A Polymorphic Type System and Compilation Scheme for Record Concatenation

by

Edward Osinski

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

May 2006

Benjamin F. Goldberg

to my wife and best friend, Beata

Acknowledgments

This dissertation would not have been possible without the help of numerous people. Finishing was an long and arduous process. I would like to thank the members of the Griffin committee, namely Robert Dewar, Benjamin Goldberg, Malcolm Harrison, Edmond Schonberg, and Dennis Shasha, for the many stimulating discussions we had. It was my work in Griffin that turned me on to advanced programming languages and type systems.

I am indebted to Malcolm Harrison, who was my advisor in the early years, and Ben Goldberg, who was my advisor in the later ones. My thesis greatly benefited from their comments. I would also like to thank several of my fellow graduate students for their time and encouragement and helping make my stay as a graduate student enjoyable. In particular, Hseu-Ming Chen, for his ongoing friendship, Allen Leung, for his sharp wit and helpful technical discussions, from which I learned a great deal of type theory, and Zhe Yang, who was always willing to provide his technical expertise when I was struggling with a proof. I would like to thank the chair of the Computer Science department at NYU, Margaret Wright, for striking just the right note of menace in the departmental letters urging me to finish. These gave me extra incentive. Anina Karmen and Rosemary Amico were helpful and patient in helping me navigate the bureaucratic maze.

My manager, Andrew Howell, deserves a special note of thanks for the warm and friendly work environment he has fostered, and for his patience and understanding when I needed to take time off to work on my dissertation. My co-workers Andrew Hagerty, Ernest Lee, Alana Libonati and Adam Zelezen are also deserving of thanks for making work a pleasant experience and for not starting a betting pool on when (or whether) I would finish. I know it must have been tempting.

My gratitude goes to my mother Wanda for encouraging me to go for a PhD in the first place, and most especially my wife Beata, who always believed I could finish; for her love and encouragement, and for her patient endurance of many evenings and weekends of me spending time with my dissertation instead of with her.

Abstract

The notion of records, which are used to organize closely related groups of data so the group can be treated as a unit, and also provide access to the data within by name, is almost universally supported in programming languages. However, in virtually all cases, the operations permitted on records in statically typed languages are extremely limited. Providing greater flexibility in dealing with records, while simultaneously retaining the benefits of static type checking is a desirable goal.

This problem has generated considerable interest, and a number of type systems dealing with records have appeared in the literature. In this work, we present the first polymorphic type system that is expressive enough to type a number of complex operations on records, including three forms of concatenation and natural join. In addition, the precise types of the records involved are inferred, to eliminate the burden of explicit type declarations. Another aspect of this problem is an efficient implementation of records and their associated operations. We also present a compilation method which accomplishes this goal.

Table of Contents

List of Figures

List of Appendices

Chapter 1

Introduction

The history of programming languages is a study in ever increasing expressiveness. The earliest programming languages (e.g. assembly) were close to the machine. That is, translation to the machine code that was understood by the computer was relatively straightforward. In fits and starts, languages evolved into higher-level beasts which dealt with increasingly powerful abstractions. These abstractions could be used to represent entities and actions in various problem domains. As the ability to do this in a "natural" fashion increased, the gap between higher-level languages and machine language grew larger. This required increasingly intricate translation techniques in order to generate executable code.

One simple but effective step in this direction was the use of records. Records provide a way to group a number of related values together, and be able to refer to each value with a comprehensible name.

One fruitful avenue of exploration leads to static type systems. It is natural for humans to categorize things in terms of their similarities and differences. In fact, the act of categorization is so basic that it is usually automatic and unconscious. A static type system is used to categorize each value in the language at compile time, and limits what can be done with a value. These constraints actually have several benefits:

1. they prevent the programmer from making some sorts of mistakes, e.g. adding a string to a list

2. they reduce the amount of run-time checking that needs to be done to check for the mistakes of the sort mentioned in the previous item

3. they can also be used to guide the translation to machine language and increase the efficiency of the generated code

4. in the hands of an experienced programmer, a static type system can be used to enforce various design invariants

Most non-machine-level programming languages have some sort of type system. One major way in which they differ is when (and if) type errors (i.e. attempts to use values in ways not sanctioned by the type system) are detected. This can be done either at run-time or at compile-time. Languages such as Lisp and Perl have run-time checking, while most conventional languages such as FORTRAN, C, Ada, ML, etc. check for type errors at compile-time. Of course, it is possible to have a mixture of the two. An example of this is Java.

Another dimension is the degree of strictness in the type system. For example, C is notorious for the ease in allowing the use of a value of one type as if it were another. Languages such as Ada, ML, and Scheme are at the other extreme.

Most interesting work has been done in type systems which detect violations at compile-time. These are known as static type systems. Because there is less information available at compile-time than at run-time, static type systems are necessarily more restrictive than dynamic ones. There will be programs which verifiably do not cause any type errors at run-time that will be rejected by the static type checking phase. Type systems in conventional imperative languages such as Pascal, C/C++, Ada, etc. are particularly restrictive. Ironically, this is the case even though C's type system, for example, is not particularly strict.

One particularly interesting approach has been the development of type inference systems, in languages such as SML [38], Haskell [45] and Miranda [54, 55]. These are static type systems where the types of variables need not be declared, as they are inferred from context. The motivation of type inference is to reduce the inconvenience of variable declarations while retaining all the benefits of static type systems. Interestingly, even though it is clearly more difficult to have a system that infers all types rather than simply checking that the types variables are explicitly assigned, languages with type inference typically have more expressive power than those that rely on extensive type declarations.

1.1 Beginnings

The beginnings of modern type theory have their origins in the lambda calculus conceived by Church [11], as part of a theory of computable functions. This theory did not have types. The syntax of the pure[1] lambda calculus is:

$$\begin{array}{rcll} E & ::= & x & \text{variable} \\ & | & EE & \text{application} \\ & | & \lambda x.E & \text{abstraction} \end{array}$$

Unless otherwise indicated, the name of the syntactic category will be used to range over the set of terms in that category. For example, E will range over the set of lambda calculus terms, and x ranges over the set of term variables. By convention, we will also use M and N to range over terms, and x, y, z to range over variables. Application is left-associative, so $E_1E_2 \ldots E_n$ means $((\ldots(E_1E_2)\ldots)E_n)$.

In the style of proof system used here, we have rules (and axioms). A *rule* has the general form

$$(\textbf{name}) \quad \frac{\textit{antecedent}(s)}{\textit{consequent}}$$

The *antecedents* above the horizontal line, and the *consequent* below, are judgements. Such a rule means that the proofs of the antecedents (if they exist) can be combined to prove the consequent. Another way of reading it is "if the antecedents are true, then so is the consequent". It is often more useful to think of rules in the opposite direction, i.e. in order to prove the consequent via a given rule,[2] the antecedent(s) must be proven. Each rule generally has a name as indicated. In the case where there are no antecedents in a rule, we use the abbreviated notation:

$$(\textbf{name}) \quad \textit{judgement}$$

[1]The term pure refers to the lack of predefined constants or variables.

[2]This caveat is necessary as more than one rule may apply to a given consequent

Such rules are called *axioms.* The exact judgement forms allowed depend on the particular inference system in question. In the simply-typed $\lambda -$ calculus, there is one judgement form:

$$A \vdash M : \tau$$

In this judgement form, M is an expression, and A is an *environment*, which is a set of variable type bindings, written $x : \tau$, where x is a variable, and τ is the associated type. We read this judgement as saying that expression M has type τ in environment A.

A *proof* in this system consists of a finite tree of judgements, where each judgement is either an instance of an axiom, or follows from judgements above it in the tree by the application of a rule. Each of the judgements in this tree is said to be proven.

There is one significant reduction rule for the untyped lambda calculus[3]:

$$(\beta) \quad (\lambda x.M)N \longrightarrow [x \mapsto N]M$$

$[x \mapsto N]$ is our notation for a *substitution*, which can be considered a function on terms. When applied to a term M, this function returns the term obtained by replacing all free occurrences of variable x with N in M. We use the symbol $\longrightarrow\!\!\!\!\!\rightarrow$ to mean the reflexive, transitive closure of $\longrightarrow$. We say:

- A term M is in *normal form* if there is no N such that $M \longrightarrow N$.
- A term M has a normal form N if $M \twoheadrightarrow N$ and N is a normal form.

[3]Actually, there are also the α-rule and the η-rule, but these do not even begin to approach the usefulness of the β-rule, and are unnecessary for our discussion.

One problem with the untyped λ-calculus is that it is too powerful in some sense. For example, it is possible to write terms that have no normal forms:

$$(\lambda x.x\ x)(\lambda x.x\ x)$$

This causes difficulties when attempting to provide models of the untyped λ-calculus. Adding types to the λ-calculus is an attempt to curtail its unruliness. There are two distinct styles of type systems, typified by the two versions of the earliest (and simplest) type system for the λ-calculus: the *simply-typed λ-calculus*, or $\lambda^{\rightarrow}$. These are due to Curry and Feys [13] and Church [10]. In a Church-style system, types are explicit in the syntax, while in a Curry-style system, also referred to as a *type assignment system*, types are implicit and are inferred from context. The process of determining the possible type(s) of a term is known as the *type inference* problem.

Many type systems have Church-style and Curry-style counterparts[4]. Informally, terms in the Curry-style system have less type information than its Church-style counterpart. A transformation of terms in a Church-style system into the corresponding terms in the Curry-style system is defined in terms of what is known as an *erasure function*.

[4]This is not always the case, however. For example, the system of intersection types described in [3,12] has no obvious Church-style counterpart. Conversely, the Calculus of Constructions, a Church-style system at the apex of Barendregt's cube, has no Curry version.

$\lambda^{\rightarrow}$-Church has the syntax:

$$\begin{array}{lcll} E & ::= & x & \text{identifier} \\ & | & E\,E & \text{application} \\ & | & \lambda x : \tau . E & \text{abstraction} \end{array}$$

$$\begin{array}{lcll} \tau & ::= & t & \text{type constant} \\ & | & \tau \rightarrow \tau & \text{function type} \end{array}$$

Note that this is the syntax of the untyped lambda calculus, with the addition of explicitly tagging variable bindings with their type, represented by τ. The type expression $\tau_1 \rightarrow \tau_2 \rightarrow \ldots \rightarrow \tau_n$ is used as an abbreviation for $\tau_1 \rightarrow (\tau_2 \rightarrow (\ldots \rightarrow \tau_n)\ldots)$.

The type rules for this system are:

$$(\textbf{var}) \qquad A \vdash x : \tau \quad \text{if } x : \tau \in A$$

$$(\textbf{app}) \quad \frac{A \vdash M : \tau' \rightarrow \tau \qquad A \vdash N : \tau'}{A \vdash M\,N : \tau}$$

$$(\textbf{abs}) \quad \frac{A, x : \tau' \vdash M : \tau}{A \vdash (\lambda x : \tau' . M) : \tau' \rightarrow \tau}$$

The syntax of $\lambda^{\rightarrow}$-Curry is the same as that of the untyped lambda calculus. Its type rules are the same as those for $\lambda^{\rightarrow}$-Church, with the modification that rule abs uses the Curry syntax for the λ-term in the consequent. the modified abs rule is:

$$(\textbf{abs}) \quad \frac{A, x : \tau' \vdash M : \tau}{A \vdash (\lambda x . M) : \tau' \rightarrow \tau}$$

Note that it is possible to give the same lambda term different types in the Curry system.

We can informally define the erasure function transforming terms from $\lambda^{\rightarrow}$-Church to terms in $\lambda^{\rightarrow}$-Curry as the function which lops off all occurrences of "$: \tau$" in the term.

1.2 The Hindley-Milner type system

1.2.1 Type abstraction

In $\lambda^{\rightarrow}$, there are many terms which have overly restrictive types. The problem is similar to one often encountered in languages such as Pascal and C. An example is a function that determines the length of a list. When written in Pascal or C, we need to fix the element type of the list even though the `length` routine never examines any element. The code for determining the length of a list of integers is identical to code which determines the length of a list of any other type, yet we are forced to write multiple definitions of these routines if we wish to apply them to different sorts of lists. One step on the road toward eliminating this restriction is to extend $\lambda^{\rightarrow}$ with type abstraction. This would allow us to write a function which could then be applied to any list type. We need additional syntax for terms to allow for type abstraction and application:

$$\begin{array}{rcll} E & ::= & \ldots & \\ & \mid & E\ \tau & \text{type application} \\ & \mid & \Lambda\alpha.E & \text{type abstraction} \end{array}$$

The syntax of types also needs to be extended. To represent the type of a term of the form $\Lambda\alpha.E$, we will use the notation $\Pi\alpha.\tau$, where τ is the type of E. We now face an important choice of what kinds of types we can apply such a term to. In particular, we can allow application to a type of the form $\Pi\beta.\tau'$, or restrict it to only Π-less types. The former choice leads to the Church-style system $\lambda^{\rightarrow,\Pi}$, an example of a *predicative* type system, while the latter leads to System F, or $\lambda 2$, which is *impredicative*. A predicative definition of a type is built up of simpler types; this is not the case for an impredicative definition. The differences between these two systems are vast. The type inference problem for the former system is decidable, whereas for System F, the question has been open until fairly recently, when it was answered in the negative [60]. There are also difficulties in developing semantic models for System F which do not occur for $\lambda^{\rightarrow,\Pi}$. For example, it is not possible to interpret terms of System F in a set-theoretic way.

Despite its nice properties, $\lambda^{\rightarrow,\Pi}$ is not really any more powerful than $\lambda^{\rightarrow}$. The reason for this is simple. We can have polymorphic terms in $\lambda^{\rightarrow,\Pi}$, but we can use each of these terms at only one type. For example,

$$\Lambda\alpha.\lambda x : \alpha.x$$

is polymorphic, but we cannot bind it to a variable. This is because the only method we have to bind a value to a variable is function application, and unlike in System F, functions may be applied only to monomorphic values. In other words, we cannot pass the term above as an argument to a function.

The Hindley-Milner (HM) type system [36] is an extension of $\lambda^{\rightarrow,\Pi}$ that solves this problem. Alternatively, it can be viewed as a predicative restriction of System

F. It accomplishes these dual goals by allowing the binding of polymorphic terms to variables, which can then be used multiple times, but restricting where this binding can occur to a specific syntactic construct, the `let`. The HM type system is the first formal type system to be implemented in a widely-used[5] programming language, ML [37]. It has a number of desirable properties, including the principal type property, decidable type inference and, in fact, a simple type inference algorithm. It also has the interesting feature that the translation of a well-typed ML expression is independent of how polymorphic types are instantiated.

For some time, it was believed that ML type inference was doable in polynomial time. Counter-examples to this conjecture were found by various researchers. Kanellakis and Mitchell [30] showed that the problem was PSPACE-hard. It was finally proven to be DEXPTIME-complete in [33], and independently, in [31].

We will describe the type system and various enhancements in terms of type rules for a 'core' of ML. The syntax of expressions is:

$$\begin{array}{rcll} E & ::= & x & \text{identifier} \\ & | & E_1\ E_2 & \text{function application} \\ & | & \lambda x.E & \text{function abstraction} \\ & | & \texttt{let}\ x = E_1\ \texttt{in}\ E_2 & \text{let expression} \\ & | & \texttt{fix}\ x.E & \text{fix-point} \end{array}$$

The abstract syntax of types and type schemes, which will be used in the type rules

[5]Widely used by academic standards.

$$
\begin{array}{lcl}
\mathrm{FV}(\sigma) & = & \text{set of free variables in type scheme } \sigma \\
\mathrm{FV}(A) & = & \bigcup_{(\mathtt{x}:\sigma)\in A} \mathrm{FV}(\sigma) \\
\mathit{Inst}(\forall\alpha_1,\ldots,\alpha_n\,.\,\tau) & = & [\alpha_1 \mapsto \tau_i,\ldots,\alpha_n \mapsto \tau_n]\tau \\
\mathit{Gen}(A,\sigma) & = & \forall\alpha_1,\ldots,\alpha_n\,.\,\sigma \qquad \text{where } \{\alpha_1,\ldots,\alpha_n\} = \mathrm{FV}(\sigma) - \mathrm{FV}(A)
\end{array}
$$

Figure 1.1: Functions used in type rules

to be presented, are:

$$
\begin{array}{rcll}
\tau & ::= & t & \text{type constant} \\
 & | & \alpha & \text{type variable} \\
 & | & \tau \to \tau & \text{function type} \\
 & | & \tau \times \cdots \times \tau & \text{cartesian product type} \\
 & | & \tau[\tau,\ldots,\tau] & \text{type application}
\end{array}
$$

$$
\begin{array}{rcll}
\sigma & ::= & \tau & \text{simple type} \\
 & | & \forall\alpha\,.\,\sigma & \text{quantified type}
\end{array}
$$

In the type rules, the metavariables α and β will range over type variables, and t will range over type constants. Similarly, τ and σ will be used as metavariables representing types and type schemes, respectively. We write $\forall\alpha_1,\ldots,\alpha_n$ as shorthand for $\forall\alpha_1.\ldots.\forall\alpha_n$, and likewise for quantifiers other than $\forall$ to be introduced later.

To keep ML predicative, types and type schemes are distinguished. Type schemes are only allowed in certain contexts. A is used to represent an environment, which is a set of items, the form of which will differ among the type systems

Judgement	**Read as:**
$\vdash A\ \mathbf{env}$	A is a well-formed environment.
$A \vdash \tau\ \mathbf{type}$	Given environment A, we can infer that τ is a well-formed type.
$A \vdash E : \tau$	Given environment A, we can infer that expression E is of type τ.

Figure 1.2: Core ML judgements

discussed. We use the notation $A, item$ as shorthand for $A \bigcup \{item\}$. Judgements for ML are of the forms indicated in Table 1.2.

The shorthand $\vdash \ldots$ means $\varnothing \vdash \ldots$.

In the basic ML type system, the environment formation rules are:

$$(\textbf{empty-env}) \qquad \varnothing\ \mathbf{env}$$

$$(\textbf{var-env}) \qquad \frac{A\ \mathbf{env}}{(A, x : \sigma)\ \mathbf{env}} \quad x : \sigma \notin A$$

$$(\textbf{type-env}) \qquad \frac{A\ \mathbf{env}}{(A, c\ \mathbf{type})\ \mathbf{env}} \quad c\ \mathbf{type} \notin A$$

In the sequel, it is understood that all environments and types mentioned in rules are well-formed unless otherwise indicated.

The type rules for this system are:

$$(\textbf{var}) \qquad \frac{(x : \sigma) \in A}{A \vdash x : \tau} \; \tau = \mathit{Inst}(\sigma)$$

$$(\textbf{app}) \qquad \frac{A \vdash E_1 : \tau' \to \tau \qquad A \vdash E_2 : \tau'}{A \vdash (E_1\, E_2) : \tau}$$

$$(\textbf{abs}) \qquad \frac{A, x : \tau \vdash E : \tau'}{A \vdash (\lambda x \,.\, E) : \tau \to \tau'}$$

$$(\textbf{let}) \quad \frac{A \vdash E_1 : \tau \qquad A, x : \sigma \vdash E_2 : \tau'}{A \vdash (\texttt{let}\ x = E_1\ \texttt{in}\ E_2) : \tau'} \; \sigma = \mathit{Gen}(A, \tau)$$

This set of rules is syntax directed. In other words, the correct rule to apply can always be determined syntactically. There is no need to do further analysis or to "guess" which rule to apply.

A number of functions are used in the type rules. They are defined in Figure 1.1. Others will be defined in the appropriate sections as necessary. The functions defined on type schemes are extended to work with environments in the usual way.

There is a well-known type inference algorithm for ML, due to Milner. This algorithm, **W**, makes use of *unification*, due to Robinson [51], in an essential way.

W takes as input the expression form in the left-hand side of Figure 1.3, and performs the actions specified on the right.

The auxiliary functions *FreshVar*, *Inst*, and *Unify* have the types:

```
$\mathbf{W}[\![x]\!]A$                          $\Rightarrow$  return $(x, \mathit{Inst}(Ax))$

$\mathbf{W}[\![fe]\!]A$                         $\Rightarrow$  $(\theta_f, \tau_f) = W[\![f]\!]A$
                                                   $(\theta_e, \tau_e) = \mathbf{W}[\![E]\!](\theta_f A)$
                                                   $\beta = \mathit{FreshVar}()$
                                                   $\theta = \mathit{Unify}(\theta_e \tau_f, \tau_e \to \beta)$
                                                   return $(\theta \circ \theta_e \circ \theta_f, \theta\beta)$

$\mathbf{W}[\![\lambda x.e]\!]A$                $\Rightarrow$  $\tau = \mathit{FreshVar}()$
                                                   $(\theta, \tau_e) = \mathbf{W}[\![E]\!](A \cup \{x : \tau\})$
                                                   return $(\theta, \theta\tau \to \tau_e)$

$\mathbf{W}[\![\texttt{let}\, x = E_1 \,\texttt{in}\, E_2]\!]A$  $\Rightarrow$  $(\theta_1, \tau_1) = \mathbf{W}[\![E_1]\!]A$
                                                   $(\theta_2, \tau_2) = \mathbf{W}[\![E_2]\!](\theta_1 A \cup \{x : \mathit{Gen}(\theta_1 A, \tau_1)\})$
                                                   return $(\theta_2 \circ \theta_1, \tau_2)$

$\mathbf{W}[\![\texttt{fix}\, x.E]\!]A$         $\Rightarrow$  $\beta = \mathit{FreshVar}()$
                                                   $(\theta, \tau) = \mathbf{W}[\![E]\!](A \cup \{x : \beta\})$
                                                   $\theta' = \mathit{Unify}(\theta\beta, \tau)$
                                                   return $(\theta' \circ \theta, \theta'\tau)$
```

Figure 1.3: Algorithm **W**

FreshVar	`unit` $\rightarrow$ `type`
Inst	`type-scheme` $\rightarrow$ `type`
Unify	`type` $\times$ `type` $\rightarrow$ `substitution`

FreshVar returns a new type variable.

Inst is defined:

$$\mathit{Inst}(\forall \alpha_1, \ldots, \alpha_n \,.\, \tau) = [\alpha_i \mapsto \beta_i]\tau$$

where all β_i are fresh type variables.

Unify obeys the properties:

1. If $\mathit{Unify}(\tau_1, \tau_2)$ succeeds, it returns a substitution θ which *unifies* τ_1 and τ_2, i.e. $\theta\tau_1 = \theta\tau_2$. Also, θ involves only variables in τ_1 and τ_2.

2. If there exists a substitution θ which unifies τ_1 and τ_2, the $\mathit{Unify}(\tau_1, \tau_2)$ succeeds, yielding a substitution θ_1 such that, for some substitution θ_2, $\theta = \theta_2\theta_1$.

It may seem at first glance that the language presented above is extremely limited in its expressiveness, because it lacks common language features, such as ML datatype declarations, expression sequences (i.e. $E_1; E_2; \ldots; E_n$), `if` expressions, a module system, mutable variables, or even any built-in types. However, some of these deficiencies can be remedied by populating the environment with predefined types and terms.

1.3 Haskell

One of the weak points of the ML type system is its treatment of overloaded operations. These are sets of related operations which cannot be given a single polymorphic type. One set of examples are arithmetic operations such as addition. Integer and floating point addition are two distinct operations for which it is customary to use the symbol +.

$$\texttt{int} \times \texttt{int} \rightarrow \texttt{int}$$

$$\texttt{float} \times \texttt{float} \rightarrow \texttt{float}$$

These two types are instances of the type scheme

$$\forall \alpha \, . \, \alpha \times \alpha \rightarrow \alpha$$

Unfortunately, addition is not definable for all instances of this type scheme.

