{\alpha \notin ftv(C)}{\alpha \overset{[\alpha \mapsto C]}{\sim} C} \qquad \frac{\alpha \notin ftv(C)}{C \overset{[\alpha \mapsto C]}{\sim} \alpha}$$

$$\frac{C_1 \overset{u}{\sim} D_1 \quad uC_2 \overset{u'}{\sim} uD_2}{C_1C_2 \overset{u'u}{\sim} D_1D_2}$$

Figure 4.4: Kind Preserving Unification

t such that $u = ts$. Since types are represented as Herbrand terms we can use the standard kind preserving unification algorithm, as shown in Figure 4.4, to calculate most general unifiers. We write $C_1 \overset{u}{\sim} C_2$ for the most general unifier u of C_1 and C_2. The following theorem is by Robinson [Robinson, 1965]:

Theorem 4.6.1. *The algorithm in Figure 4.4 returns a most general unifier if it exists. It fails precisely when there's no such unifier.*

With the given unification algorithm we can directly use the type inference algorithm W for qualified types by Jones [Jones, 1992]. For completeness we include the type inference algorithm in Figure 4.5. The type inference rules can be interpreted as an attribute grammar where each judgment of the form $P \mid sA \vdash^W e : \tau$ is a semantic rule where assignment A and expression e are inherited, while the predicates P, the substitution s, and the type τ are synthesized. The algorithm W calculates principal types with respect to the typing rules presented in Figure 4.3.

In the (let^W) rule, we use a generalization function that quantifies free variables not present in the type assignment:

$$gen(A, \varphi) = \forall \overline{\alpha}.\varphi \quad \text{where } \overline{\alpha} = ftv(\varphi) \backslash ftv(A)$$

$$(var^W) \quad \frac{(x : \forall \overline{\alpha}.P \Rightarrow \tau) \in A \qquad fresh(\overline{\beta}) \quad s = [\overline{\alpha \mapsto \beta}]}{sP \mid A \vdash^W x : s\tau}$$

$$(\rightarrow E^W) \quad \frac{P \mid sA \vdash^W e : \tau \quad Q \mid tsA \vdash^W e' : \tau' \qquad (t\tau) \overset{u}{\sim} (\tau' \rightarrow \alpha) \quad fresh(\alpha)}{u(tP \cup Q) \mid utsA \vdash^W e\ e' : u\alpha}$$

$$(\rightarrow I^W) \quad \frac{P \mid sA_x, x : \alpha \vdash^W e : \tau \quad fresh(\alpha)}{P \mid sA \vdash^W \lambda x.e : s\alpha \rightarrow \tau}$$

$$(let^W) \quad \frac{P \mid sA \vdash^W e : \tau \quad \sigma = gen(sA, P \Rightarrow \tau) \qquad Q \mid tsA_x, x : \sigma \vdash^W e' : \tau'}{Q \mid tsA \vdash^W \textbf{let}\ x = e\ \textbf{in}\ e' : \tau'}$$

Figure 4.5: Type Inference Algorithm W

Jones proves that the type inference algorithm W in Figure 4.5 is both sound and complete with respect to the typing rules in Figure 4.3 [Jones, 1992]:

Theorem 4.6.2. *The algorithm in Figure 4.5 infers a principal type for a given expression e and assignment A. It fails precisely when no type exists for e under the assignment A.*

4.7 Examples of Inferred Types

To give a taste of the type system, in this section we provide some expressions along with their inferred types (separated from the expression with a single line). As the type inference algorithm

does not check the satisfiability of predicates (a problem, addressed in detail in the next chapter) some expression, though incorrect, do have types inferred for them. More complex expressions (like *join*) also demonstrate how unwieldy inferred types can grow and thus the need for type improvement and type simplification.

1. Selecting a field from a record literal:

$$\frac{\langle a = 2,\ b = True\rangle \cdot a}{\begin{array}{l} P = \rho_1 \textbf{ has } (a : Int),\ \rho_2 \textbf{ lacks } b,\ \rho_1 \simeq (\!|b : Bool \mid \rho_2|\!),\ (\!|\,|\!) \textbf{ lacks } a,\ \rho_2 \simeq (\!|a : Int \mid (\!|\,|\!)|\!) \\ \tau = Int \end{array}}$$

2. Concatenating two records then extending the result:

$$\frac{\lambda x.\lambda y.\langle a = 7 \mid \langle x \parallel y\rangle\rangle}{\begin{array}{l} P = \rho_1 \textbf{ lacks } a,\ \rho_2 \simeq (\!|a : Int \mid \rho_1|\!),\ \rho_1 \simeq (\rho_3 \parallel \rho_4),\ \rho_3 \# \rho_4 \\ \tau = Rec\ \rho_3 \rightarrow (Rec\ \rho_4 \rightarrow Rec\ \rho_2) \end{array}}$$

3. A quite complex expression: *Join*

$$\frac{\begin{array}{l} join \equiv \lambda r.\lambda s.\textbf{let } h = \langle r^* \backslash \langle r^* \backslash s^*\rangle\rangle \textbf{ in} \\ \quad \{\langle tr \parallel \langle ts \backslash tr\rangle\rangle \mid ts \leftarrow s,\ tr \leftarrow r,\ tr \cdot [h] == ts \cdot [h]\} \end{array}}{\begin{array}{l} P = \rho_4 \simeq (\rho_1 \backslash \rho_8),\ \rho_8 \simeq (\rho_1 \backslash \rho_2),\ \rho_5 \simeq (\rho_1 \backslash \rho_6),\ \rho_6 \simeq (\rho_1 \backslash \rho_4),\ \rho_4 \leq \rho_1,\ \rho_4 \simeq (\rho_1 \backslash \rho_8), \\ \quad \rho_8 \simeq (\rho_1 \backslash \rho_2),\ \rho_5 \simeq (\rho_2 \backslash \rho_7),\ \rho_7 \simeq (\rho_2 \backslash \rho_4),\ \rho_4 \leq \rho_2,\ \rho_4 \simeq (\rho_1 \backslash \rho_8),\ \rho_8 \simeq (\rho_1 \backslash \rho_2), \\ \quad \rho_3 \simeq (\rho_1 \parallel \rho_9),\ \rho_1 \# \rho_9,\ \rho_9 \simeq (\rho_2 \backslash \rho_1) \\ \tau = \{Rec\ \rho_1\} \rightarrow \{Rec\ \rho_2\} \rightarrow \{Rec\ \rho_3\} \end{array}}$$

Chapter 5

Checking Satisfiability of Predicates

The type inference algorithm *W* ignores the satisfiability of predicates, a serious challenge since the power of the language derives in part from the expressiveness of type predicates, hence, we begin the chapter with formalizing the notion of satisfiability and stating that the problem of checking satisfiability is NP-complete both in the number of row variables and in the number of labels. In order to remove exponential complexity in the number of labels, we introduce a language restriction, but argue that the majority of useful programs still remain valid under the restriction. Next, we develop an algorithm (algorithm Q) for checking satisfiability in the restricted system. The algorithm consists of two main parts: (1) a set constraint solver, and (2) a field type constraint solver. Nested records require special treatment, so we follow the general description of algorithm Q with a section on how to handle them. We conclude the chapter with a semi-formal complexity analysis, where we mainly argue for the practical usefulness of the algorithm, despite its exponential complexity.

Let us begin our discussion on satisfiability with a motivating example. Take the following expression:

$$e \quad \equiv \quad \langle a = 1 \rangle \cdot b$$

We are trying to select a field from a record literal that does not possess that field, that is, we

violate the precondition of the *field selection* operation. If we attempted to evaluate the expression e using the evaluation rules presented in Chapter 3, we would get stuck before reaching a value term, hence, the expression e is *ill-typed*. The problem is *not* that *field selection* is applied to something other than a record (it is applied to a record), but rather that the record in question does not have the field b. The standard Milner type system can express the first constraint, that *field selection* must be applied to a record, but it cannot express the second one, that the field being selected must be present in the record. Type predicates are used to capture these additional constraints on types, constraints that cannot be expressed using the Milner type system only. Let's look at the type inferred by the type inference algorithm W for the expression e:

$$\rho_1 \simeq (\!|a : Int \mid (\!|\,|\!)|\!), \ \rho_1 \ \textbf{has} \ (b : \alpha) \ \Rightarrow \ \alpha$$

Since the type inference algorithm is an extension of the standard Damas-Milner one, if the algorithm succeeds in inferring a type for an expression, then none of the constraints expressible in the Milner type system are violated in the expression (a basic property of type inference). Nevertheless, it is *still* possible that some constraints, expressed using type predicates, are violated by the expression in question. In the case of the concrete example, it is easy to see that there is no instantiation of the row variable ρ_1 that would simultaneously satisfy both predicates: that ρ_1 must be equal to $(\!|a : Int \mid (\!|\,|\!)|\!)$ and that ρ_1 must have the field b. In other words, the type predicates inferred for the expression e are *unsatisfiable* (a notion we formalize later). Informally, when a set of predicates is unsatisfiable it means that the preconditions of some basic record operations are violated (and vice versa).

In the rest of the chapter, we formalize the notion of satisfiability of predicates, propose a

language restriction to eliminate one source of exponential complexity in checking satisfiability, describe the algorithm Q for checking the satisfiability of predicates, and analyze the complexity of the proposed algorithm.

5.1 Definition of Satisfiability

In this section we formalize the notion of satisfiability of predicates and prove that the problem of checking satisfiability is NP-complete. We define the satisfiability of a set of predicates P as:

$$\textbf{sat}\ P \ \text{ iff } \ \exists u.u \in \{s \mid s \in Subst,\ \Vdash sP\}$$

That is, a set of predicates P is satisfiable if there is at least one instantiation of row variables that makes all predicates in the set true (that is, entailed by the entailment relation).

Theorem 5.1.1. *Given a term e, where* $P \mid sA \vdash^W e : \tau$*, the problem* **sat** P *is NP-complete.*

Even more importantly, we observe, based on the reductions used in the proof, that the decision procedure is exponential *both* in the number of row variables *and* in the number of labels mentioned in P.

We could now add the requirement **sat** P to the (let^W) rule of the type inference algorithm W to ensure satisfiability of predicates for inferred types. This extension of the algorithm, necessary it might seem, is nevertheless a non-obvious step, since it changes the complexity of type-checking from the Damas-Milner 'quasi-linear' (that is, exponential in theory, but overwhelmingly linear in practice) to NP-complete. However, checking the satisfiability of predicates is polynomial, if all record operations are performed on operands with ground types, which is always the case in expressions that can cause the evaluation of record operations. Exploiting this

fact, it is possible to keep the type-checking algorithm polynomial, by accepting programs with type errors that are guaranteed not to interfere with evaluation, an approach taken both in [Buneman and Ohori, 1996] and [Makholm and Wells, 2005]. We do not side-step the exponential complexity caused by type-checking polymorphic record operations, rather, by introducing certain restrictions, we offer a system that is almost as expressive as the unrestricted system, and though still NP-complete, we argue that exponential complexity rarely arises in practice.

5.2 Mapping to Set Expressions

Rows describe sets of labels and predicates on rows can be thought of as set constraints. We would like to formalize this intuition by introducing a mapping ϕ from row expressions to set expressions. The language of set expressions that is used as the target of the mapping is:

$$e ::= \varnothing \mid \{l\} \mid x \mid (e \cup e) \mid (e \cap e) \mid (e \backslash e)$$

With the above definition in mind, we define the mapping ϕ from row expressions to set expressions inductively (notice the convention of mapping row variable ρ_i to set variable ρ'_i):

$$\begin{aligned}
\phi(\langle\!|\rangle) &= \varnothing \\
\phi(\rho_i) &= \rho'_i \\
\phi(r - l) &= \phi(r) \backslash \{l\} \\
\phi(\langle\!| l : \tau \mid r |\!\rangle) &= \{l\} \cup \phi(r) \\
\phi(r_1 \parallel r_2) &= \phi(r_1) \cup \phi(r_2) \\
\phi(r_1 \backslash r_2) &= \phi(r_1) \backslash \phi(r_2)
\end{aligned}$$

The following examples show the results of applying ϕ to various row expressions:

Row Expression	**Set Expression (ϕ)**
$\rho_1 \parallel \rho_2$	$\rho'_1 \cup \rho'_2$
$(\!\|a : Int \mid (\!\|b : Bool \mid \rho_1\|\!)\|\!)$	$\{a\} \cup (\{b\} \cup \rho'_1)$
$\rho_1 \parallel (\!\|a : Int \mid (\!\|\|\!)\|\!)$	$\rho'_1 \cup (\{a\} \cup \varnothing)$
$(\rho_1 - b) \parallel (\rho_1 \backslash (\!\|a : Int \mid \rho_2\|\!))$	$(\rho'_1 \backslash \{b\}) \cup (\rho'_1 \backslash (\{a\} \cup \rho'_2))$

We extend ϕ to substitutions (restricted to row variables):

$$\phi(s) \quad = \quad [\overline{\phi(\rho_i) \mapsto \phi(r_i)}]$$

... and predicates (we treat all derived predicates in their expanded form, hence we only have to handle row equality):

$$\phi(r_1 \simeq r_2) \quad = \quad (\phi(r_1) = \phi(r_2))$$

Thus $\phi(s)$ is a mapping from set variables to set expressions, while $\phi(P)$ is a system of set constraints. An important property of our mapping is expressed by the following lemma:

Lemma 5.2.1. *Under all substitutions s, for a row expression r and a label ℓ, $\ell \in \phi(sr)$ if and only if $(\ell : \tau)$ **in** sr for some τ.*

Informally, what the lemma says is that the mapping ϕ is well-behaved, in the sense that the presence/absence of fields in row expressions is preserved by the mapping.

5.3 A Simplifying Language Restriction

Theorem 5.1.1 asserts that checking the satisfiability of predicates is exponential in the number of labels appearing (mentioned) in the predicates. Our goal in this section is to remove this

source of complexity from the system, for it would be unreasonable to put an *a priori* upper bound on the number of labels appearing in expressions.

One obvious solution would be to require all fields with the same label to have the same type, in the manner of Haskell. Although this would achieve our goal (we do not prove this here), it would make the system too inflexible. From the programmer's point of view, unrelated field selections would be forced to have the same type, greatly reducing the usefulness of the system.

Our solution is to require only those field selections that *are potentially related through record construction* to have the same type. We try to make this restriction clear through the following example, which, though is type-correct under the unrestricted system, violates our restriction:

$$f \quad \equiv \quad \lambda t.\lambda u.\lambda v.\langle t \parallel u\rangle \cdot a == 10 \wedge \langle t \parallel v\rangle \cdot a == \text{True}$$

We require the two field selections to have the same type because they can potentially refer to the same field, as it is the case when t has field a. On the other hand, the following, modified, example passes our restriction, since now t cannot have the field a (we extend t with the field a) so the two field selections are not related:

$$f' \quad \equiv \quad \lambda t.\lambda u.\lambda v.\langle t \parallel u\rangle \cdot a == 10 \wedge \langle t \parallel v\rangle \cdot a == \text{True}$$
$$\wedge \langle a = 7 \mid t\rangle == \langle a = 7, b = 5\rangle$$

Extending t with the field a is just a roundabout way of saying that the record t cannot have a. Naturally, in a full-fledged language with type ascription, there is no need for such workarounds, instead, the programmer could achieve the same effect by simply *ascribing* the intended type to

the given term (in this case t). For example (using Haskell-like syntax):

$$f'' \quad \equiv \quad \lambda t.\lambda u.\lambda v.\langle((t :: \rho \textbf{ lacks } a \Rightarrow Rec\ \rho) \parallel u\rangle \cdot a == 10 \wedge \langle t \parallel v\rangle \cdot a == \text{True}$$

With the above restriction imposed, checking satisfiability of predicates is polynomial in the number of labels. We will show this by presenting an algorithm with the required complexity in the sections that follow. Formally, satisfiability in the restricted system, as captured by the algorithm for satisfiability, relates to satisfiability in the unrestricted system as:

$$\textbf{sat}_{ALG}\ P \ \text{ iff } \ \textbf{sat}\ P \wedge \forall s' \in \lceil P' \rceil . \exists s \in \lceil P \rceil . \phi(s') = \phi(s)$$

where we get P' from P by replacing all occurrences of base types χ^* with type variables α_i^*, fresh for each occurrence. In effect, P' is a form of P, where the field type constraints have been relaxed. Informally, what the definition says is that for each satisfying substitutions for P' there must exist a satisfying substitution for P that assigns the same set of labels to each row variable. Since $\textbf{sat}_{ALG}\ P$ implies **sat** P, but **sat** P does not imply $\textbf{sat}_{ALG}\ P$, the algorithm for satisfiability is sound but not complete with respect to the unrestricted system.

With the restriction imposed, the algorithm still correctly type-checks all examples provided in the thesis, including the definitions of polymorphic relational operators, and a host of useful programs with record and relational operations. Actually, we claim even more. We claim that the restriction, in some sense, actually *improves* the language by rejecting programs that are rarely useful in practice.

5.4 The Algorithm Q

Type predicates can be unsatisfiable, either because there are conflicting constraints on the presence/absence of record fields, or because there are conflicting type constraints on record fields. Thus, we will first check if $\phi(P)$ is satisfiable (since obviously $\neg$**sat** $\phi(P)$ implies $\neg$**sat** P), then, using information gathered from the previous step, we will move on to check field type constraints. Special care needs to be taken to handle nested records, which we will address after discussing the first two phases. The algorithm Q we are about to describe implements the language restriction explained in Section 5.3.

There already exist several decision procedures to solve a system of set constraints [Aiken and Wimmers, 1992; Aiken, 1994], however, none of them has *all* of the properties that we find important. Our algorithm exhibits all of them: (1) mixing set constraint solving with unification, (2) producing a normal form that can be directly used for type simplification and improvement, (3) being compositional, or resumable, that is, adding a new set constraint does not require starting over, (4) having good error-reporting capabilities by clearly identifying conflicting constraints, and (5) exploiting assumptions on external database schemas (explained in Section 5.11).

5.4.1 Pseudo Code for Algorithm Q

The steps of the algorithm for satisfiability are summarized in the following high-level outline:

1. Identify base variables in $\phi(P)$
2. Map row predicates P to set constraints $\phi(P)$
3. Identify independent (base) variables in $\phi(P)$
4. Calculate normal form ($\hat{\psi}$) for set expressions in $\phi(P)$
5. Solve set constraints $\phi(P)$ now in normal form (or **fail**)
6. Build row construction graph G from P
7. Identify connected components in G using output from Step 5
8. Unify field types belonging to the same component (or **fail**)
9. If row variables unify in Step 8, add new set constraints and resume at Step 5, otherwise **finish**

5.5 A Normal Form for Set Expressions

Let $\mathcal{L}$ be the universal set of all labels with *base sets* $H_1, ..., H_n \subseteq \mathcal{L}$. We define 2^n *regions* $R_0, ..., R_{2^n-1}$ by $R_j = \bigcap_i \hat{H}_i$ where $\hat{H}_i = H_i$ if the ith bit is set in the binary representation of j, and $\hat{H}_i = \overline{H_i}$ otherwise. Informally, regions are just the contiguous areas of a Venn diagram, as illustrated in Figure 5.1 and Table 5.1 for two base sets.

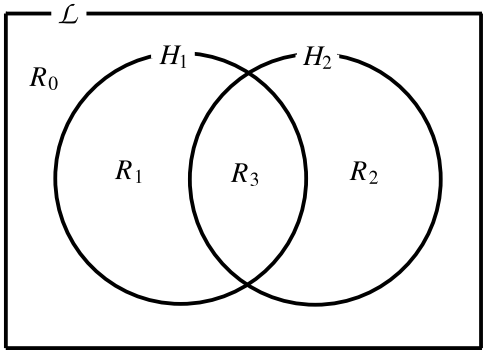

Figure 5.1: Regions Defined by Base Sets H_1 and H_2

$$R_0 = \overline{H_1} \cap \overline{H_2}$$

$$R_1 = H_1 \cap \overline{H_2}$$

$$R_2 = \overline{H_1} \cap H_2$$

$$R_3 = H_1 \cap H_2$$

Table 5.1: Regions Defined by Base Sets H_1 and H_2

The following theorem can be found in many textbooks on set theory:

Theorem 5.5.1. *For base sets $H_1, ..., H_n$ and regions $R_0, ..., R_{2^n-1}$ defined as above:*

1. $\mathcal{L} = \bigcup_i R_i$,

2. *regions are pair-wise disjoint, and*

3. *any set expression on $H_1, ..., H_n$, built using only set union, intersection, complement, and difference, is equivalent to the union of some regions $\bigcup_{i \in I} R_i$.*

An important corollary of the theorem that will be exploited later is that, since regions partition $\mathcal{L}$, each label ℓ belongs to *exactly one* of the regions R_i.

We define a mapping ψ_B from *homogeneous* set expressions (set expressions without label constants) to *provisional normal forms* where B is a set of base sets $H_1, ..., H_n$, and $R_0, ..., R_{2^n-1}$ are regions defined on the base sets:

$$\begin{aligned}
\psi_B(\varnothing) &= \varnothing \\
\psi_B(H_i) &= \textstyle\bigcup_{i\in\mathcal{I}} R_i \;\; \textit{where } \mathcal{I} = \{k \mid R_k \subseteq H_i\} \\
\psi_B(e \cap f) &= \psi_B(e) \cap \psi_B(f) = \textstyle\bigcup_{i\in(\mathcal{I}^e \cap \mathcal{I}^f)} R_i \\
\psi_B(e \cup f) &= \psi_B(e) \cup \psi_B(f) = \textstyle\bigcup_{i\in(\mathcal{I}^e \cup \mathcal{I}^f)} R_i \\
\psi_B(e \setminus f) &= \psi_B(e) \setminus \psi_B(f) = \textstyle\bigcup_{i\in(\mathcal{I}^e \setminus \mathcal{I}^f)} R_i
\end{aligned}$$

It is important to note that we use the definitions of regions to decide whether $R_j \subseteq H_k$ holds for some region R_j and some base set H_k (it is easy to check that $R_j \subseteq H_k$ holds if and only if $R_j = ... \cap H_k \cap ...$).

We remark here that provisional normal forms can be effectively implemented by their index sets ($\mathcal{I}$) represented as bit vectors (of length 2^n) thus allowing the performance of set operations on provisional normal forms as bit-wise logical operations on bit vectors. Bit vectors also provide a succinct way of representing provisional normal forms (a useful fact in discussing examples). For example, in bit vector notation, `0101` stands for $\bigcup_{i\in\{1,3\}} R_i$, that is, a region appears in the union if the corresponding bit is set in the bit vector. Further examples (assuming there are two base sets, hence, four regions) are presented in Table 5.2.

Provisional normal forms cannot handle label constants in set expressions. For this reason, we extend our normal form with *label presence* index sets $\mathcal{I}_\ell$ for each label $\ell \in L_P$, where L_P

Set Expression	Provisional Normal Form (ψ_B)	Bit Vector
$\varnothing$	$\bigcup_{i\in\varnothing} R_i$	`0000`
H_1	$\bigcup_{i\in\{1,3\}} R_i$	`0101`
H_2	$\bigcup_{i\in\{2,3\}} R_i$	`0011`
$H_1 \cup H_2$	$\bigcup_{i\in\{1,2,3\}} R_i$	`0111`
$H_1 \cap H_2$	$\bigcup_{i\in\{3\}} R_i$	`0001`
$H_1 \backslash H_2$	$\bigcup_{i\in\{1\}} R_i$	`0100`
$\overline{H_1}$	$\bigcup_{i\in\{0,2\}} R_i$	`1010`

Table 5.2: Examples for Provisional Normal Forms

stands for the set of labels appearing in P. These presence sets, like index sets, are determined by the set expression at hand. The format of our normal form thus becomes:

$$\hat{\psi}_B(e) = \bigcup_{i\in\mathcal{I}^e}(R_i \backslash L_P) \cup \bigcup_{\ell\in L_P}\bigcup_{i\in\mathcal{I}^e_\ell} R_i \cap \{\ell\}$$

We define $\hat{\psi}_B$ inductively as:

For $\hat{\psi}_B(\varnothing)$, let $\mathcal{I} = \varnothing$, with $\mathcal{I}_\ell = \varnothing$ for all $\ell \in L_P$.

For $\hat{\psi}_B(\{l\})$, let $\mathcal{I} = \varnothing$, with $\mathcal{I}_l = \{0, ..., 2^n - 1\}$ and $\mathcal{I}_\ell = \varnothing$ for all $l \neq \ell \in L_P$.

For $\hat{\psi}_B(H_j)$, let $\mathcal{I} = \{k \mid R_k \subseteq H_j\}$, with $\mathcal{I}_\ell = \mathcal{I}$ for all $\ell \in L_P$.

$\hat{\psi}_B(e \cap f) = \hat{\psi}_B(e) \cap \hat{\psi}_B(f) =$

$\bigcup_{i\in\mathcal{I}^e\cap\mathcal{I}^f}(R_i \backslash L_P) \cup \bigcup_{\ell\in L_P}\bigcup_{i\in\mathcal{I}^e_\ell\cap\mathcal{I}^f_\ell} R_i \cap \{\ell\}$

The cases for $\hat{\psi}_B(e \cup f)$ etc. are defined analogously.

An important property of the normal form defined by $\hat{\psi}$ is that the union (intersection, etc.) of

two expressions in normal form is again in normal form. Furthermore, the following lemma guarantees that $\hat{\psi}$ preserves the meaning of set expressions:

Lemma 5.5.2. *For a set expression e and label ℓ, $\ell \in e$ iff $\ell \in \hat{\psi}_B(e)$.*

The index set $\mathcal{I}^e_\ell$ for a label ℓ in the normal form of $\hat{\psi}_B(e)$ of a set expression e plays an important part in the algorithm, since it tells us exactly which one of the regions a label must belong to in order to belong to a set expression:

Lemma 5.5.3. *For a set expression e and a label $\ell \in L_P$, $\ell \in e$ iff $\ell \in \bigcup_{i \in \mathcal{I}^e_\ell} R_i$, where $\mathcal{I}^e_\ell$ is defined by the normal form $\hat{\psi}_B(e)$.*

Although formidable looking, normal forms can also be concisely represented by their index sets ($\mathcal{I}$ together with $\mathcal{I}_\ell$ for each $\ell \in L_P$) represented as a collection of bit vectors (of length 2^n), in other words, as a bit matrix, thus allowing the performance of set operations on normal forms as bit-wise logical operations on these bit matrices. Furthermore, in the bit matrix notation we will use the generic label name '*' to mark the index set for all unmentioned labels in P. Examples for normal forms in bit matrix notation are presented in Table 5.3 for predicates:

$$P = \rho_3 \simeq (\!|a : Int \mid \rho_1|\!),\ \rho_4 \simeq \rho_3 \backslash \rho_2$$

where there are two base sets (ρ'_1 and ρ'_2), and also $L_P = \{a\}$, that is, only the label a is mentioned in P.

5.6 Solving Set Constraints

If no labels appear in the set of constraints P, then P is trivially satisfiable by the substitution that maps each row variable to the empty row (an observation also made in [Van den Bussche

Set Expression	Normal Form	Bit Matrix
$\varnothing$	$\bigcup_{i\in\{\}} R_i\backslash\{a\} \cup \bigcup_{i\in\{\}} R_i \cap \{a\}$	a:0000 *:0000
ρ'_1	$\bigcup_{i\in\{1,3\}} R_i\backslash\{a\} \cup \bigcup_{i\in\{1,3\}} R_i \cap \{a\}$	a:0101 *:0101
ρ'_2	$\bigcup_{i\in\{2,3\}} R_i\backslash\{a\} \cup \bigcup_{i\in\{2,3\}} R_i \cap \{a\}$	a:0011 *:0011
$\{a\} \cup \rho'_1$	$\bigcup_{i\in\{1,3\}} R_i\backslash\{a\} \cup \bigcup_{i\in\{0,1,2,3\}} R_i \cap \{a\}$	a:1111 *:0101
$(\{a\} \cup \rho'_1)\backslash\rho'_2$	$\bigcup_{i\in\{1\}} R_i\backslash\{a\} \cup \bigcup_{i\in\{0,1\}} R_i \cap \{a\}$	a:1100 *:0100

Table 5.3: Examples for Normal Forms in Bit Matrix Notation

and Waller, 1999]).

Otherwise, $\phi(P)$ is a collection of set equality constraints of the form $e = f$ that we no set out to solve. Using Lemma 5.5.2, $e = f$ iff $\hat{\psi}_B(e) = \hat{\psi}_B(f)$. Now, using Lemma 5.5.3, $\ell \in e$ iff $\ell \in \bigcup_{i\in\mathcal{I}^e_\ell} R_i$, and similarly $\ell \in f$ iff $\ell \in \bigcup_{i\in\mathcal{I}^f_\ell} R_i$. Now suppose $\ell \in R_k$ where $k \in \mathcal{I}^e_\ell$ and $k \notin \mathcal{I}^f_\ell$. This would imply that $\ell \in e$ while $\ell \notin f$, which would violate the constraint $e = f$. Thus, it must be the case that $\ell \notin R_k$. In general, ℓ cannot belong to any of the regions $\mathcal{I}^e_\ell \uplus \mathcal{I}^f_\ell$ without violating the constraint $e = f$.

The last observation leads us to an algorithm to check the satisfiability of the set constraints $\phi(P)$ as follows:

1. Initialize $\Theta(\ell)$ to $\{0, ..., 2^n - 1\}$ for each $\ell \in L_P$.

2. For each set equality constraint $(e = f) \in \phi(P)$, and for each $\ell \in L_P$, perform the update $\Theta(\ell) \leftarrow \Theta(\ell)\backslash(\mathcal{I}^e_\ell \uplus \mathcal{I}^f_\ell)$, where $\mathcal{I}^e_\ell$ and $\mathcal{I}^f_\ell$ are defined by $\hat{\psi}_B(e)$ and $\hat{\psi}_B(f)$, respectively.

3. If $\Theta(\ell) = \varnothing$ for any ℓ, then $\phi(P)$ is not satisfiable, otherwise it is satisfiable.

$\Theta(\ell)$ tells us which regions the label ℓ can be in. Before processing any of the set equalities, ℓ can be in any of the regions, hence the initial value for $\Theta(\ell)$ is the set of indexes of all regions. By processing the set equalities the algorithm 'narrows down' the set of regions each label can belong to. If $\Theta(\ell) = \varnothing$, then ℓ cannot be in any of the regions, a contradiction, meaning that the constraints are inconsistent. From an error-reporting point of view, the algorithm can report (1) exactly which labels have inconsistent constraints on them (those for which $\Theta(\ell) = \varnothing$), and (2) exactly which set equalities are responsible for the inconsistency (those that caused the update of some $\Theta(\ell)$ to $\varnothing$).

Observe, that the global constraints captured by Θ can also be represented by a set expression in normal form (using the special name '*' for unmentioned labels):

$$\hat{\psi}(\Theta) = \bigcup_{i \in \Theta(*)} R_i \backslash L_P \cup \bigcup_{\ell \in L_P} \bigcup_{i \in \Theta(\ell)} R_i \cap \{\ell\}$$

This will be convenient in Chapter 6 when we will introduce *cleansed* normal forms using Θ.

5.7 Selecting Base Sets

In order to solve set constraints, we need to convert set expressions to normal form. But to do that we first need to select base sets in terms of which all set expressions can be expressed. The naïve approach would be to take all the variables (that stand for unknowns sets) appearing in $\phi(P)$ as base sets, but since the number of regions is exponential in the number of base sets, it is crucial to keep their number small.

A closer examination of the structure of predicates generated by the base operations reveals that most row variables in P only name other row expressions and thus could be completely

1. Initialize $Deps(X)$ to $\{vs(e) \mid (X = e) \in \phi(P)\}$ for all $X \in vs(\phi(P))$, and initialize BV_P to $vs(\phi(P))$.

2. If there is $V \in BV_P$ with a dependency set $d \in Deps(V)$ such that $d \neq \emptyset$ and $V \notin d$ (a non-recursive dependency), then

   (a) update $BV_P \leftarrow BV_P \backslash \{V\}$,

   (b) and for all $X \in BV_P$, update

$$Deps(X) \leftarrow \{d \in Deps(X) \mid V \notin d\}$$
$$\cup \{d_X \backslash \{V\} \cup d_V \mid d_X \in Deps(X), V \in d_X, d_V \in Deps(V), V \notin d_V\}$$

3. If BV_P has changed then goto Step 2, otherwise stop.

Figure 5.2: Algorithm for Identifying Base Variables

eliminated by substitution. Those variables that cannot be substituted away, *independent* variables, will form a suitable collection of base sets. We make a slight shift of terminology now as we start talking about base *variables* instead of base sets. The reason for this shift is that we would like to emphasize the fact that base sets will correspond to certain set variables in $\phi(P)$ and that we are going to identify these set variables by analyzing the set constraints in $\phi(P)$.

We assume, without loss of generality, that the right-hand side of all equations in $\phi(P)$ is either a single variable or the empty set. Let $vs(e)$ stand for the set of variables appearing in e. The set of independent variables in $\phi(P)$ can be identified (actually, approximated) by the algorithm presented in Figure 5.2.

When the algorithm terminates, BV_P will be the set of independent variables that now can be used as base variables in calculating normal forms for set expressions (of course, now set equalities must be processed in order of their dependencies to ensure that the normal form of a given variable is available in expressions that use that variable).

To clarify how the algorithm works, we present a sample run of the algorithm on the set of constraints $\{A = B,\ B = C \cup D,\ A = A \cap C\}$. $Deps(X)$ describes the set of dependency sets for variable X while BV_P is our current approximation of base variables. Since the dependencies of variables that are no longer in BV_P are never considered again, we will only show dependencies for variables that are still in BV_P. The initial values of *Deps* and BV_P are:

$$\begin{aligned} BV_P &= \{A,\ B,\ C,\ D\} \\ Deps(A) &= \{\{B\},\ \{A,\ C\}\} \\ Deps(B) &= \{\{C,\ D\}\} \\ Deps(C) &= \{\} \\ Deps(D) &= \{\} \end{aligned}$$

Next, in Step 2 of the algorithm, we discover that the variable A has a dependency $\{B\}$ that is not empty and that does not have A as a member. After performing the updates, described in Step 2.a and Step 2.b, the new values of *Deps* and BV_P are:

$$\begin{aligned} BV_P &= \{B,\ C,\ D\} \\ Deps(B) &= \{\{C,\ D\}\} \\ Deps(C) &= \{\} \\ Deps(D) &= \{\} \end{aligned}$$

Our next candidate for elimination is variable B since it has a non-empty, non-recursive depen-

dency. After performing the necessary updates:

$$\begin{aligned} BV_P &= \{C,\ D\} \\ Deps(C) &= \{\} \\ Deps(D) &= \{\} \end{aligned}$$

There are no more suitable candidates for substituting away, so we return our approximation of base variables, the set $\{C, D\}$. We talk about approximation, since it can be the case that a variable we identified as base variable is in fact *not* an independent variable, that is, it can be expressed in terms of other variables. However, this is not a serious problem, since all we care about is to quickly select a suitably small set of base variables in terms of which all other variables can be expressed using our normal form.

5.8 Checking Field Type Constraints

Type predicates on rows reflect the way records are constructed using basic record operations. Thus, it is possible to build a 'goes into', or row construction, graph G, showing which records contribute fields to some other record, by analyzing the predicates. If we learn about the type of a field in a given record, we can deduce, by looking at the graph, that some other record must also have the same type for the given field. For example, in the record expression $e = \langle x \parallel y \rangle$, if the field a has type *Int* in x, then it must also have the same type in e. Notice that field type information flows in the other direction as well, that is, if we discover the type of $e \cdot a$, then we learned the type of a in x *or* y (since only one of them can have a), hence G is undirected.