ML treats addition (and several similar operations) as special cases. Addition is overloaded on both integers and floating point values. Programmers cannot define their own overloaded operations. In addition, this sort of overloading interferes with type inference. In ML, the function

```
fun add (x, y) = x + y
```

cannot be typed, because there is no way to assign a single most general type to `add`. The programmer can choose either integer or floating point addition with the addition of an explicit type constraint:

```
fun add (x, y : int) = x + y
```

This gives ML enough information to determine that integer addition is meant in this case.

Another example is the operation of equality. This operation is defined on many, but not all types. For example, there is no non-trivial computable definition of equality on functions[6]. All instances of equality have types which are instances

[6]There is, of course, extensional equality, but this is not computable.

of the type scheme:

$$\forall \alpha \,.\, \alpha \times \alpha \rightarrow \texttt{bool}$$

Unlike the case with addition, equality is defined on an infinite number of types. ML's solution to this is to make equality a special case. Equality is predefined on most types. We can abstract over all of these types by using an *equality types variable*. Whereas the syntax for a regular type variable is a single quote followed by the variable name, the syntax for an equality type variable is two single quotes followed by the variable name.

A clean solution to these problematic areas exists in Haskell. Haskell allows a controlled form of overloading which can accommodate operations which are defined on a finite set of types, such as addition, as well as equality, which is defined on an infinite set of types. In Haskell, we can define a *class*, which declares a set of operations on a type, and *instances* of the class, which are specific definitions of the definitions in the class for a specific type. For example, we can define the class:

```
class Addable a where
    (+) :: a → a → a
```

and instances

```
instance Addable Int where
    (+) = ... –– definition of integer addition
instance Addable Float where
    (+) = ... –– definition of floating point addition
```

We can now abstract over all types which have addition:

```
sum3 x y z = (x + y) + z
```

The type of `sum3` is $\forall\alpha\,.\,\texttt{Addable}\ a \Rightarrow a \rightarrow a \rightarrow a \rightarrow a$. We read this as "for all types a which are in the class `Addable` the function takes three a's and returns an a. Equality is handled similarly:

```
class Eq a where
    (==) :: a → a → Bool
```

Equality is defined on most basic types, e.g., integers, booleans, etc. Equality is also defined on lists of any type for which equality is defined. This is accomplished with the definition:

```
instance Eq a ⇒ Eq [a] where
    []      == []      = True
    (x::xs) == (y::ys) = x == y && xs == ys
    _       == _       = False
```

The notation `[a]` denotes a list of type `a`. Informally, we can read the first line as saying that if type `a` is an instance of the `Eq` class, then so is `[a]`. The definition of equality on lists is recursive. Note that in the second clause, we invoke the equality operation twice, at different types. In the expression `x ==y`, we are invoking equality at type `a`, and in the expression `xs ==ys`, we are simply making a recursive call. That is, we are invoking it at type `[a]`.

1.4 Qualified types

This area has spawned a great deal of research in the recent past. The term *qualified type* was coined by Mark Jones [25, 29]. This system subsumes and generalizes Haskell's type system.

In the HM type system, if $f(\alpha)$ is a type, then $\forall\alpha\,.\,f(\alpha)$ represents the set of types:

$$\{f(\tau) \mid \tau \text{ is a type}\}$$

There are times when we wish to restrict α to a subset of all types. Mark Jones expresses this as:

$$\pi(\alpha) \Rightarrow f(\alpha)$$

where π is a type predicate. The interpretation of this is the set of types:

$$\{f(\tau) \mid \tau \text{ is a type such that } \pi(\tau) \text{ holds}\}$$

In [29], individual predicates are written $\pi = p\ \tau_1\tau_2\ldots\tau_n$ describing an n-place relation p between the types $\tau_1\tau_2\ldots\tau_n$. There is an entailment relation ($\Vdash$) between sets of predicates P, Q, whose properties depend on the specific system. As a shorthand, where sets of predicates are required, individual predicates π may be used to indicate the singleton set containing that predicate.

Jones proves various useful results about such a system, as long as the predicate sets obey several simple properties:

monotonicity: $\dfrac{P \supseteq P'}{P \Vdash P'}$

transitivity: $\dfrac{P \Vdash Q \text{ and } Q \Vdash R}{P \Vdash R}$

closure property: $\dfrac{P \Vdash Q}{\theta P \Vdash \theta Q}$ for any type substitution θ

The idea of predicates on types that satisfy the conditions mentioned above has wide applicability, encompassing such disparate features as Haskell type classes, flexible records, and subtyping.

Features	**Predicates**
Haskell type classes	$C\ \tau$
flexible records	r `lacks` l
subtyping	$\tau' \subseteq \tau$

A system that supports subtyping has the predicate form $\tau' \subseteq \tau$, which means that type τ' is a *subtype* of type τ. Such a system typically has the inference rule of *subsumption*:

$$(\mathbf{sub})\quad \frac{P|A \vdash e : \tau' \quad P \Vdash \tau' \subseteq \tau}{P|A \vdash e : \tau}$$

These three systems are more fully described in [29].

The type judgements in the system of qualified types are of the form:

$$P|A \vdash E : \tau$$

Note that the left-hand side of the judgement has a P in addition to the usual environment A. P is a set of predicates satisfied in the environment. The judgement can be read as follows: "Expression E has type τ in the presence of type assignment A and predicates P."

Figure 1.4 has the syntax-directed rules.

The two functions *Inst* and *Gen* are similar to their counterparts in the HM type rules, the difference being that *Inst* now returns an instantiated type *and* context, and *Gen* considers free variables in the context in addition to the rest of the type.

One of the properties proven for this system is the existence of principal types for the type inference algorithm given. The definition of a principal type retains its

$$(\mathbf{var}) \qquad \frac{(x:\sigma) \in A}{P|A \vdash x : \tau} \quad (P \Rightarrow \tau) = \mathit{Inst}(\sigma)$$

$$(\mathbf{app}) \qquad \frac{P|A \vdash E_1 : \tau' \to \tau \qquad P|A \vdash E_2 : \tau'}{P|A \vdash (E_1\, E_2) : \tau}$$

$$(\mathbf{abs}) \qquad \frac{P|A, x : \tau' \vdash E : \tau}{P|A \vdash (\lambda x.E) : \tau' \to \tau}$$

$$(\mathbf{let}) \quad \frac{P|A \vdash E_1 : \tau \qquad P'|A, x : \sigma \vdash E_2 : \tau'}{P'|A \vdash (\texttt{let}\; x = E_1 \;\texttt{in}\; E_2) : \tau'} \quad \sigma = \mathit{Gen}(A, P \Rightarrow \tau)$$

Figure 1.4: Typing rules for qualified types

intuitive meaning, but details change due to the need to deal with predicates. We need a few definitions:

Constrained type scheme A pair of the form $(P|\sigma)$, where P is a set of predicates, and σ is a type scheme.

Generic instance A qualified type $R \Rightarrow \tau$ is a generic instance of the constrained type $(P|\forall\alpha_1, \ldots, \alpha_n.Q \Rightarrow \tau')$ if there exist types τ_i such that $R \Vdash P \cup [\alpha_i \mapsto \tau_i]Q$ and $\tau = [\alpha_i \mapsto \tau_i]\tau'$.

Generality The constrained type scheme $(Q|\sigma')$ is more general than $(P|\sigma)$ if every generic instance of $(Q|\sigma')$ is also a generic instance of $(P|\sigma)$.

A principal type scheme for a term M under type assignment A is a constrained type scheme $(P|\sigma)$ such that $P|A \vdash M : \sigma$, and $(P'|\sigma') \leqslant (P|\sigma)$ whenever $P'|A \vdash M : \sigma'$.

The typical code generation strategy for terms in such a system involves the translation of predicates into *evidence*. That is, a value of a type of the form $\forall\alpha.\pi(\alpha) \Rightarrow \tau$ is translated into a function taking a parameter representing the predicate $\pi(\alpha)$, and returning a value of type τ.[7] The evidence is used to implement the capabilities implied by the predicate. For example, the evidence for the Haskell predicate `Show` τ is typically a function of type $\tau \rightarrow$ `String` which returns the string representation of its argument. The Haskell function

```
prList xs = concat (map show xs)
```

has the type

$$\forall\alpha\,.\,\texttt{Show}\;\alpha \Rightarrow [\alpha] \rightarrow \texttt{String}$$

The types of the functions used in the body of `prList` are

Name	Type
`concat`	$\forall\alpha\,.\,[[\alpha]] \rightarrow [\alpha]$
`map`	$\forall\alpha,\beta\,.\,(\alpha \rightarrow \beta) \rightarrow ([\alpha] \rightarrow [\beta])$
`show`	$\forall\alpha\,.\,\texttt{Show}\;\alpha \Rightarrow \alpha \rightarrow \texttt{String}$

`prList` would be translated into a function whose type is

$$\forall\alpha\,.\,(\alpha \rightarrow \texttt{String}) \rightarrow ([\alpha] \rightarrow \texttt{String})$$

The translated function would be

[7]Typically, τ is a function type, and a simple optimization is to increase the arity of the function, i.e. passing the evidence as an extra parameter rather than to generate a curried function.

```
    prList f xs = concat (map (show f) xs)
```

`f` is the evidence parameter corresponding to the `Show` α predicate. Note that it is passed to the expanded show function, because `show`'s type has the predicate `Show` α.

Strictly speaking, therefore, qualified types do not increase the power of a typing system, in that they do not increase the set of typable terms. It does, however, increase the convenience (which is admittedly hard to define or measure) of using a language, because it allows the programmer to omit certain parameters with the knowledge that the type-directed translation process will fill them in correctly.

There are a number of further issues raised. The system, as described, generates translations with redundancies which can cause inefficiencies. For example, consider the let rule in Figure 1.4. If $P' \Vdash Q$, for some Q such that $Q \subseteq P$, then there is no need to pass in any evidence parameters represented by Q, since they are guaranteed to be the same for every use of the variable. In addition, depending on the nature of the predicates in a given system, it may be possible to simplify the set of predicates in a qualified type. For example, in a naive implementation of the system of extensible records described in [29], the function

```
fun f x = (x.a, x.a)
```

would have the inferred type:

$$\forall \alpha, \beta, r. \langle r \texttt{ lacks } a \rangle \Rightarrow \{\texttt{a} : \alpha, \texttt{a} : \beta; r\} \Rightarrow \alpha \times \beta$$

although because a field name may not be used multiple times within a record, the simpler type

$$\forall \alpha, r. \langle r \texttt{ lacks } a \rangle \Rightarrow \{\texttt{a} : \alpha; r\} \Rightarrow \alpha \times \alpha$$

can be used without any loss of generality. Simplification is subject to ongoing research [24].

Another issue is the possibility that for an inferred type of the form $P \Rightarrow \tau$, the set of predicates P may be unsatisfiable. This possibility may be ignored without violating any correctness criteria, because a value of such a type cannot ever be used if we cannot present evidence for the predicates. Nevertheless, it would be desirable for the system to warn the programmer of such types, since they are likely to indicate programming errors.

1.5 Contribution

Our contribution is the development of a polymorphic type system for records which supports a number of operations, including three kinds of concatenation, natural join, and, in fact, most of the record operations discussed in the literature. This is the first system to combine such a large set of powerful record operations. We prove that the type system is sound and present a type inference algorithm. The principal difficulty for type inference is the determination of predicate entailment. We give two algorithms to accomplish this. The second of these is used in the compilation algorithm, where we translate our implicitly typed record calculus into a simpler, explicitly typed implementation calculus where the predicates become evidence parameters. We prove this implementation calculus to be sound with respect to its semantics. Although record concatenation can be found in the literature, and a compilation method for polymorphic records is described by Ohori in [41], this is the first work in which a compilation method for concatenation is discussed.

Chapter 2

A Simply-typed record calculus

Before discussing the full system designed in this dissertation, we define a simplified simply-typed system in this chapter, and prove various properties. The essential simplification is that all types in the first-order system are monomorphic. This precludes us from expressing various constraints of types, but we will introduce the predicate forms that do make sense in this system, and solve the entailment relation for them. This gives us the opportunity to introduce various issues that will be the concern of subsequent chapters.

We distinguish between expressions and values, which are a subset of expressions. This distinction will be useful when we define the semantics of the language. Types are those standard from the simply-typed lambda calculus, augmented with record types. A record type is the record type constructor $\{\cdot\}$ applied to a *row*, which is a finite mapping from labels to types. The notation

$$l_1 : \tau_1, \ldots, l_n : \tau_n,$$

where $l_i \neq l_j$ for all $i, j \in \{1..n\}$ such that $i \neq j$, denotes the row which is defined on the set of labels $\{l_1, \ldots, l_n\}$, and which maps label l_i to type τ_i, for all $i \in \{1..n\}$. Each $l : \tau$ element is a *field*. We say that field $l : \tau$ has label l and type τ. Because a row is a map, the order in which fields are written is not relevant. There is a natural view of a map as a set of fields whose labels are unique.

Expressions and types are defined in Figure 2.1.

We need a set of type rules to tell us if an expression is well-formed, and what its type is. Figure 2.2 is a set of the usual set of type rules for the simply-typed lambda-calculus[1] augmented with several rules for record operations.

In order to type some record operations, we need to ensure that the types involved satisfy certain conditions, which we call *predicates*. For example, our record concatenation operation is strict; i.e., two records may be concatenated only if their fields have no labels in common. The simply-typed system has two sorts of predicates:

- $row_1 \# row_2$ – which means that row_1 and row_2 have disjoint domains. In other words, the labels in row_1 and row_2 are disjoint.
- $row_1 \blacktriangleright row_2$ – which means that row_1 consists of all the fields in row_2, with matching types, and possibly others. We read this as "row_1 is coerceable to row_2".

We let π range over predicates. We have an entailment relation on predicates which we write $\Vdash \pi$. This means that predicate π is true. $\nVdash$ is the negation of this relation. For example,

[1]See [4] for details.

$$\begin{array}{llll}
E & ::= & V & \text{value} \\
 & \mid & E\,E & \text{function application} \\
 & \mid & \{l = E, \ldots, l = E\} & \text{record} \\
 & \mid & E.l & \text{field selection} \\
 & \mid & E \,\&\, E & \text{record concatenation} \\
 & \mid & E \backslash l & \text{field elimination} \\
 & \mid & (E :> \tau) & \text{record coercion} \\
 & & & \\
V & ::= & c & \text{constant} \\
 & \mid & x & \text{variable} \\
 & \mid & \lambda x : \tau \,.\, E & \text{lambda abstraction} \\
 & \mid & \{l = V, \ldots, l = V\} & \text{record value} \\
 & & & \\
\tau & ::= & t & \text{predefined types} \\
 & \mid & \tau \to \tau & \text{function type} \\
 & \mid & \{\mathit{row}\} & \text{record type} \\
 & & & \\
\mathit{row} & ::= & \emptyset & \text{empty row} \\
 & \mid & l : \tau & \text{field} \\
 & \mid & \mathit{row}, \mathit{row} & \text{row concatenation}
\end{array}$$

Figure 2.1: Simply-typed expressions

$$(\textbf{var}) \qquad A \vdash x : \tau \quad (x : \tau) \in A$$

$$(\textbf{abs}) \qquad \frac{A, (x : \tau_1) \vdash e : \tau_2}{A \vdash (\lambda x : \tau_1 . e) : \tau_1 \to \tau_2}$$

$$(\textbf{app}) \qquad \frac{A \vdash e : \tau' \to \tau \quad A \vdash e' : \tau'}{A \vdash (e\, e') : \tau}$$

$$(\textbf{record}) \qquad \frac{A \vdash e_i : \tau_i \quad \forall i \in 1..n}{A \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}} \quad l_1, \ldots, l_n \text{ unique}$$

$$(\textbf{sel}) \qquad \frac{A \vdash e : \{(l : \tau), \mathit{row}\} \quad \Vdash \mathit{row} \# (l : \tau)}{A \vdash (e.l) : \tau}$$

$$(\textbf{restrict}) \qquad \frac{A \vdash e : \{(l : \tau), \mathit{row}\} \Vdash \mathit{row} \# (l : \tau)}{A \vdash (e \backslash l) : \{\mathit{row}\}}$$

$$(\textbf{concat}) \qquad \frac{A \vdash e_1 : \{\mathit{row}_1\} \quad A \vdash e_2 : \{\mathit{row}_2\} \quad \Vdash \mathit{row}_1 \# \mathit{row}_2}{A \vdash (e_1 \,\&\, e_2) : \{\mathit{row}_1, \mathit{row}_2\}}$$

$$(\textbf{coerce}) \qquad \frac{A \vdash e : \{\mathit{row}'\} \quad \Vdash \mathit{row}' \blacktriangleright \mathit{row}}{A \vdash (e :> \{\mathit{row}\}) : \{\mathit{row}\}}$$

Figure 2.2: Type rules for simply-typed records

$$(\textbf{disjoint})\quad \frac{\{l_1,\ldots,l_n\}\cap\{l'_1,\ldots,l'_m\}=\varnothing}{\Vdash (l_1:\tau_1,\ldots,l_n:\tau_n)\mathrel{\#}(l'_1:\tau'_1,\ldots,l'_m:\tau'_m)}$$

$$(\textbf{coerceable})\quad \frac{\{l_1:\tau_1,\ldots,l_n:\tau_n\}\supseteq\{l'_1:\tau'_1,\ldots,l'_m:\tau'_m\}}{\Vdash (l_1:\tau_1,\ldots,l_n:\tau_n)\blacktriangleright(l'_1:\tau'_1,\ldots,l'_m:\tau'_m)}$$

Figure 2.3: Predicate rules for simply-typed records

$$\begin{array}{lcl}
c\,V & \longrightarrow & \delta(c,V) \\
(\lambda x:\tau.E)\,V & \longrightarrow & [x\mapsto V]E \\
\{\overrightarrow{l=V}\}\mathbin{\&}\{\overrightarrow{l'=V'}\} & \longrightarrow & \{\overrightarrow{l=V},\overrightarrow{l'=V'}\} \\
\{l_1=V_1,\ldots,l=V,\ldots,l_n=V_n\}.l & \longrightarrow & V \\
\{l_1=V_1,\ldots,l_n=V_n\}\backslash l_i & \longrightarrow & \begin{array}{l}\{l_1=V_1,\ldots,l_{i-1}=V_{i-1},\\ l_{i+1}=V_{i+1},\ldots,l_n=V_n\}\end{array} \\
\{l_1=V_1,\ldots,l_n=V_n\} :> \{l_{i_1}:\tau_{i_1},\ldots,l_{i_m}:\tau_{i_m}\} & \longrightarrow & \{l_{i_1}=V_{i_1},\ldots,l_{i_m}=V_{i_m}\}
\end{array}$$

Figure 2.4: Simply-typed notions of reduction

$$\Vdash (a:\mathit{int},b:\mathit{real})\mathrel{\#}(c:\mathit{bool})$$
$$\not\Vdash (a:\mathit{int},b:\mathit{real})\mathrel{\#}(b:\mathit{bool})$$
$$\Vdash (a:\mathit{int},b:\mathit{real})\blacktriangleright(b:\mathit{real})$$
$$\not\Vdash (a:\mathit{int},b:\mathit{real})\blacktriangleright(b:\mathit{bool})$$
$$\not\Vdash (a:\mathit{int},b:\mathit{real})\blacktriangleright(c:\mathit{bool})$$

The predicate entailment rules are given in Figure 2.3.

We use a syntactic rewriting scheme, as described in [61], to define the semantics

$$
\begin{array}{rcl}
\mathcal{E} & ::= & [\,] \\
& | & \mathcal{E}\, e \\
& | & V\, \mathcal{E} \\
& | & \mathcal{E} \,\&\, E \\
& | & V \,\&\, \mathcal{E} \\
& | & \mathcal{E}.l \\
& | & \mathcal{E}\backslash l \\
& | & \{l_1 = V_1, \ldots, l_{i-1} = V_{i-1}, l_i = \mathcal{E}, l_{i+1} = E_{i+1}, \ldots, l_n = E_n\}
\end{array}
$$

Figure 2.5: Untyped Context

of this language. The rewriting scheme is built up in several stages. At the lowest level, we have a set of rewriting rules called *notions of reduction*. We write this relation on pairs of programs like so:

$$e_1 \longrightarrow e_2$$

For our simply-typed system, the notions of reduction are given in Figure 2.4. The function δ is used to abstract the semantics of any predefined functions, e.g. addition.

We cannot simply use these rules as is because they do not explain how to reduce subterms. The rules apply to any program which matches one of the forms on the right side of the relation symbol. However, there are programs which do not match any of these forms, yet which may contain subexpressions which do match. An example of this is the expression

$$\{a = (\lambda x : \mathit{int} \,.\, \mathit{succ}\ x)1\} \,\&\, (\{b = 2\} \,\&\, \{c = 3\})$$

This does not match the notion

$$\{\overrightarrow{l = V}\} \,\&\, \{\overrightarrow{l' = V'}\} \;\longrightarrow\; \{\overrightarrow{l = V}, \overrightarrow{l' = V'}\}$$

(where $\overrightarrow{l = V}$ means $l_1 = V_1, \ldots, l_n = V_n$) because the right operand of the outermost concatenation does not match the form

$$\{\overrightarrow{l' = V'}\}$$

Adding more notions of reduction to cover more cases does not help, because subterms where we would like to apply a notion can be nested arbitrarily deep. Instead, we define how to rewrite programs which contain subexpressions for which a notion is applicable.

Consider the program above. There are two subexpressions where we can apply a notion of reduction. They are:

$$(\lambda x : \mathit{int} \,.\, \mathit{succ}\ x)\ 1 \quad \text{and}$$
$$\{b = 2\} \,\&\, \{c = 3\}$$

We have a choice as to which subexpression to rewrite first. For this particular language, the choice is actually irrelevant, because all subexpressions must eventually be evaluated, and termination is guaranteed. However, this would not be the case if we add side-effects or an if-expression to the language, among other things. We will therefore choose a strict order of evaluation. There are two guidelines:

1. the operands of a function or operation are evaluated before it is applied

2. subexpressions are evaluated in a left-to-right order

We formalize these guidelines by defining another relation on pairs of programs, written

$$e_1 \longmapsto e_2.$$

We restrict the subexpressions where the notions of reduction are applicable by defining a *context*. Informally, a context is an expression with a "hole", which is written [], where one of its subexpressions would be. We denote contexts by $\mathcal{E}$ and $\mathcal{C}$. $\mathcal{E}[e]$ denotes the context $\mathcal{E}$ with its hole filled in with e.

In our case, the hole is the place in the expression where we allow a notion of reduction to be applied. The evaluation context $\mathcal{E}$ that we will use is defined in Figure 2.5. We can now define $\longmapsto$:

$$\mathcal{E}[e_1] \longmapsto \mathcal{E}[e_2] \quad \text{iff } e_1 \longrightarrow e_2$$

The reflexive and transitive closure of $\longmapsto$ is $\longmapsto\!\!\!\!\rightarrow$.

Finally, we define the evaluation of a program e as the partial function *eval*:

$$\mathit{eval}(e) = v \quad \text{iff } e \longmapsto\!\!\!\!\rightarrow v$$

Here is an example of the use of these evaluation rules:

$$\begin{array}{ll} & (\lambda x : \{a : \mathit{int}\} \,.\, (x \,\&\, \{b = x.a + 1\}))\, \{a = 2\} \\ \longmapsto & \{a = 2\} \,\&\, \{b = \{a = 2\}.a + 1\} \\ \longmapsto & \{a = 2\} \,\&\, \{b = 2 + 1\} \\ \longmapsto & \{a = 2\} \,\&\, \{b = 3\} \\ \longmapsto & \{a = 2, b = 3\} \end{array}$$

Theorem 2.0.1 (Progress). *If E is a closed, well-typed term, then either E is a value, or there exists some E' such that $E \longrightarrow E'$.*

Proof. The proofs of the Theorems in this chapter are subsumed by the proofs of the corresponding Theorems of Chapter 4 and are therefore not repeated. In general, proofs of propositions, lemmas and theorems which do not appear in the text can be found in Appendix B. $\square$

Theorem 2.0.2 (Subject Reduction). *If* $A \vdash E : \tau$ *and* $E \longmapsto E'$*, then* $A \vdash E' : \tau$.