The row equality predicates in P that we are interested in when building our graph G are one of the forms:

1. $\rho_i \simeq (\!|\ell : \tau \mid \rho_j|\!)$

2. $\rho_i \simeq (\rho_j \parallel \rho_k)$

3. $\rho_i \simeq (\rho_j \backslash \rho_k)$

The vertices of the graph G are row variables, and an edge between two vertices signals a potential equality of some field types in the connected rows. Formally, $G = (V, E)$, where $V = ftv^{\mathsf{row}}(P)$, and E is defined as follows:

$$(\rho_i \simeq (\!|\ell : \tau \mid \rho_j|\!)) \in P \Rightarrow (\rho_i, \rho_j) \in E$$

$$(\rho_i \simeq (\rho_j \parallel \rho_k)) \in P \Rightarrow (\rho_i, \rho_j) \in E \wedge (\rho_i, \rho_k) \in E$$

$$(\rho_i \simeq (\rho_j \backslash \rho_k)) \in P \Rightarrow (\rho_i, \rho_j) \in E$$

Vertices are labelled with field type information as follows:

$$\rho_i(\ell) = \{\tau \mid (\rho_i \simeq (\!|\ell : \tau \mid \rho_j|\!)) \in P\}$$

The row construction graph G is used to determine which field types must unify. Given a label ℓ, and connected vertices ρ_1 and ρ_2, if $\ell \in \phi(\rho_1)$ and also $\ell \in \phi(\rho_2)$, then the field ℓ must have the same type in both rows, that is, all types in $\rho_1(\ell) \cup \rho_2(\ell)$ must unify. In general, for a given label ℓ, if we can decide whether $\ell \in \phi(\rho_i)$ for each vertex ρ_i, then we can identify connected components C_j^ℓ in G where $\rho_i \in C_j^\ell$ implies $\ell \in \phi(\rho_i)$. Vertices that belong to the same component C_j^ℓ must have the same field type for ℓ, that is, all types in $\bigcup_{\rho_i \in C_j^\ell} \rho_i(\ell)$ must unify.

In order to identify the connected components C_j^ℓ, we need to be able to answer the question whether $\ell \in \phi(\rho_i)$. More precisely, we are asking the question: does 'ρ_i has field ℓ in all valid instantiations of ρ_i' follow from the set of type predicates P? We can answer this question by

taking advantage of a by product of the set constraint solving algorithm in Section 5.6, that is, $\Theta(\ell)$, which tells us to which regions ℓ can belong to. Using Lemma 5.5.3, $\ell \in \hat{\psi}_B(\phi(\rho_i))$ if and only if ℓ belongs to one of the regions in the index set $\mathcal{I}_\ell^{\phi(\rho_i)}$.

Now the question 'is ℓ in $\phi(\rho_i)$?' can be answered in three different ways:

1. If $\Theta(l) \subseteq \mathcal{I}_\ell^{\phi(\rho_i)}$, then **yes**.

2. If $\Theta(l) \cap \mathcal{I}_\ell^{\phi(\rho_i)} = \varnothing$, then **no**.

3. Otherwise, **maybe**.

In order to proceed without resorting to back-tracking (that is, branching on every **maybe** answer), and thus to exponential complexity in the number of labels; our algorithm takes a **maybe** answer for **yes** and considers ambiguous cases as if the label ℓ has positively appeared in the given row. This greatly simplifies the algorithm and is at the heart of the language restriction discussed earlier.

Once we identified all the field types that must unify we perform unification and if it fails, then we can conclude that the predicates in P are not satisfiable due to conflicting field type constraints. In case of success, the resulting (referred to as *improving*) substitution s_P will play an important part in type improvement, as discussed in Chapter 6.

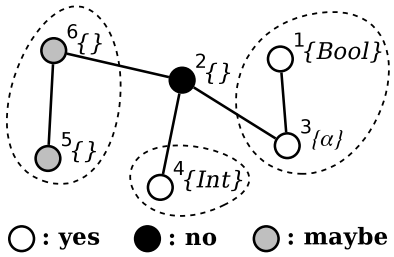

Figure 5.3: Connected Components for Field *a*

To further understanding, in Figure 5.3 we show a row construction graph with connected components for field *a* encircled. The graph is generated by the following expression:

$$\textbf{if } z^1{\cdot}a \textbf{ then } \langle x^2 \parallel z^1\rangle^3{\cdot}a \textbf{ else } g\ \langle a = 2 \mid x^2\rangle^4\ \langle x^2 \parallel y^5\rangle^6$$

where numerical superscripts link vertices to their corresponding record expressions. The resulting substitution in this case is $s_P = [\alpha \mapsto \mathit{Bool}]$.

5.9 Handling Nested Records

Consider the following example using nested records:

$$e \quad \equiv \quad \textbf{if } x == y \textbf{ then } f\ ((x{\cdot}a){\cdot}b) \textbf{ else } g\ \langle b = 2 \mid y{\cdot}a\rangle$$

Although not immediately obvious, the expression *e* is ill-typed since it has conflicting presence/absence constraints on the field *b*. Both *x* and *y* are records (we select the field *a* from both) of the same type (we compare them for equality) meaning that the field *a* must have the same type in both of them. However, given the same record type for field *a*, we cannot simultaneously select from it (requiring its presence) and extend it with (requiring its absence) the field *b*.

The problem is that in the set constraint solving phase, the algorithm does not yet know the important constraint that the record types for field a must be the same. This additional constraint is discovered only in the field type constraint checking phase.

To correctly handle nested records, we have to continue analyzing the improving substitution s_P returned by the field type constraint checking phase. The basic idea is that if s_P maps one row variable ρ_x to another ρ_y, then we have to enforce the additional set constraint $\phi(\rho_x) = \phi(\rho_y)$. Formally, we define the additional set constraints as

$$K = \{\phi(\rho_x) = \phi(\rho_y) \mid [\rho_x \mapsto \rho_y] \in s_P\}.$$

We can process the additional set constraints K without re-checking all previous set constraints. If the system of set constraints $\phi(P)$ extended with K is unsatisfiable, we fail. Otherwise, we apply the improving substitution s_P to P and move on to the field type constraint checking phase. We repeat this cycle until $K = \emptyset$, which is guaranteed to happen since each substitution reduces the number of type variables in P. If the algorithm successfully terminates, the improving substitution eventually returned is defined to be the *composition* of improving substitution from each cycle.

Nested records raise the possibility of recursive record types. Take the following expression:

$$e_r \quad \equiv \quad \lambda t.t.a == t$$

The type of field a in the record t is the same as the type of t itself. There is nothing in the design or the assumptions of algorithm Q that would prevent it from handling recursive record types, and indeed it does handle them correctly. It is an altogether different question whether recursive record types serve any useful purpose and whether they should be included in the language. It

remains as future work to consider the pros and cons of recursive record types in the context of a purely functional database programming language.

5.10 Soundness and Completeness

The following theorem states that algorithm Q, described in the previous sections, correctly identifies sets of predicates that are not satisfiable (with regard to the entailment relation presented in Section 4.4):

Theorem 5.10.1 (Soundness)**.** *If* ¬**sat** *P then Algorithm Q will report failure.*

Completeness, because of the language restriction presented in Section 5.3, is a bit trickier business. As it stands, algorithm Q is *deliberately* not complete, since it rejects programs that are well-typed but violate the language restriction. However, it is possible to modify the type system in such a way that it coincides with the algorithm. The main idea is to use *two* rows (not just one) to describe record types: one for field *presence* and one for field *types*. Then we can modify the type of record concatenation so that it requires all fields (present *or* absent) in its operands to have the same type:

$$\langle \_ \parallel \_ \rangle :: \rho_3 \simeq \rho_1 \parallel \rho_2,\ \rho_1 \# \rho_2 \Rightarrow$$

$$\mathit{Rec}\ \rho_1\ \rho_4 \rightarrow \mathit{Rec}\ \rho_2\ \rho_4 \rightarrow \mathit{Rec}\ \rho_3\ \rho_4$$

With the modified type system in mind, we can state the following theorem:

Theorem 5.10.2 (Completeness)**.** *Assuming a modified type system with two rows per record type, if* **sat** *P then Algorithm Q will succeed.*

5.11 Complexity of Algorithm Q

The determining factors in the complexity of constraint satisfiability checking are

1. the number of predicates (p),

2. the number of base variables ($n \leq p$), and

3. the number of labels ($l \leq p$).

Each predicate is of size $O(1)$ (the typing of basic operations guarantee this). Furthermore, we assume that field types are of $O(1)$ so that unification can be considered a unit operation. Next, we proceed with a detailed complexity analysis of the algorithm.

Converting to set constraints is $O(p)$. Identifying base variables is $O(p^2)$: in each iteration (requiring $O(p)$ steps) we get rid of one variable, and there can be at most p variables). Calculating the normal forms and processing the predicates is $O(2^n lp)$, since each bit matrix used to represent normal forms is of size $O(2^n l)$ (there are 2^n regions and there is one presence set for each label). Building the row construction graph G is $O(p)$ (assuming an efficient graph representation using hash tables and adjacency lists), since it consists of analyzing each predicate and building the graph. Finding the connected components in G for each label (with checking label presence for each vertex) is again $O(2^n lp)$ (each presence check is $O(2^n)$ with $O(p)$ vertices and $O(l)$ labels, while finding l connected components using depth-first search is $O(lp)$). Without nested records, the complexity of algorithm is thus $O(2^n lp)$. Since the presence of nested records might require restarting the algorithm, in the worst-case $O(p)$ times (for each row variable), we conclude that the complexity of algorithm Q is $O(2^n lp^2)$.

The algorithm is clearly exponential in the number of base variables (which determines the number of regions used in describing normal forms) so we would like to put forward arguments as to why we rarely expect to see this exponential complexity arise in practice. The gist of our argument for practicality hinges on the fact that the number of base variables is proportional to the number of function parameters from polymorphic function definitions. To see this, all one has to consider is that independent row variables (ones that are not the result of some type-level operation) can only originate from polymorphic function arguments (informally, uncertainty must come from the 'outside'). Functions that take even as many as a dozen arguments are quite rare in practice, so we expect the algorithm to perform well on average programs. Our expectation is also supported by the experience we gained when using the implementation of the algorithm. Also, we would like to re-iterate here the importance of the language restriction introduced in Section 5.3 that removed one source of exponential complexity (on the number of labels).

In database programming, it is quite common to refer to relations defined in some external database. With the language presented in this paper, it now becomes possible to do this without access to database schema information, relying on the type-checking algorithm to enforce that external relations are used in a consistent manner. From a complexity point of view, this poses a new problem, since database applications regularly use dozens (if not hundreds) of external relations so our previous argument for practicality (upper bound on polymorphic function arguments) no longer holds. However, this only has to be the case if we treat database schemas as completely unstructured, which would ignore the useful fact that most relational databases are normalized (to some extent, at least). In fact, if we impose the strict condition on database

schemas that *any two relations can have at most one attribute in common*, then the number of *distinct* regions becomes *quadratic* in the number of external relations, making the type-checking algorithm practical again. Severe as it might seem, the restriction is not completely unrealistic, since database schemas in third normal form and using numeric surrogate keys often already satisfy the condition, and if not, can be brought into conformance by normalizing, introducing surrogate keys, and renaming attributes.

5.12 Sample Run

In this section we provide a sample run of the algorithm on a simple expression to clarify each step of the algorithm. The expression we are going to analyze is the following ill-typed expression:

$$\lambda x.x!a == x!b$$

The inferred type is:

$$P = \rho_1 \textbf{ has } (a : \alpha),\ \rho_2 \simeq \rho_1 - a,\ \rho_1 \textbf{ has } (b : \beta),\ \rho_2 \simeq \rho_1 - b$$

$$\tau = Rec\ \rho_1 \rightarrow Bool$$

First, we convert the predicates to their expanded forms:

$$P = (\!|a : \alpha \mid (\!|)\!| \simeq \rho_1 \backslash (\rho_1 - a),\ \rho_2 \simeq \rho_1 - a,\ (\!|b : \beta \mid (\!|)\!| \simeq \rho_1 \backslash (\rho_1 - b),\ \rho_2 \simeq \rho_1 - b$$

Next, we convert the predicates to set constraints $\phi(P)$:

$$\phi(P) = \{a\} \cup \varnothing = \rho_1' \backslash (\rho_1' \backslash \{a\}),\ \rho_2' \simeq \rho_1' \backslash \{a\},\ \{b\} \cup \varnothing = \rho_1' \backslash (\rho_1' \backslash \{b\}),\ \rho_2' \simeq \rho_1' \backslash \{b\}$$

By analyzing the set constraints, we can identify the set of base variables, in this case: $B = \{\rho_1'\}$.

Set Expression	Bit Matrix
$\varnothing$	a:00 b:00 *:00
$\{a\}$	a:11 b:00 *:00
$\{b\}$	a:00 b:11 *:00
ρ'_1	a:01 b:01 *:01
$(\rho'_2 =)\ \rho'_1 \backslash \{a\}$	a:00 b:01 *:01
$\rho'_1 \backslash (\rho'_1 \backslash \{a\})$	a:01 b:00 *:00
$(\rho'_2 =)\ \rho'_1 \backslash \{b\}$	a:01 b:00 *:01
$\rho'_1 \backslash (\rho'_1 \backslash \{b\})$	a:00 b:01 *:00

Table 5.4: Normal Forms in Bit Matrix Notation

In order to process the set constraints, we need to calculate the normal forms of set expressions, presented in Table 5.4. There is one base variable that defines two regions, and there are two labels (a and b) mentioned in P, which means that our bit matrices are relatively small.

All that remains is to process set constraints, now with both sides of the equation in normal form. Initially, there are no constraints on the presence of labels in any of the regions, so $\Theta(a) =$ `11` and $\Theta(b) =$ `11`. Next, we process the set constraints (order is not important):

1. First, let's process constraint $\{a\} \cup \varnothing = \rho'_1 \backslash (\rho'_1 \backslash \{a\})$, now in normal form:

$$\begin{matrix} \texttt{a:11} \\ \texttt{b:00} \\ \texttt{*:00} \end{matrix} = \begin{matrix} \texttt{a:01} \\ \texttt{b:00} \\ \texttt{*:00} \end{matrix}$$

   Thus, we learn that label a cannot appear in region R_0 (since that would violate the equation), and we update $\Theta(a)$ to `01`.

2. Processing set constraint $\{b\} \cup \varnothing = \rho'_1 \backslash (\rho'_1 \backslash \{b\})$ is analogous to the previous step, with the result of updating $\Theta(b)$ to `01`.

3. Finally, we process set constraint $(\rho'_2 =)\ \rho'_1 \backslash \{a\} = \rho'_1 \backslash \{b\}\ (= \rho'_2)$:

$$\begin{matrix} \texttt{a:00} \\ \texttt{b:01} \\ \texttt{*:01} \end{matrix} = \begin{matrix} \texttt{a:01} \\ \texttt{b:00} \\ \texttt{*:01} \end{matrix}$$

   From this we learn that neither a nor b can be in region R_1. Thus, we update $\Theta(a)$ to `00` and $\Theta(b)$ to `00`.

Now that we have processed all the set constraints, we can conclude, by looking at Θ, that they are unsatisfiable since both $\Theta(a) = \varnothing$ and $\Theta(b) = \varnothing$. Not only that, we can report that we have inconsistent presence/absence constraints for labels a and b.

5.13 Summary

In this chapter we described algorithm Q, an algorithm for checking the satisfiability of predicates derived by type inference algorithm W. The internal data structures built by algorithm Q, especially Θ and G, will play an important role in the coming chapters, since they capture useful information about the structure of the predicates, the relation between row variables, and the 'relevance' of each predicate.

Chapter 6

Type Improvement and Simplification

The types inferred by the type inference algorithm *W* (presented in Chapter 4) are not always as accurate or concise as they could be. In this chapter we formalize the notions of *accuracy* and *conciseness* of qualified types, and develop algorithms that *improve* and *simplify* said types. The idea of treating type improvement and simplification as orthogonal to type inference in the framework of the theory of qualified types was discussed by Jones in [Jones, 1995]. The definitions (but not the algorithms, since they are specific to our system) presented in this chapter were inspired by Jones's work.

6.1 Type Improvement

We will begin with a motivating example. Take the following expression:

$$e \quad \equiv \quad \lambda x.\lambda f.7 + f\ (x{\cdot}a)\ (x{\cdot}a)$$

The type inferred for expression *e* is:

$$P \Rightarrow \tau \quad \equiv \quad \rho_1\ \mathbf{has}\ (a : \alpha),\ \rho_1\ \mathbf{has}\ (a : \beta) \Rightarrow \mathit{Rec}\ \rho_1 \rightarrow (\alpha \rightarrow \beta \rightarrow \mathit{Int}) \rightarrow \mathit{Int}$$

By analyzing the structure of *e*, we can conclude that the two input parameters of the function f must be of the same type, that of the field *a* in the record *x*. Thus, a more accurate type for

expression e would be (P' is a *set*, thus repeated predicates have been removed):

$$P' \Rightarrow \tau' \quad \equiv \quad \rho_1 \textbf{ has } (a : \alpha) \Rightarrow \mathit{Rec}\ \rho_1 \rightarrow (\alpha \rightarrow \alpha \rightarrow \mathit{Int}) \rightarrow \mathit{Int}$$

The original principal type τ is not as accurate as it could be, because the type inference algorithm does not (since it cannot) take into consideration certain additional type equality constraints (here, equality of field types) that follow from the type predicates P. As a result, there are instances of τ where the type predicates P are not satisfiable, for example, those instances that map type variables α and β to conflicting types. Thus, although the principal type τ is strictly more general than the improved type τ', this additional generality is illusory since we are forced to map type variables α and β to the same type anyway in order to satisfy the type predicates P.

A more accurate (or improved) type is thus one that takes into account *at least some* of the type equality constraints that follow from the type predicates, but are hidden from the type inference algorithm. The type that has taken *all* such constraints into account is the *principal satisfiable type*. Next, we develop these notions formally.

Recall from Section 4.4, that for a type to be well-formed only row variables (or the empty row) can appear in it and not arbitrary row expressions. According to this rule, the types $\mathit{Rec}\ (\!|\!|)$ and $\mathit{Rec}\ \rho_1 \rightarrow \mathit{Rec}\ \rho_2$ are well-formed, while the types $\mathit{Rec}\ (\rho_1 - l)$ and $\mathit{Rec}\ (\rho_1 \parallel \rho_2)$ are not. In order to preserve the well-formedness of types during type improvement, we define a *row-restricted* substitution as a substitution that maps row variables to other row variables (or to the empty row): that is, a row-restricted substitution does not map row variables to arbitrary row expressions. For example, the substitution $[\rho_1 \mapsto \rho_2]$ is row-restricted, while the substitution $[\rho_1 \mapsto (\!|a : \mathit{Int} \mid \rho_2|\!)]$ is not.