Theorem 2.0.3 (Type Soundness). *If* $A \vdash E : \tau$*, then* $E \longmapsto\!\!\!\!\rightarrow V$ *and* $A \vdash V : \tau$.

Chapter 3

Overview of a Polymorphic Record Calculus

In this chapter, we informally describe the various operations on records that are supported by our system. Because it makes essential use of the predicate forms $\_ \# \_$, $\_ \parallel \_$, and $\_ = \_$, we will refer to the system as $\lambda^{\#,\parallel,=}$.

3.1 Overview

Our system of records, $\lambda^{\#,\parallel,=}$, is based on rows, introduced by Wand [56]. The type system uses qualified types, as described by Mark Jones [26]. A row is a finite map from labels to types. The record type constructor $\{\cdot\}$ applied to a row yields a record type. Our records are unordered. That is, there is no notion of whether one field in a record precedes another.

Note that a row is not a type; it is considered a different *kind*. Informally, a kind is the type of a type. The kind of an entity determines what sort of operations is may be subjected to. For example, the most basic kind is the kid of simple types, typically written $*$. Examples of types with this kind are `int`, `bool`, and `int` $\rightarrow$ `bool` $\rightarrow$ `int`. The function type constructor, $\rightarrow$, is not of kind $*$. It takes two types (i.e., the argument and result types) and yields a type. For example, `int` $\rightarrow$ `bool` is considered the application of $\rightarrow$ to `int` and `bool`. This kind of $\rightarrow$ is therefore $* \rightarrow * \rightarrow *$. We assign a new kind to rows: R. In this framework, the record type constructor has kind $R \rightarrow *$.

To support polymorphic operations on records, we can use row variables when we have a record whose components are not all statically known. For example,

$$\{l : \tau, \rho\}$$

is the type of a record which contains a field l of type τ, and possibly other fields, represented by row variable ρ.

A contribution of our work is that, unlike in previous systems using rows, [18, 20, 28, 56, 57], etc., the concatenation of two rows (which must satisfy a disjointness condition) is itself a row. This allows us to express various record operations in a natural way.

3.2 Basic operations

The most basic record operation is field selection. We will use the notation $\_.l$ to denote the name of the field selection operation, specialized for label l. In general, if an operation is written using any notation other than the standard prefix format, we will write the name of the operation by writing an instance of the expression and replacing all arguments with $\_$. An attempt to fix a type for field selection may result in

$$(\_.l) : \forall \alpha, \rho\,.\,\{l : \alpha, \rho\} \rightarrow \alpha$$

However, because we do not allow a record to have two fields with the same label, we must ensure that ρ does not also have a field labeled l. We accomplish this by the use of a predicate:

$$(\_.l) : \forall \alpha, \rho\,.\,\rho \mathrel{\#} l \Rightarrow \{l : \alpha, \rho\} \rightarrow \alpha$$

The predicate $\rho \mathrel{\#} l$ asserts that the domain of ρ, i.e., the labels in the fields denoted by ρ, do not include l. In general, the $\#$ predicate asserts that the sets of labels denoted by the two operands are disjoint. When either operand is a row variable or row, as in the case above, the domain of that row variable or row is denoted, respectively. If either operand is a single label, the singleton set containing that label is denoted. Labelsets are ranged over by *lset*.

Because functional languages do not support in-place modification of fields, it would be nice to have a way to create a record value which has all the fields of an existing record value, except for one field whose new value is specified. We can give this operation a type:

$$(\_\ \texttt{update}\ l = \_) : \forall \alpha, \rho\,.\,\{\rho, l : \alpha\} \times \alpha \rightarrow \{\rho, l : \alpha\}$$

A related operation is record extension. This takes a record and a new field value, and constructs a new record value which has all the fields of the old value and a new field. We have the choice of prohibiting the record operand from having the new field, or allowing it. We call the former operation *strict* extension and the latter *non-strict* extension. The strict version is easier to type:

$$(\_\ \texttt{with}\ l = \_) : \forall \alpha, \rho \,.\, \rho \mathrel{\#} l \Rightarrow \{\rho\} \times \alpha \rightarrow \{\rho, l : \alpha\}$$

Our system can also give the latter operation a type, which requires the predicate form $lset_1 \parallel lset_2$, which is read "$lset_1$ is parallel to $lset_2$". This predicate asserts that the two labelsets are equal. (We do not use the standard = symbol the two labelsets are equal to avoid overloading it – a third predicate using it is coming up.)

$$(\_\ \texttt{with}\ l := \_) : \forall \alpha, \rho, \rho' \,.\, (\rho_l, \rho'_l) \parallel l \Rightarrow \{\rho, \rho_l\} \times \alpha \rightarrow \{\rho, l : \alpha\}$$

The complementary basic operation is *restriction*. This operation removes a field with the given label l from a record. Just as for extension, we have the choice of requiring that the record contain a field with this label or not. We will call the first choice *strict* restriction. Our system can give this operation an appropriate type:

$$(\_\backslash l) : \forall \alpha, \rho \,.\, \rho \mathrel{\#} l \Rightarrow \{l : \alpha, \rho\} \rightarrow \{\rho\}$$

We can also give an appropriate type to a relaxed version, which we will call *non-strict*:

$$(\_ - l) : \forall \alpha, \rho, \rho' \,.\, \rho \mathrel{\#} l, (l : \alpha) \blacktriangleright \rho' \Rightarrow \{\rho, \rho'\} \rightarrow \{\rho\}$$

Note that the predicate $\rho \mathrel{\#} \rho'$ must be satisfied for the type of the function operand above, i.e., $\{\rho, \rho'\}$, to be well-formed. This predicate is not explicit in the qualification of the type above, however, because it is implied by the combination of the two that are explicitly mentioned. The first predicate prohibits ρ from having a field labeled with l, and the second predicate restricts ρ' to either being the empty row, or consisting solely of the field $l : \alpha$. Therefore it must be true that the two row variables have no labels in common. We say that $\rho \# \rho'$ is *entailed* by the two predicates $\rho \# l$ and $(l : \alpha) \blacktriangleright \rho'$.

3.3 Concatenation

In addition to defining record extension, which adds a single field to an existing record, we can use the power of our system to define concatenation. Concatenation is generally regarded as one of the more difficult record operations to statically type. There are several different concatenation operations described in the literature. The simplest and most common is *symmetric* concatenation [20], in which the two operands are prohibited from having any fields with the same label. In our system, this is enforced by the $\#$ predicate. The type of symmetric concatenation is:

$$(\_\,\&\,\_) : \forall \rho_1, \rho_2 \,.\, \rho_1 \# \rho_2 \Rightarrow \{\rho_1\} \times \{\rho_2\} \rightarrow \{\rho_1, \rho_2\}$$

A more complex variant allows fields to overlap, but only if their types are the same. For fields that overlap, the value of the field in the second operand is selected. We will call this variant *compatible concatenation* and write the operation $|\&|$. For example,

$$\{\mathtt{a} = 5, \mathtt{b} = \text{"}\mathtt{Hi}\text{"}\} \;|\&|\; \{\mathtt{b} = \text{"}\mathtt{Bye}\text{"}, \mathtt{c} = 2.3\}$$

would be legal, and result in

$$\{\mathtt{a} = 5, \mathtt{b} = \text{"}\mathtt{Bye}\text{"}, \mathtt{c} = 2.3\}.$$

However,

$$\{\mathtt{a} = 5, \mathtt{b} = \text{"}\mathtt{Hi}\text{"}\} \;|\&|\; \{\mathtt{b} = 0, \mathtt{c} = 2.3\}$$

Symbol	**Description**	**Qualified Type**
$\_ \,\&\, \_$	symmetric	$\rho_1 \# \rho_2 \Rightarrow \{\rho_1\} \times \{\rho_2\} \to \{\rho_1, \rho_2\}$
$\_ \,\vert\&\vert\, \_$	compatible	$\rho_1 \# \rho_2, \rho_1 \# \rho_3, \rho_2 \# \rho_3$ $\Rightarrow \{\rho_1, \rho_2\} \times \{\rho_2, \rho_3\} \to \{\rho_1, \rho_2, \rho_3\}$
$\_ \,\&\&\, \_$	unrestricted	$\rho_1 \# \rho_2, \rho_1 \# \rho_3, \rho_2 \# \rho_3, \rho_3 \parallel \rho_3'$ $\Rightarrow \{\rho_1, \rho_3'\} \times \{\rho_2, \rho_3\} \to \{\rho_1, \rho_2, \rho_3\}$

Figure 3.1: Types for different concatenation operations

would be prohibited, because field `b` is a string in the first operand, and an integer in the second. To give compatible concatenation a proper type, we need another predicate form: $lset_1 \parallel lset_2$. This asserts that the two labelsets are equal. Just as for the disjointness predicate, we extend this to take rows and single labels as operands.

The type of compatible concatenation is:

$$(\_ \,|\&|\, \_) : \forall \rho_1, \rho_2, \rho_3 \,.\, \rho_1 \# \rho_2, \rho_1 \# \rho_3, \rho_2 \# \rho_3 \Rightarrow (\{\rho_1, \rho_2\} \times \{\rho_2, \rho_3\}) \to \{\rho_1, \rho_2, \rho_3\}$$

Finally, there is the variant which has no restrictions with respect to field overlap. As before, values (and types) of fields in the second operand take precedence. This variant, which we call *unrestricted concatenation*, would allow the previous prohibited example:

$$\{\mathtt{a} = 5, \mathtt{b} = \text{"}\mathtt{Hi}\text{"}\} \,|\&|\, \{\mathtt{b} = 0, \mathtt{c} = 2.3\} \quad \longrightarrow \quad \{\mathtt{a} = 5, \mathtt{b} = 0, \mathtt{c} = 2.3\}$$

Our system can statically type each of these variants. The type schemes for each are summarized in Figure 3.1.

In any of these variants, when we statically know the type of the right operand (either operand for symmetric concatenation), it is possible to dispense with the polymorphic concatenation operation and define concatenation as a series of record extensions of the right flavor. For example,

$$x \mathbin{\&} \{l_1 = e_1, \ldots, l_n = e_n\}$$

can be rewritten as

$$(\ldots(x \;\texttt{with}\; l_1 = e_1) \ldots \texttt{with}\; l_n = e_n)$$

It is only when the operands' types are not statically known where we need polymorphic concatenation.

3.4 Join

Our system can also type the relational join operation (assuming an equality operation on records). This is essentially a variant of concatenation. The parameters of the operation are two sequences of records, and it returns a sequence of the "joined" records. Let us initially ignore the "sequence" part and consider an operation $\texttt{join}_0$ that takes two records and returns a single record. The type of this would be[1]:

$$\texttt{join}_0 : \forall \rho_1, \rho_2, \rho_3 \,.\, \rho_1 \# \rho_2, \rho_1 \# \rho_3, \rho_2 \# \rho_3 \Rightarrow (\{\rho_1, \rho_2\} \times \{\rho_2, \rho_3\}) \rightarrow \{\rho_1, \rho_2, \rho_3\}$$

In order to type the actual join operation, we need to consider two complications:

[1]Note that this is identical to the type for compatible concatenation

1. The operands and result are actually sequences of records rather than single records.

2. We need to be able to test the common fields of the two parameters for equality.

The most obvious way of dealing with point (1) is for join to take two lists (of records) and return a list (of records). In the presentation of our system, we do not consider lists or other type constructors (except for $\rightarrow$), as it would distract from the crucial points. In SML [38], the ability to compare two values of some arbitrary type for equality is expressed by *equality type variables*. This is considered one of the weak points of SML. For an explanation, see [2]. In Haskell [45], the class `Eq` is used for this purpose. We do not consider classes either in our presentation, for the same reason. Nevertheless, it should be straightforward to extend this system with the various features found in the type systems of languages such as SML and Haskell.

If we combine our system with Haskell's type system, we can give the join operation the type:

$$\begin{gathered}\forall \rho_1, \rho_2, \rho_3 \,.\, (\mathtt{Eq}\,\{\rho_2\}, \rho_1 \mathrel{\#} \rho_2, \rho_1 \mathrel{\#} \rho_3, \rho_2 \mathrel{\#} \rho_3) \\ \Rightarrow [\{\rho_1, \rho_2\}] \times [\{\rho_2, \rho_3\}] \rightarrow [\{\rho_1, \rho_2, \rho_3\}]\end{gathered}$$

In Haskell, a list of τ is written $[\tau]$, and the predicate $\mathtt{Eq}\,\tau$ means that there exists an equality operation operating on values of type τ.

We can write the code for this in Haskell, using the lower-level operation `rcomm`. This operation takes two compatible records and returns a 4-tuple which contains the non-common fields of the first operand, the common field of the first, the common fields of the second, and the remainder of the second record value. For example, evaluation of

$$\texttt{rcomm}\,\{\mathsf{a}=1,\mathsf{b}=2\}\,\{\mathsf{b}=3,\mathsf{c}=4\}$$

results in

$$(\{\mathsf{a}=1\},\{\mathsf{b}=2\},\{\mathsf{b}=3\},\{\mathsf{c}=4\}).$$

The type of this operation is:

$$\begin{gathered}\forall \rho_1,\rho_2,\rho_3\,.\,(\rho_1 \# \rho_2, \rho_1 \# \rho_3, \rho_2 \# \rho_3)\\ \Rightarrow (\{\rho_1,\rho_2\}\times\{\rho_2,\rho_3\}) \rightarrow (\{\rho_1\}\times\{\rho_2\}\times\{\rho_2\}\times\{\rho_3\})\end{gathered}$$

A sample implementation of join is:

```
join xs ys =
    [ x1 & y2 & y3 | (x1, x2, y2, y3) ← segments, x2 == y2 ]
    where segments = [ rcomm x y | (x,y) ← xys ]
          xys = concat (map (\ x → map (\ y → (x,y)) ys) xs)
```

(`concat` takes a list of lists and concatenates them.)

Our system also understands that join is associative. That is, it infers the same type scheme for the two functions `f` and `g`:

```
f (x, y, z) = join (join (x, y), z)
g (x, y, z) = join (x, join (y, z))
```

The type scheme for both `f` and `g` is

$$\begin{aligned}&\forall \rho_1,\rho_2,\rho_3,\rho_4,\rho_5,\rho_6,\rho_7\,.\,\texttt{Eq}\,\{\rho_3,\rho_4\},\texttt{Eq}\,\{\rho_2,\rho_4,\rho_6\},\ldots\\ &\Rightarrow ([\{\rho_1,\rho_2,\rho_3,\rho_4\}]\times[\{\rho_3,\rho_4,\rho_5,\rho_6\}]\times[\{\rho_2,\rho_4,\rho_6,\rho_7\}])\\ &\rightarrow [\{\rho_1,\rho_2,\rho_3,\rho_4,\rho_5,\rho_6,\rho_7\}]\end{aligned}$$

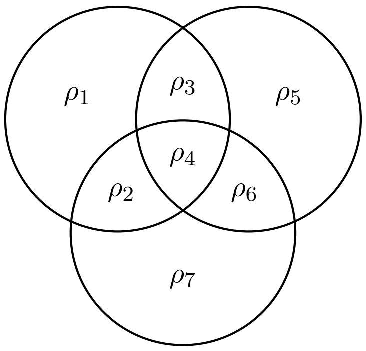

Figure 3.2: Venn diagram for associative join

where the ... represents the necessary disjointness predicates.

The relationship between the various row variables can better be visualized using the Venn diagram in Figure 3.2. The three sets represent the three rows in the parameters, represented as sets of fields. For right to left, top to bottom order, the sets represent the first, second and third arguments, respectively. The various ρ variables label various intersections of the three rows.

As we can see from the Venn diagram, the row variables are all pairwise disjoint. Because $\#$ is symmetric, i.e., $\rho \# \rho' \leftrightarrow \rho' \# \rho$, $(7 \times 6)/2 = 21$ disjointness predicates are necessary.

3.5 Optional arguments

We can also express the types of functions with optional arguments. To do this, we use the $\blacktriangleright$ predicate.

A function which takes multiple keyword parameters can be emulated by a function which takes a single record as its argument. The fields of this record correspond to the keyword parameters. If we denote the optional and mandatory parts of this record by row_O and row_M, respectively, and the function returns a value of type τ, then the type of the function is:

$$\forall \rho \,.\, row_O \blacktriangleright \rho \Rightarrow \{row_M, \rho\} \rightarrow \tau$$

It must be the case that row_M and row_O have no fields in common; this is of course statically checkable.

As an example, if we have a function `f` which takes mandatory keyword arguments $\mathtt{a} : \tau_a$ and $\mathtt{b} : \tau_b$, and optional keyword arguments $\mathtt{c} : \tau_c$ and $\mathtt{d} : \tau_d$, then `f`'s type would be:

$$\forall \rho \,.\, (\mathtt{c} : \tau_c, \mathtt{d} : \tau_d) \blacktriangleright \rho \Rightarrow \{\mathtt{a} : \tau_a, \mathtt{b} : \tau_b, \rho\} \rightarrow \tau$$

The predicate constrains ρ to be a subset of the row $(\mathtt{c} : \tau_c, \mathtt{d} : \tau_d)$. The function argument, therefore, must be a record with fields `a` and `b`, and possibly `c` and/or `d`, each of which must be of the appropriate type. The type for `f` will admit no other type for its argument.

Note that the predicates $\rho \mathrel{\#} \mathtt{a}$ and $\rho \mathrel{\#} \mathtt{b}$ must be satisfied for $\{\mathtt{a} : \tau_a, \mathtt{b} : \tau_b, \rho\}$ to be a well-formed type. However, they are not required to be explicitly stated in the predicate set because they are entailed by the coercion predicate.

To clarify how this scheme would work, we show how to translate a function which takes optional arguments into our system:

If we have a function

```
fun f { a, b, c = "xyz", d = 0 } = ...
```

where `a` and `b` are mandatory, and `c` and `d` are optional with the specified defaults, we can translate this into

```
fun f r = let val r' = { c = "xyz", d = 0 } |&| r
              val a = r'.a
              val b = r'.b
              val c = r'.c
              val d = r'.d
          in  ...
          end
```

`r'` will have the values for all arguments. If an optional argument is omitted, the concatenation

```
 { c = "xyz", d = 0 } |&| r
```

will ensure that the default values are provided.

`f` would have the type

$$\forall \rho, \rho' \,.\, (\rho, \rho') = (\texttt{c} : \texttt{string}, \texttt{d} : \texttt{int}), \rho \mathrel{\#} \rho' \Rightarrow \{\texttt{a} : \tau_a, \texttt{b} : \tau_b, \rho\} \to \tau,$$

In Section 5.3.1, page 74, we will see that this is equivalent to

$$\forall \rho \,.\, (\texttt{c} : \texttt{string}, \texttt{d} : \texttt{int}) \blacktriangleright \rho \Rightarrow \{\texttt{a} : \tau_a, \texttt{b} : \tau_b, \rho\} \to \tau$$

Chapter 4

A Second-Order Record Calculus

4.1 Overview

We first describe our type system for $\lambda^{\#,\|,=}$, and follow by developing a type inference system which derives the most general type allowed by the type rules. The type inference rules rely on the $P \Vdash Q$ entailment relation between sets of predicates. An algorithm for deciding entailment is given in section 5.5. In section 5.3, we describe a problem with ambiguous types and how this problem is solved by making use of the implicit dependency information in predicates. In Chapter 6, we consider the augmentation of the system with evidence, in preparation for compilation. The type rules of the original type system and the type inference algorithm are easily augmented with rules to handle evidence. We show that the algorithm given for entailment cannot be so augmented, due to its reliance on classical logic instead of constructive logic. Section 6.4 discusses the difference between these two systems of logic. In section 6.5, we show that is it possible to restrict the use of classical logic to a portion of the algorithm where its power is not a liability.

$$
\begin{array}{rcll}
E & ::= & V & \text{value} \\
 & | & E\,E & \text{function application} \\
 & | & \textbf{let } x = E \textbf{ in } E & \text{let expression} \\
 & | & \{l = E, \ldots, l = E\} & \text{record} \\
 & | & E.l & \text{field selection} \\
 & | & E \backslash l & \text{field elimination} \\
\\
V & ::= & c & \text{constant} \\
 & | & x & \text{variable} \\
 & | & \lambda x.E & \text{lambda abstraction} \\
 & | & \{l = V, \ldots, l = V\} & \text{record value}
\end{array}
$$

Figure 4.1: Syntax of expressions and values

4.2 Type system elements

For the full system, we use system OML described in [26], which is the core ML type system extended with qualified types. We in turn add rows, row variables, and labelsets. Our predicates involve rows and labelsets. Figures 4.1 give the syntax of expressions and values. The syntax of types and rows is provided by Figure 4.2. Figure 4.3 defines the syntax of predicates.

We will see later that type equality, row equality and row coercion predicates can be eliminated. A *labelset element* is either a label or a row variable. *le* ranges over labelset elements. Similarly, *re* ranges over *row elements*, which are either a field or a row variable. We will, when convenient, abuse the notation for labelsets by writing a row where a labelset is expected. This is shorthand for the domain

τ	$::=$	t	predefined type
	$\mid$	α	type variable
	$\mid$	$\tau \rightarrow \tau$	function type
	$\mid$	$\{row\}$	record type
row	$::=$	$\varnothing$	empty row
	$\mid$	re	row element
	$\mid$	row, row	disjoint row union
$lset$	$::=$	$\varnothing$	empty set
	$\mid$	le	labelset element
	$\mid$	$lset, lset$	disjoint labelset union
le	$::=$	l	single label
	$\mid$	ρ	domain of row variable
re	$::=$	$l : \tau$	field
	$\mid$	ρ	row variable

Figure 4.2: Syntax of types and rows

$$\begin{array}{rcll}
\tau & = & \tau & \text{type equality} \\
row & = & row & \text{row equality} \\
row & \blacktriangleright & row & \text{coerceable rows} \\
lset & \# & lset & \text{disjoint label sets} \\
lset & \| & lset & \text{equal label sets}
\end{array}$$

Figure 4.3: Predicate syntax

of the row. We will enclose labelsets and rows in parentheses where necessary to clarify grouping. Because we interpret a row as a finite mapping from labels to types, and the comma separating rows as disjoint union, the order of fields in a row is not significant. We will therefore rearrange them as convenient, and sometimes write a row as a sequence of more than two elements. For example, each of the following denote the same row:

$$(l_1 : \tau_1, l_2 : \tau_2, l_3 : \tau_3, \rho)$$
$$(((l_1 : \tau_1, l_2 : \tau_2), l_3 : \tau_3), \rho)$$
$$((l_1 : \tau_1, l_2 : \tau_2), (l_3 : \tau_3, \rho))$$
$$(\rho, l_1 : \tau_1, l_2 : \tau_2, l_3 : \tau_3)$$
$$(\rho, (l_2 : \tau_2, l_3 : \tau_3), l_1 : \tau_1)$$

π ranges over predicates. P and Q range over sets of predicates. X and Y range over sets of type and row variables. The notation $A \backslash x$ means A with any occurrence of $x : \tau$ removed, and $A|_X$ is A restricted to the set of variables in X.

Qualified types are:

$$\eta \quad ::= \quad P \Rightarrow \tau$$

We use η to denote qualified types, rather than ρ, as in [26], because ρ is already occupied by row variables in our system.

Type schemes are:

$$\sigma \quad ::= \quad \forall\, X \,.\, \eta$$

We abbreviate a type scheme with no quantified variables $\forall\, \varnothing \,.\, \eta$ as η, and the qualified type with no predicates $\varnothing \Rightarrow \tau$ as τ.

Definition: We use the notation $\mathrm{TRV}(K)$ to denote the set of free type and row variables in K, which may be a type, qualified type, type scheme, row, or environment. The notion of free variable is defined in the usual way.

4.3 Predicates

4.3.1 Disjointness

The disjointness predicate, written $\mathit{lset}_1 \# \mathit{lset}_2$, is the underlying predicate used in ensuring that rows are well-formed. This predicate is used in the types of all interesting record operations. The remaining predicates are supporting players at best.

4.3.2 Type, row and labelset equality

It is possible for two rows to be equal, yet not have a most general unifier. There are numerous substitutions for the row variables in rows (ρ_1, ρ_2) and (ρ_3, ρ_4) which make them equal, yet there is no single most general unifier. We cannot simply unify ρ_1 with ρ_3, and ρ_2 with ρ_4, because this does not capture all cases. For example, the substitution

$$[\rho_1 \mapsto \mathtt{a} : \tau,\ \rho_2 \mapsto \mathtt{b} : \tau,\ \rho_3 \mapsto (\mathtt{a} : \tau, \mathtt{b} : \tau),\ \rho_4 \mapsto \varnothing]$$

will also unify the two rows. This situation suggests the need for a row equality predicate, which allows us to substitute one row for another that it is equal to. A motivating example is:

```
    fun f (a, b) = (a & b) \ l
```

The type of `a & b` must be of both of the forms:

- $\{l : \tau, \mathit{row}\}$, where $\mathit{row} \# l$
- $\{\mathit{row}_1, \mathit{row}_2\}$, where $\{\mathit{row}_1\}$ and $\{\mathit{row}_2\}$ are the types of a and b, respectively.

In the absence of the row equality predicate, we can choose either of the following two types for f:

- $\forall \alpha, \rho_1, \rho_2 \,.\, \rho_1 \# \rho_2, \rho_1 \# l, \rho_2 \# l \Rightarrow (\{l : \alpha, \rho_1\} \times \{\rho_2\}) \rightarrow \{\rho_1, \rho_2\}$
- $\forall \alpha, \rho_1, \rho_2 \,.\, \rho_1 \# \rho_2, \rho_1 \# l, \rho_2 \# l \Rightarrow (\{\rho_1\} \times \{l : \alpha, \rho_2\}) \rightarrow \{\rho_1, \rho_2\}$

Unfortunately, there is no single most general type. The ability to specify a row equality predicate allows us to assign this function the most general type:

$$\forall \alpha, \rho, \rho_1, \rho_2 . (\rho_1, \rho_2) = (l : \alpha, \rho) \Rightarrow (\{\rho_1\} \times \{\rho_2\}) \rightarrow \{\rho\}$$

The two previous type schemes are both instances of this one. We can obtain each of the two type schemes above by applying the two substitutions:

$$[\rho_1 \mapsto (l : \tau, \rho_1'),\ \rho_2 \mapsto \rho_2',\ \rho \mapsto (\rho_1', \rho_2')] \quad \text{and}$$
$$[\rho_1 \mapsto \rho_1',\ \rho_2 \mapsto (l : \tau, \rho_2'),\ \rho \mapsto (\rho_1', \rho_2')]$$

to the most general type scheme and renaming the row variables.

Note that the equality predicate cannot always be reduced, e.g., via unification. We know that the concatenation of the two rows has a field labeled l, but we do not know which of the two rows is responsible for contributing it to the concatenation. The row equality predicate captures this information.

We also have the labelset equality predicate, written $\|$, because it is also convenient to have the same substitutability for labelsets. This allows us to define one of the asymmetric versions of record concatenation.

4.3.3 Coercion

We do not have subsumption, in which coercion is implicit, but we do assume the existence of a predefined generic thinning operation:

$$\texttt{thin} : \forall \rho_1, \rho_2 . \rho_1 \blacktriangleright \rho_2 \Rightarrow \{\rho_1\} \rightarrow \{\rho_2\}$$

which can be used explicitly in the cases where subtyping is desired.

4.4 Well-formedness

Unlike the case for core ML, the question of whether a type is well-formed cannot be decided purely on the basis of syntax, because of the complication caused by rows. Specifically, the elements of a row must be disjoint in order for the row to be well-formed. In the case of fixed fields, this can be determined syntactically. For example,

$$(l_1 : \tau,\ l_2 : \tau)$$

is a well-formed row, but

$$(l_1 : \tau,\ l_2 : \tau,\ l_1 : \tau)$$

is not.

Because we cannot syntactically determine if two row variables are disjoint, the presence of row variables in rows means that determining the well-formedness of the rows is not possible by syntactic means alone.

We therefore need inference rules that tell us when rows are well-formed, and because other constructs may contain rows, when these other constructs are well-formed as well. These are given in Figures 4.4, 4.5, 4.6, and 4.7. The judgements

$$P \vdash \tau\ \mathtt{TYPE}$$

$$P \vdash \mathit{row}\ \mathtt{ROW}$$

$$P \vdash \mathit{lset}\ \mathtt{LSET}$$

state that τ, *row*, and *lset* is a well-formed type, row, and labelset, respectively, in the context P, and

$$\vdash P\ \mathtt{PSET}$$

states that P is a well-formed row predicate set. In the sequel, we require that all types, rows, labelsets and row predicate sets occurring in the premises of inference rules be well-formed without stating this explicitly. It will be possible to prove the well-formedness of any types, etc., occurring in the conclusion of the rules from the premises and the well-formedness of the entities therein.

The essential property of a well-formed predicate set is given by the following proposition:

Proposition 4.4.1. *For any well-formed predicate set P,*

1. *if $(\mathit{row}_1, \mathit{row}_2)$ occurs in P, then $P \Vdash \mathit{row}_1 \mathrel{\#} \mathit{row}_2$.*

2. *if $(\mathit{lset}_1, \mathit{lset}_2)$ occurs in P, then $P \Vdash \mathit{lset}_1 \mathrel{\#} \mathit{lset}_2$.*

$$(\textbf{WF-type-tyconst}) \qquad \vdash c\ \texttt{TYPE}$$

$$(\textbf{WF-type-tyvar}) \qquad \vdash \alpha\ \texttt{TYPE}$$

$$(\textbf{WF-type-fun}) \qquad \frac{P \vdash \tau_1\ \texttt{TYPE} \quad P \vdash \tau_2\ \texttt{TYPE}}{P \vdash \tau_1 \to \tau_2\ \texttt{TYPE}}$$

$$(\textbf{WF-type-rec}) \qquad \frac{P \vdash \mathit{row}\ \texttt{ROW}}{P \vdash \{\mathit{row}\}\ \texttt{TYPE}}$$

$$(\textbf{WF-type-qtype}) \qquad \frac{\vdash \pi\ \texttt{PSET} \quad P \vdash \eta\ \texttt{TYPE}}{P \vdash \pi \Rightarrow \eta\ \texttt{TYPE}}$$

$$(\textbf{WF-type-tyscheme}) \qquad \frac{P \vdash \eta\ \texttt{TYPE}}{P \vdash \forall X\,.\,\eta\ \texttt{TYPE}}$$

Figure 4.4: Well-formed types, qualified types and type schemes

$$(\textbf{WF-row-}\varnothing) \qquad \vdash \varnothing\ \texttt{ROW}$$

$$(\textbf{WF-row-rowvar}) \qquad \vdash \rho\ \texttt{ROW}$$

$$(\textbf{WF-row-field}) \qquad \frac{P \vdash \tau\ \texttt{TYPE}}{P \vdash l : \tau\ \texttt{ROW}}$$

$$(\textbf{WF-row-union}) \qquad \frac{P \vdash \mathit{row}_1\ \texttt{ROW} \quad P \vdash \mathit{row}_2\ \texttt{ROW} \quad P \Vdash \mathit{row}_1 \# \mathit{row}_2}{P \vdash (\mathit{row}_1, \mathit{row}_2)\ \texttt{ROW}}$$

Figure 4.5: Well-formed rows

$$(\textbf{WF-lset-}\varnothing) \qquad \vdash \varnothing\ \texttt{LSET}$$

$$(\textbf{WF-lset-label}) \qquad \vdash l\ \texttt{LSET}$$

$$(\textbf{WF-lset-rowvar}) \qquad P \vdash \rho\ \texttt{LSET}$$

$$(\textbf{WF-lset-union}) \qquad \frac{P \vdash \mathit{lset}_1\ \texttt{LSET} \quad P \vdash \mathit{lset}_2\ \texttt{LSET} \quad P \Vdash \mathit{lset}_1 \mathrel{\#} \mathit{lset}_2}{P \vdash (\mathit{lset}_1, \mathit{lset}_2)\ \texttt{LSET}}$$

Figure 4.6: Well-formed labelsets

$$(\textbf{WF-pset-}\varnothing) \qquad \vdash \varnothing\ \texttt{PSET}$$

$$(\textbf{WF-pset-=}) \qquad \frac{\vdash P\ \texttt{PSET} \quad P \vdash \mathit{row}_1\ \texttt{ROW} \quad P \vdash \mathit{row}_2\ \texttt{ROW}}{\vdash P \cup \{\mathit{row}_1 = \mathit{row}_2\}\ \texttt{PSET}}$$

$$(\textbf{WF-pset-}\blacktriangleright) \qquad \frac{\vdash P\ \texttt{PSET} \quad P \vdash \mathit{row}_1\ \texttt{ROW} \quad P \vdash \mathit{row}_2\ \texttt{ROW}}{\vdash P \cup \{\mathit{row}_1 \blacktriangleright \mathit{row}_2\}\ \texttt{PSET}}$$

$$(\textbf{WF-pset-\#}) \qquad \frac{\vdash P\ \texttt{PSET} \quad P \vdash \mathit{lset}_1\ \texttt{LSET} \quad P \vdash \mathit{lset}_2\ \texttt{LSET}}{\vdash P \cup \{\mathit{lset}_1 \mathrel{\#} \mathit{lset}_2\}\ \texttt{PSET}}$$

$$(\textbf{WF-pset-}\|) \qquad \frac{\vdash P\ \texttt{PSET} \quad P \vdash \mathit{lset}_1\ \texttt{LSET} \quad P \vdash \mathit{lset}_2\ \texttt{LSET}}{\vdash P \cup \{\mathit{lset}_1 \parallel \mathit{lset}_2\}\ \texttt{PSET}}$$

Figure 4.7: Well-formed row predicate sets

$$
\begin{array}{lcl}
c\,V & \longrightarrow & \delta(c, V) \\
(\lambda x.E)\,V & \longrightarrow & [x \mapsto V]E \\
\textbf{let } x = V \textbf{ in } E & \longrightarrow & [x \mapsto V]E \\
\{l_1 = V_1, \ldots, l = V, \ldots, l_n = V_n\}.l & \longrightarrow & V \\
\{l_1 = V_1, \ldots, l_n = V_n\} \backslash l_i & \longrightarrow & \begin{array}{l}\{l_1 = V_1, \ldots, l_{i-1} = V_{i-1}, \\ l_{i+1} = V_{i+1}, \ldots, l_n = V_n\}\end{array}
\end{array}
$$

Figure 4.8: Notions of reduction for $\lambda^{\#,\|,=}$

$$
\begin{array}{rcl}
\mathcal{E} & ::= & [] \\
& | & \mathcal{E}\,e \\
& | & V\,\mathcal{E} \\
& | & \textbf{let } x = \mathcal{E} \textbf{ in } E \\
& | & \mathcal{E}.l \\
& | & \mathcal{E} \backslash l \\
& | & \{l_1 = V_1, \ldots, l_{i-1} = V_{i-1}, l_i = \mathcal{E}, l_{i+1} = E_{i+1}, \ldots, l_n = E_n\}
\end{array}
$$

Figure 4.9: Evaluation context for $\lambda^{\#,\|,=}$

4.5 Semantics

We define the semantics for $\lambda^{\#,\|,=}$ using the same method as in Chapter 2. The differences are in the notions of reduction, which are given in Figure 4.8, and the evaluation context, which is defined in Figure 4.9.

We first need to establish two properties of type derivations:

Lemma 4.5.1 (Type Substitution). *If $P|A \vdash E : \tau$, and θ is an arbitrary sub-*

stitution, then $\theta P|\theta A \vdash E : \theta\tau$.

Lemma 4.5.2 (Value Substitution). *If* $P|A, x : (\forall\overrightarrow{\alpha}.Q \Rightarrow \tau) \vdash E : \tau'$*, when* $x \notin \mathcal{D}(A)$*,* $P|A \vdash V : \tau$ *and* $\overrightarrow{\alpha} \cap (\text{TRV}(A) \cup \text{TRV}(P)) = \varnothing$*, then* $P|A \vdash [x \mapsto V]E : \tau'$.

We use the previous two Lemmas in the proof of the Subject Reduction Lemma:

Lemma 4.5.3 (Subject reduction). *If* $P|A \vdash E : \tau$*, and* $E \longrightarrow E'$*, then* $P|A \vdash E' : \tau$.

Subject reduction is not sufficient to establish type soundness. To do that, we need to prove that we cannot assign a type for expressions that are "wrong". These are expressions whose evaluation gets stuck; that is, expressions E for which there is no E' such that $E \longmapsto E'$. Because this property is undecidable in general, we will conservatively approximate it with the set of expressions containing a subexpression $c\,V$ for which $\delta(c, V)$ is not defined. We say that such expressions are *faulty*.

The proof of type soundness rests on another property, stated in the following lemma:

Lemma 4.5.4 (Uniform Evaluation). *For closed* E*, if there is no* E' *such that* $E \longmapsto\!\!\!\!\rightarrow E'$ *and* E' *is faulty, then either* E *diverges, or* $E \longmapsto\!\!\!\!\rightarrow V$.

We can now state the main theorem of this chapter:

Theorem 4.5.1 (Syntactic Type Soundness). *If* $\vdash E : \tau$*, then* $E \longmapsto\!\!\!\!\rightarrow V$ *and* $\vdash V : \tau$.

4.6 Type System Mechanics

4.6.1 Type rules

There is a complication in developing a syntax-directed set of rules, caused by the proposed rule

$$(\textbf{eqtype}) \quad \frac{P|A \vdash E : \tau' \quad P \Vdash \tau' = \tau}{P|A \vdash E : \tau}$$

It is not possible to eliminate this rule in a trivial way. We cannot simply fold this into one of the other rules. We can fold it into *every* other inference rule, but this would make the inference rules messy. An equivalent solution is described in [29:section 10.5] to deal with the subsumption rule. Rather than complicating the inference rules, the expression is decorated with calls to a function which has the same effect as the rule. In Jones' case, each subexpression e' of e is replaced by the call $\textit{coerce}\, e'$. The type of *coerce* is $\forall \alpha, \beta \,.\, \alpha \subseteq \beta \Rightarrow \alpha \rightarrow \beta$. Type inference is then performed on the preprocessed term. In our case, we define the function *equerse*, of type $\forall \alpha, \beta \,.\, \alpha = \beta \Rightarrow \alpha \rightarrow \beta$. The preprocessing step is described in Figure 4.11. In the remainder of this work, we assume that we are always dealing with preprocessed terms.

The type rules given in Figure 4.12 are a superset of the syntax-directed rules in Jones [26]. The additional rules involve records. In fact, the preliminary development of the type system parallels the one in [26]. There are no type rules for record operations such as concatenation, because they are assumed to be part of the default environment, given in Figure 4.10.

Operation	Type
$\_ . l$	$\forall \rho, \alpha . \rho \# l \Rightarrow \{\rho, l : \alpha\} \rightarrow \alpha$
$\_ \backslash l$	$\forall \rho, \alpha . \rho \# l \Rightarrow \{\rho, l : \alpha\} \rightarrow \{\rho\}$
$\_ \& \_$	$\forall \rho_1, \rho_2 . \rho_1 \# \rho_2 \Rightarrow \{\rho_1\} \times \{\rho_2\} \rightarrow \{\rho_1, \rho_2\}$

Figure 4.10: Default environment

$$\begin{array}{lcl}
EQ[x] & = & EQ[\mathit{equerse}\, x] \\
EQ[E_1 E_2] & = & EQ[E_1] EQ[E_2] \\
EQ[\lambda x.E] & = & \lambda x.EQ[E] \\
EQ[\mathbf{let}\ x = E_1\ \mathbf{in}\ E_2] & = & \mathbf{let}\ x = EQ[E_1]\ \mathbf{in}\ EQ[E_2] \\
EQ[\{l = E_1, \ldots, l = E_n\}] & = & \{l = EQ[E_1], \ldots, l = EQ[E_n]\} \\
EQ[E.l] & = & EQ[E].l \\
EQ[E \backslash l] & = & EQ[E] \backslash l
\end{array}$$

Figure 4.11: Preprocessing step to make the **eqtype** rule explicit

These rules use the *Gen* function, which is defined:

$$Gen(A, \eta) = \forall (\mathrm{TRV}(\eta) - \mathrm{TRV}(A)).\eta$$

Theorem 4.6.1 (Preservation of Well-Formedness). *The type rules preserve well-formedness. That is, for every type rule which concludes $P|A \vdash E : \tau$, if any predicates, type assumptions, and types appearing in the premises or introduced in the conclusion are well-formed, then P, $A|_X$, and τ are well-formed, where X is the set of free variables in E.*

$$(\mathbf{var}) \qquad \frac{x : \sigma \in A \quad (P \Rightarrow \tau) \leqslant \sigma}{P | A \vdash x : \tau}$$

$$(\mathbf{app}) \qquad \frac{P | A \vdash E_1 : \tau' \to \tau \quad P | A \vdash E_2 : \tau'}{P | A \vdash (E_1\, E_2) : \tau}$$

$$(\mathbf{abs}) \qquad \frac{P | A \backslash x, x : \tau_1 \vdash E : \tau_2}{P | A \vdash (\lambda x.E) : \tau_1 \to \tau_2}$$

$$(\mathbf{let}) \qquad \frac{P | A \vdash E_1 : \tau' \quad P' | A \backslash x, x : \sigma \vdash E_2 : \tau}{P' | A \vdash (\mathbf{let}\ x = E_1\ \mathbf{in}\ E_2) : \tau} \quad \sigma = Gen(A, P \Rightarrow \tau')$$

$$(\mathbf{record}) \qquad \frac{P | A \vdash E_i : \tau_i \quad \forall i \in 1..n}{P | A \vdash \{l_1 = E_1, \ldots, l_n = E_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}} \quad l_1, \ldots, l_n \text{ distinct}$$

$$(\mathbf{sel}) \qquad \frac{P | A \vdash E : \{l : \tau, row\} \quad P \Vdash row \# l}{P | A \vdash E.l : \tau}$$

$$(\mathbf{extract}) \qquad \frac{P | A \vdash E : \{l : \tau, row\} \quad P \Vdash row \# l}{P | A \vdash E \backslash l : \{row\}}$$

Figure 4.12: Syntax-directed inference rules

$$(\textbf{var})\qquad \frac{(x : \forall X\,.\,P \Rightarrow \tau) \in A}{[X \mapsto Y]P | A \vdash^{W} x : [X \mapsto Y]\tau}\ Y \text{ new}$$

$$(\textbf{app})\qquad \frac{P|\theta_1 A \vdash^{W} E : \tau \quad Q|\theta_2\theta_1 A \vdash^{W} F : \tau' \quad (\theta, R) = \mathit{Simp}[\![\theta_2\tau = \tau' \to \alpha]\!]}{(R, \theta(\theta_2 P, Q))|(\theta\theta_2\theta_1 A) \vdash^{W} (E\ F) : \theta\alpha}\ \alpha \text{ new}$$

$$(\textbf{abs})\qquad \frac{P|\theta(A\backslash x, x : \alpha) \vdash^{W} E : \tau}{P|\theta A \vdash^{W} (\lambda x.E) : \theta\alpha \to \tau}\ \alpha \text{ new}$$

$$(\textbf{let})\qquad \frac{P|\theta A \vdash^{W} E : \tau' \quad Q|\theta'(\theta A\backslash x, x : \sigma) \vdash^{W} F : \tau \quad \sigma = \mathit{Gen}(\theta A, P \Rightarrow \tau')}{Q|\theta'\theta A \vdash^{W} (\texttt{let}\ x = E\ \texttt{in}\ F) : \tau}$$

$$(\textbf{record})\qquad \frac{P_i|(\theta_i \ldots \theta_1 A) \vdash^{W} E_i : \tau_i \quad \forall i \in 1..n}{Q|(\theta_n \ldots \theta_1 A) \vdash^{W} \{l_1 = E_1, \ldots, l_n = E_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}$$

$$\text{where } Q = P_n, \theta_n(P_{n-1}, \theta_{n-1}(\ldots, \theta_1 P_1)\ldots)$$

$$(\textbf{sel})\qquad \frac{P|\theta A \vdash^{W} E : \tau' \quad (\theta', R) = \mathit{Simp}[\![\tau' = \{l : \alpha, \rho\}]\!]}{(R, \theta'(\rho \# l, P))|(\theta'\theta A) \vdash^{W} E.l : \theta'\alpha}\ \alpha, \rho \text{ new}$$

$$(\textbf{extract})\qquad \frac{P|\theta A \vdash^{W} E : \tau' \quad (\theta', R) = \mathit{Simp}[\![\tau' = \{l : \alpha, \rho\}]\!]}{(R, \theta'(\rho \# l, P))|(\theta'\theta A) \vdash^{W} E\backslash l : \{\theta'\rho\}}\ \alpha, \rho \text{ new}$$

Figure 4.13: Type inference algorithm

4.7 Type Inference Algorithm

The rules in Figure 4.13 describe a type inference algorithm in the style of [46], which is an adaptation of the type inference algorithm found in [26]. The rules describe an attribute grammar, where the environment A and the expression to be typed are inherited attributes, while the rest, i.e., substitutions, predicate sets and types, are synthesized attributes. Figure 4.14 gives a more traditional presentation of the algorithm.

The most common type inference question to be asked is: given a set of predicates P, environment A, and an expression E, does there exist a type τ such that

$$P|A \vdash E : \tau$$

This is the *principal type* question.

For a whole program analysis, P will be empty for our system. If we have modules that we wish to compile separately which have parameters however, it may be the case that the parameters impose a predicate environment in which to elaborate the module. In such a case, P would not be empty.