Next, we define the set of satisfiable instances of a type τ with regard to predicates P as:

$$\lfloor \tau \rfloor_P = \{ s\tau \mid s \in \mathit{Subst},\ \Vdash sP \}$$

We call a row-restricted substitution s an *improving* substitution, and the type $s\tau$ an *improved* type, if the substitution does not change the set of satisfiable instances of the type:

$$\lfloor s\tau \rfloor_{sP} = \lfloor \tau \rfloor_P$$

We call the improving substitution s the *principal improving* substitution, and the type $s\tau$ the *principal satisfiable type*, if for all improving substitutions u there exists a row-restricted substitution t such that $u\tau = ts\tau$. Observe, that the principal satisfiable type is unique (up to renaming of type variables, of course), but the principal improving substitution is not necessarily so (for example, if $[\alpha \mapsto \beta]$ improves the type $\alpha \to \beta \to \mathit{Int}$, so does $[\beta \mapsto \alpha]$). This hardly matters in practice, since *any* principal improving substitution will give the principal satisfiable type.

Jones states in [Jones, 1995] that in general (under an arbitrary system of predicates) it can be undecidable to find the principal satisfiable type and sometimes it does not even exist. Fortunately, this is not the case with the type system presented in this thesis. For all well-typed expressions of the language, a principal satisfiable type *does* exists and it *can* be found using the algorithm we describe later (in Section 6.2).

6.1.1 Representative Cases

Before we develop the algorithm for type improvement, we present three examples that embody the three main ‘sources’ of additional type equality constraints that are hidden from the type inference algorithm, and thus are not reflected in the inferred type.

Field Type Constraints

Take the following expression:

$$e_1 \quad \equiv \quad \lambda x.\lambda y.\textbf{if } \langle x \parallel y \rangle \cdot a == 3 \textbf{ then } x \cdot a \textbf{ else } y \cdot b$$

The type inferred for expression e_1 is:

$$\rho_3 \textbf{ has } (a : \mathit{Int}),\ \rho_3 \simeq (\rho_1 \parallel \rho_2),\ \rho_1 \# \rho_2,\ \rho_1 \textbf{ has } (a : \alpha),\ \rho_2 \textbf{ has } (b : \alpha)$$

$$\Rightarrow \mathit{Rec}\ \rho_1 \rightarrow \mathit{Rec}\ \rho_2 \rightarrow \alpha$$

In this example, we select the field *a* from the concatenation of records *x* and *y* to compare it to an integer (thus pinning down its type). Hence, the field *a* must have type *Int* in *either x* or *y* (since they are disjoint). But we also select *a* from *x*, meaning that *x* must have *a*, which, together with our previous observation, implies that the expression $x \cdot a$ has type *Int*. Thus, we can conclude from the constraints on field types that the return type of the function e_1 cannot be anything but *Int*. The row-constrained substitution $[\alpha \mapsto \mathit{Int}]$ is therefore an improving substitution. As a matter of fact, the improving substitution $s_1 = [\alpha \mapsto \mathit{Int}]$ also happens to be a principal improving substitution since it takes *all* type equality constraints into account.

The type inference algorithm is too general to perform the kind of reasoning we did in the previous paragraph and as a result the constraint that the type variable α is in fact equal to *Int* is hidden from it. In general, when two field selections refer to the same record field the resulting types must unify. As the above example tries to demonstrate, it is often a non-trivial task (since it can involve reasoning with arbitrary record expressions) to decide whether two field selections actually refer to the same field. Field type equality constraints are thus potential sources of improving substitutions.

Empty Row Constraints

A markedly different kind of ‘hidden’ constraint is when one row variable is constrained to be equal to the empty row by the type predicates. Take, for example, the following expression:

$$e_2 \quad \equiv \quad \lambda x.\langle x \parallel x \rangle$$

The type inferred for expression e_2 is:

$$\rho_2 \simeq (\rho_1 \parallel \rho_1),\ \rho_1 \# \rho_1$$
$$\Rightarrow \mathit{Rec}\ \rho_1 \rightarrow \mathit{Rec}\ \rho_2$$

In this expression, we concatenate the record x to itself. Record concatenation requires its operands to be disjoint, so ρ_1, the type of x, must be a row that is disjoint with itself. There is only one row that is disjoint with itself, and that is the empty row $(\!|\,|\!)$, thus the type of both the record x and the return value is the empty record $\mathit{Rec}\ (\!|\,|\!)$. As a consequence, the improving substitution in this case is $s_2 = [\rho_1 \mapsto (\!|\,|\!),\ \rho_2 \mapsto (\!|\,|\!)]$ (which also happens to be a principal one).

Clearly, the case represented by the expression e_2 has nothing to do with field type constraints (no fields are even mentioned in it), rather, it is the result of a field presence/absence constraint, namely, disjointness. In general, type predicates can constraint a particular row variable to be equal to the empty row under any valid instantiation of the type, thus we can improve the type by substituting said row variable with the empty row. Discovering empty row constraints is again outside the scope of the type inference algorithm, and thus it too provides a potential opportunity for type improvement.

Same Row Constraints

Type predicates can constrain two row variables to be the same in ways that are not accessible to the type inference algorithm. Consider the following expression:

$$e_3 \quad \equiv \quad \lambda x.\lambda f.f\ \langle a = 7 \mid x\rangle\ \langle a = 2 \mid x\rangle$$

The type inferred for expression e_3 is:

$$\rho_1\ \textbf{lacks}\ a,\ \rho_2 \simeq (\!|a : \mathit{Int} \mid \rho_1|\!),\ \rho_3 \simeq (\!|a : \mathit{Int} \mid \rho_1|\!)$$
$$\Rightarrow \mathit{Rec}\ \rho_1 \rightarrow (\mathit{Rec}\ \rho_2 \rightarrow \mathit{Rec}\ \rho_3 \rightarrow \alpha) \rightarrow \alpha$$

The two input parameters of the function f are extensions of the same record x with the field a of type *Int*, therefore, the parameters should be of the same type. We could draw the same conclusion by looking at the type predicates and realizing that the right-hand sides of the row equalities defining row variables ρ_2 and ρ_3 are exactly the same. Using our observations, we can improve the type inferred for e_3 by the improving substitution $s_3 = [\rho_3 \mapsto \rho_2]$, which is also a principal improving substitution.

In general, by analyzing the type predicates it can sometimes be concluded that some row variables must always refer to the same row (because, for example, they consist of the same fields and the have matching types for all of their fields). In these cases, we can improve the type by substituting away some row variables by other (equivalent) row variables. Like in the previous cases, the type inference algorithm is not aware of these additional type constraints so they can be exploited for potential type improvement.

6.2 Algorithm for Type Improvement

In order to find the principal satisfiable type, we will have to take into consideration all type equivalence constraints implied by the type predicates. The three main sources of these additional type constraints (field type, empty row, and same row) were outlined in the previous section so now we can concentrate on identifying the principal improving substitution that takes all three sources into account. We proceed by breaking the problem up into finding the three improving substitutions: s_ℓ, $s_{(\!|}$, and $s_\simeq$, that capture the consequences of all the field type, empty row, and same row constraints, respectively. The composition of the three substitutions $s = s_\ell s_{(\!|} s_\simeq$ will be a principal improving substitution which, when applied to the type, will yield the principal satisfiable type. In the rest of the section we assume that we are trying to improve the qualified type $P \Rightarrow \tau$ and that it has already been established (using the algorithm described in Chapter 5) that P is satisfiable.

6.2.1 Finding the Improving Substitution s_ℓ (Field Types)

Finding the substitution s_ℓ is easy, since it has already been done by the algorithm for checking the satisfiability of predicates, that is, algorithm Q. Among other things, algorithm Q must also check field type constraints (since they might be unsatisfiable) by trying to unify types that refer to the same field (see Section 5.8). The resulting substitution is exactly the one that we are looking for, because it takes *all* field type constraints into account.

6.2.2 *Finding the Improving Substitution $s_{(\!|\,|\!)}$ (Empty Row)*

In order to decide which row variables are necessarily equal to the empty row, we will again take advantage of information gathered by algorithm Q. Take example e_2 from Section 6.1.1:

$$e_2 \quad \equiv \quad \lambda x.\langle x \parallel x \rangle$$

The set of predicates for expression e_2 is:

$$P \quad \equiv \quad \rho_2 \simeq (\rho_1 \parallel \rho_1),\ \rho_1 \# \rho_1$$

The system of set constraints $\phi(P)$ is (with the *disjoint* predicate expanded for processing):

$$\phi(P) \quad \equiv \quad \rho_2' = \rho_1' \cup \rho_1',\ \rho_1' = \rho_1' \backslash \rho_2'$$

Analyzing P we can see that there is one base variable ρ_1 (since ρ_2 can be expressed in terms of ρ_1), and thus two regions $R_1 = \overline{\rho_1'}$ and $R_2 = \rho_1'$. Next, we convert set expressions to normal form (notice, that in order to calculate the normal form of ρ_2', we use the normal form of ρ_1'):

Set Expression	**Normal Form**
ρ_1'	`*:01`
$\rho_2' \quad (= \rho_1' \cup \rho_1')$	`*:01`
$\rho_1' \backslash \rho_2'$	`*:00`

When algorithm Q tries to satisfy the set constraint $\rho_1' = \rho_1' \backslash \rho_2'$ (in normal form: `*:01` = `*:00`) it discovers that the region R_2 must be empty, since it appears on the left-hand side but not on the right-hand side. In other words, the constraint on unmentioned labels is that they cannot appear in region R_2. Algorithm Q records this as $\Theta =$ `*:10`. Now, if region R_2 is

constrained to be empty, then it can be removed from the normal form of ρ_1 without changing its meaning, yielding the normal form `*:00`. But `*:00` is exactly the normal form of the empty set! Thus, we can conclude that set ρ_1' must be the empty set in order to satisfy the constraints in $\phi(P)$, and, consequently, the row ρ_1 must be equal to the empty row in order to satisfy the predicates in P.

In general, we would like to be able to tell, simply by looking at its normal form, whether a set (and thus the corresponding row) is constrained to be empty. As the previous example shows, in order to do this, we have to take into consideration the global label presence/absence constraints, captured by the algorithm Q. When we apply the consequences of the global label presence/absence constraints to a normal form, we say we *cleanse* it, turning it into a *cleansed* normal form. Formally, for some set expression e, we define the cleansed normal form as:

$$\hat{\psi}^c(e) = \hat{\psi}(e) \cap \Theta$$

For example, the cleansed normal form of ρ_2' is `*:00` (that is, unsurprisingly, ρ_2' is also empty). Obviously, cleansing normal forms only makes sense once we have processed all set constraints, that is, after algorithm Q has finished (since Θ must reflect *all* label presence/absence constraints).

Now, if the cleansed normal form of a set is equal to the empty set, then the corresponding row variable is necessarily equal to the empty row and can be safely substituted away. Formally, we define the improving substitution $s_{(\!|\,|\!)}$ as (where $\rho \in ftv^{\mathsf{row}}(P)$):

$$s_{(\!|\,|\!)} = \{[\rho \mapsto (\!|\,|\!)] \mid \hat{\psi}^c(\phi(\rho)) = \hat{\psi}(\varnothing)\}$$

6.2.3 Finding the Improving Substitution $s_{\simeq}$ (Same Row)

A simple way of finding rows that are necessarily the same would be to compare their normal forms for equality. After all, one might reasonably expect that two set expressions whose normal forms match represent the same set. This is in fact the case, but, unfortunately, there are cases where two set expressions represent the same set, yet, their normal forms do not match. For example, the set expressions $e_1 = X \cup a$ and $e_2 = a$ (assuming X is the only base variable) have normal forms `a:11` `*:01` and `a:11` `*:00`, respectively. Now, if the set X is constrained to be the empty set, then it is easy to see that sets e_1 and e_2 are exactly the same. Thus, when looking for sets that are the same, what we need to compare is not the normal forms, but rather the cleansed normal forms (since they take global field presence/absence constraints into consideration).

When the cleansed normal forms of two set variables are the same, it means that under all substitutions they have the same set of labels. The problem is, that this does not guarantee that the corresponding rows also have matching field *types*. Therefore, in order to decide row equality, we have to take field types into consideration as well. This is possible, since during satisfiability checking, algorithm Q builds a row construction graph G (see Section 5.8) which contains information on field types in individual rows. As part of checking field type constraints the algorithm also calculates connected components for each field C_i^ℓ. Rows belonging to a component C_j^ℓ must have the same, unique type $\tau(C_j^\ell)$ for field ℓ (these unique field types are determined through unification, whose result, the substitution s_ℓ, we discussed in Section 6.2.1).

With the aforementioned in mind, after algorithm Q has finished, it now makes sense to ask for the type of field ℓ in row ρ (for which type we will use the notation $\rho{\cdot}\ell$ to mirror field selection on records). Formally:

$$\rho{\cdot}\ell = \tau(C_k^\ell) \qquad \textit{where } \rho \in C_k^\ell$$

Two row variables, whose corresponding set variables have matching cleansed normal forms and whose field types are the same for all labels mentioned in P, are considered the same and one can be safely substituted away with the other. The following is the definition of the improving substitution $s_\simeq$ (where $\rho_x, \rho_y \in \mathit{ftv}^{\mathsf{row}}(P)$):

$$s_\simeq = \{[\rho_x \mapsto \rho_y] \mid \hat{\psi}^c(\phi(\rho_x)) = \hat{\psi}^c(\phi(\rho_y)),\ \forall \ell \in \mathit{Labels}(P).\rho_x{\cdot}\ell = \rho_y{\cdot}\ell\}$$

Actually, the above definition is incorrect, since it generates mappings between any two pairs in a set of equivalent row variables. Thus, if ρ_3 is equal to ρ_2 and also to ρ_1, it will include both mapping $[\rho_3 \mapsto \rho_2]$ and $[\rho_3 \mapsto \rho_1]$. To make the substitution well-defined, we assume there exists some arbitrary ordering on row variables (for example, lexicographical), and require that, if there are several mappings for a row variable in $s_\simeq$, then it should map to the smallest one (according to the chosen ordering).

To further understanding, we walk through the steps of finding the improving substitution for the example presented in Section 6.1.1:

$$e_3 \quad \equiv \quad \lambda x.\lambda f.f\ \langle a = 7 \mid x\rangle\ \langle a = 2 \mid x\rangle$$

The type predicates P inferred for expression e_3 are:

$$P \quad \equiv \quad \rho_1\ \textbf{lacks}\ a,\ \rho_2 \simeq (\!|a : \mathit{Int} \mid \rho_1|\!),\ \rho_3 \simeq (\!|a : \mathit{Int} \mid \rho_1|\!)$$

The system of set constraints $\phi(P)$ is (with the *lacks* predicate expanded for processing):

$$\phi(P) \quad \equiv \quad \rho'_1 = \rho'_1 \backslash \{a\},\ \rho'_2 = \{a\} \cup \rho'_1,\ \rho'_3 = \{a\} \cup \rho'_1$$

The only base variable is ρ'_1, so the normal forms of the relevant set expressions are as follows:

Set Expression	Normal Form
ρ'_1	a:01 *:01
$\rho'_1 \backslash \{a\}$	a:00 *:01
$\rho'_2 \quad (= \{a\} \cup \rho'_1)$	a:11 *:01
$\rho'_3 \quad (= \{a\} \cup \rho'_1)$	a:11 *:01

After algorithm Q has processed the constraints in $\phi(P)$, the global constraints are: $\Theta =$ a:10
*:11 .

(The only relevant constraint in this case is $\rho'_1 = \rho'_1 \backslash \{a\}$, that is, the fact that ρ'_1 lacks field *a*.)

Using the value of Θ we can calculate the cleansed normal forms of the set variables:

Set Expression	Cleansed Normal Form
ρ'_1	a:00 *:01
ρ'_2	a:10 *:01
ρ'_3	a:10 *:01

By looking at the cleansed normal forms, we can conclude that sets ρ'_2 and ρ'_3 represent the same set of labels, and, consequently, rows ρ_2 and ρ_3 have the same set of fields.

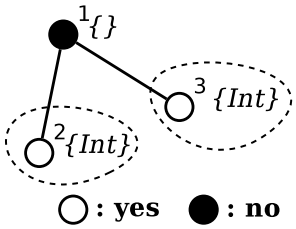

Figure 6.1: Row Construction Graph, Expression: $\lambda x.\lambda f.f\ \langle a = 7 \mid x^1\rangle^2\ \langle a = 2 \mid x^1\rangle^3$

Next, we have to check whether the field types of rows ρ_2 and ρ_3 match. The row construction graph (with vertices labelled with types for field a and connected components encircled) for predicates P is presented in Figure 6.1 (like in Section 5.8 numerical superscripts link vertices to their corresponding record expressions). By looking at the picture, we can conclude that the type of field a in ρ_2 is *Int*, that is, $\rho_2 \cdot a = Int$, and, similarly, $\rho_3 \cdot a = Int$. Thus, rows ρ_2 and ρ_3 are indeed the same, so we can substitute one away with the other using the improving substitution $[\rho_3 \mapsto \rho_2]$.

6.3 Type Simplification

While experimenting with the language and the type system, one quickly discovers that quite often the inferred type includes predicates that are superfluous. Informally, a superfluous predicate does not capture any relevant information, that is, it does not put any additional constraint on the type variables that is not already captured by other predicates. Take the following, motivating example:

$$e_1 \quad \equiv \quad \langle a = 7 \mid \langle\rangle\rangle$$

The type inferred for e_1 is:

$$(\!|\,\,\textbf{lacks}\ a,\ \rho_1 \simeq (\!|a : \mathit{Int} \mid (\!|\,|\!)\,|\!) \Rightarrow \mathit{Rec}\ \rho_1$$

Obviously, the predicate $(\!|\,\,\textbf{lacks}\ a$ is trivially true under all instantiations, thus it does not constrain the set of satisfiable instances in any way. Therefore, the following type is equivalent to the previous one:

$$\rho_1 \simeq (\!|a : \mathit{Int} \mid (\!|\,|\!)\,|\!) \Rightarrow \mathit{Rec}\ \rho_1$$

Formally, two sets of predicates P and Q are equivalent if they define the same set of satisfiable instances (this definition slightly differs from the one offered by Jones in [Jones, 1995]):

$$P \sim Q \quad \text{iff} \quad \lfloor \tau \rfloor_P = \lfloor \tau \rfloor_Q$$

A predicate π is thus superfluous if $P \sim P \backslash \{\pi\}$. A set of predicates P is minimal if there is no predicate in P that is superfluous.