We may also wish to ask if an expression has a *principal typing*. This asks if, given an expression E, do there exist P, A, and τ such that

$$P|A \vdash E : \tau$$

For satisfiability, discussed in section 5.6, we ask an intermediate question. Given an environment A and expression E, do there exist P and τ such that

$$P|A \vdash E : \tau$$

$$
\begin{array}{lll}
\mathbf{W}\llbracket E\rrbracket A & = & (P,\,\theta,\,\tau) \\
\hline
\mathbf{W}\llbracket x\rrbracket A & = & ([X \mapsto Y]P,\,[\,],\,[X \mapsto Y]\tau) \\
 & & \text{where}\quad (x : \forall X\,.\,P \Rightarrow \tau) \in A \\
 & & \phantom{\text{where}\quad} Y \text{ new} \\
\mathbf{W}\llbracket EF\rrbracket A & = & ((R, \theta(\theta_2 P, Q)),\,\theta\theta_2\theta_1,\,\theta\alpha) \\
 & & \text{where}\quad (P,\,\theta_1,\,\tau) = \mathbf{W}\llbracket E\rrbracket A \\
 & & \phantom{\text{where}\quad} (Q,\,\theta_2,\,\tau') = \mathbf{W}\llbracket F\rrbracket(\theta_1 A) \\
 & & \phantom{\text{where}\quad} \alpha \text{ new} \\
 & & \phantom{\text{where}\quad} (\theta, R) = \mathit{Simp}\llbracket\theta_2\tau = \tau' \to \alpha\rrbracket \\
\mathbf{W}\llbracket \lambda x.E\rrbracket A & = & (P,\,\theta,\,\theta\alpha \to \tau) \\
 & & \text{where}\quad (P,\,\theta,\,\tau) = \mathbf{W}\llbracket E\rrbracket(A\backslash x, x : \alpha) \\
 & & \phantom{\text{where}\quad} \alpha \text{ new} \\
\mathbf{W}\llbracket \texttt{let}\ x = E\ \texttt{in}\ F\rrbracket A & = & (Q,\,\theta'\theta,\,\tau) \\
 & & \text{where}\quad (P,\,\theta,\,\tau') = \mathbf{W}\llbracket E\rrbracket A \\
 & & \phantom{\text{where}\quad} (Q,\,\theta',\,\tau) = \mathbf{W}\llbracket F\rrbracket(\theta A\backslash x, x : \sigma) \\
 & & \phantom{\text{where}\quad} \sigma = \mathit{Gen}(\theta A, P \Rightarrow \tau') \\
\mathbf{W}\llbracket \{l_1 = E_1, \ldots, l_n = E_n\}\rrbracket A & = & (Q,\,\theta_n \ldots \theta_1,\,\{l_1 : \tau_1, \ldots, l_n : \tau_n\}) \\
 & & \text{where}\quad (P_i,\,\theta_i,\,\tau_i) = \mathbf{W}\llbracket E_i\rrbracket(\theta_i \ldots \theta_1 A) \\
 & & \phantom{\text{where}\quad}\qquad \text{forall } i \in 1..n \\
 & & \phantom{\text{where}\quad} Q = P_n, \theta_n(P_{n-1}, \theta_{n-1}(\ldots, \theta_1 P_1)\ldots) \\
\mathbf{W}\llbracket E.l\rrbracket A & = & ((R, \theta'(\rho \# l, P)),\,\theta'\theta,\,\theta'\alpha) \\
 & & \text{where}\quad (P,\,\theta,\,\tau') = \mathbf{W}\llbracket E\rrbracket \\
 & & \phantom{\text{where}\quad} (\theta', R) = \mathit{Simp}\llbracket\tau' = \{l : \alpha, \rho\}\rrbracket \\
 & & \phantom{\text{where}\quad} \alpha, \rho \text{ new} \\
\mathbf{W}\llbracket E\backslash l\rrbracket A & = & ((R, \theta'(\rho \# l, P)),\,\theta'\theta,\,\{\theta'\rho\}) \\
 & & \text{where}\quad (P,\,\theta,\,\tau') = \mathbf{W}\llbracket E\rrbracket \\
 & & \phantom{\text{where}\quad} (\theta', R) = \mathit{Simp}\llbracket\tau' = \{l : \alpha, \rho\}\rrbracket \\
 & & \phantom{\text{where}\quad} \alpha, \rho \text{ new}
\end{array}
$$

Figure 4.14: Type inference algorithm (traditional style)

In general, we need to be able to determine if $P \Vdash Q$ for arbitrary predicates. This is traditionally defined in terms of inference rules, and an algorithm is provided which is proven to yield the same results as the system of inference rules. We provide a set of inference rules in section 5.4. However, it is not clear that these inference rules are complete. That is, it may be the case that π holds whenever P does, but there is no derivation of

$$P \Vdash \pi$$

Therefore, in section 5.5, we describe an alternative formulation of the entailment relation which is not based on rules of inference and is complete.

4.8 Implied predicates

It is possible to use the information embedded in a type to reduce the number of predicates that need to be explicitly given. For example, consider the qualified type

$$\rho_1 \# \rho_2 \Rightarrow (\{\rho_1\} \times \{\rho_2\}) \to \{\rho_1, \rho_2\}$$

Using the well-formedness criteria for types, we can infer that $\rho_1 \# \rho_2$ must be the case in order for the result type of the function, $\{\rho_1, \rho_2\}$, to be well-formed. We can use this fact to rewrite the qualified type above as simply

$$(\{\rho_1\} \times \{\rho_2\}) \to \{\rho_1, \rho_2\}$$

The type is abbreviated only for display purposes; this is a shorthand for the fully qualified type, which can be recovered by adding any predicates that are implied by the well-formedness criteria. We will make extensive use of this observation to simplify the presentation of types.

Chapter 5

Predicates

In this chapter, we discuss *entailment* of sets of predicates, which we denote by $P \Vdash Q$. Informally, entailment means that any substitution that makes the all of the predicates in P true makes all predicates in Q true as well. There are various problems that come up in determining entailment, and we discuss how they can be overcome. In addition, we discuss *satisfiability* of sets of predicates. Informally, a set of predicates P is satisfiable if there exists a substitution which makes all of the predicates in P true.

5.1 Semantics

Well-formed rows are finite maps from labels to types, or equivalently, sets of (l, τ) pairs such that there is at most one pair (l, τ) in a well-formed row for any given label l. This implies that when we concatenate two rows, we have the implicit constraint that the two rows have no labels in common. It will be convenient to make this constraint explicit; we will therefore expand the semantic domain of rows to allow such duplicates, and disallow them by the use of explicit constraints. The expanded domain of a row will be a finite multiset of $l \times \tau$ pairs, or, equivalently, a finite multimap of labels to types. A multiset can be viewed as a map from multiset elements to non-negative integers which represent the number of occurrences of the element. Syntactically, a multiset can be written as a sequence, where it is understood that the order of elements does not matter. A multimap from labels to types is the same as a map from labels to multisets of types.

The set operations $\in$, $\cup$, $\cap$ and $\subseteq$ have multiset counterparts, written $\dot{\in}$, $\dot{\cup}$, $\dot{\cap}$ and $\dot{\subseteq}$. We write a multiset just like a set, except that we use the brackets $\{\!\!\{\cdot, \cdot\}\!\!\}$. We commandeer the empty set symbol $\varnothing$ to do double duty and denote the empty multiset as well.

5.1.1 Multiset semantics

If T is a multiset containing n occurrences of x, we say that $T(x) = n$. Similarly, if T is a multimap, then we write $T(x)$ to denote the multiset of elements that T maps x to. Note that each set S can be viewed as a multiset $\dot{S}$ where

$$\begin{aligned} x \in S \quad &\Longrightarrow \quad \dot{S}(x) = 1 \\ x \notin S \quad &\Longrightarrow \quad \dot{S}(x) = 0 \end{aligned}$$

The semantics of the basic multiset operations and relations are given in Figure 5.1.

Like the corresponding set operations, multiset union and intersection are commutative and associative. Multiset union distributes over multiset intersection. However, unlike the corresponding set operation, multiset intersection does not distribute over multiset union. Finally, $A \mathbin{\dot{\cap}} A = A$ for all A, but $B \mathbin{\dot{\cup}} B \dot{\neq} B$ for all non-empty multisets B.

$$\begin{array}{lcl}
x \mathrel{\dot\in} A & \leftrightarrow & A(x) > 0 \\
A = B & \leftrightarrow & \forall x. A(x) = B(x) \\
C = A \mathbin{\dot\cup} B & \leftrightarrow & \forall x. C(x) = A(x) + B(x) \\
C = A \mathbin{\dot\cap} B & \leftrightarrow & \forall x. C(x) = \min(A(x), B(x)) \\
A \mathrel{\dot\subseteq} B & \leftrightarrow & \forall x. A(x) \leqslant B(x)
\end{array}$$

Figure 5.1: Multiset semantics

5.1.2 Interpretations of predicates

The semantics of a syntactic entity depends on its syntactic structure, as well as on the values assign to any free variables that it may contain.

We define the semantics of such an entity as a mapping from the syntax to a semantic formula in the appropriate domain which may contain free variables. We use $[\![S]\!]$ to denote the semantics of entity S. In many cases, we will also provide an *interpretation*, which is a mapping from variables to semantic values in the appropriate domain(s). We use $\mathcal{I}$ to range over interpretations, and we use $\mathcal{I}[\![S]\!]$ to denote the semantics of entity S with respect to interpretation $\mathcal{I}$. This is simply the same as $[\![S]\!]$, except that any free variables in $[\![S]\!]$ are replaced with their image in $\mathcal{I}$.

Figure 5.2 lists the semantic domains of the various syntactic entities in our system.

Because there is some overlap between the syntactic categories of *row* and *lset*, we will subscript the semantic brackets with *ROW* and *LS* to distinguish them when not clear from context. The semantics of the predicates is given in Figure 5.3.

Note that an interpretation maps row variables ρ to a finite map (from labels

Syntactic entity	Domain of $\mathcal{I}[\![$Syntactic entity$]\!]$
lset	finite multiset of labels
row	finite multimap from labels to types
π	truth values
ρ	finite map from labels to types
α	non-recursive types

Figure 5.2: Predicate semantic domains

to types), whereas it maps rows to a finite multimap (from labels to types). This is done because a row can be made up of multiple row elements, each of which can be a map. We need to accommodate the possibility that two or more of these elements may map the same label to a type. We do not consider such rows well-formed, but this is prevented by explicit row disjointness predicates rather than by hidden restrictions on the domain.

5.2 Definitions

lset ranges over sets of labels. We write $\rho_1, \ldots, \rho_n, l_1, \ldots, l_m$, or some permutation thereof, to denote the set $\{l_1, \ldots, l_m\} \cup \mathcal{D}(\rho_1, \ldots, \rho_n)$. In a similar spirit, we will abuse the notation where convenient by specifying a row where a set of labels is expected. The set of labels that is meant in such cases is the domain of the row.

Two rows are equal if they denote the same mapping. $lset_1 \# lset_2$ is true if the sets $lset_1$ and $lset_2$ are disjoint, and $row_1 \blacktriangleright row_2$ is true if the map denoted by row_1 is a superset of the map denoted by row_2.

$$
\begin{array}{lcl}
\mathcal{I}[\![\varnothing]\!]_{LS} & \equiv & \varnothing \\
\mathcal{I}[\![\varnothing]\!]_{ROW} & \equiv & \varnothing \\
\mathcal{I}[\![l]\!]_{LS} & \equiv & \{\!\cdot l \cdot\!\} \\
\mathcal{I}[\![\rho]\!]_{ROW} & \equiv & \begin{cases} \bot & \text{if } \mathcal{I}\rho(l) = \varnothing \\ \tau & \text{if } \mathcal{I}\rho(l) = \{\!\cdot \tau \cdot\!\} \\ * & \text{if } |\mathcal{I}\rho(l)| > 1 \end{cases} \\
\mathcal{I}[\![\rho]\!]_{LS} & \equiv & \mathcal{D}(\mathcal{I}(\rho)) \\
\mathcal{I}[\![l : \tau]\!]_{ROW} & \equiv & \{\!\cdot (l, \tau) \cdot\!\} \\
\mathcal{I}[\![l : \tau]\!]_{LS} & \equiv & \{\!\cdot l \cdot\!\} \\
\mathcal{I}[\![lset_1, lset_2]\!]_{LS} & \equiv & \mathcal{I}[\![lset_1]\!]_{LS} \mathbin{\dot\cup} \mathcal{I}[\![lset_2]\!]_{LS} \\
\mathcal{I}[\![row_1, row_2]\!]_{ROW} & \equiv & \mathcal{I}[\![row_1]\!]_{ROW} \mathbin{\dot\cup} \mathcal{I}[\![row_2]\!]_{ROW} \\
\mathcal{I}[\![lset_1 \# lset_2]\!] & \equiv & \mathcal{I}[\![lset_1]\!]_{LS} \mathbin{\dot\cap} \mathcal{I}[\![lset_2]\!]_{LS} = \varnothing \\
\mathcal{I}[\![lset_1 \parallel lset_2]\!] & \equiv & \mathcal{I}[\![lset_1]\!]_{LS} = \mathcal{I}[\![lset_2]\!]_{LS} \\
\mathcal{I}[\![row_1 \blacktriangleright row_2]\!] & \equiv & \mathcal{I}[\![row_1]\!]_{ROW} \mathbin{\dot\supseteq} \mathcal{I}[\![row_2]\!]_{ROW} \\
\mathcal{I}[\![row_1 = row_2]\!] & \equiv & \mathcal{I}[\![row_1]\!]_{ROW} = \mathcal{I}[\![row_2]\!]_{ROW} \\
\mathcal{I}[\![P]\!] & \equiv & \bigwedge_{\pi \in P} \mathcal{I}[\![\pi]\!]
\end{array}
$$

Figure 5.3: Row predicate semantics

F and V range over non-empty fixed and variable portions of a row, respectively. That is,

$$\begin{array}{rcl} F & ::= & l : \tau \mid l : \tau, F \\ V & ::= & \rho \mid \rho, V \end{array}$$

R ranges over non-empty rows.

We say that $l \in F$ if $F = l_1 : \tau_1, \ldots, l_n : \tau_n$, and $l = l_i$ for some $1 \leqslant i \leqslant n$.

Definition: $\mathcal{L}(\cdot)$ is the set of labels explicitly occurring in its argument. We define it for row, type, and predicate arguments in Figure 5.4, and extend it in the natural way for sets of predicates.

The empty predicate set will sometimes be written as `OK`, and a simplified unsatisfiable predicate will be written as `Fail`. `Fail` will not appear in programs; it is used primarily in the predicate algorithms.

5.3 Functional dependencies

In a traditional qualified type system, a type scheme $\forall X.P \Rightarrow \tau$ is *unambiguous* if all the variables in X which appear free in P also appear free in τ. That is, $\mathrm{TRV}(P) \cap X \subseteq \mathrm{TRV}(\tau) \cap X$. Ambiguous type schemes are problematic because when instantiated, we do not have enough information to pick unique types for the type variables in $X \cap (\mathrm{TRV}(P) - \mathrm{TRV}(\tau))$. This may lead to a lack of coherence, which means that the same expression can have differing semantics, depending on how the ambiguous type variables are instantiated. However, this is not necessarily always the case. There are cases where the traditional notion of ambiguity does

$$
\begin{aligned}
\mathcal{L}(P) &= \bigcup_{\pi \in P} \mathcal{L}(\pi) \\
\mathcal{L}(\mathit{lset}_1 \mathbin{\#} \mathit{lset}_2) &= \mathcal{L}(\mathit{lset}_1) \cup \mathcal{L}(\mathit{lset}_2) \\
\mathcal{L}(\mathit{lset}_1 \parallel \mathit{lset}_2) &= \mathcal{L}(\mathit{lset}_1) \cup \mathcal{L}(\mathit{lset}_2) \\
\mathcal{L}(\mathit{row}_1 = \mathit{row}_2) &= \mathcal{L}(\mathit{row}_1) \cup \mathcal{L}(\mathit{row}_2) \\
\mathcal{L}((\mathit{le}_1, \ldots, \mathit{le}_n)) &= \bigcup_{i \in 1..n} \mathcal{L}(\mathit{le}_i) \\
\mathcal{L}((\mathit{re}_1, \ldots, \mathit{re}_n)) &= \bigcup_{i \in 1..n} \mathcal{L}(\mathit{re}_i) \\
\mathcal{L}(l) &= \{l\} \\
\mathcal{L}(l : \tau) &= \{l\} \cup \mathcal{L}(\tau) \\
\mathcal{L}(\alpha) &= \varnothing \\
\mathcal{L}(\tau_1 \to \tau_2) &= \mathcal{L}(\tau_1) \cup \mathcal{L}(\tau_2) \\
\mathcal{L}(\{\mathit{row}\}) &= \mathcal{L}(\mathit{row})
\end{aligned}
$$

Figure 5.4: Explicit labels

not lead to incoherence. For example, consider the type scheme:

$$\forall \rho, \rho'.(l : \texttt{int}) = (\rho, \rho') \Rightarrow \{\rho\} \to \texttt{int}$$

This would be considered ambiguous, because the quantified variable ρ' appears in the predicate but not in the body of the type. However, any instantiation of ρ uniquely determines ρ'.

In [27], this idea was developed for the case of multi-parameter type classes [16] in Haskell [45]. In [15], Duck et al prove that, under certain conditions, functional dependencies allow for sound and decidable type inference. We adapt the ideas in these papers for use in our system. The major difference is that whereas functional dependencies are imposed by the user in Haskell, they occur "naturally" and can be inferred automatically in $\lambda^{\#,\|,=}$.

We will use the notation in [27]; if the set of variables X uniquely determines the disjoint set of variables Y, we express this by writing $X \rightsquigarrow Y$. For the above example, we would say that $\rho \rightsquigarrow \rho'$ (and also that $\rho' \rightsquigarrow \rho$). In our system, we can infer a set of functional dependencies for each instance of a row equality predicate. Given a row equality predicate

$$(\rho_1, \ldots, \rho_n, l_1 : \tau_1, \ldots, l_m : \tau_m) = (\rho'_1, \ldots, \rho'_{n'}, l'_1 : \tau'_1, \ldots, l'_{m'} : \tau'_{m'})$$

let

$$\begin{array}{rcl} S & = & \{\rho_1, \ldots, \rho_n\} \\ T & = & \{\rho'_1, \ldots, \rho'_{n'}\} \\ S_i & = & \{\rho_1, \ldots, \rho_{i-1}, \rho_{i+1}, \rho_n\} \\ T_i & = & \{\rho'_1, \ldots, \rho'_{i-1}, \rho'_{i+1}, \rho'_{n'}\} \end{array}$$

Then we can infer the set of functional dependencies:

$$\{(S \cup T_i) \rightsquigarrow \rho'_i \mid i \in 1..n'\} \cup \{(S_i \cup T) \rightsquigarrow \rho_i \mid i \in 1..n\}$$

For example, given the predicate

$$(\rho_1, \rho_2, l : \tau) = (\rho_3, \rho_4, \rho_5)$$

we can infer the functional dependencies

$$\{\rho_2, \rho_3, \rho_4, \rho_5\} \rightsquigarrow \rho_1$$
$$\{\rho_1, \rho_3, \rho_4, \rho_5\} \rightsquigarrow \rho_2$$
$$\{\rho_1, \rho_2, \rho_4, \rho_5\} \rightsquigarrow \rho_3$$
$$\{\rho_1, \rho_2, \rho_3, \rho_5\} \rightsquigarrow \rho_4$$
$$\{\rho_1, \rho_2, \rho_3, \rho_4\} \rightsquigarrow \rho_5$$

In short, knowing any four of the row variables, we can immediately determine the fifth.

Let us consider another example: the unreduced predicate $(\rho_1, \rho_2) = (\rho_1, \rho_2)$. If we apply the rules above, we get the dependencies:

$$\{\rho_1, \rho_2\} \rightsquigarrow \rho_2$$
$$\{\rho_1, \rho_2\} \rightsquigarrow \rho_1$$

These dependencies are correct, but trivial. Better results would be obtained by reducing the predicate, which would result in its disappearance, as it is always satisfied, and no functional dependencies would then be generated.

We can now define an improved notion of an unambiguous type scheme. The closure of a set of variables Z with respect to the functional dependencies implied by a predicate set P, written Z_P^+, is the smallest set such that $Z \subseteq Z_P^+$, and, if $X \rightsquigarrow Y$ is implied by P, and $X \subseteq Z_P^+$, then $Y \subseteq Z_P^+$. This is the set of variables that is uniquely determined, either directly or indirectly, by the variables in Z.

A type scheme $\forall X.P \Rightarrow \tau$ is unambiguous if $\mathrm{TRV}(P) \subseteq Z_P^+$, where $Z = \mathrm{TRV}(\tau)$.

The closure of a set of variables with respect to a set of predicates can easily be calculated:

Given a set of predicates, we can determine the functional dependencies as shown above. Let F be a set of function dependencies, and Z be a set of row variables. We can calculate Z_P^+ by the iterative algorithm:

$$\begin{array}{lcl} Z_P^0 & = & Z \\ Z_P^{n+1} & = & Z_P^n \cup \{\rho \mid X \rightsquigarrow \rho \in P, X \subseteq Z_P^n\} \\ Z_P^+ & = & Z_P^n \quad \text{where } Z_P^n = Z_P^{n+1} \end{array}$$

Given a type scheme $\forall X\,.\,P \Rightarrow \tau$, we will call qualified variables in P that

1. do not appear in τ,
2. are uniquely determined by variables in τ, and
3. occur only once in the predicate,

uninteresting. When we display a type scheme, we do not need to name any uninteresting variables, and may choose to display them as _ (or ...) instead.

5.3.1 Row Coercion

The notion of functional dependencies in the previous section gives us a way to eliminate row coercion by rewriting it in terms of more fundamental predicates, namely row equality and labelset disjointness. We can regard the row coercion predicate as a shorthand. The translation is:

$$row_1 \blacktriangleright row_2 \quad \Longrightarrow \quad (row_1 = (row_2, \rho)), (\rho \# row_2)$$

The new row variable ρ appears only in these two predicates. Because it is uniquely determined by the variables in row_1 and row_2, there is no danger of it causing ambiguity when it appears in the predicates of a type scheme.

An example of this translation is the type for the optional arguments example from page 43, where row_O and row_M are rows specifying the optional and mandatory fields, respectively. This type can be rewritten

$$\forall \rho, \rho' \,.\, row_O = (\rho, \rho') \Rightarrow \{row_M, \rho\} \rightarrow \tau$$

The type for the `thin` operation, introduced on page 52, can be rewritten as

$$\forall \rho_1, \rho_2 \,.\, \{\rho_1, \rho_2\} \rightarrow \{\rho_2\}$$

In both cases, we omit the implied disjointness predicate.

In the sequel, we will continue to use the row coercion predicate where convenient. It is understood to be shorthand for the translation above. In fact, it will often be convenient for types to be displayed with the coercion predicate where possible instead of in their "translated" form.

5.4 Inference rules

We provide here our first formulation of the predicate entailment relation, which is based on inference rules. The rules, grouped by predicate type, are given in Figures 5.5, 5.6, and 5.7. In addition, we have the rule:

$$(\textbf{elem}) \quad \frac{\pi \in P}{P \Vdash \pi}$$

Proposition 5.4.1. *Using the predicate inference rules in Figures 5.5, 5.6, and 5.7, together with rule* **elem**, *if* $P \Vdash \pi$, *then* $\models \forall \mathcal{I}.\mathcal{I}[\![P]\!] \rightarrow \mathcal{I}[\![\pi]\!]$, *i.e. the predicate inference rules are sound with respect to the predicate semantics in Figure 5.3.*

$$\textbf{(\#-null)} \qquad P \Vdash \varnothing \# \mathit{lset}$$

$$\textbf{(\#-fields)} \qquad P \Vdash l \# l' \quad l \neq l'$$

$$\textbf{(\#-sym)} \qquad \frac{P \Vdash \mathit{lset}_1 \# \mathit{lset}_2}{P \Vdash \mathit{lset}_2 \# \mathit{lset}_1}$$

$$\textbf{(\#-eq)} \qquad \frac{P \Vdash \mathit{lset}_1 \# \mathit{lset}_2 \quad P \Vdash \mathit{lset}_2 \parallel \mathit{lset}_2'}{P \Vdash \mathit{lset}_1 \# \mathit{lset}_2'}$$

$$\textbf{(\#-compose)} \qquad \frac{P \Vdash \mathit{lset}_1 \# \mathit{lset} \quad P \Vdash \mathit{lset}_2 \# \mathit{lset} \quad P \Vdash \mathit{lset}_1 \# \mathit{lset}_2}{P \Vdash (\mathit{lset}_1, \mathit{lset}_2) \# \mathit{lset}}$$

$$\textbf{(\#-decompose)} \qquad \frac{P \Vdash (\mathit{lset}_1, \mathit{lset}_2) \# \mathit{lset} \quad P \Vdash \mathit{lset}_1 \# \mathit{lset}_2}{P \Vdash \mathit{lset}_1 \# \mathit{lset}}$$

Figure 5.5: Predicate rules for $\#$

.

Proof. A straightforward induction of the structure of predicate derivations. □

Lemma 5.4.1. *Predicate entailment, as defined in Figures 5.5, 5.6, and 5.7, and rule* **elem***, satisfy the closure, monotonicity, and transitivity properties. To reiterate, these properties are:*

Monotonicity: *If $P \supseteq P'$, then $P \Vdash P'$.*

Transitivity: *If $P_1 \Vdash P_2$ and $P_2 \Vdash P_3$, then $P_1 \Vdash P_3$.*

Closure property: *If $P \Vdash P'$, then $\theta P \Vdash \theta P'$ for any substitution θ.*

Proof.

Monotonicity: This follows from the **elem** rule.

(**‖-field**) $$P \Vdash l \parallel l$$

(**‖-var**) $$P \Vdash \rho \parallel \rho$$

(**‖-sym**) $$\frac{P \Vdash \mathit{lset}_1 \parallel \mathit{lset}_2}{P \Vdash \mathit{lset}_2 \parallel \mathit{lset}_1}$$

(**‖-trans**) $$\frac{P \Vdash \mathit{lset}_1 \parallel \mathit{lset}_2 \quad P \Vdash \mathit{lset}_2 \parallel \mathit{lset}_3}{P \Vdash \mathit{lset}_1 \parallel \mathit{lset}_3}$$

(**‖-pd-null**) $$\frac{P \Vdash \mathit{lset} \parallel \mathit{lset}' \quad P \Vdash \mathit{lset} \# \mathit{lset}'}{P \Vdash \mathit{lset} \parallel \varnothing}$$

(**‖-pc**) $$\frac{P \Vdash \mathit{lset}_1 \parallel \mathit{lset}_2 \quad P \Vdash \mathit{lset}_1 = (\mathit{lset}_2, \mathit{lset}_2')}{P \Vdash \mathit{lset}_2' \parallel \varnothing}$$

(**‖-coerce**) $$\frac{P \Vdash \mathit{lset}_1 \parallel (\mathit{lset}_2, \mathit{lset}_1') \quad P \Vdash \mathit{lset}_2 \parallel (\mathit{lset}_1, \mathit{lset}_2')}{P \Vdash \mathit{lset}_1 \parallel \mathit{lset}_2}$$

(**‖-compose**) $$\frac{P \Vdash \mathit{lset}_1 \parallel \mathit{lset}_2 \qquad P \Vdash \mathit{lset} \# \mathit{lset}_1}{P \Vdash (\mathit{lset}, \mathit{lset}_1) \parallel (\mathit{lset}, \mathit{lset}_2)}$$

Figure 5.6: Predicate rules for ‖

.

(eq-field) $$\frac{P \Vdash \tau_1 = \tau_2}{P \Vdash l : \tau_1 = l : \tau_2}$$

(eq-var) $$P \Vdash \rho = \rho$$

(eq-sym) $$\frac{P \Vdash row_1 = row_2}{P \Vdash row_2 = row_1}$$

(eq-trans) $$\frac{P \Vdash row_1 = row_2 \quad P \Vdash row_2 = row_3}{P \Vdash row_1 = row_3}$$

(eq-pd-null) $$\frac{P \Vdash row \parallel row' \quad P \Vdash row \# row'}{P \Vdash row = \varnothing}$$

(eq-pc) $$\frac{P \Vdash row_1 \parallel row_2 \quad P \Vdash row_1 = (row_2, row'_2)}{P \Vdash row'_2 = \varnothing}$$

(eq-coerce) $$\frac{\begin{array}{c} P \Vdash row_1 = (row_2, row'_1) \quad P \Vdash row_2 = (row_1, row'_2) \\ P \Vdash row_2 \# row'_1 \quad P \Vdash row_1 \# row'_2 \end{array}}{P \Vdash row_1 = row_2}$$

(eq-compose) $$\frac{P \Vdash row_1 = row_2 \quad P \Vdash row \# row_1}{P \Vdash (row, row_1) = (row, row_2)}$$

Figure 5.7: Predicate rules for row equality

.

$$\textbf{(eq-type-con)} \qquad P \Vdash t = t$$

$$\textbf{(eq-type-var)} \qquad P \Vdash \alpha = \alpha$$

$$\textbf{(eq-fun)} \qquad \frac{P \Vdash \tau_{11} = \tau_{21} \quad P \Vdash \tau_{12} = \tau_{22}}{P \Vdash (\tau_{11} \to \tau_{12}) = (\tau_{21} \to \tau_{22})}$$

$$\textbf{(eq-tuple)} \qquad \frac{P \Vdash \tau_i = \tau_i' \quad \forall i \in \{1..n\}}{P \Vdash (\tau_1 \times \ldots \times \tau_n) = (\tau_1' \times \ldots \times \tau_n')}$$

$$\textbf{(eq-record)} \qquad \frac{P \Vdash row_1 = row_2}{P \Vdash \{row_1\} = \{row_2\}}$$

Figure 5.8: Predicate rules for type equality

.

Transitivity: The derivation of $P \Vdash Q$ is a forest of derivation trees, one for each predicate π in Q. We can generate a derivation forest for $P_1 \Vdash P_3$ by starting with the derivation forest for $P_2 \Vdash P_3$, and replacing each occurrence of any predicate $\pi \in P_2$ by the derivation tree for $P_1 \Vdash \pi$, which we have from the premises.

Closure property: Satisfaction of the closure property can be seen by induction on the structure of derivations.

□

Note that the premise of the **#-decompose** rule is also the conclusion of the **#-compose** rule, and its conclusion is one of the premises. This causes difficulties for any algorithm, because we cannot always decompose complex queries into simpler ones. Unfortunately, both of these rules are necessary. For example, consider the entailment

$$P \Vdash (\rho_1, \rho_2) \# l_2$$

where

$$P = \{(l_1 \parallel (\rho_1, \rho_3)), (\rho_1 \# \rho_2), (\rho_1 \# \rho_3), (\rho_2 \# l_2)\}$$

A derivation of this is

$$\dfrac{\dfrac{\dfrac{\dfrac{P \Vdash l_2 \# l_1 \quad P \Vdash l_1 \parallel (\rho_1, \rho_3)}{P \Vdash l_2 \# (\rho_1, \rho_3)}\;\textbf{\#-eq}}{P \Vdash (\rho_1, \rho_3) \# l_2}\;\textbf{\#-sym}}{P \Vdash \rho_1 \# l_2}\;\textbf{\#-decompose} \qquad P \Vdash \rho_2 \# l_2 \quad P \Vdash \rho_1 \# \rho_2}{P \Vdash (\rho_1, \rho_2) \# l_2}\;\textbf{\#-compose}$$

It would not be possible to derive this without using both **#-compose** and **#-decompose** rules.

The row and labelset equality rules present another difficulty: there are potentially many derivations of a particular judgement $P \Vdash lset_1 \parallel lset_2$, but except in some trivial cases, the forms of the $lset_1$ and $lset_2$ are of no use in deciding how to guide the search.

Basing an algorithm directly on a set of inference rules appears to be problematic. Instead, we take a different approach in the next section.

5.5 A different approach: checking validity

In seeking to establish the truth of a proposition, there are generally two available methods: syntactic and semantic.

The syntactic (aka proof-theoretic) method involves defining a set of inference rules which can be used to construct proofs of propositions. We need to ensure that the inference rules are sound with respect to the semantics of propositions. That is, we need to ensure that each inference rule respects the semantics of the propositions, which simply means that, for each inference rule, the consequent is guaranteed to be true whenever the antecedents are.

On the other hand, the semantic (aka model-theoretic) approach attempts to determine whether the proposition is true, without regard to a particular set of inference rules. One advantage of the syntactic approach is that in limiting oneself to a weaker system of rules (or logic, as it is generally called), it may be easier to prove termination properties of the proof system. Conversely, limiting oneself to a weaker system also runs the risk that the inference rules are not *complete*. That is, there may be propositions which are true but that the set of rules cannot prove.

In our case, it seems that relying on inference rules does not make termination of the system easy to prove. We therefore try the semantic approach. Rather than using inference rules to determine $P \Vdash \pi$, we will determine if π follows from P by using an algorithm that calculates if $\models \forall \mathcal{I}.\mathcal{I}[\![P]\!] \rightarrow \mathcal{I}[\![\pi]\!]$ in the structure of sets. In other words, our approach will be model- rather than proof-theoretic. The algorithm transforms the entailment judgement into a formula in first order logic which respects the stated semantics of predicates. Specifically,

$$P \Vdash \pi \quad \rightarrow \quad \forall \mathcal{I}.\mathcal{I}[\![P]\!] \rightarrow \mathcal{I}[\![\pi]\!]$$

Along the way, we will decompose and simplify the formula so that the search space is substantially reduced. We write P æ Q to denote that the algorithm described below succeeds when given the arguments P and Q. We read æ as *algorithmically entails.* We will prove that the æ relation is sound and complete with respect to the semantics defined by $\mathcal{I}[\![\cdot]\!]$. In other words,

$$P \;\text{æ}\; \pi \quad \leftrightarrow \quad \models \forall \mathcal{I}.\mathcal{I}[\![P]\!] \rightarrow \mathcal{I}[\![\pi]\!]$$

for all P, π. Because æ is complete, it may find that π follows from P in cases where the inference rules do not.

Earlier, we defined a row to be a partial finite map from labels to types. In the context of the predicate entailment algorithm, we will use a different but equivalent semantics. To review, a row is written as a sequence of fields and/or row variables, and, under an interpretation, a row variable denotes a finite set of fields, which are label $\times$ type pairs, written "label : type". A well-formed row *row*, i.e., one for which $A \vdash \textit{row}\ \texttt{ROW}$, has the additional constraint that no two fields may have the same label. It will be convenient to loosen this constraint, and regard rows as being a total map from labels to *field types.* A field type captures the relevant aspects of a multiset of types. We use $\hat{\tau}$ to range over field types. The domain of field types is given in Figure 5.9. Because we are ultimately only interested in well-formed rows, i.e., rows which map labels to either the empty set or to a singleton type, once we know that a row contains more than one field with the same label, we are not concerned with the exact nature of this multiset. We will therefore denote any such multiset by $*$ (pronounced "clash").

For example, the (non-well-formed) row

$$(l_1 : \tau_1, l_2 : \tau_2, l_1 : \tau_1)$$

can be viewed as mapping l_1 to $*$, l_2 to τ_2, and all other labels to $\bot$.

Element	**Field with given label in the record**
$\perp$	no such field exists
τ	a single field exists
$*$	multiple fields exist

Figure 5.9: Field type domain

The example row above is not well-formed because it contains two fields labeled l_1. The fact that the types are the same does not help. A set containing the two identical types would have a cardinality of one; hence the need to use multisets, and with it, the possibility of confusion when using standard set brackets $\{\ldots\}$, or unfamiliarity when using a non-standard notation. We pick this notation rather than using multisets because our notation is more concise, and also because there is no well-known notation for multi-sets that distinguishes it from sets. Representing a singleton set containing τ as simply τ is actually quite natural in this context. We can regard the domain of field types as the domain of type, lifted, with two distinguished elements: $\perp$ and $*$.

5.5.1 Conversion to field predicates

The semantic interpretation of a row is a finite partial map from labels to types, and predicates are interpreted as relations on these maps. Because such a map is equivalent to a set of pairs with some additional constraints, by the axiom of extensionality [17], we have the following equivalences, one for each of the three sorts of predicates:

$$\begin{array}{rcl}
\mathcal{I}[\![lset_1 \parallel lset_2]\!] & \equiv & \mathcal{I}[\![lset_1]\!] = \mathcal{I}[\![lset_2]\!] \\
& \equiv & \forall l. l \in \mathcal{I}[\![lset_1]\!] \leftrightarrow l \in \mathcal{I}[\![lset_2]\!] \\
\mathcal{I}[\![lset_1 \# lset_2]\!] & \equiv & \mathcal{I}[\![lset_1]\!] \cap \mathcal{I}[\![lset_2]\!] = \varnothing \\
& \equiv & \forall l. \neg(l \in \mathcal{I}[\![lset_1]\!] \wedge l \in \mathcal{I}[\![lset_2]\!]) \\
\mathcal{I}[\![row_1 = row_2]\!] & \equiv & \mathcal{I}[\![row_1]\!] = \mathcal{I}[\![row_2]\!] \\
& \equiv & \forall l. (\exists \tau_1, \tau_2. l : \tau_1 \in \mathcal{I}[\![row_1]\!] \wedge l : \tau_2 \in \mathcal{I}[\![row_2]\!] \wedge \tau_1 = \tau_2) \vee \\
& & \quad ((\forall \tau. l : \tau \notin \mathcal{I}[\![row_1]\!]) \wedge (\forall \tau. l : \tau \notin \mathcal{I}[\![row_2]\!]))
\end{array}$$

For each equivalence, the right side can be viewed as a conjunction of an infinite set of pointwise versions of the semantics of the original predicate. It will be convenient to define a purely syntactic translation of a predicate (and also labelsets, rows, row variables, and sets of predicates) into their pointwise version at a specific label, called a *slice*. $K@l$ denotes this syntactic translation of entity K into its slice at label l. This translation introduces a number of new entities:

- A slice of a labelset is a *label presence*. lp ranges over label presences. The semantics of a label presence is a truth value denoting the presence or absence of a label in a labelset. We use `t` and `f` to denote the values *true* and *false*, respectively.

- A slice of a row is a *field*. fld ranges over fields. The semantics of a field is the field type the label maps to in the row.

- A slice of a row variable is a *field variable*, indexed by row variable and label, denoted by $\boldsymbol{\phi}(\rho, l)$. This field variable is distinct from $\boldsymbol{\phi}(\rho', l')$ iff either ρ and

fe	$::=$	τ	type
	$\mid$	$\perp$	absent field
	$\mid$	ϕ	field variable
lpe	$::=$	$\texttt{t}$	label present
	$\mid$	$\texttt{f}$	label not present
	$\mid$	ϕ	field variable
fld	$::=$	$\varnothing$	empty field
	$\mid$	fe	field element
	$\mid$	fld, fld	disjoint field union
lp	$::=$	$\varnothing$	empty label presence
	$\mid$	lpe	label presence element
	$\mid$	lp, lp	disjoint label presence union

Figure 5.10: Syntax of fields and label presences

ρ' denote distinct variables, or l and l' denote distinct labels. ϕ ranges over field variables[1] We saw in the previous chapter that the meaning of a row variable differs depending on whether it is encountered in a row or labelset context. Analogously, the semantics of a field variable under an interpretation is either a label presence or a field, depending on its context.

We will also need, for reasons which will be revealed in section 5.5.3, page 101,

[1] "phi" sounds like "field". Also, the line through the circle is reminiscent of slicing, n'est-ce pas?

shorthand	canonical form
$\mathit{fe}/\mathtt{f}$	$\bot$
$\mathit{fe}/\bot$	$\bot$
$\mathit{fe}/\mathtt{t}$	fld
fe/τ	fld
fe/ϕ	$\mathit{fld}/\{\phi\}$
$\mathit{fe}/LP/LP'$	$\mathit{fe}/(LP \cup LP')$
$\mathit{fe}/(\mathit{lp}/LP)$	fe/LP

Figure 5.11: Restriction shorthand translation

restricted field elements and *restricted label presence elements*, written lpe/LP and fe/LP, respectively. The LP is a set of label presences. We say that fe/LP restricts fe by LP. We will write fe/fe' as a shorthand for the restriction of fe by the appropriate set of label presences LP. In this case, we say that fe' is the restricting field element. We can also restrict by a label presence element. The meaning of the shorthand is given in Figure 5.11. By convention, ϕ/LP has $\phi \in LP$.

The syntax of fields and label presences is given in Figure 5.10. Slicing is defined in Figure 5.13, and the semantics of slices are given in Figure 5.14. We define the semantics of the translation so that it matches the pointwise semantics. In other words, we want the following equivalence to hold:

$$\mathcal{I}[\![\pi]\!] = \forall l.\mathcal{I}[\![\pi@l]\!]$$

Indeed, this hope is justified by the proposition below:

$$
\begin{array}{lcl}
\mathrm{WFC}[(\mathit{lpe}_1,\ldots,\mathit{lpe}_n)] & = & \{\mathit{lpe}_i \# \mathit{lpe}_j \mid 1 \leqslant i,j \leqslant n, i \neq j\} \\
\mathrm{WFC}[(\mathit{fe}_1,\ldots,\mathit{fe}_n)] & = & \{\mathit{fe}_i \# \mathit{fe}_j \mid 1 \leqslant i,j \leqslant n, i \neq j\} \\
\mathrm{WFC}[\mathit{lp}_1 \# \mathit{lp}_2] & = & \mathrm{WFC}[\mathit{lp}_1] \cup \mathrm{WFC}[\mathit{lp}_2] \\
\mathrm{WFC}[\mathit{lp}_1 \parallel \mathit{lp}_2] & = & \mathrm{WFC}[\mathit{lp}_1] \cup \mathrm{WFC}[\mathit{lp}_2] \\
\mathrm{WFC}[\mathit{fld}_1 = \mathit{fld}_2] & = & \mathrm{WFC}[\mathit{fld}_1] \cup \mathrm{WFC}[\mathit{fld}_2] \\
\mathrm{WFC}[F] & = & \bigcup_{\pi \in F} \mathrm{WFC}[\pi]
\end{array}
$$

Figure 5.12: Well-formedness constraints

Proposition 5.5.1.

$$\mathcal{I}[\![\pi]\!] = \forall l.\mathcal{I}[\![\pi@l]\!]$$

We will also need to show that slicing maintains another property – well-formedness. Well-formedness of a field predicate set is defined analogously to the well-formedness of a row predicate set (on page 53). We define $\mathrm{WFC}[K]$ to be the well-formedness predicates implied by entity K. $\mathrm{WFC}[K]$ is defined in Figure 5.5.1.

A field predicate set F is well-formed if whenever a field $(\mathit{fe}_1,\ldots,\mathit{fe}_n)$ or label presence $(\mathit{lpe}_1,\ldots,\mathit{lpe}_n)$ occurs in F, then $\forall\mathcal{I}.\mathcal{I}[\![F]\!] \rightarrow \mathcal{I}[\![\mathit{fe}_i \# \mathit{fe}_j]\!]$ for all $1 \leqslant i,j \leqslant n$ such that $i \neq j$. Equivalently, F is well-formed if $\forall\mathcal{I}.\mathcal{I}[\![F]\!] \rightarrow \mathcal{I}[\![\mathrm{WFC}[F]]\!]$.

Proposition 5.5.2. *Slice well-formedness preservation If row predicate set P is well-formed, then so is $P@l$, for any label l.*

Because the semantics of fields and label presences is independent of the order of the elements, we are justified in arranging said elements in whatever order is convenient. In addition, we define "field former" and "label presence former" notations, analogous to the "set former" notation, using parentheses to enclose the former rather than set braces. For example,

$$(fe | fe \in \{fe_1, \ldots, fe_n\}, fe \neq \bot)$$

is shorthand for:

$$(fe'_1, \ldots, fe'_m) \quad \text{where } \{fe'_1, \ldots, fe'_m\} = \{fe | \in \{fe_1, \ldots, fe_n\}, fe \neq \bot\}$$

Where necessary, we will distinguish the unsliced form of a predicate from the sliced form. The former will be known as a *row predicate*; the latter as a *field predicate*. We will continue to use the term "predicate" where the form can be determined from context, or when we do not want to distinguish between sliced and unsliced forms.

5.5.2 Restriction to finite set of labels

A crucial step on the road to an algorithm is the ability to restrict the universe of labels in expressions of the form

$$\forall \mathcal{I}. \forall l. \mathcal{I}[\![P@l]\!] \rightarrow \forall l. \mathcal{I}[\![\pi@l]\!]$$

to a finite set. The intuition that allows us to do this is that slices at any two labels which do not explicitly appear in the predicates "act" the same. That is,

Proposition 5.5.3.

$$(\forall \mathcal{I}. \mathcal{I}[\![\pi@l_1]\!]) \leftrightarrow (\forall \mathcal{I}. \mathcal{I}[\![\pi@l_2]\!])$$

if l_1 and l_2 are not in $\mathcal{L}(\pi)$.

Proof. The slices $\pi@l_1$ and $\pi@l_2$ are the same except that, for all ρ, each occurrence of $\boldsymbol{\phi}(\rho, l_1)$ in the former is replaced by $\boldsymbol{\phi}(\rho, l_2)$ in the latter. This can be seen by

$$
\begin{array}{lcl}
P@l & = & \{\pi@l \mid \pi \in P\} \\[6pt]
(lset_1 \mathbin{\#} lset_2)@l & = & (lset_1@l) \mathbin{\#} (lset_2@l) \\
(lset_1 \parallel lset_2)@l & = & (lset_1@l) \parallel (lset_2@l) \\
(row_1 = row_2)@l & = & (row_1@l) = (row_2@l) \\[6pt]
(le_1, \ldots, le_n)@l & = & (le_1@l, \ldots, le_n@l) \\
(re_1, \ldots, re_n)@l & = & (re_1@l, \ldots, re_n@l) \\[6pt]
\rho@l & = & \boldsymbol{\phi}(\rho, l) \\
l'@l & = & \begin{cases} \mathtt{t} & \text{if } l = l' \\ \mathtt{f} & \text{otherwise} \end{cases} \\
(l' : \tau)@l & = & \begin{cases} \tau & \text{if } l = l' \\ \bot & \text{otherwise} \end{cases}
\end{array}
$$

Figure 5.13: Slice definitions

inspecting the translation rules for slices, and the semantics of field predicates. This is necessarily the case only if l_1, l_2 do not occur in π. For example, consider slices

$$
\begin{array}{lcl}
\mathcal{I}[\![\phi(\rho, l)]\!]_{FLD} & = & \mathcal{I}[\![\rho]\!](l) \\
\mathcal{I}[\![\phi(\rho, l)]\!]_{LP} & = & l \mathrel{\dot{\in}} \mathcal{D}(\mathcal{I}[\![\rho]\!]) \\
\mathcal{I}[\![\mathtt{t}]\!] & = & \mathit{true} \\
\mathcal{I}[\![\mathtt{f}]\!] & = & \mathit{false} \\
\mathcal{I}[\![\tau]\!] & = & \{\!\cdot \tau \cdot\!\} \\
\mathcal{I}[\![\bot]\!] & = & \varnothing \\
\mathcal{I}[\![*]\!] & = & * \\
\mathcal{I}[\![\varnothing]\!]_{FLD} & = & \varnothing \\
\mathcal{I}[\![\varnothing]\!]_{LP} & = & \mathit{false} \\
\mathcal{I}[\![(\mathit{lpe}_1, \ldots, \mathit{lpe}_n)]\!] & = & \bigvee_{i \in 1..n} \mathcal{I}[\![\mathit{lpe}_i]\!] \\
\mathcal{I}[\![(\mathit{fe}_1, \ldots, \mathit{fe}_n)]\!] & = & \dot{\bigcup}_{i \in 1..n} \mathcal{I}[\![\mathit{fe}_i]\!] \\
\mathcal{I}[\![\mathit{lp}_1 \mathrel{\#\!\!\!\#} \mathit{lp}_2]\!] & = & \neg\mathcal{I}[\![\mathit{lp}_1]\!] \vee \neg\mathcal{I}[\![\mathit{lp}_2]\!] \\
\mathcal{I}[\![\mathit{lp}_1 \parallel \mathit{lp}_2]\!] & = & \mathcal{I}[\![\mathit{lp}_1]\!] \leftrightarrow \mathcal{I}[\![\mathit{lp}_2]\!] \\
\mathcal{I}[\![\mathit{fld}_1 = \mathit{fld}_2]\!] & = & \mathcal{I}[\![\mathit{fld}_1]\!] = \mathcal{I}[\![\mathit{fld}_2]\!] \\
\mathcal{I}[\![\mathit{fld}/\mathit{LP}]\!] & = & \begin{cases} \mathcal{I}[\![\mathit{fld}]\!] & \text{if } \wedge_{\mathit{lp} \in \mathit{LP}} \mathcal{I}[\![\mathit{lp}]\!]_{LP} \\ \varnothing & \text{otherwise} \end{cases} \\
\mathcal{I}[\![\mathit{lp}/\mathit{LP}]\!] & = & \begin{cases} \mathcal{I}[\![\mathit{lp}]\!] & \text{if } \wedge_{\mathit{lp}' \in \mathit{LP}} \mathcal{I}[\![\mathit{lp}']\!]_{LP} \\ \mathit{false} & \text{otherwise} \end{cases} \\
\mathcal{I}[\![P]\!] & = & \bigwedge_{\pi \in P} \mathcal{I}[\![\pi]\!]
\end{array}
$$

Figure 5.14: Field predicate semantics

of predicate $(l_a) \mathbin{\#} (l_b, \rho)$ at several labels:

$$\begin{aligned}
(l_a \mathbin{\#} (l_b, \rho))@l_a &= (l_a@l_a) \mathbin{\#} (l_b@l_a, \rho@l_a) \\
&= \mathtt{t} \mathbin{\#} (\mathtt{f}, \boldsymbol{\phi}(\rho, l_a)) \\
(l_a \mathbin{\#} (l_b, \rho))@l_b &= (l_a@l_b) \mathbin{\#} (l_b@l_b, \rho@l_b) \\
&= \mathtt{f} \mathbin{\#} (\mathtt{t}, \boldsymbol{\phi}(\rho, l_b)) \\[1ex]
(l_a \mathbin{\#} (l_b, \rho))@l_1 &= (l_a@l_1) \mathbin{\#} (l_b@l_1, \rho@l_1) \\
&= \mathtt{f} \mathbin{\#} (\mathtt{f}, \boldsymbol{\phi}(\rho, l_1)) \\
(l_a \mathbin{\#} (l_b, \rho))@l_2 &= (l_a@l_2) \mathbin{\#} (l_b@l_2, \rho@l_2) \\
&= \mathtt{f} \mathbin{\#} (\mathtt{f}, \boldsymbol{\phi}(\rho, l_2))
\end{aligned}$$

Note that although the slices at l_1 and l_2 have the same form, the other two slices are different.