Although, in general, it is difficult to say under what conditions should one set of predicates be considered simpler than some other (different metrics give different results, not to mention one's subjective taste), we nevertheless decided to settle on one particular measure: the number of predicates in the set. Thus, in this thesis, type simplification will always mean (unless otherwise stated) disposing of superfluous predicates and thus reducing the number of predicates in the type. Formally, the set of predicates P is simpler than Q, written as $P \leq Q$, if $P \subseteq Q$ and $P \sim Q$. Since finding *a* minimal set of predicates is easier than finding the *smallest* minimal set (because finding the smallest set would require enumerating *all* minimal subsets of P), what we will be looking for during type simplification is not *the* minimum (smallest), but rather *a* minimal (irreducible) set of predicates. Thus, there will be no 'principal simplified type.'

From now on, when we present inferred types, we will assume that satisfiability checking (described in Chapter 5) and type improvement (described in sections 6.1 and 6.2) have already taken place, that is, the presented types will always be principal satisfiable types.

6.3.1 Representative Cases

In this section, we present several examples for type simplification, where we arrive at a minimal set of predicates using semi-formal reasoning. Bear in mind, that our goal is not to give an exhaustive list of cases to be handled later, but rather to demonstrate the various ways a predicate can become superfluous.

Constant Predicates

In some cases, there are no row variables mentioned in the predicates that cannot be substituted away using row equality predicates. Take the following example:

$$e_1 \quad \equiv \quad \langle a = 7 \mid \langle b = 2 \mid \langle\rangle\rangle\rangle \cdot b$$

The improved type for expression e_1 is:

$$\begin{aligned} &\rho_1 \textbf{ has } (b : \mathit{Int}),\ \rho_1 \simeq (\!|a : \mathit{Int} \mid \rho_2|\!),\ \rho_2 \textbf{ lacks } a, \\ &\rho_2 \simeq (\!|b : \mathit{Int} \mid (\!|\,|\!)|\!),\ (\!|\,|\!) \textbf{ lacks } b \\ &\quad \Rightarrow \mathit{Int} \end{aligned}$$

In this example, we select a field from a record literal that is completely determined at compile-time. Intuitively, if the operation is type-correct, the type of the expression should simply be the type of the selected field without any type predicates whatsoever. More formally, observe

that there are no independent row variables in the predicate set: ρ_2 is $(\!|b : Int \mid (\!|\,|\!)|\!)$, and ρ_1 (an extension of ρ_2) is $(\!|a : Int \mid (\!|b : Int \mid (\!|\,|\!)|\!)|\!)$. In other words, we are dealing with constant predicates that are necessarily true (since we already checked that they are satisfiable). Necessarily true predicates do not capture any relevant type constraints, hence the type of expression e_1 after type simplification is simply *Int*.

Predicate Cancellation

Sometimes, the precondition of some basic operation is guaranteed by the postcondition of some other basic operation. In this situation, the predicate expressing the precondition becomes superfluous. Take the following example:

$$e_2 \quad \equiv \quad \lambda x.\langle a = 2 \mid x!a\rangle \cdot a$$

The improved type for expression e_2 is:

$$\rho_2 \textbf{ lacks } a,\ \rho_2 \simeq \rho_1 - a \ {}^{(1)},\ \rho_1 \textbf{ has } (a : \alpha),$$
$$\rho_3 \textbf{ has } (a : Int),\ \rho_3 \simeq (\!|a : Int \mid \rho_2|\!) \ {}^{(2)}$$
$$\Rightarrow Rec\ \rho_1 \rightarrow Int$$

In this example, there are two instances of predicate cancellation (cancelling pairs of predicates highlighted): (1) we extend the record $x!a$ with the field a that the record is guaranteed to lack, and (2) we select from the record $\langle a = 2 \mid ...\rangle$ the field a that the record is guaranteed to have. Actually, the only constraint expression e_2 puts on record x is that it must have the field a. Consequently, the simplified type of expression e_2 is: ρ_1 **has** $(a : \alpha) \Rightarrow Rec\ \rho_1 \rightarrow Int$.

Unreachable Predicates

A quite common case is when some predicates become superfluous because the type variables they constrain are no longer reachable (a notion we will make more precise later) from the Milner type. Take the following example:

$$e_3 \quad \equiv \quad \lambda x.\textbf{let } f = \lambda y.\langle x \parallel y\rangle \textbf{ in } x$$

The improved type for expression e_3 is:

$$\rho_3 \simeq (\rho_1 \parallel \rho_2),\ \rho_1 \# \rho_2 \Rightarrow \mathit{Rec}\ \rho_1 \rightarrow \mathit{Rec}\ \rho_1$$

In expression e_3, inside the function f, we concatenate the record x to the record y, but there is no way to supply the value of y from the outside world since f never escapes the scope of the *let* expression. A closer examination of the situation reveals that the row variable ρ_2 is not mentioned in the type and cannot be constructed from ρ_1, that is, ρ_2 is unreachable (and so is ρ_3). As a result, the type predicates, although they are not constant, do not constrain the type of record x in any way. The simplified type of e_3 is thus only $\mathit{Rec}\ \rho_1 \rightarrow \mathit{Rec}\ \rho_1$.

Parallel Construction

When a row is constructed in several different ways from the same rows, it often happens that only certain predicates are required to describe the construction of the row, making predicates that lie on ‘parallel construction paths’ superfluous. Take the following example:

$$e_4 \quad \equiv \quad \lambda x.\lambda y.\textbf{if } \mathit{True} \textbf{ then } \langle x \parallel y\rangle \textbf{ else } \langle\langle x \backslash y\rangle \parallel y\rangle$$

The improved type for expression e_4 is:

$$\rho_3 \simeq (\rho_1 \parallel \rho_2),\ \rho_1 \# \rho_2,\ \rho_3 \simeq (\rho_4 \parallel \rho_2),\ \rho_4 \# \rho_2,\ \rho_4 \simeq (\rho_1 \backslash \rho_2)$$
$$\Rightarrow \mathit{Rec}\ \rho_1 \rightarrow \mathit{Rec}\ \rho_2 \rightarrow \mathit{Rec}\ \rho_3$$

Predicates mirror the construction of records at the level of rows. For instance, the row ρ_3 is constructed in two different ways (as a result of constructing the same record in the two branches of the *if* expression). Observe, however, that only one way of construction is actually needed in this situation, because in both cases the resulting record is the concatenation of the records *x* and *y*. Consequently, during type simplification, we have a choice between two minimal sets of predicates (representing the two different ways of construction):

- $\rho_3 \simeq (\rho_1 \parallel \rho_2),\ \rho_1 \# \rho_2 \Rightarrow \mathit{Rec}\ \rho_1 \rightarrow \mathit{Rec}\ \rho_2 \rightarrow \mathit{Rec}\ \rho_3$

- $\rho_3 \simeq (\rho_4 \parallel \rho_2),\ \rho_1 \# \rho_2,\ \rho_4 \simeq (\rho_1 \backslash \rho_2) \Rightarrow \mathit{Rec}\ \rho_1 \rightarrow \mathit{Rec}\ \rho_2 \rightarrow \mathit{Rec}\ \rho_3$

The type simplification algorithm halts as soon as it has found a minimal set of predicates, and there is no guarantee that it will find the smallest possible set. Thus, the actual minimal set chosen by the algorithm can be considered somewhat of an accident of implementation.

6.4 Algorithm for Type Simplification

The goal of type simplification is to find a minimal set of predicates by removing superfluous predicates from a given set of predicates P. The algorithm for type simplification consists of the following steps: (1) identify *reachable* predicates using the Milner type; (2) identify *relevant* predicates among reachable predicates; and (3) provide support using *constructor* predicates for unsupported row variables appearing in the relevant predicates or in the Milner type. In the rest of the chapter, we assume that satisfiability checking and type improvement have already taken place before attempting type simplification.

6.4.1 Identifying Reachable Predicates

Informally, *reachable* predicates are those predicates that may directly or indirectly constrain some type variables appearing in the Milner type. Predicates that are not reachable are guaranteed to be superfluous, since they capture constraints on type variables that cannot be instantiated through unification. Take the following expressions and their improved types:

Expression	**Improved Type**
$e_1 \equiv \langle a : Int \mid \langle\rangle \rangle \cdot a$	$P_1 \equiv \rho_1$ **has** $(a : Int),\ (\!\|\|\!)$ **lacks** $a,\ \rho_1 \simeq (\!\|a : Int \mid (\!\|\|\!)\|\!)$
	$\tau_1 \equiv Int$
$e_2 \equiv \lambda x.x!a$	$P_2 \equiv \rho_1$ **has** $(a : \alpha),\ \rho_2 \simeq \rho_1 - a$
	$\tau_2 \equiv Rec\ \rho_1 \rightarrow Rec\ \rho_2$
$e_3 \equiv \lambda x.$**let** $f = \lambda y.\langle x \parallel y \rangle$ **in** $x \cdot a$	$P_3 \equiv \rho_3 \simeq (\rho_1 \parallel \rho_2),\ \rho_1 \# \rho_2,\ \rho_1$ **has** $(a : \alpha)$
	$\tau_3 \equiv Rec\ \rho_1 \rightarrow \alpha$

For expression e_1, there are no type variables in the Milner type τ_1 (since *Int* is a monotype), so no predicate in P_1 is reachable. For expression e_2, both predicates in P_2 are reachable since they constrain row variables ρ_1 and ρ_2 appearing in the Milner type τ_2. As for expression e_3, only the predicate ρ_1 **has** $(a : \alpha)$ is reachable.

To better understand why unreachable predicates are necessarily superfluous, take for example the predicate $\rho_1 \# \rho_2$ from P_3. Since ρ_2 is unknown, and there is *no way* that it can become known, what predicate $\rho_1 \# \rho_2$ says is that row ρ_1 is disjoint with *some* other row ρ_2. But we do not know anything about row ρ_2! Hence, we can safely ignore predicate $\rho_1 \# \rho_2$ since it is always true that row ρ_1 is disjoint with *some* row ρ_2.

We can calculate the set of reachable predicates $Rch \subseteq P$, relative to some type τ, using the following algorithm (where *rvs* is the set of currently reachable row variables, and predicates like *lacks* or *has* are processed in their expanded form):

1. Initialize $Rch = \emptyset$, and $rvs = ftv^{\text{row}}(\tau)$.

2. If there is a predicate $\pi = (r_1 \simeq r_2) \in P$, such that the row expression r_1 is not a single row variable and $\emptyset \neq ftv^{\text{row}}(r_1) \subseteq rvs$ (or the same holds for row expression r_2), then update:

   (a) $Rch \leftarrow Rch \cup \{\pi\}$

   (b) $rvs \leftarrow rvs \cup \{ftv^{\text{row}}(\pi)\}$

3. If *Rch* has changed in the last iteration, go to Step 2, otherwise, halt.

The restriction in Step 2 of the algorithm, that we do not consider a row equality predicate reachable if the reachable side consists of only a single row variable, makes the definition of reachable

predicates much tighter. For example, without the restriction, the predicate $\pi = \rho_1 \simeq (\rho_2 \parallel \rho_3)$ would be considered reachable, even if only ρ_1 appears in the Milner type, which is undesirable, since predicate π is completely irrelevant unless we know something about both ρ_2 and ρ_3. Also, notice that, according to the algorithm, if a predicate does not contain row variables (the set of free row variables on both sides of *row equality* is empty), then it cannot be reachable. Finally, if the there are no row variables in the Milner type τ, then the set of reachable predicates is necessarily empty.

6.4.2 Identifying Constructor Predicates

In Section 5.7 we presented an algorithm for identifying base variables in a set of predicates. Informally, base variables are independent row variables, that is, row variables in terms of which all other *derived* (non-base) row variables can be expressed. Take for example the following set of predicates:

$$P \quad \equiv \quad \rho_3 \simeq \rho_1 - a,\ \rho_1 \textbf{ has } (a : \alpha),\ \rho_4 \simeq \rho_2 \parallel \rho_1,\ \rho_2 \# \rho_1$$

The base variables in P are ρ_1 and ρ_2, while the derived variables are ρ_3 and ρ_4. The predicates $Ctr \subseteq P$ that are used to 'construct' all derived row variables are called *constructor* predicates. In the above example, there are two constructor predicates: $Ctr = \rho_3 \simeq \rho_1 - a,\ \rho_4 \simeq \rho_2 \parallel \rho_1$. We remark that, since the set of base variables is not unique (explained in Section 5.7), neither are the set of constructor predicates. For instance, take predicates $\rho_1 \simeq \rho_2 - a,\ \rho_2 \simeq (\!|a : \alpha \mid \rho_1|\!)$: either row variable ρ_1 or ρ_2 can be used as the base variable, the choice being arbitrary.

6.4.3 Identifying Relevant Predicates

As algorithm Q processes set equality constraints in $\phi(P)$ (see Section 5.6), it records the consequences (in the form of disallowing the presence of certain labels in certain regions) of said constraints. However, not all constraints convey new information about the problem. Only when the normal forms on the two sides of a set equality constraint are different, can we learn something new. However, in the case of constructor predicates, the two sides always have the same normal form, since we use one side to define the other side (a derived variable). In a sense, constructor predicates are assumed to be true and we check the remaining predicates against them. Now, if the processing of a non-constructor predicate leaves the global presence/absence constraints Θ unchanged, then the predicate in question is irrelevant, in the sense that its consequences follow from the predicates processed before it. Unfortunately, since it depends on the processing order, this approach does not, in itself, necessarily find a minimal set of relevant predicates.

In order to find a minimal set of relevant predicates, we have to extend Θ, so that we can capture the *evolution* of label presence/absence constraints. The idea is, that after each update of Θ, we also record which predicate caused the change (this information will also form the basis of error-reporting in Chapter 7). Actually, we need to do more: when updating Θ with the consequences of some predicate π, we need to check whether it *subsumes* the consequences of some previous predicate (which we are now able to do, using the log of updates to Θ). Consider the following set of predicates:

$$P \quad \equiv \quad \rho_3 \simeq \rho_1 \parallel \rho_2,\ \rho_1 \textbf{ lacks } a,\ \rho_3 \textbf{ lacks } a$$

Set Expression	Normal Form
ρ'_1	`a:0101` `*:0101`
ρ'_2	`a:0011` `*:0011`
$\rho'_3 \quad (= \rho'_1 \cup \rho'_2)$	`a:0111` `*:0111`
$\rho'_1 \backslash \{a\}$	`a:0000` `*:0101`
$\rho'_3 \backslash \{a\}$	`a:0000` `*:0111`

Table 6.1: Normal Forms for Set Expressions in P

Since ρ_1 is a subset of ρ_3, the predicate ρ_3 **lacks** a subsumes the predicate ρ_1 **lacks** a. To see how we can discover this fact, take a look at the steps taken by algorithm Q:

1. Convert predicates P to set constraints $\phi(P) = \{\rho'_3 = \rho'_1 \cup \rho'_2,\ \rho'_1 = \rho'_1 \backslash \{a\},\ \rho'_3 = \rho'_3 \backslash \{a\}\}$.

2. Identify the set of base variables as $BV = \{\rho_1,\ \rho_2\}$. The set of constructor predicates is thus $Ctr = \{\rho_3 \simeq \rho_1 \parallel \rho_2\}$.

3. Calculate the normal forms of the set expressions in Table 6.1 (we processes the *lacks* predicate in its expanded form).

4. After processing the constraint $\pi_1 \equiv \rho'_1 = \rho'_1 \backslash \{a\}$, the global constraints are $\Theta =$ `a:1010` `*:1111`. (We do not have to process the constructor predicate $\rho'_3 = \rho'_1 \cup \rho'_2$, since we used it to define the normal form of ρ'_3.) In other words, we learned that the field a cannot be in regions R_1 and R_3, that is, in ρ'_1.

5. Next, the constraint $\pi_2 \equiv \rho'_3 = \rho'_3 \backslash \{a\}$ changes the global constraints to $\Theta = \begin{smallmatrix} \texttt{a:1000} \\ \texttt{*:1111} \end{smallmatrix}$. However, the information gathered from the constraint π_2 subsumes that of constraint π_1. This is because the set of 'forbidden' regions $\{1,\ 3\}$ implied by π_1 for label a, is a subset of the set of the forbidden regions $\{1,\ 2,\ 3\}$ implied by π_2 for the same label. The constraint π_2 captures all the constraints as π_1, thus it makes π_1 superfluous.

We can now define *relevant* predicates $Rel \subseteq Rch$ as those non-constructor predicates that cause an update to the global constraints Θ during set constraint solving *and* whose effect on the global constraints is not subsumed by some other predicate (which we ensure by keeping track of the evolution of global constraints).

6.4.4 Putting It All Together

After we have identified relevant predicates, we are still not done, as the last step is to provide support for row variables mentioned in the relevant predicates or in the Milner type. The problem is that relevant predicates, by definition, exclude constructor predicates, so using only relevant predicates in the simplified type might result in 'dangling' row variables, that is, row variables without support (way of construction). For instance, consider the expression $e \equiv \lambda x.x!a$. The improved type for expression e is $P \Rightarrow \tau \equiv \rho_2 \simeq \rho_1 - a,\ \rho_1$ **has** $(a : \alpha) \Rightarrow Rec\ \rho_1 \rightarrow Rec\ \rho_2$. The row variable ρ_1 is the only base variable, and predicate $\rho_2 \simeq \rho_1 - a$ is the only constructor predicate. The only relevant predicate in this case is $Rel = \rho_1$ **has** $(a : \alpha)$. However, it would be incorrect to use ρ_1 **has** $(a : \alpha) \Rightarrow Rec\ \rho_1 \rightarrow Rec\ \rho_2$ as the simplified type, since it leaves row variable ρ_2 dangling. The problem is that $P \nsim Rel$, which violates the fundamental requirement

of type simplification, that it should not alter the set of satisfiable instances.