Moving on to the semantics, observe that every occurrence of l in $\mathcal{I}[\![\pi@l]\!]$ is in an atomic subformula of the form $l \in \mathcal{I}[\![\rho]\!]$, for some ρ. Furthermore, the remaining atoms and logical connectives of the formula are the same if l does not occur in π. The values of these atoms are fixed; they do not depend on the particular interpretation.

For example, let us examine the semantics of these four predicates:

$$\begin{aligned}
\mathcal{I}[\![(l_a \mathbin{\#} (l_b, \rho))@l_a]\!] &= \neg\mathtt{t} \vee \neg(\mathtt{f} \vee \exists\tau.(l_a, \tau) \in \mathcal{I}[\![\rho]\!]) \\
\mathcal{I}[\![(l_a \mathbin{\#} (l_b, \rho))@l_b]\!] &= \neg\mathtt{f} \vee \neg(\mathtt{t} \vee \exists\tau.(l_b, \tau) \in \mathcal{I}[\![\rho]\!]) \\[1ex]
\mathcal{I}[\![(l_a \mathbin{\#} (l_b, \rho))@l_1]\!] &= \neg\mathtt{f} \vee \neg(\mathtt{f} \vee \exists\tau.(l_1, \tau) \in \mathcal{I}[\![\rho]\!]) \\
\mathcal{I}[\![(l_a \mathbin{\#} (l_b, \rho))@l_2]\!] &= \neg\mathtt{f} \vee \neg(\mathtt{f} \vee \exists\tau.(l_2, \tau) \in \mathcal{I}[\![\rho]\!])
\end{aligned}$$

In general, the only parts of the formula which depend on the interpretation are of one of the two forms:

$$\{\tau \mid (l,\tau) \in \mathcal{I}[\![\rho]\!]\}$$

$$\exists \tau.(l,\tau) \in \mathcal{I}[\![\rho]\!]$$

Instead of quantifying over all interpretations, we can replace each occurrence of

$$(l,\tau) \in \mathcal{I}[\![\rho]\!]$$

in the formula with a variable indexed by ρ, and universally quantify over all such variables:

$$\forall \mathcal{I}.\mathcal{I}[\![\pi @ l_1]\!] \quad = \quad \forall \mathcal{I}.\forall x_{\rho_1},\ldots,x_{\rho_n}.\mathcal{I}[\![\, \mathit{Varify}^{l_1}(\pi @ l_1)]\!]$$

It is clear that the formulas corresponding to $\pi @ l_1$ and $\pi @ l_2$ can thus both be translated into the same form. Therefore,

$$(\forall \mathcal{I}.\mathcal{I}[\![\pi @ l_1]\!]) = (\forall \mathcal{I}.\mathcal{I}[\![\pi @ l_2]\!])$$

if l_1 and l_2 do not occur in π.

$\square$

Because this is the case, we only need to consider the labels that explicitly appear in the problem, and one additional arbitrary label that does not. We give two examples to demonstrate the need for this extra label:

1. It may be the case that there no explicit labels appear in the predicate sets. For example, consider the predicate sets:

$$\begin{aligned} P &= \{\rho_1 \# \rho_2\} \\ Q &= \{\rho_1 \# \rho_3\} \end{aligned}$$

   If we do not have the extra label, then $\forall \mathcal{I}. \bigwedge_{l \in \varnothing} \mathcal{I}[\![P]\!] \rightarrow \bigwedge_{l \in \varnothing} \mathcal{I}[\![Q]\!]$ is trivially true for any P and Q.

2. Even if explicit labels appear in the predicate sets, use of an additional label gives us information on whether fields whose labels do not explicitly appear may exist in a row. For example, consider the existence of a field labeled `b` in the rows $\texttt{a} : \tau$ and ρ. There is definitely no such field in the former row. For the latter row, however, the question is open in the absence of further information.

   An entailment may hold for each explicit label, but may not hold for other possible labels. For example, consider the predicate sets

$$\begin{aligned} P &= \{\rho \mathrel{\#} l\} \\ Q &= \{l : \tau = (l : \tau, \rho)\} \end{aligned}$$

   Clearly, it is not the case that $P \Vdash Q$. A counterexample is $[\rho \mapsto (l' : \tau')]$, for any $l' \neq l$. However, if we consider just the slice at l, which is the only explicit label, we get

$$\begin{aligned} P@l &= \{\boldsymbol{\phi}(\rho, l) \mathrel{\#} \texttt{t}\} \\ Q@l &= \{\tau = (\tau, \boldsymbol{\phi}(\rho, l))\} \end{aligned}$$

   It *is* the case that $P@l \Vdash Q@l$, because $\boldsymbol{\phi}(\rho, l)$ must be $\bot$ for $P@l$ to be true, which allows us to simplify $(\tau, \boldsymbol{\phi}(\rho, l))$ to τ, which in turn forces Q to be true.

   Relying on just this slice would lead us to an incorrect conclusion. Looking at an "extra" slice at l', we get

$$\begin{aligned} P@l' &= \{\boldsymbol{\phi}(\rho, l') \mathrel{\#} \texttt{f}\} \\ Q@l' &= \{\varnothing = \boldsymbol{\phi}(\rho, l')\} \end{aligned}$$

   $P@l'$ is true for any value of $\boldsymbol{\phi}(\rho, l')$. This is clearly not the case for $Q@l'$; any non-$\bot$ value of $\boldsymbol{\phi}(\rho, l')$ makes it false.

Throughout the course of the next few sections, we will need the capability to refer to a particular set of distinct labels. Rather than overloading the l symbol with two different rôles, we will use an indexed $\mathbf{l}(i)$ to indicate a specific label. Two such labels with different indices denote distinct labels. We also impose an ordering on the set of labels $\mathbf{l}(i)$:

$$\mathbf{l}(i) \leqslant \mathbf{l}(j) \quad \text{iff } i \leqslant j$$

The usual l remains a metavariable which ranges over the set of labels.

We now define $\mathcal{L}^+(\cdot)$, which is the explicit set of labels occurring in its argument, and one additional label. For convenience, we will use $\mathbf{l}(0)$ to denote this label, and define it to be strictly smaller than all other labels:

$$\mathcal{L}^+(P \cup Q) = \mathcal{L}(P \cup Q) \uplus \{\mathbf{l}(0)\}, \text{ where } \forall l \in \mathcal{L}(P \cup Q).\mathbf{l}(0) < l$$

We can now conveniently state the lemma which allows us to consider only a finite set of labels:

Lemma 5.5.1.

$$\forall \mathcal{I}.\forall l.\mathcal{I}[\![\pi @ l]\!] \quad \leftrightarrow \quad \forall \mathcal{I}. \bigwedge_{l \in \mathcal{L}^+(\pi)} \mathcal{I}[\![\pi @ l]\!]$$

Proof. We can split $\forall l.\mathcal{I}[\![\pi @ l]\!]$ into a conjunction of two subformulas:

$$\Big(\bigwedge_{l \in \mathcal{L}(\pi)} \mathcal{I}[\![\pi @ l]\!]\Big) \wedge \Big(\bigwedge_{l \notin \mathcal{L}(\pi)} \mathcal{I}[\![\pi @ l]\!]\Big)$$

We pick an arbitrary label which is not in $\mathcal{L}(\pi)$, an exemplar, if you will, to represent all of the labels not in $\mathcal{L}(\pi)$. Let us call this label l_0. By Proposition 5.5.3, this is equivalent to

$$\forall \mathcal{I}.\mathcal{I}[\![\pi @ l_0]\!] \quad = \quad \forall \mathcal{I}. \bigwedge_{l \in \mathcal{L}^+(\pi)} \mathcal{I}[\![\pi @ l]\!]$$

□

Lemma 5.5.2.

$$\begin{aligned} & \models \forall \mathcal{I}.(\forall l.\mathcal{I}[\![P@l]\!]) \rightarrow (\forall l.\mathcal{I}[\![Q@l]\!]) \\ \leftrightarrow \quad & \models \forall \mathcal{I}.(\textstyle\bigwedge_{l \in \mathcal{L}^+(P \cup Q)} \mathcal{I}[\![P@l]\!]) \rightarrow (\textstyle\bigwedge_{l \in \mathcal{L}^+(P \cup Q)} \mathcal{I}[\![Q@l]\!]) \end{aligned}$$

A *well-formed field* is one which does not have multiple field elements of the form τ. Similarly, a well-formed labelset does not have multiple `t` elements. A *well-formed field predicate set* is a field predicate set for which each field and labelset expression is well-formed.

In the previous chapter, we defined $\mathrm{TRV}(K)$ to denote the set of free row and type variables in entity K. We are now replacing row variables by field variables, and will use $\mathrm{TFV}(K)$ to denote the free type and field variables in entity K.

We will employ an abuse of notation analogous to the one described in the previous chapter where we permitted ourselves to write a row where a labelset was expected. A field expression in a label presence context is defined to be `f` if the field expression evaluates to $\{\}$, and `t` otherwise.

Let $\mathcal{L}^+(P \cup Q) = \{\mathbf{l}(0), \ldots, \mathbf{l}(N)\}$. With each row variable ρ in $\mathrm{TRV}(P \cup Q)$, associate distinct field variables $\boldsymbol{\phi}(\rho, \mathbf{l}(0)), \ldots, \boldsymbol{\phi}(\rho, \mathbf{l}(N))$.

We say that the sliced versions of K are the set $\{K@l \,|\, l \in L\}$. For the particular case of sets of predicates P, we define

$$P@L = \bigcup_{l \in L} P@l$$

We use F, G and H to range over sets of field predicates.

Proposition 5.5.4. $\mathcal{I}[\![lset]\!]_{LS} \cap L = \bigcup_{l \in L} \{l \,|\, \mathcal{I}[\![lset@l]\!]_{LP}\}$

Proof.

$$\begin{aligned}\textstyle\bigcup_{l\in L}\{l \mid \mathcal{I}[\![lset@l]\!]_{LP}\} &= \textstyle\bigcup_{l\in L}\{l \mid l \in \mathcal{I}[\![lset]\!]_{LS}\} \quad \text{by definition of } \mathcal{I}[\![lset@l]\!]_{LP} \\ &= \mathcal{I}[\![lset]\!]_{LS} \cap L\end{aligned}$$

□

Proposition 5.5.5. $\mathcal{I}[\![row]\!]|_L = \bigcup_{l\in L}\{(l,\tau) \mid \mathcal{I}[\![row@l]\!] = \tau\}$

Proof.

$$\begin{aligned}\textstyle\bigcup_{l\in L}\{(l,\tau) \mid \mathcal{I}[\![row@l]\!] = \tau\} &= \textstyle\bigcup_{l\in L}\{(l,\tau) \mid (l,\tau) \in \mathcal{I}[\![row]\!]\} \\ &\qquad \text{by definition of } \mathcal{I}[\![row@l]\!] \\ &= \mathcal{I}[\![row]\!]|_L\end{aligned}$$

□

Theorem 5.5.1.

$$\models \forall\mathcal{I}.\mathcal{I}[\![P]\!] \to \mathcal{I}[\![Q]\!] \quad \leftrightarrow \quad \models \forall\mathcal{I}.(\bigwedge_{l\in\mathcal{L}^+(P\cup Q)} \mathcal{I}[\![P@l]\!]) \to (\bigwedge_{l\in\mathcal{L}^+(P\cup Q)} \mathcal{I}[\![Q@l]\!])$$

Proof. Follows directly from Proposition 5.5.1 and Lemma 5.5.2. □

We are now ready to apply the crucial transformation of the problem: replace each predicate in the sentence

$$\forall\mathcal{I}.(\mathcal{I}[\![P]\!] \to \mathcal{I}[\![Q]\!])$$

by the conjunction of all of its slices:

$$\forall\mathcal{I}.(\bigwedge_{l\in\mathcal{L}^+(P\cup Q)} \mathcal{I}[\![P@l]\!]) \to (\bigwedge_{l\in\mathcal{L}^+(P\cup Q)} \mathcal{I}[\![Q@l]\!])$$

The meaning of these predicates on fields and booleans are analogous to the corresponding predicates on rows and labelsets, respectively. This is summarized in Figure 5.14.

We will use as our example the predicates in the entailment on page 80. We repeat the predicate sets, renaming the labels for convenience:

$$\begin{aligned} P &= \{(\mathbf{l}(1) \parallel (\rho_1, \rho_3)), (\rho_1 \# \rho_2), (\rho_1 \# \rho_3), (\rho_2 \# l_2)\} \\ Q &= \{(\rho_1, \rho_2) \# \mathbf{l}(2)\} \end{aligned}$$

For this example,

$$\mathcal{L}^+(P \cup Q) = \{\mathbf{l}(0), \mathbf{l}(1), \mathbf{l}(2)\}$$

Here are the slices of P and Q:

$$\begin{aligned}
P@\mathbf{l}(0) &= \{\, \mathtt{f} \parallel (\boldsymbol{\phi}(\rho_1, \mathbf{l}(0)), \boldsymbol{\phi}(\rho_3, \mathbf{l}(0))), \\
&\quad \boldsymbol{\phi}(\rho_1, \mathbf{l}(0)) \# \boldsymbol{\phi}(\rho_2, \mathbf{l}(0)), \\
&\quad \boldsymbol{\phi}(\rho_1, \mathbf{l}(0)) \# \boldsymbol{\phi}(\rho_3, \mathbf{l}(0)), \\
&\quad \boldsymbol{\phi}(\rho_2, \mathbf{l}(0)) \# \mathtt{f} \,\} \\
P@\mathbf{l}(1) &= \{\, \mathtt{t} \parallel (\boldsymbol{\phi}(\rho_1, \mathbf{l}(1)), \boldsymbol{\phi}(\rho_3, \mathbf{l}(1))), \\
&\quad \boldsymbol{\phi}(\rho_1, \mathbf{l}(1)) \# \boldsymbol{\phi}(\rho_2, \mathbf{l}(1)), \\
&\quad \boldsymbol{\phi}(\rho_1, \mathbf{l}(1)) \# \boldsymbol{\phi}(\rho_3, \mathbf{l}(1)), \\
&\quad \boldsymbol{\phi}(\rho_2, \mathbf{l}(1)) \# \mathtt{f} \,\} \\
P@\mathbf{l}(2) &= \{\, \mathtt{f} \parallel (\boldsymbol{\phi}(\rho_1, \mathbf{l}(2)), \boldsymbol{\phi}(\rho_3, \mathbf{l}(2))), \\
&\quad \boldsymbol{\phi}(\rho_1, \mathbf{l}(2)) \# \boldsymbol{\phi}(\rho_2, \mathbf{l}(2)), \\
&\quad \boldsymbol{\phi}(\rho_1, \mathbf{l}(2)) \# \boldsymbol{\phi}(\rho_3, \mathbf{l}(2)), \\
&\quad \boldsymbol{\phi}(\rho_2, \mathbf{l}(2)) \# \mathtt{t} \,\} \\
Q@\mathbf{l}(0) &= \{\, (\boldsymbol{\phi}(\rho_1, \mathbf{l}(0)), \boldsymbol{\phi}(\rho_2, \mathbf{l}(0))) \# \mathtt{f} \,\} \\
Q@\mathbf{l}(1) &= \{\, (\boldsymbol{\phi}(\rho_1, \mathbf{l}(1)), \boldsymbol{\phi}(\rho_2, \mathbf{l}(1))) \# \mathtt{f} \,\} \\
Q@\mathbf{l}(2) &= \{\, (\boldsymbol{\phi}(\rho_1, \mathbf{l}(2)), \boldsymbol{\phi}(\rho_2, \mathbf{l}(2))) \# \mathtt{t} \,\}
\end{aligned}$$

5.5.3 Field predicate simplification

For our problem P **æ** Q, once P and Q are decomposed into the sliced versions F and G, we can simplify the problem by using a reduction algorithm on F and applying any substitutions thus generated to G. This reduction procedure will also be useful in a later stage of the algorithm. We will define the reduction relation in several stages.

First, we define a one-step reduction relation. We write $\pi \longrightarrow (\theta, H)$ to mean

$$\begin{array}{lcll}
(\mathit{fe}_1, \ldots, \mathit{fe}_n, \bot) & \longrightarrow & (\mathit{fe}_1, \ldots, \mathit{fe}_n) & \text{where } n \geqslant 1 \\
(\mathit{lpe}_1, \ldots, \mathit{lpe}_n, \texttt{f}) & \longrightarrow & (\mathit{lpe}_1, \ldots, \mathit{lpe}_n) & \text{where } n \geqslant 1 \\
(\phi_1, \ldots, \phi_n, \tau, \mathit{fe}_1, \ldots, \mathit{fe}_m) & \longrightarrow & \begin{array}{l}([\phi_1 \mapsto \bot, \ldots, \phi_n \mapsto \bot], \\ \ (\tau, \mathit{fe}_1, \ldots, \mathit{fe}_m))\end{array} & \text{where } n \geqslant 1, m \geqslant 0 \\
(\phi_1, \ldots, \phi_n, \texttt{t}, \mathit{lpe}_1, \ldots, \mathit{lpe}_m) & \longrightarrow & \begin{array}{l}([\phi_1 \mapsto \bot, \ldots, \phi_n \mapsto \bot], \\ \ (\texttt{t}, \mathit{lpe}_1, \ldots, \mathit{lpe}_m))\end{array} & \text{where } n \geqslant 1, m \geqslant 0
\end{array}$$

Figure 5.15: Field and label presence simplification

that predicate π reduces to field predicate set H in one step via application of substitution θ. When H is a singleton $\{\pi'\}$, we will usually omit the set braces and simply write π'. If the substitution is omitted, it is understood to be the null substitution. Similarly, if the predicate set is omitted, it is understood to be the empty set. We solemnly promise not to omit both in the same rule. Furthermore, we overload $\longrightarrow$ so that it also applies to fields and label presences: $\mathit{fld} \longrightarrow (\theta, \mathit{fld}')$ and $\mathit{lp} \longrightarrow (\theta, \mathit{lp}')$.

We will define the one-step reduction relation in three steps.

1. We define some reduction rules based solely on the value of one of the two arguments of a predicate:

   - Field expressions can be simplified by removing redundant $\bot$'s. For example, the field expression

     $$(\bot, \tau, \phi, \bot)$$

     can be simplified to (τ, ϕ). The general rule is that an occurrence of $\bot$

can be removed as long as it is not the last field element.

- Similarly, label presence expressions can be simplified by removing redundant `f` elements. For example, the label presence expression

  $$(\mathtt{f}, \mathtt{t}, \phi, \mathtt{f})$$

  can be simplified to

  $$(\mathtt{t}, \phi)$$

  Just as for field expressions, all occurrences of `f` but the last can be removed from a label presence expression.

- Any predicate which has $(\tau, \phi_1, \ldots, \phi_n)$ as one of its arguments reduces to the same predicate with that argument replaced by τ, via the substitution $[\phi_i \mapsto \bot \mid i \in 1..n]$. For example,

  $$lp \parallel (\tau, \phi_1, \phi_2) \longrightarrow ([\phi_1 \mapsto \bot, \phi_2 \mapsto \bot], lp \parallel \tau)$$

These reduction rules are formalized in Figure 5.15.

2. If we reduce a field or label presence expression, we can reduce any predicate which has that expression as one of its operands. Formally, we have the rules

$$\frac{\mathit{fld}_1 \longrightarrow (\theta, \mathit{fld}_1')}{\mathit{fld}_1 = \mathit{fld}_2 \longrightarrow (\theta, \mathit{fld}_1' = \mathit{fld}_2)}$$

$$\frac{\mathit{fld}_2 \longrightarrow (\theta, \mathit{fld}_2')}{\mathit{fld}_1 = \mathit{fld}_2 \longrightarrow (\theta, \mathit{fld}_1 = \mathit{fld}_2')}$$

$$\frac{lp_1 \longrightarrow (\theta, lp_1')}{lp_1 \parallel lp_2 \longrightarrow (\theta, lp_1' \parallel lp_2)}$$

$$\frac{lp_2 \longrightarrow (\theta, lp_2')}{lp_1 \parallel lp_2 \longrightarrow (\theta, lp_1 \parallel lp_2')}$$

$$\frac{lp_1 \longrightarrow (\theta, lp_1')}{lp_1 \# lp_2 \longrightarrow (\theta, lp_1' \# lp_2)}$$

$$\frac{lp_2 \longrightarrow (\theta, lp_2')}{lp_1 \# lp_2 \longrightarrow (\theta, lp_1 \# lp_2')}$$

3. We can simplify predicates such as

$$(fe_1, fe_2) = (fe_a, fe_b)$$

The key observation is that we can "split" each field element in the predicate into a set of new field, each of which represents the "intersection" of the original field element with one of the field elements occurring on the opposite side of the equality. In the case of the predicate above, we can split fe_1 into $(\mathit{fld}_{1a}, \mathit{fld}_{1b})$, for some appropriate fields fld_{1a} and fld_{1b}, and similarly for the rest of the field elements in the predicate, giving us:

$$\begin{aligned}(fe_1, fe_2) = (fe_a, fe_b) \quad \longrightarrow \quad & \{(fe_1 = (\mathit{fld}_{1a}, \mathit{fld}_{1b})),\ (fe_2 = (\mathit{fld}_{2a}, \mathit{fld}_{2b})), \\ & (fe_a = (\mathit{fld}_{1a}, \mathit{fld}_{2a})),\ (fe_b = (\mathit{fld}_{1b}, \mathit{fld}_{2b}))\})\end{aligned}$$

We need to determine what would be appropriate values for fld_{1a}, etc. For fld_{1a}, we want something that represents the intersection of the field elements fe_1 and fe_a. If either of these field elements evaluate to $\bot$, then fld_{1a} should as well. Otherwise, fe_1 and fe_a must both evaluate to some type τ, in which case fld_{1a} should as well. The restricted field construct that we had the foresight to define in Figure 5.14 has the correct semantics:

$$\mathit{fld}/LP = \begin{cases} \mathcal{I}[\![\mathit{fld}]\!] & \text{if } \wedge_{lp \in LP} \mathcal{I}[\![lp]\!]_{LP} \\ \varnothing & \text{otherwise} \end{cases}$$

Specifically, fld_{1a} would be $\mathit{fe}_1/\mathit{fe}_a$, and similarly for the other intersections. We will show that all necessary disjoint conditions are satisfied by this substitution.