The way to handle dangling row variables is to use constructor predicates to provide support for them. Since constructor predicates, by definition, are able to construct all derived (non-base) row variables from base row variables, we can use them to construct dangling row variables. In the following algorithm, P' denotes the simplified set of predicates being built, *drv* is the set of dangling row variables, *BV* is the set of base variables, *Ctr* is the set of constructor predicates, and *Rel* is the set of relevant predicates. We also assume that each constructor predicate has the row variable on its left-hand side:

1. Initialize $P' = Rel$ and $drv = (ftv^{\text{row}}(Rel) \cup ftv^{\text{row}}(\tau))\backslash BV$.

2. If $drv = \varnothing$, then halt.

3. With a row variable $\rho \in drv$ and predicate $\pi = (\rho \simeq r) \in Ctr$, update:

   (a) $P' \leftarrow P' \cup \{\pi\}$

   (b) $drv \leftarrow (drv \cup ftv^{\text{row}}(\pi))\backslash\{\rho\}\backslash BV$

4. Goto Step 2.

Observe, that in the initialization step, we do not consider base row variables as dangling, since, by definition, they cannot (and need not) be constructed from other row variables.

When the above algorithm finishes, P' will be a minimal set of predicates that defines the same set of satisfiable instances as P, formally, $P' \leq P$ and there is no $Q \subset P'$ such that $Q \sim P$. We call $P' \Rightarrow \tau$ the *simplified* type.

Chapter 7
Explaining Type Errors

In this chapter, we will turn to the problem of generating informative error messages for ill-typed expressions. We do not present a fully-fledged algorithm (it could well be the subject of a separate dissertation on its own), only a proposal for explaining complex errors using an interactive approach. There has been a lot of research on how to generate useful error messages in polymorphic languages, including a proposal for an interactive Q&A-like system by Beaven and Stansifer in [Beaven and Stansifer, 1993]. Pure unification based type inference algorithms are actually constraint solvers where the constraints are all type equality constraints. In a qualified type system the type checking algorithm has to deal with both type equality constraints *and* predicates on row variables. Type errors in these systems translate to an unsatisfiable, conflicting constraint set. The chief task of the type checker is to somehow report the *cause* of the conflict in a way that the programmer will find it useful in locating the error in her program. The type system of the language presented in this thesis is an extension of that of ML, thus it felt natural to base the type explanation algorithm off one designed for ML [Beaven and Stansifer, 1993], which we will briefly discuss in the next section.

7.1 Explaining Type Errors in Polymorphic Languages

The type inference algorithm *W* presented in Section 4.6, like most type checking algorithms, is syntax directed and compositional: the type of an expression is calculated bottom-up from the types of its subexpressions. When type inference fails in ML, the reason for failure is always a unification failure: two types that should be the same fail to unify. The result of successful type unification is a substitution that summarizes the consequences of type equalities. The type assignment provides the context in which type inference takes place, one of its roles being the communication of type constraints between different subexpressions of an expression, that is, between different branches of the abstract syntax tree. In practice, during type checking there is always an initial context (type assignment) which contains type bindings for the primitives and library functions of the language.

The general approach taken in [Beaven and Stansifer, 1993] is to augment data structures used by the type inference algorithm with information that can later be used by explanation functions to pinpoint the cause of type errors. Nodes in the abstract syntax tree are annotated by the type inferred for their subexpressions by the type inference algorithm. This way we can now what types were unified as a consequence of each expression. Nodes that represent identifiers are treated somewhat differently, because we also have to keep track of type variable renamings that take place whenever a generic type variable (introduced by let-bound polymorphism) in the type of the identifier is instantiated. Substitutions are represented as a list of atomic bindings of type variables to types. This ensures that there is always a single cause for any binding, simplifying error explanation. Each atomic binding is annotated with a pointer into the abstract syntax tree

to the node that caused the unification call that resulted in the binding in question. This way we can trace the evolution of a type to its final form.

There were two explanation functions defined in [Beaven and Stansifer, 1993]: *Why* and *How*. The function *Why* tries to explain why a certain expression was assigned the type it was assigned, while the function *How* attempts to answer the question how a type variable got assigned a certain type. The function *Why* uses structural information from the annotated abstract syntax tree to answer questions, while the function *How* uses annotated atomic bindings to follow the evolution of types (representing information flow between different parts of an expression). Rather than providing a detailed treatment of the explanation functions, we adhere to the old adage of "a picture is worth a thousand words" (*not* a Chinese proverb!) and present a sample interaction with the system in Figure 7.1 (taken from [Beaven and Stansifer, 1993]). The expression that caused the type error (attempting to add an integer to a boolean) for which we are seeking explanation is the following:

```
(fn a => +((fn b => if b then b else a) true, 3))
```

The explanation functions in the original paper were not interactive (they simply generated a depth-first traversal of all the reasons leading to a type error), but they could easily be converted to ask the user which branch of the explanation should be explored next. Obviously, as programs get larger, this Q&A style becomes more advantageous.

7.2 Type Errors and Qualified Types

The theory of qualified types extends the standard Milner type with type predicates to constrain the instantiation of universally quantified type variables (see Chapter 4). The ability to statically

```
A type error was detected in the application ‘(+ (#,3))’.
The domain of the function ‘+’ is not unifiable with the type of the argument ‘((# true) ,3)’.
Domain of function is ‘(int*int)’.
The argument has type ‘(bool*int)’.
**Why does the function ‘+’ have type ‘((int*int)->int)’?
  The identifier ‘+’ was assigned type ‘((int*int)->int)’ as part of the initial environment.
**Why does the argument ‘((# true) ,3)’ have type ‘(bool*int)’?
  Type of the pair ‘((# true) ,3)’ is determined by type of each element.
  **Why does the first element ‘((fn b => #) true)’ have type ‘bool’?
    The type of an application is the range of the function.
    The function ‘(fn b => if #)’ has type ‘(bool->bool)’.
    **Why does the fuction ‘(fn b => if #)’ have type ‘(bool->bool)’?
       The type of a function definition is determined by the type inferred for
       the formal parameter ‘b’ and by the type of the function body ‘if b then b else a’.
       **Why does formal parameter ‘b’ have type ‘bool’?
         The type of a formal parameter is inferred from its use.
         The variable ‘b’ was initially assigned type variable ‘’b’.
         **How did type variable ‘’b’ come to be bound to type ‘bool’?
           This binding arose during analysis of ‘if b then b else a’.
           Since the expression ‘b’ must have type ‘bool’ and the type variable ‘’b’ is its type,
           then the type variable ‘’b’ must stand for ‘bool’.
           **Why does the expression ‘b’ have type ‘bool’?
             [Elided.]
      **Why does functions body ‘if b then b else a’ have type ‘bool’?
        The type of a conditional is determined by the the types of its branches which must unify.
        **Why does the "then" branch ‘b’ have type ‘bool’?   [Elided.]
        **Why does the "else" branch ‘a’ have type ‘bool’?   [Elided.]
    **Why does the second element ‘3’ have type ‘int’?
      All Integer constants have type ‘int’.
```

Figure 7.1: Explanation of a Type Error

type check basic record operations derives from the fact that we can precisely describe their types using type predicates. As a direct consequence, the type inference algorithm *W* alone is not able to decide whether an expression is ill-typed or not. In other words, being accepted by algorithm *W* is a necessary but not sufficient condition for well-typedness. It is the task of the satisfiability checking algorithm Q (see Chapter 5) to ensure that there exists an instantiation of type variables that make all type predicates true. If there is no such instantiation, then the type predicates are unsatisfiable, and thus should be rejected by the type checker. When an expression is ill-typed because of a type unification failure, we can use the approach presented in [Beaven and Stansifer, 1993] to explain the type error. On the other hand, if it is ill-typed because of the unsatisfiability of predicates, we need a different approach.

The problem of how to explain why a set of type predicates (in our case, row predicates) can be broken into the following, somewhat orthogonal, subproblems: (1) showing which expression introduced originally the predicates in the final predicate set; (2) identifying a conflicting set of predicates; and (3) revealing, step by step, the conflict (contradiction) in the conflicting set. Before we develop algorithms for addressing these problems, we would like to emphasize that what we are trying to do is explaining type errors caused by the unsatisfiability of predicates. The reason for this is that, if the type inference algorithm *W* itself fails, then the approach described in [Beaven and Stansifer, 1993], without the *slightest* of modifications, is 100 per cent applicable to task of explaining type errors in our system (another triumph of modularity for the theory of qualified types). That is, in the cases we will handle, the type inference algorithm has successfully derived a qualified type for the ill-typed expression in question.

7.2.1 Showing the Origins of Predicates

Each type predicate that appears in the final set of predicates returned by the type inference algorithm must have been introduced by some subexpression. After its original introduction, a type predicate can change several times as type variables in the predicate go through an evolution similar to the one experienced by type variables in the type assignment. The approach in [Beaven and Stansifer, 1993] takes care of keeping track of how type variables get bound to their final type, so all that needs to be done is to augment type predicates with a pointer into the abstract syntax tree to the expression that introduced them into the current predicates set, and a list of instantiations of generic type variables used in the predicate. This way each predicate can be traced back to its original source and form, and then, using information on the evolution of type variables, can be shown to acquire its final form.

The information where a predicate is originated from becomes interesting when reviewing the set of predicates that caused the conflict. The investigative user then could be provided the option to be presented with explanations on how each predicate has come into being, and got into its current form. The function *Where* could be used to answer the question where a certain predicate comes from.

The above described rosy situations is bit complicated by the fact that it is possible that predicates that have been introduced by different expressions, 'collapse' into a single predicate because of the result of type substitutions. For example, the predicate set ρ_1 **lacks** a, ρ_2 **lacks** a becomes simply ρ_1 **lacks** a if we apply the substitution $[\rho_2 \mapsto \rho_1]$ to it. In other words, some predicates in the final set might actually stand for several predicates, introduced at different

points of the program. Can this create problem? As it turns out, hardly. After all, the main problem is that the expressions in the program generated a conflicting set of predicates, and it does not really matter if more than one expression contributed a certain predicate to the final set, as long as we can link the predicate to at least one of its contributors, which we can.

7.2.2 Identifying Conflicting Predicates

From the point of view of error explanation, it is beneficial to be able to work with a minimal set of conflicting predicates (that is, where no subset is unsatisfiable). To reveal the contradiction in a larger set of predicates, it is useful to be presented with a smaller subset that still exhibits the conflict. Recall that we have already faced a similar problem during type simplification (see Chapter 6) when we were trying to identify a minimal set of predicates that expresses the same constraints as some larger set. As it turns out, the exact same algorithm, without modification, works when the predicates in question are not satisfiable. This is due to the particular way algorithm Q identifies unsatisfiable sets of predicates: when there are conflicting presence/absence constraints on a particular field, it manifests itself as a global constraint telling us that the particular field cannot appear in any of the regions. From the point of view of type simplification, this a perfectly legitimate constraint, so it is meaningful to ask: “what is a minimal subset of predicates that still expresses the same (contradictory) constraints on the field in question?” When there are several fields with conflicting presence/absence constraints, we can identify separate, and most likely different, conflicting sets for each. The user then could be presented with a choice to choose for which field we should present and explain the conflicting set of predicates.

7.2.3 Revealing the Contradiction

After identifying a minimal conflicting set of predicates for some field ℓ, the next step is to show *why* they are in conflict. As it turns out, this is simpler than it might seem at first. After all, when there is a conflict, it means that there is no instantiation of row variables that would satisfy the predicates. Because the conflict concerns the presence/absence of fields in rows, it is possible to provide a step by step explanation by making assumptions on the base row variables, then proceeding with the evaluation of constructor predicates, until some relevant predicate is violated (using the classification of predicates introduced in Chapter 6). In other words, we depend on algorithm Q and the algorithms used for type simplification to break up predicates into various categories so that we can explain the contradiction. Informally, base row variables are like independent variables in a system of equations, and we reveal that the equations are unsolvable by considering all possible combinations of values for the independent variables. Luckily, since we are only interested in the presence/absence of a particular field ℓ (we are not even interested in its type), the number of possibilities is only *two* per row variable. Take, for example, the following ill-typed expression (we concatenate the record x to itself, forcing it to be the empty record, so it cannot have the field a):

$$e_1 \quad \equiv \quad \lambda x.\langle x \parallel x\rangle \cdot a$$

The type inferred for expression e_1:

$$P_1 \quad \equiv \quad \rho_2 \ \textbf{has}\ (a : \alpha),\ \rho_2 \simeq (\rho_1 \parallel \rho_1),\ \rho_1 \# \rho_1$$

$$\tau_1 \quad \equiv \quad \mathit{Rec}\ \rho_1 \rightarrow \alpha$$

The conflicting set in this case is the whole of P_1. To clarify how the error explanation could work, we present an extract from a hypothetical type error explanation session in Figure 7.2.

```
[Elided.]
The following predicate set is in conflict for field ‘a’:
 (a)  r2 has (a:a)
 (b)  r2 = (r1 || r1)
 (c)  r1#r1
Explanation:

 (1)  r1 has (a:b)     Assumption.
 Contradiction!        (1) + (c)

 (1)  r1 lacks a       Assumption.
 (2)  r2 = (r1 || r1)  (b)
 (3)  r2 lacks a       (1) + (2)
 Contradiction!        (3) + (a)
```

Figure 7.2: Explaining Predicate Conflicts Using Assumptions

It is not always necessary to make assumptions when explaining the contradiction in a set of type predicates. Actually, in most cases, the contradiction can be revealed without complicating the explanation with any assumptions at all. The reason for this is that there can be *trivial* type predicates that simply require the presence/absence of a particular field in a base row variable. In that case, there is no need to make the opposite assumption since it will obviously fail. Take the following ill-typed expression:

$$e_2 \quad \equiv \quad \lambda x.\lambda y.\langle x \parallel y\rangle \cdot a + x \cdot a == y \cdot a$$

```
[Elided.]
The following predicate set is in conflict for field ‘a’:
 (a)  r1#r2
 (b)  r1 has (a:Int)
 (c)  r2 has (a:Int)
Explanation:
 (1)  r1 has (a:Int)   (b)
 (2)  r2 has (a:Int)   (c)
 Contradiction!        (1) + (2) + (a)
```

Figure 7.3: Explaining Predicate Conflicts Without Assumptions

The type inferred for expression e_2:

$$P_2 \quad \equiv \quad \rho_3 \textbf{ has } (a : Int),\ \rho_3 \simeq (\rho_1 \parallel \rho_2),\ \rho_1 \# \rho_2,\ \rho_1 \textbf{ has } (a : Int),\ \rho_2 \textbf{ has } (a : Int)$$

$$\tau_2 \quad \equiv \quad Rec\ \rho_1 \rightarrow Rec\ \rho_2 \rightarrow Bool$$

The expression is ill-typed since we are selecting the field *a* from both records *x* and *y*, ensuring that they are *not* disjoint, while attempting to concatenate them. The two base row variables in this case are ρ_1 and ρ_2, but there is no need to make any assumptions during the explanation of the conflict, since there are two trivial *has* predicates that put field *a* in both ρ_1 and ρ_2. We present an extract from a hypothetical type error explanation session for e_2 in Figure 7.3 that does not use assumptions.

The hypothetical error explanation sessions presented in figures 7.2 and 7.3 look suspiciously like formal proofs where we start with a set of axioms, inference rules, and try to arrive at a contradiction. In fact, this is the case, and although the task of deriving proofs like the ones presented might seem daunting, closer examination reveals that our ‘proofs’ are always of an

extremely simple structure. To see this, first, observe that given base row variables we can always use constructor predicates to calculate the values of derived row variables. Second, we are only dealing with the presence/absence of single field at any time, so practically we are working inside a propositional logic system, which greatly simplifies the application of inference rules. Finally, the set we are working with is conflicting, so every instantiation of row variables is guaranteed to make at least one predicate false.

Chapter 8
Conclusions

We have presented a language that supports polymorphic record operations through basic record operations, such as field selection, field deletion, record extension, symmetric record concatenation, record difference, and record projection. We have demonstrated, through examples, how the basic operations can be used to define polymorphic relational operators such as *join* and *divide*. We have described the type system of the language, an application of the theory of qualified types with ML-style type inference, and presented a set of type predicates that describe the type-correct usage of basic operations. We have showed that checking the satisfiability of type predicates is exponential, not only in the number of row variables, but also in the number of labels. To eliminate exponential complexity in the number of labels, thus making the language useful for practical purposes, we have imposed a restriction on the set of valid expressions in the language. We have argued, based on experience with the implementation, that the restriction poses no obstacle in writing the kind of programs the language was intended for. We have presented a predicate satisfiability checking algorithm for the restricted language. The complexity of the algorithm is polynomial in the number of record labels and in the number of predicates, and exponential only in the number of independent row variables. The latter, we have argued, is proportional to the number of function arguments in polymorphic function

definitions, a relatively small number, thus making the algorithm practical. To make inferred types more accurate and concise, we have developed algorithms for improving and simplifying types. We have illustrated how type improvement can make types more precise, and also that type simplification finds a minimal set of type predicates. In the last chapter, we have outlined an approach to explaining type errors in the presence of complex record operations as an extension of a type explanation solution for ML. The type-inference algorithm, the predicate satisfiability checking algorithm, and the type improvement and simplification algorithms have all been implemented in Haskell and thoroughly tested, which has greatly increased our confidence in the results we have presented.

8.1 Future Work

Although some work has been done on the implementation of the interactive type explanation algorithm presented in Chapter 7, it is far from complete and definitely not yet an integral part of the system. As with all interactive type error explanation systems, only experience with actual users can tell us how useful it really is, and suggest points for improvement.

More research needs to done on how to efficiently implement polymorphic record operations, which is something that must be addressed before any practical implementation of the language is attempted. It is known that for the record calculus of Gaster and Jones [Gaster and Jones, 1996], there exists an efficient compilation method that is based on the fact that *lacks* predicates can be translated to integer offsets in records for runtime field access, allowing complete type erasure at compile-time. Unfortunately, it is unlikely that such a simple and efficient method of compilation can be found for the stronger record calculus of the system in the current thesis.

Bibliography

Alexander Aiken. Set constraints: Results, applications, and future directions. In *PPCP'94: Principles and Practice of Constraint Programming, Proceedings*, pages 326–335. Springer, 1994. ISBN 3-540-58601-6.

Alexander Aiken and Edward L. Wimmers. Solving systems of set constraints. In *LICS'92: IEEE Symposium on Logic in Computer Science*, pages 329–340. IEEE Press, 1992.

Malcolm P. Atkinson and Peter Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2):105–170, 1987. ISSN 0360-0300.

Malcolm. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. An approach to persistent programming. In *Readings in object-oriented database systems*, pages 141–146. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

François Bancilhon and Peter Buneman, editors. *Advances in Database Programming Languages*. ACM Press, New York, NY, 1990.

D. Bartels and J. Robie. Persistent objects and object-oriented databases for C++. *C++ Report*, 4(7):49–50, 52–56, 1992.

Mike Beaven and Ryan Stansifer. Explaining type errors in polymorphic languages. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 2(1-4):17–30, 1993. ISSN 1057-4514.

Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, Reading, MA, 1995.

Peter Buneman and Atsushi Ohori. Polymorphism and type inference in database programming. *ACM Transactions on Database Systems*, 21(1):30–76, 1996. ISSN 0362-5915.

Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*. CRC Press, Boca Raton, FL, 1997.

Edgar F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6): 377–387, 1970. ISSN 0001-0782.

Robert M. Colomb. *Deductive Databases and their Applications*. Taylor & Francis, London, UK, 1998.

William R. Cook and Ali H. Ibrahim. Integrating programming languages & databases: What's the problem? `http://www.cs.utexas.edu/~wcook/projects/dbpl/`, 2005.

Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL '82: Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

Chris J. Date. *Relational Database: Writings 1985-1989*. Addison-Wesley, 1990.

Chris J. Date. *Introduction to Database Systems (7th edition)*. Addison Wesley, 1999.

Chris J. Date and Edgar F. Codd. The relational and network approaches: Comparison of the application programming interfaces. In *FIDET '74: Proceedings of the 1974 ACM SIGFIDET*

(now SIGMOD) workshop on Data description, access and control, pages 83–113, New York, NY, USA, 1975. ACM Press.

Chris J. Date and Hugh Darwen. Into the great divide. In *Relational Database Writings: 1989-1991*, pages 155–168. Addison-Wesley, Reading, MA, 1992a.

Chris J. Date and Hugh Darwen. *Relational Database Writings: 1989-1991*. Addison-Wesley, Reading, MA, 1992b.

Chris J. Date and Hugh Darwen. *Relational Database Writings: 1991-1994*. Addison-Wesley, 1995.

Chris J. Date and Hugh Darwen. *Foundation for Object / Relational Databases: The Third Manifesto*. Addison-Wesley, 1998.

Chris J. Date, Hugh Darwen, and David McGoveran. *Relational Database Writings: 1994-1997*. Addison-Wesley, 1998.

Klaus R. Dittrich. Object-oriented database systems: The notion and the issues. In *On Object-Oriented Database System*, pages 3–12. Springer-Verlag, 1991.

Benedict R. Gaster and Mark P. Jones. A Polymorphic Type System for Extensible Records and Variants. Technical Report NOTTCS-TR-96-3, Department of Computer Science, Nottingham, 1996.

Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *POPL '91: Principles of Programming Languages, Proceedings*, pages 131–142. ACM Press, 1991.

Rod Johnson. *J2EE Development Without EJB*. Hungry Minds Inc, U.S., 2004.

Mark P. Jones. A theory of qualified types. In *ESOP'92: European Symposium on Programming, Proceedings*, pages 287–306. Springer, 1992.

Mark P. Jones. Simplifying and improving qualified types. In *Functional Programming Languages and Computer Architecture*, pages 160–169, 1995.

Mark P. Jones and John C. Peterson. Hugs 98: A functional programming system based on haskell 98 - user manual, 1999. URL `http://citeseer.ist.psu.edu/334009.html`.

Simon Peyton Jones and et al. *Haskell 98 Language and Libraries, the Revised Report*. Cambridge University Press, 2003.

Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1978.

Won Kim. Object-oriented database systems: Strengths and weaknesses. *Journal of Object-Oriented Programming*, pages 21–29, July 1991.

Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004.

Robert Kowalski. Predicate logic as a programming language. *Information Processing*, 74: 569–574, 1974.

Daan Leijen. First-class labels for extensible rows. Technical Report UU-CS-2004-51, Department of Computer Science, Universiteit Utrecht, 2004.

Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *DSL'99: 2nd USENIX Conference on Domain Specific Languages*, pages 109–122, 1999. Also appeared in ACM SIGPLAN Notices 35, 1, (Jan. 2000).

Henning Makholm and J. B. Wells. Type inference, principal typings, and let-polymorphism for first-class mixin modules. In *ICFP'05: International Conference on Functional programming, Proceedings*, pages 156–167. ACM Press, 2005.

Jim Melton and Alan R. Simon. *SQL: 1999 - Understanding Relational Language Components*. Morgan Kaufmann, 2001.

Randy Meyers. The new C: Introducing C99. *C/C++ Users Journal*, 18(10):49–53, October 2000.

Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

Jack Minker. Logic and databases: Past, present, and future. *AI Magazine*, 18(3):21–47, 1997.

Lajos Nagy and Ryan Stansifer. Polymorphic type inference for the relational algebra in the functional database programming language Neon. In *Proceedings of the 2006 ACM Symposium of Applied Computing*, pages 673–679, New York, NY, USA, 2005. ACM Press.

Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, 1995. ISSN 0164-0925.

Atsushi Ohori and Peter Buneman. Type inference in a database programming language. In *LFP '88: LISP and Functional Programming, Proceedings*, pages 174–183. ACM Press, 1988.

Benjamin Pierce. *Types and Programming Languages*. The MIT Press, 2002.

François Pottier. A constraint-based presentation and generalization of rows. In *LICS'03: Logic in Computer Science, Proceedings*, pages 331–340, 2003.

Raghu Ramakrishnan and S. Sudarshan. Top-Down vs. Bottom-Up Revisited. In Vijay Saraswat and Kazunori Ueda, editors, *Proceedings of the 1991 International Symposium on Logic Programming*, pages 321–335. MIT, 1991.

Shuping Ran, Paul Brebner, and Ian Gorton. The rigorous evaluation of Enterprise Java Bean technology. In *ICOIN'01: International Conference on Information Networking, Proceedings*, page 93, 2001.

Manuel Reimer. Implementation of the database programming language Modula/R on the personal computer Lilith. *Software—Practice and Experience*, 14(10):945–956, October 1984.

Raymond Reiter. Towards a logical reconstruction of relational database theory. In *On Conceptual Modelling*, pages 191–233. Springer-Verlag, 1982.

Didier Rémy. Type checking records and variants in a natural extension of ML. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 77–88, New York, NY, USA, 1989. ACM Press.

Didier Rémy. Typing record concatenation for free. In *POPL'92: Principles of Programming Languages, Proceedings*, pages 166–176. ACM Press, 1992.

J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of ACM*, 12(1):23–41, 1965. ISSN 0004-5411.

Amr Sabry. What is a purely functional language? *Journal of Functional Programming*, 8(1): 1–22, 1998.

Joachim W. Schmidt. Some high level language constructs for data of type relation. *ACM Transactions on Database Systems*, 2(3):247, 1977.

Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The implementation of mercury, an efficient purely declarative logic programming language. In *ILPS Workshop: Implementation Techniques for Logic Programming Languages*, 1994.

Mads Torgersen. Language integrated query: unified querying across data sources and programming languages. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 736–737, New York, NY, USA, 2006. ACM Press.

Jan Van den Bussche and Stijn Vansummeren. Polymorphic type inference for the named nested relational calculus. *ACM Transactions on Computational Logic*, 2005.

Jan Van den Bussche and Emmanuel Waller. Type inference in the polymorphic relational algebra. In *PODS'99: Principles of Database Systems, Proceedings*, pages 80–90. ACM Press, 1999.

Ghica van Emde Boas-Lubsen and Peter van Emde Boas. Compiling horn-clause rules in ibm's business system 12 and early experiment in declarativeness. In *SOFSEM '98: Proceedings of*

the 25th Conference on Current Trends in Theory and Practice of Informatics, pages 68–88, London, UK, 1998. Springer-Verlag. ISBN 3-540-65260-4.

Stijn Vansummeren. On the complexity of deciding typability in the relational algebra. *Acta Informatica*, 41(6):367–381, 2005. ISSN 0001-5903.

Philip Wadler. Monads for functional programming. In *Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School*. Springer-Verlag, 1993.

Mitchell Wand. Complete type inference for simple objects. In *LICS'87: Logic in Computer Science, Proceedings*, pages 37–44. IEEE Press, 1987.

Mitchell Wand. Corrigendum: Complete type inference for simple objects. In *IEEE Symposium on Logic in Computer Science (LICS), Edinburgh, Scotland*, page 132, 1988.

Appendix A

Overview of the Relational Model and Algebra

In the relational model a *tuple* is a mapping from *attributes* to values of a specific *type*. The *heading* of a tuple is a mapping from attributes to types, thus the domain of a heading is a set of attributes. A *relation* is a *finite* set of tuples with matching headings. Since all tuples in a relation have the same heading, we define the heading of a relation to be that of its tuples.

An integral part of the relational model is the *relational algebra*, a set of operations on relations. Codd in his seminal paper [Codd, 1970] identified eight relational operators, sometimes referred to as the original relational algebra. One of Codd's main concerns was ensuring the *safety* of relational algebra, meaning that given finite input relations, no relational algebra expression should result in an infinite relation. The safety property holds for the algebra introduced by Codd, and it is traditionally required of newly invented operators as well. Throughout the decades, authors introduced extended several versions of the original relational operators along with brand new ones. In this description, we will focus on the original set, as introduced by Codd (which will nevertheless not prevent us from discussing some newer operators as well), because the list of potential relational operators is open ended. It is traditional [Date, 1999] to divide the original relational operators into two groups: (1) traditional set operators (*union*, *intersection*, *difference*, and *Cartesian product*), and (2) special relational operators (*restrict*,

project, *join*, and *divide*).

In the following, $h(t)$ stands for the heading of the tuple t, and $h(r)$ stands for the heading of the relation r. We write $t[h]$ for the projection of the tuple t on the heading h. The expression $t.a$ denotes the value of attribute a in the tuple t.

A.1 Union

Relational *union* is a slightly restricted form of standard set union in the sense that we require the heading of the two operands to be the same. Formally:

$$r \cup s \quad \equiv \quad \{t \mid t \in r \vee t \in s\} \quad \textit{where } h(r) = h(s)$$

An example for *union* (the traditional way to represent relations pictorially is in tabular format):

A

Name	*Age*	*City*
Jones	30	Melbourne
Jones	45	Miami

B

Name	*Age*	*City*
Jones	45	Miami
Adams	27	Orlando

A ∪ B

Name	*Age*	*City*
Jones	45	Miami
Jones	30	Melbourne
Adams	27	Orlando

A.2 Intersection

Relational *intersection* is a slightly restricted form of standard set intersection in the sense that we require the heading of the two operands to be the same. Formally:

$$r \cap s \quad \equiv \quad \{t \mid t \in r \wedge t \in s\} \quad \textit{where } h(r) = h(s)$$

An example for *intersection*:

A

Name	*Age*	*City*
Smith	30	Melbourne
Jones	45	Miami

B

Name	*Age*	*City*
Jones	45	Miami
Adams	27	Orlando
Jones	37	Melbourne

A ∩ B

Name	*Age*	*City*
Jones	45	Miami

A.3 Difference

Relational *difference* is a slightly restricted form of standard set difference in the sense that we require the heading of the two operands to be the same. Formally:

$$r \setminus s \quad \equiv \quad \{t \mid t \in r \wedge t \notin s\} \quad \textit{where } h(r) = h(s)$$

An example for *difference*:

A

Name	*Age*	*City*
Smith	30	Melbourne
Jones	45	Miami

B

Name	*Age*	*City*
Jones	45	Miami
Adams	27	Orlando
Baker	37	Melbourne

A\B

Name	*Age*	*City*
Smith	30	Melbourne

A.4 Cartesian Product

Relational *Cartesian product* is similar to Cartesian product defined on sets, with the important difference that the headings of the operands must be *disjoint*. Another important difference is that, unlike traditional Cartesian product, the relational version is *commutative*. The reason for this is that tuples are represented as mappings so there is no ordering on the attributes, thus the concatenation of two tuples is actually the union of two mappings, a commutative operation. Formally:

$$r \times s \quad \equiv \quad \{t \mid t[h(r)] \in r \wedge t[h(s)] \in s\} \quad \textit{where } h(r) \cap h(s) = \emptyset$$

An example for *Cartesian Product*:

A

Name	*Age*
Smith	30
Jones	45

B

City
Orlando
Melbourne

A $\times$ **B**

Name	*Age*	*City*
Smith	30	Melbourne
Jones	45	Melbourne
Smith	30	Orlando
Jones	45	Orlando

A.5 (Natural) Join

Relational *join* is a special relational operator with no corresponding set operator. The result of joining two input relations is an output relation whose heading is the union of the headings of the input relations and that contains ‘matching’ tuples from both input relations, that is, those tuples that have the same values for their common attributes. Traditionally, when the required relation between common attributes is that their values must be equal, we call the operation *natural join*, or *equi-join*. Unless otherwise stated, when talking about *join* we will mean *natural join*. It is interesting to note here, that the definition of *join* is actually the same as that of *Cartesian*

product, with the restriction on the headings of operands being disjoint removed:

$$r \bowtie s \quad \equiv \quad \{t \mid t[h(r)] \in r \wedge t[h(s)] \in s\}$$

An example for *join*:

A

Name	*Age*
Smith	30
Jones	45

B

Name	*Car*
Jones	Ford
Jones	Porsche
Adams	Toyota
Smith	Suzuki

A ⋈ B

Name	*Age*	*Car*
Smith	30	Suzuki
Jones	45	Ford
Jones	45	Porsche

A.6 Restriction

Relational *restriction* takes a relation and a condition and then selects those tuples from a relation that satisfy the given condition. The condition is represented by a function θ from tuples to the logical values *true* or *false*. To guarantee finiteness, it is traditional to require θ to be effectively computable. During introduction to relational algebra, it is customary to restrict θ to conditional

expressions built using only attribute names, simple comparison operators (=, ≠, >, etc.), and logical connectives (∧, ∨, etc.) The formal definition of *restriction* is:

$$\sigma_{\theta}(r) \quad \equiv \quad \{t \mid t \in r \wedge \theta(t)\}$$

An example for *restriction*:

A

Name	*Age*	*City*
Jones	45	Miami
Adams	27	Orlando
Baker	37	Melbourne

$\sigma_{(Age>30 \wedge City\neq Miami)}(\mathbf{A})$

Name	*Age*	*City*
Baker	37	Melbourne

A.7 Projection

Relational *projection* takes an input relation and a set of attribute names and yields an output relation that consist of the tuples of the input relation projected on the given set of attribute names. It is commonly required that the given set of attribute names be a *subset* of the heading of the input relation. For this reason, *projection* can be thought of as 'vertical restriction' (in the tabular depiction of relations, attributes form columns). The formal definition is:

$$\pi_{\{a_1,...,a_n\}}(r) \quad \equiv \quad \{t[\{a_1, ..., a_n\}] \mid t \in r\} \quad \textit{where } \{a_1, ..., a_n\} \subseteq h(r)$$

An example for *projection*:

A

Name	*Age*	*City*
Smith	30	Melbourne
Jones	45	Miami
Baker	37	Melbourne

$\pi_{\{City\}}(\mathbf{A})$

City
Melbourne
Miami

A.8 Division

Relational *division* is a special relational operator with no corresponding set operator. The semantics of division seems a bit involved at first, but it is actually quite simple and useful in practice. The relational operator *division* takes two relations, the *dividend* and the *divisor*. It is only defined for relations where the heading of the divisor is a *subset* of that of the dividend. The result of *division* is an output relation whose heading is the *difference* of the headings of the dividend and the divisor, and that contains exactly those tuples which, when extended with any tuple from the divisor, will appear in the dividend in their extended form. In practice, *division* is often used to formulate queries involving the requirement ‘all ...’, for example, “Employees

who work on all projects". The formal definition of *division* is:

$$r \div s \quad \equiv \quad \{t \mid \forall t_s \in s.\exists t_r \in r.t_r[h(s)] = t_s \wedge t_r[h(r)\backslash h(s)] = t\} \quad \textit{where } h(s) \subset h(r)$$

An example for *division*:

B

Name	*Car*
Jones	Ford
Jones	Toyota
Adams	Suzuki
Smith	Ford

B

Car
Toyota
Ford

A ÷ B

Name
Jones

A.9 'Non-Standard' Relational Operators

In this section we attempt to give a taste of the kind of diversity in relational operators that any language designer must face when attempting to incorporate relational algebra into a general-purpose programing language. We can only hope to give a sample of the various relational operators that were proposed in the literature throughout the years, each proposal being either a variation on or an improvement of a previous operator, with the occasional proposal for a truly novel one. We would like to emphasize that the list presented here is not complete, neither can

it be one. There is an ongoing research in the area with new relational operators being proposed, and older ones falling out of favor. Thus, our goal here is merely to demonstrate, by enumerating operators that each legitimately could claim to be included in any practical implementation of relational algebra, the inherent limitations of any approach that attempts to support only some *specific* set of relational operators as opposed to providing the *means* to express those operators.

A.9.1 Projecting Away and Renaming Attributes

It is customary to define an operation for projecting *away* attributes analogously to projection. As in the case of projection, we will again require that the set of attributes being projected away be a subset of the heading of the relation in question. Formally:

$$\hat{\pi}_{\{a_1,...,a_n\}}(r) \quad \equiv \quad \{t[h(r)\backslash\{a_1, ..., a_n\}] \mid t \in r\} \quad \textit{where } \{a_1, ..., a_n\} \subseteq h(r)$$

Although omitted by Codd in his original paper, it soon became obvious that renaming of attributes is an important operation so it is almost always included in any implementation of relational algebra. The heading of the relation being renamed must have the old attribute name and the new attribute name must not appear in the heading in order for the operation to be well defined. Formally:

$$\rho_{a/b}(r) \quad \equiv \quad \{t[a/b] \mid t \in r\} \quad \textit{where } a \in h(r) \textit{ and } b \notin h(r)$$

A.9.2 Variations on join: semijoin, antijoin, and compose

The *composition* of relations r and s is defined to be the join of r and s with the common attributes *projected away* from the result of the join. Formally:

$$r \hat{\bowtie} s \quad \equiv \quad \hat{\pi}_{(h(r) \cap h(r))}(r \bowtie s)$$

The *semijoin* of relations r and s, written as $r \ltimes s$ is defined similarly to join, except, we project the result of the join on the heading of the first operand, r. Formally:

$$r \ltimes s \quad \equiv \quad \pi_{(h(r))}(r \bowtie s)$$

The *antijoin* of relations r and s, written as $r \triangleright s$ is the dual of semijoin in the sense that it is the restriction of r to all the tuples that *do not* have a matching pair in s. Formally:

$$r \triangleright s \quad \equiv \quad \{t[h(r)] \mid t[h(r)] \in r \wedge t[h(s)] \notin s\}$$

A.9.3 Improved division: the Small Divide

Date and Darwen describe their improved version of the division operator in [Date and Darwen, 1998]. The original version of Codd only worked on relations where the divisor was a subset of the dividend. It also gave somewhat counterintuitive results when faced with pathological cases. For example, when asking for suppliers that supply *all* purple parts (an example, taken from [Date and Darwen, 1998]), and there are no purple parts in our inventory, logically speaking, suppliers that do not supply *any* parts at all should also be returned. To handle cases like this, a generalized version of division, called the *Small Divide*, a ternary relational operator was

introduced, defined in the following way:

$$gd(q, r, s) \quad \equiv \quad q \backslash \pi_{(h(q))}((q \times r) \backslash s) \quad \textit{where} \;\; h(q) \cap h(r) = \emptyset \;\; \textit{and} \;\; h(s) = h(q) \cup h(r)$$

A.9.4 Extension

A common operation in relational algebra is to extend a relation with a new attribute whose value is computed from other attributes for each tuple. The way to do this is to provide a function that takes a tuple and yields the value of the new attribute (again, the function should be effectively computable to ensure the safety of the resulting algebra). Formally:

$$ext(r, a, f) \quad \equiv \quad \{t \mid t[h(r)] \in r \wedge t.a = f(t)\} \quad \textit{where} \;\; a \notin h(r)$$

An example for *extension*:

A

Name	*Age*	*Monthly*
Jones	45	4,500
Adams	27	3,200
Baker	37	4,000

$ext(\mathbf{A}, Yearly, (\lambda t.t{\cdot}Monthly * 12))$

Name	*Age*	*Monthly*	*Yearly*
Jones	45	4,500	54,000
Adams	27	3,200	38,400
Baker	37	4,000	48,000

Appendix B

Proofs

Theorem 5.1.1. *Given a term e, where $P \mid sA \vdash^{W} e : \tau$, the problem* **sat** *$P$ is NP-complete.*

Proof. Membership in NP follows from the observation that given a ground instance of P, all predicates on rows can be checked in polynomial type.

Completeness is shown by reduction from SAT. We provide two different reductions, one which uses a constant number of distinct labels, and one which uses a constant number of polymorphic variables. This way we demonstrate that exponential complexity of satisfiability checking can be caused *either* by the number of polymorphic variables *or* by the number of distinct labels.

Complexity in Number of Variables The reduction presented here uses only a constant number of distinct labels (actually, just one) to show the complexity of satisfiability checking purely in the number of polymorphic variables.

First, we define auxiliary functions for heading union, intersection, and complement (relative to the arbitrarily chosen base set $\langle a \rangle$):

$$\mathit{union}\ x\ y = \langle \langle x \backslash y \rangle \parallel y \rangle$$

$$\mathit{intersection}\ x\ y = \langle x \backslash \langle x \backslash y \rangle \rangle$$

$$\mathit{complement}\ e = \langle \langle a \rangle \backslash e \rangle$$

The corresponding types of the auxiliary functions show that operations performed at the value level are correctly mirrored at the type level by appropriate operations and predicates on row variables:

$$\begin{aligned}
&\mathit{union} :: \rho_3 \simeq \rho_1 \backslash \rho_2,\ \rho_4 \simeq \rho_3 \parallel \rho_2,\ \rho_3 \# \rho_2 \Rightarrow \\
&\qquad \mathit{Rec}\ \rho_1 \rightarrow \mathit{Rec}\ \rho_2 \rightarrow \mathit{Rec}\ \rho_4 \\
&\mathit{intersection} :: \rho_3 \simeq \rho_1 \backslash \rho_2,\ \rho_4 \simeq \rho_1 \backslash \rho_3,\ \Rightarrow \\
&\qquad \mathit{Rec}\ \rho_1 \rightarrow \mathit{Rec}\ \rho_2 \rightarrow \mathit{Rec}\ \rho_4 \\
&\mathit{complement} :: \rho_2 \simeq (\!|a|\!),\ \rho_3 \simeq \rho_2 \backslash \rho_1,\ \Rightarrow \\
&\qquad \mathit{Rec}\ \rho_1 \rightarrow \mathit{Rec}\ \rho_3
\end{aligned}$$

Using the definitions of row operations (Section 4.3) and type predicates (Section 4.4) it is easy to verify that the auxiliary functions do perform the corresponding set operations both at the value *and* at the type level. For example, the type of *union* $\langle a \rangle\ \langle\rangle$ will be $(\!|a|\!)$.

Using the auxiliary functions we can now define a reduction from an arbitrary boolean formula F with boolean variables $\{b_1, b_2, ..., b_n\}$ to a corresponding term e^F with free variables $\{x_1, x_2, ..., x_n\}$ as follows:

$$\begin{aligned}
e^F \quad &\equiv \quad \mathit{Tr}(F) == \mathit{Tr}(\mathit{true}) \qquad \textbf{where} \\
&\qquad \mathit{Tr}(\mathit{true}) = \langle a \rangle \\
&\qquad \mathit{Tr}(\mathit{false}) = \langle\rangle \\
&\qquad \mathit{Tr}(b_i) = x_i \\
&\qquad \mathit{Tr}(\phi_1 \vee \phi_2) = (\mathit{union}\ \mathit{Tr}(\phi_1)\ \mathit{Tr}(\phi_2)) \\
&\qquad \mathit{Tr}(\phi_1 \wedge \phi_2) = (\mathit{intersection}\ \mathit{Tr}(\phi_1)\ \mathit{Tr}(\phi_2)) \\
&\qquad \mathit{Tr}(\neg\phi) = (\mathit{complement}\ \mathit{Tr}(\phi))
\end{aligned}$$

The translation from F to e^F is clearly polynomial. Also, since each auxiliary function is well-typed and is used in a type-correct way, it must be the case that $P_v \mid sA \vdash^W e : Bool$ where P_v captures the type constraints on row variables, and sA provides the type assignment for the free variables x_i in e^F.

We now prove that **sat** P_v if and only if F is satisfiable. First, observe that Tr is an isomorphism between two boolean algebras, logic operations and set operations, where the type $(\!|\!)$ corresponds to the empty set and $(\!|a|\!)$ to the universal set. If F is satisfiable, then there is a truth assignment $M(b_i)$ which maps each b_i to either *true*, or *false* such that $M(F) = true$. Since Tr is an isomorphism, a type assignment A where $A(x_i)$ has type of $Tr(M(b_i))$ will ensure that $Tr(F)$ has type of $Tr(true)$ proving that **sat** σ_v. The other direction is proved in a symmetric way.

Complexity in Number of Labels We now provide a reduction from an arbitrary boolean formula F to a term e^F that uses only a constant number of variables (in this case, only four). Without loss of generality we assume that F uses only operations *negation* and *conjunction*.

We assume the following constants to be defined:

$$b \;::\; Bool \to Bool$$

$$i \;::\; Int \to Int$$

$$* \;::\; \beta_1 \to \beta_2 \to \beta_2 \quad (infix)$$

The constants b and i will be used to 'fix' the type of record fields, while the function '$*$' will be used to build a single expression out of a set of expressions. We introduce four variables

$\{w_1, w_2, w_3, w_4\}$ and define the following shorthands on them:

$$x_1 \equiv \langle w_1 \parallel w_2 \rangle$$

$$x_2 \equiv \langle w_1 \parallel w_3 \rangle$$

$$x_3 \equiv \langle w_2 \parallel w_4 \rangle$$

$$x_4 \equiv \langle w_3 \parallel w_4 \rangle$$

To simplify the presentation of our argument, we will use tables to show field types in record expressions, where rows correspond to record expressions, columns to fields, and cells show the type of the given field in the given record expression. Also, we will use the convention of referring to the type of field l_i in x_1 as α_i. For example, the constraints on field types generated by the expression:

$$var(l_i) \equiv (x_1 \cdot l_i) * (b \;\; x_2 \cdot l_i) * (i \;\; x_3 \cdot l_i)$$

can be represented by the following table:

	l_i
x_1	α_i
x_2	*Bool*
x_3	*Int*

Because records w_1 and w_2 are disjoint (we concatenate them in $x_1 \equiv \langle w_1 \parallel w_2 \rangle$) and l is in x_1, it follows that l must be in exactly one of w_1 and w_2. If $l \in w_1$, then it has to have type *Bool* in x_1, because $x_2 \cdot l \equiv \langle w_1 \parallel w_3 \rangle \cdot l$ has type *Bool*. Similarly, if l is in w_2, then it must have type *Int* in x_1. Labels introduced using this technique will correspond to the boolean variables and subexpressions of the formula F, where the type *Bool* (*Int*) in x_1 for a given label l will represent the logical value *false* (*true*).

Next we show how to represent *negation* and *conjunction*. Negation is a simple matter, once we realize that for any given label l, the type of l in x_4 is the 'negation' of its type in x_1, that is $x_1 \cdot l$ has type *Bool* if and only if $x_4 \cdot l$ has type *Int* (and $x_1 \cdot l$ has type *Int* if and only if $x_4 \cdot l$ has type *Bool*). Exploiting this, we can define the 'negation' of a label l_1, that is, a label l_2 where $x_1 \cdot l_2$ has type *Int* if and only if $x_1 \cdot l_1$ has type *Bool*:

$$not(l_1, l_2) \equiv (x_4 \cdot l_1 == x_1 \cdot l_2)$$

Conjunction is a bit more involved. Suppose we would like to represent the 'conjunction' of labels l_1 and l_2. That is, we would like to define a label l_3 where $x_1 \cdot l_3$ has type *Int* if and only if $x_1 \cdot l_1$ and $x_2 \cdot l_2$ both have type *Int*. We claim that the following expression achieves exactly this:

$$and(l_1, l_2, l_3) \equiv e_1 * e_2 * e_3$$

$$\textbf{where} \qquad (\ell_x, \ell_y \ \ fresh)$$

$$e_1 \equiv (x_1 \cdot l_1 == x_1 \cdot \ell_x) * (i \ \ x_2 \cdot \ell_x) * (x_1 \cdot l_3 == x_3 \cdot \ell_x)$$

$$e_2 \equiv (x_1 \cdot l_2 == x_1 \cdot \ell_y) * (i \ \ x_2 \cdot \ell_y) * (x_1 \cdot l_3 == x_3 \cdot \ell_y)$$

$$e_3 \equiv (x_1 \cdot l_3) * (x_1 \cdot l_1 == x_2 \cdot l_3) * (x_1 \cdot l_2 == x_3 \cdot l_3)$$

To better understand what is going on, we show the tabular representation of the field type constraints:

	ℓ_x	ℓ_y	l_3
x_1	α_1	α_2	α_3
x_2	*Int*	*Int*	α_1
x_3	α_3	α_3	α_2

First, observe, that through the construction of x_i's, in any given column a type appearing in the first row must be equal to exactly one of the types that appear in the second or the third row.

With this in mind, we first show that if either α_1 or α_2 is *Bool*, then α_3 must also be *Bool*. For, if α_1 is *Bool*, then α_3 must also be *Bool* since α_1 is equal to either *Int* or α_3 in column ℓ_x. Similarly if α_2 is *Bool*, then α_3 is also *Bool* because of column ℓ_y. Next, if both α_1 and α_2 are *Int*, then α_3 must be *Int*, because of column l_3. On the other hand, if α_3 is *Bool*, then either α_1 or α_2 must be *Bool*, because of column l_3. And finally, if α_3 is *Int* then both α_1 and α_2 must be *Int*, because of columns ℓ_x and ℓ_y.

Now we are ready to define a reduction from a boolean formula F with boolean variables $\{b_1, b_2, ..., b_n\}$ to a corresponding expression e^F. One technical remark before the formal definition: the translation of a boolean subexpression ϕ returns a *pair* of an expression *and* a record label by which the translated subexpression can subsequently be referred to. During the translation we introduce fresh record labels for subexpression as needed, except for boolean variables b_i for which we use the fixed record labels l_i, and for *true* and *false*, for which we use the fixed labels l_t and l_f respectively:

$$Tr(true) = (i \;\; x_1 \cdot l_t, l_t)$$

$$Tr(false) = (b \;\; x_1 \cdot l_f, l_f)$$

$$Tr(b_i) = (var(l_i), l_i)$$

$$Tr(\neg\phi) = (e_\phi * not(\ell_\phi, \ell), \ell) \quad (\ell \; fresh) \quad \textbf{where}$$

$$(e_\phi, \ell_\phi) = Tr(\phi)$$

$$Tr(\phi_1 \wedge \phi_2) = (e_{\phi_1} * e_{\phi_2} * and(\ell_{\phi_1}, \ell_{\phi_2}, \ell), \ell) \quad (\ell \; fresh) \quad \textbf{where}$$

$$(e_{\phi_1}, \ell_{\phi_1}) = Tr(\phi_1)$$

$$(e_{\phi_2}, \ell_{\phi_2}) = Tr(\phi_2)$$

Using Tr, we define e^F as:

$$e^F \equiv e * (i \;\; x_1 \cdot \ell_F)$$

$$\textbf{where} \;\; (e, \ell_F) = Tr(F)$$

The translation from F to e^F is clearly polynomial. Also, it is easily verified that e^F must have an inferred type since the functions b, i, $*$, and $(==)$ were all used in a type-correct way. This means that $P_l \mid sA \vdash^W e^F : Int$ where P_l captures the type constraints on field types, and sA provides the type assignment for the free variables $\{w_1, w_2, w_3, w_4\}$ in e^F.

We now prove that **sat** P_l if and only if F is satisfiable. If F is satisfiable, then there is a truth assignment M that makes F true. Let's define a type assignment A' where l_i has type *Int* in w_1 if $M(b_i) = true$, otherwise l_i has type *Bool* in w_2. Since we demonstrated earlier that the expressions for *negation* and *conjunction* are isomorphic to the corresponding logic operations at the level of field types (with *Bool* representing *false* and *Int* representing *true*) it follows that there can be no inconsistencies regarding field type constraints in P_l, that is P_l is satisfiable. The other direction is proved in a symmetric way. □

Lemma 5.5.2. *For a set expression e and label ℓ, $\ell \in e$ iff $\ell \in \hat{\psi}_B(e)$.*

Proof. We will proceed by induction on the form of the set expression e. (Recall, that each label ℓ belongs to exactly one region R_i.)

If $e = \emptyset$ then, by definition of $\hat{\psi}_B$:

$$\hat{\psi}_B(\varnothing) = \bigcup_{i \in \varnothing} R_i \backslash L_P \cup \bigcup_{\ell \in L_P} \bigcup_{i \in \varnothing} R_i \cap \{\ell\} = \varnothing$$

If $e = \{\ell\}$ then, by definition of $\hat{\psi}_B$:

$$\hat{\psi}_B(\{\ell\}) = \bigcup_{i \in \varnothing} R_i \backslash L_P \cup \bigcup_{i \in \{0,\ldots,2^n-1\}} R_i \cap \{\ell\} = \mathcal{L} \cap \{\ell\} = \{\ell\}$$

If $e = H_j$ then, by definition of $\hat{\psi}_B$:

$$\hat{\psi}_B(\{\ell\}) = \bigcup_{i \in \{k \mid R_k \subseteq H_j\}} R_i \backslash L_P \cup \bigcup_{\ell \in L_P} \bigcup_{i \in \{k \mid R_k \subseteq H_j\}} R_i \cap \{\ell\} =$$

$$(H_j \backslash L_P) \cup \bigcup_{\ell \in L_P} H_j \cap \{\ell\} = (H_j \backslash L_P) \cup (H_j \cap L_P) = H_j$$

If $e = f \cap g$, then by the induction hypothesis, the lemma holds for $\hat{\psi}_B(f)$ and $\hat{\psi}_B(g)$, which, combined by the definition of intersection on sets, yields that the lemma holds for $\hat{\psi}_B(e)$.

We handle set union and difference analogously to set intersection. □

Lemma 5.5.3. *For a set expression e and a label $\ell \in L_P$, $\ell \in e$ iff $\ell \in \bigcup_{i \in \mathcal{I}^e_\ell} R_i$, where $\mathcal{I}^e_\ell$ is defined by the normal form $\hat{\psi}_B(e)$.*

Proof. Recall the definition of $\hat{\psi}_B(e)$ (we name the parts so that we can refer back to them):

$$N_1 = \bigcup_{i \in \mathcal{I}^e} R_i \backslash L_P$$

$$N_2 = \bigcup_{\ell \in L_P} \bigcup_{i \in \mathcal{I}^e_\ell} R_i \cap \{\ell\}$$

$$\hat{\psi}_B(e) = N_1 \cup N_2$$

Using Lemma 5.5.2, $\ell \in e \Leftrightarrow \ell \in \hat{\psi}_B(e)$.

Also, since $\ell \in L_P$, then obviously $\ell \notin N_1$, thus $\ell \in \hat{\psi}_B(e) \Leftrightarrow \ell \in N \Leftrightarrow \ell \in \bigcup_{i \in \mathcal{I}^e_\ell} R_i \cap \{\ell\}$.

But $\ell \in \bigcup_{i \in \mathcal{I}^e_\ell} R_i \cap \{\ell\} \Leftrightarrow \ell \in \bigcup_{i \in \mathcal{I}^e_\ell} R_i$, which is exactly what we set out to prove.

□

Theorem 5.10.1 (Soundness)**.** *If* ¬**sat** *P then Algorithm Q will report failure.*

Sketch. Recall, that **sat** *P* means that there is a substitution *s* that makes $\Vdash sP$ true.

First, from Lemma 5.2.1, if $\phi(P)$ is not satisfiable, then *P* cannot be satisfiable since they both put the same constraints on label presence and absence. Therefore, reusing the argument from

Section 5.6, if algorithm finds the system of set constraints $\phi(P)$ unsatisfiable, then P cannot be satisfiable.

Second, if $\phi(P)$ is satisfiable, P can still be unsatisfiable due to inconsistent field type constraints. Algorithm Q takes field type constraints into account by analyzing the row construction graph G. Now, would it be possible for algorithm Q to find P satisfiable, when in fact it is not? That would mean that some field type equalities were not taken into account when analyzing graph G. But this is not possible, since all relations between rows have been taken into account when building the graph. Thus, we can conclude that algorithm Q will fail if P is not satisfiable.

□