4. We can apply a transformation similar to the one in the previous point when the predicate is $\parallel$ instead of $=$. However, there is a complication. Consider the predicate

$$(\mathit{lpe}_1, \mathit{lpe}_2) \parallel (\mathit{lpe}_a, \mathit{lpe}_b)$$

We cannot have equate the field elements on the left-hand and right-hand sides to common field elements, because this would force both the labels *and types* associated with the original field elements to be the same, instead of just the labels. We will instead map the variables on opposite sides of the $\parallel$ to distinct field elements which themselves are related by $\parallel$. In this particular case, we have:

$$\begin{aligned}(lpe_1, lpe_2) \parallel (lpe_a, lpe_b) \quad \longrightarrow \quad & \{(lpe_1 = (lpe_1/lpe_a, lpe_1/lpe_b)), \\ & (lpe_2 = (lpe_2/lpe_a, lpe_2/lpe_b)), \\ & (lpe_a = (lpe_a/lpe_1, lpe_a/lpe_2)), \\ & (lpe_b = (lpe_b/lpe_1, lpe_b/lpe_2))\})\end{aligned}$$

The general reduction rules for $=$ and $\parallel$ are given in Figure 5.17.

We have now replaced a single $\parallel$ (or $=$) predicate with four $=$ predicates. However, the substitutions may cause already simplified $\parallel$ predicates to become subject to this transformation. If unchecked, this may prevent the transformation process from terminating. A closer look at what is occurring reveals that the process is terminating. Each "new" field element introduced is a restriction of an existing field element. The restriction can always be written as a set of flat (i.e., unrestricted) label presences. No new field variables are introduced. Therefore, there is a limit to the number of distinct restrictions. Because there is a finite number of unrestricted label presences, at some point, we simply run out of new restrictions. For example, if the process does not otherwise terminate, we will reach a point where we have:

$$(fe_1/LP_1, fe_2/LP_2) = (fe_a/LP_a, fe_b/LP_b)$$

and $LP_1 = LP_a$. Then the equation

$$fe_1/LP_1 = (fe_1/LP_1/(fe_a/LP_a), fe_1/LP_1/(fe_b/LP_b))$$

can be rewritten as $fe_1/LP_1 = (fe_1/LP_1, fe_1/(LP_1 \cup LP_b))$, and this can be simplified to $fe_1/(LP_1 \cup LP_b)) = \perp$.

$$
\begin{array}{rcll}
\mathtt{f} & \# & lp & \longrightarrow \mathtt{OK} \\
\phi & \# & \mathtt{t} & \longrightarrow [\phi \mapsto \bot] \\
\mathtt{t} & \# & \mathtt{t} & \longrightarrow \mathtt{Fail} \\
(\phi_1, \ldots, \phi_n) & \# & lp & \longrightarrow \{\phi_i \# lp \mid i \in 1..n\} \\
\mathtt{f} & \| & \mathtt{f} & \longrightarrow \mathtt{OK} \\
\mathtt{t} & \| & \mathtt{t} & \longrightarrow \mathtt{OK} \\
\mathtt{t} & \| & \mathtt{f} & \longrightarrow \mathtt{Fail} \\
\phi & \| & \mathtt{t} & \longrightarrow [\phi \mapsto \alpha] \qquad \alpha \text{ new} \\
(\phi, lp) & \| & (\phi, lp') & \longrightarrow lp \| lp' \\
(\phi_1, \ldots, \phi_n) & \| & \mathtt{f} & \longrightarrow [\phi_1 \mapsto \bot, \ldots, \phi_n \mapsto \bot] \\
\bot & = & \bot & \longrightarrow \mathtt{OK} \\
\alpha & = & \tau & \longrightarrow \begin{cases} [\alpha \mapsto \tau] & \text{if } \alpha \text{ does not occur in } \tau \\ \mathtt{Fail} & \text{otherwise} \end{cases} \\
\phi & = & \hat{\tau} & \longrightarrow [\phi \mapsto \hat{\tau}] \quad \text{if } \phi \text{ does not occur in } \hat{\tau} \\
t & = & \tau & \longrightarrow \begin{cases} \mathtt{OK} & \text{if } \tau \equiv t \\ \mathtt{Fail} & \text{otherwise} \end{cases} \\
\tau_1 \to \tau_2 & = & \tau & \longrightarrow \begin{cases} \{\tau_1 = \tau_1', \tau_2 = \tau_2'\} & \text{if } \tau \equiv \tau_1' \to \tau_2' \\ \mathtt{Fail} & \text{otherwise} \end{cases} \\
\{row\} & = & \tau & \longrightarrow \begin{cases} \{row@\mathbf{l}(0) = row'@\mathbf{l}(0), & \\ \ \ldots, & \text{if } \tau \equiv \{row'\} \\ \ row@\mathbf{l}(n) = row'@\mathbf{l}(n)\} & \\ \mathtt{Fail} & \text{otherwise} \end{cases} \\
\tau & = & \bot & \longrightarrow \mathtt{Fail} \\
(\phi, \hat{\tau}) & = & (\phi, \hat{\tau}') & \longrightarrow \hat{\tau} = \hat{\tau}' \\
(\phi_1, \ldots, \phi_n) & = & \bot & \longrightarrow [\phi_1 \mapsto \bot, \ldots, \phi_n \mapsto \bot]
\end{array}
$$

Figure 5.16: Field predicate simplification

$$(\mathit{lpe}_1, \ldots, \mathit{lpe}_n) \parallel (\mathit{lpe}'_1, \ldots, \mathit{lpe}'_m) \longrightarrow H \quad \text{if } n, m > 1$$

$$\text{where } H = \Big(\bigcup_{i \in 1..n} \{ \mathit{lpe}_i = (\mathit{lpe}_i / \mathit{lpe}'_1, \ldots, \mathit{lpe}_i / \mathit{lpe}'_m) \} \Big) \cup \Big(\bigcup_{j \in 1..m} \{ \mathit{lpe}'_j = (\mathit{lpe}'_j / \mathit{lpe}_1, \ldots, \mathit{lpe}'_j / \mathit{lpe}_n) \} \Big)$$

$$(\mathit{fe}_1, \ldots, \mathit{fe}_n) = (\mathit{fe}'_1, \ldots, \mathit{fe}'_m) \longrightarrow H \quad \text{if } n, m > 1$$

$$\text{where } H = \Big(\bigcup_{i \in 1..n} \{ \mathit{fe}_i = (\mathit{fe}_i / \mathit{fe}'_1, \ldots, \mathit{fe}_i / \mathit{fe}'_m) \} \cup \bigcup_{j \in 1..m} \{ \mathit{fe}'_j = (\mathit{fe}_1 / \mathit{fe}'_j, \ldots, \mathit{fe}_n / \mathit{fe}'_j) \} \Big)$$

Figure 5.17: Field equality predicate simplification

$$(\textbf{failure}) \qquad (\theta, F \uplus \{\texttt{Fail}\}) \longmapsto (\theta, \{\texttt{Fail}\})$$

$$(\textbf{unit}) \qquad \frac{\pi \longrightarrow (\theta', H)}{(\theta, F \uplus \{\pi\}) \longmapsto (\theta' \circ \theta, \theta' F \cup \theta H)}$$

Figure 5.18: Predicate set reduction

Figure 5.16 gives the remainder of the rules.

Proposition 5.5.6. *Given a well-formed field predicate set F, if a field expression occurring as either argument of any predicate in F has the form $(\mathit{fld}_1, \ldots, \mathit{fld}_n)$ then if there is more than one fld_i which is τ, then F is not satisfiable.*

Proof. Two field elements in a field F can be types only if two fields with the same label occur in the row F is a slice of. Because each distinct pair of field elements in each field is related by $\#$ in a well-formed field, the field predicate $\tau \# \tau'$ occurs in F, and this predicate is unsatisfiable.

If P is a well-formed predicate set, and F is the result of converting P to a set of field predicates, then any field expressions $(\hat{\tau}_1, \ldots, \hat{\tau}_n)$ in F correspond to a slice of a row with n elements in P. Furthermore, because of the well-formedness criteria for rows and labelsets, it is the case that $F \models \hat{\tau}_i \,\#\, \hat{\tau}_j$, for $i, j \in 1..n$ and $i \neq j$. $\square$

Lemma 5.5.3 (Single Step Simplification Equivalence). *If we have a simplification rule $\pi \longrightarrow (\theta, H)$, then for any θ', well-formed F and G, where $\theta' F = F$,*

$$(\forall \mathcal{I}.\mathcal{I}[\![\{\pi\} \uplus F]\!] \to \mathcal{I}[\![G]\!]) \quad \leftrightarrow \quad (\forall \mathcal{I}.\mathcal{I}[\![H \cup \theta F]\!] \to \mathcal{I}[\![\theta G]\!])$$

Next, we use the one-step reduction relation on predicates $\longrightarrow$ to define a one-step reduction on predicate sets: we write $(\theta, F) \longmapsto (\theta'\theta, G)$ to mean that substitution θ and predicate set F reduce to predicate set G in one step via application of substitution θ'. The rules for this relation are given in Figure 5.18.

Thirdly, we define the reflexive, transitive closure of $\longmapsto$, i.e. $\longmapsto\!\!\!\!\rightarrow$:

$$(\longmapsto\!\!\!\!\rightarrow\textbf{-reflex}) \qquad (\theta, F) \longmapsto\!\!\!\!\rightarrow (\theta, F)$$

$$(\longmapsto\!\!\!\!\rightarrow\textbf{-unit}) \qquad \frac{(\theta, F) \longmapsto (\theta', G)}{(\theta, F) \longmapsto\!\!\!\!\rightarrow (\theta', G)}$$

$$(\longmapsto\!\!\!\!\rightarrow\textbf{-trans}) \qquad \frac{(\theta_1, F_1) \longmapsto\!\!\!\!\rightarrow (\theta_2, F_2) \quad (\theta_2, F_2) \longmapsto\!\!\!\!\rightarrow (\theta_3, F_3)}{(\theta_1, F_1) \longmapsto\!\!\!\!\rightarrow (\theta_3, F_3)}$$

Finally, we define the Norm function: $\mathsf{Norm}(F) = (\theta, F')$ iff $([\,], F) \longmapsto\!\!\!\!\rightarrow (\theta, F')$, and there is no rule $(\theta, F') \longmapsto (\theta', F'')$.

Given a pair of field predicate sets (F, G), where F is the antecedent, we compute the normalized version F', where $\mathsf{Norm}(F) = (\theta, F')$. For our example, we simplify the set of predicates $P@L$:

$$\begin{array}{lcl}
\mathtt{f} \parallel (\boldsymbol{\phi}(\rho_1,\mathbf{l}(0)),\boldsymbol{\phi}(\rho_3,\mathbf{l}(0))) & \longrightarrow & ([\boldsymbol{\phi}(\rho_1,\mathbf{l}(0)) \mapsto \bot, \boldsymbol{\phi}(\rho_3,\mathbf{l}(0)) \mapsto \bot], \varnothing) \\
\multicolumn{3}{l}{[\boldsymbol{\phi}(\rho_1,\mathbf{l}(0)) \mapsto \bot](\boldsymbol{\phi}(\rho_1,\mathbf{l}(0)) \# \boldsymbol{\phi}(\rho_2,\mathbf{l}(0)))} \\
 & = & \bot \# \boldsymbol{\phi}(\rho_2,\mathbf{l}(0)) \\
 & \longrightarrow & \mathtt{OK} \\
\multicolumn{3}{l}{[\boldsymbol{\phi}(\rho_1,\mathbf{l}(0)) \mapsto \bot, \boldsymbol{\phi}(\rho_3,\mathbf{l}(0)) \mapsto \bot](\boldsymbol{\phi}(\rho_1,\mathbf{l}(0)) \# \boldsymbol{\phi}(\rho_3,\mathbf{l}(0)))} \\
 & = & \bot \# \bot \\
 & \longrightarrow & \mathtt{OK} \\
\boldsymbol{\phi}(\rho_2,\mathbf{l}(0)) \# \mathtt{f} & \longrightarrow & \mathtt{OK} \\
\boldsymbol{\phi}(\rho_2,\mathbf{l}(1)) \# \mathtt{f} & \longrightarrow & \mathtt{OK} \\
\mathtt{f} \parallel (\boldsymbol{\phi}(\rho_1,\mathbf{l}(2)),\boldsymbol{\phi}(\rho_3,\mathbf{l}(2))) & \longrightarrow & ([\boldsymbol{\phi}(\rho_1,\mathbf{l}(2)) \mapsto \bot, \boldsymbol{\phi}(\rho_3,\mathbf{l}(2)) \mapsto \bot], \varnothing) \\
\multicolumn{3}{l}{[\boldsymbol{\phi}(\rho_1,\mathbf{l}(2)) \mapsto] \# \boldsymbol{\phi}(\rho_1,\mathbf{l}(2))\boldsymbol{\phi}(\rho_2,\mathbf{l}(2))} \\
 & = & \bot \# \boldsymbol{\phi}(\rho_2,\mathbf{l}(2)) \\
 & \longrightarrow & \mathtt{OK} \\
\boldsymbol{\phi}(\rho_1,\mathbf{l}(2)) \# \boldsymbol{\phi}(\rho_3,\mathbf{l}(2)) & = & \bot \# \boldsymbol{\phi}(\rho_3,\mathbf{l}(2)) \\
 & \longrightarrow & \mathtt{OK} \\
\boldsymbol{\phi}(\rho_2,\mathbf{l}(2)) \# \mathtt{t} & \longrightarrow & ([\boldsymbol{\phi}(\rho_2,\mathbf{l}(2)) \mapsto \bot], \varnothing)
\end{array}$$

Summarizing:

$$\begin{array}{lcl}
\mathsf{Norm}(P@L) & = & ([\boldsymbol{\phi}(\rho_1,\mathbf{l}(0)) \mapsto \bot,\ \boldsymbol{\phi}(\rho_3,\mathbf{l}(0)) \mapsto \bot,\ \boldsymbol{\phi}(\rho_1,\mathbf{l}(2)) \mapsto \bot, \\
 & & \quad \boldsymbol{\phi}(\rho_2,\mathbf{l}(2)) \mapsto \bot,\ \boldsymbol{\phi}(\rho_3,\mathbf{l}(2)) \mapsto \bot], \\
 & & \{\mathtt{t} \parallel (\boldsymbol{\phi}(\rho_1,\mathbf{l}(1)),\boldsymbol{\phi}(\rho_3,\mathbf{l}(1))), \\
 & & \quad \boldsymbol{\phi}(\rho_1,\mathbf{l}(1)) \# \boldsymbol{\phi}(\rho_2,\mathbf{l}(1)), \\
 & & \quad \boldsymbol{\phi}(\rho_1,\mathbf{l}(1)) \# \boldsymbol{\phi}(\rho_3,\mathbf{l}(1))\})
\end{array}$$

Let θ be the substitution above. This needs to be applied to the consequent:

$$\theta(G@L) \quad = \quad \{(\mathtt{f} \# \mathtt{f}),\ ((\boldsymbol{\phi}(\rho_1, \mathbf{l}(1)), \boldsymbol{\phi}(\rho_2, \mathbf{l}(1))) \# \mathtt{f}),\ (\mathtt{f} \# \mathtt{t})\}$$

Each of these three predicates are trivially true. For this problem, there is no need to proceed further. Q is true whenever P is.

Observation: The normalized predicates are all in one of the forms given in Figure 5.20.

Proposition 5.5.7. *The process of computing* $\mathsf{Norm}(F)$ *for a well-formed field predicate set* F *terminates.*

Proof. We define the *size* of a predicate in Figure 5.19. Using this, we define a metric on (θ, F) which has no infinite descending chains and strictly decreases with every transformation. We call this metric the *problem size.* The problem size is a pair whose elements are:

1. The number of field variables.

2. A tuple $(c_1, c_2, \ldots, c_n)$, where c_i is the number of predicates in F of size i, and the size of the largest predicate in F is n.

   Given two such tuples, $(c_1, \ldots, c_n)$, and $(d_1, \ldots d_m)$, we define the first to be larger, or equivalently, the second to be smaller, if either $n > m$, or $n = m$ and there exists an i such that $c_i > d_i$, and $c_j = d_j$ for all $i < j <= n$. For example, the following tuples are in ascending order:

$$
\begin{aligned}
&(2, 3)\\
&(1, 1, \mathbf{1})\\
&(0, 2, 0, \mathbf{1})\\
&(1, 1, \mathbf{1}, 1)\\
&(\mathbf{2}, 1, 1, 1)\\
&(0, \mathbf{2}, 1, 1)
\end{aligned}
$$

The crucial element that causes a predicate to be larger than the previous is written in boldface.

It can be seen by inspection that this metric decreases for each of the simplification rules described above. For example, for the rule

$$(\phi, \hat{\tau}) = (\phi, \hat{\tau}') \quad \longrightarrow \quad \hat{\tau} = \hat{\tau}'$$

the metric of any set with this element decreases when this rule is applied because the second element in the problem size pair decreases, while the first element remains the same. For another example, consider either of the two rules in Figure 5.17. This reduces a single predicate to a number of other, simpler predicates, each of which is strictly smaller. $\square$

The need for the first element of the problem size pair is due to the rules which reduce to substitutions of field variables. If we simply rely on the number and size of the predicates as our metric, such rules cannot be depended on to decrease it. A substitution may make other predicates in the set more complex, because it replaces a single field variable (of size 1) with fields of size potentially much larger.

$$
\begin{array}{lcl}
\mathsf{size}(\bot) & = & 1 \\
\mathsf{size}(t) & = & 1 \\
\mathsf{size}(\phi) & = & 1 \\
\mathsf{size}(\mathtt{t}) & = & 1 \\
\mathsf{size}(\mathtt{f}) & = & 1 \\
\mathsf{size}(\tau_1 \to \tau_2) & = & 1 + \mathsf{size}(\tau_1) + \mathsf{size}(\tau_2) \\
\mathsf{size}((\hat{\tau}_1, \ldots, \hat{\tau}_n)) & = & 1 + \sum_{i \in 1..n} \mathsf{size}(\hat{\tau}_i) \\
\mathsf{size}(lp_1 \parallel lp_2) & = & \mathsf{size}(lp_1) + \mathsf{size}(lp_2) \\
\mathsf{size}(lp_1 \# lp_2) & = & \mathsf{size}(lp_1) + \mathsf{size}(lp_2) \\
\mathsf{size}(\hat{\tau}_1 = \hat{\tau}_2) & = & \mathsf{size}(\hat{\tau}_1) + \mathsf{size}(\hat{\tau}_2) \\
\mathsf{size}(\mathtt{Fail}) & = & 1
\end{array}
$$

Figure 5.19: Predicate size

$$
\begin{array}{lrcl}
(1) & \phi & \# & \phi' \\
(2) & \phi & \parallel & \phi' \\
(3) & (\phi_1/LP_1, \ldots, \phi_n/LP_n) & \parallel & \mathtt{t} \\
(4) & (\phi_1/LP_1, \ldots, \phi_n/LP_n) & = & \tau \\
(5) & (\phi_1/LP_1, \ldots, \phi_n/LP_n) & = & \phi/LP
\end{array}
$$

Figure 5.20: Field predicate normal forms

Fortunately, each new substitution eliminates a field variable, of which there is only a finite number.

The following proposition asserts that our field simplification rules are correct:

Proposition 5.5.8 (Simplification Equivalence). *If F is a well-formed field* *predicate set, and* $\mathsf{Norm}(F) = (\theta, H)$, *then*

$$(\forall \mathcal{I}.\mathcal{I}[\![F]\!] \rightarrow \mathcal{I}[\![G]\!]) \quad \leftrightarrow \quad (\forall \mathcal{I}.\mathcal{I}[\![H]\!] \rightarrow \mathcal{I}[\![\theta G]\!])$$

Proof. This is true by a trivial induction on the length of a reduction sequence, where each step is true by Lemma 5.5.3. ☐

5.5.4 Decomposition

We can decompose the universally quantified formula into a conjunction of simpler cases using the equality

$$\begin{aligned}\forall \phi.p &= (\forall \phi.(\phi = \bot) \rightarrow p) \wedge (\forall \phi.(\phi \neq \bot) \rightarrow p) \\ & [\phi \mapsto \bot]p \wedge \forall \alpha.[\phi \mapsto \alpha]p \quad \alpha \text{ new}\end{aligned}$$

Using this equality, we can eliminate all field variables from our formula:

$$\forall \vec{\phi}, \vec{\alpha}.([\![F]\!] \rightarrow [\![G]\!]) \;=\; \bigwedge_{\forall \vec{\phi_Z}, \vec{\phi_N}.\vec{\phi_Z} \uplus \vec{\phi_N} = \vec{\phi}} [\vec{\phi_Z} \mapsto \bot](\forall \vec{\alpha_N}, \vec{\alpha}.[\vec{\phi_N} \mapsto \vec{\alpha_N}]([\![F]\!] \rightarrow [\![G]\!]))$$

where $\vec{\alpha_N}$ is new.

5.5.5 Evaluation

The decomposition in the previous section yields 2^n conjuncts, where $n = |\vec{\phi}|$. Each of these conjuncts has the form:

$$\forall \vec{\alpha}.([\![F]\!] \rightarrow [\![G]\!])$$

where F has no free field variables. We can now completely evaluate F, as shown by the following proposition. If F evaluates to `Fail`, then we can discard this conjunct. Otherwise, we need to remember the evaluating substitution θ, and apply it to G, yielding θG.

Proposition 5.5.9. *If we have a field predicate set F with no free field variables, then one of the following is true:*

$$\mathsf{Norm}(F) = (\theta, \mathtt{Fail})$$
$$\mathsf{Norm}(F) = (\theta, \mathtt{OK})$$

Proof. All field expressions $(\mathit{fld}_1, \ldots, \mathit{fld}_n)$ which have no field variables can be fully evaluated to either $\bot$, some type τ, or $*$ in a field context, and either $\mathtt{t}$ or $\mathtt{f}$ in a label presence context. This can be seen by inspection of the semantic rules in Figure 5.14. Because of this, in conjunction with the simplification rule for $*$, we can limit our scrutiny to predicates whose field operands are either $\bot$ or some type τ, and predicates whose label presence operands are either $\mathtt{t}$ or $\mathtt{f}$.

With one exception, all of the applicable simplification rules reduce such a predicate to either $\mathtt{Fail}$ or $\mathtt{OK}$. The remaining rule is:

$$\tau = \tau' \quad \xrightarrow{\theta} \quad H \qquad \text{where } \tau \overset{\theta;H}{\sim} \tau'$$

The field predicate set H may be non-empty only in the case where we are unifying two record types. H needs to be simplified and decomposed, possibly yielding more field predicates. Because of the occurs check, however, we are guaranteed that the process will terminate.

The simplification problem, then, reduces to a conjunction of type unifications. The end result of each unification must be either a substitution, with no predicate set, or failure. □

In summary,

$$\forall\vec{\alpha}.[\![F]\!] \to [\![G]\!] = \begin{cases} \forall\vec{\alpha}.[\![\theta G]\!] & \text{if } \mathsf{Norm}(F) = (\theta, \mathtt{OK}) \\ \mathit{true} & \text{if } \mathsf{Norm}(F) = (\theta, \mathtt{Fail}). \end{cases}$$

Because G is a conjunction of type equalities, we can evaluate θG by substituting distinct type constants which do not appear in G for the universally quantified variables. The choice of distinct type constants for the substitutions has the effect of preventing each universally quantified variable α from being equal to any other type. Because none of the predicates are satisfied as a result of two types being non-equal, this choice for the substitution value will prevent inadvertent satisfaction of the formula. This results in a conjunct of type equalities which has no free type or field variables. The truth or falsity of each can then easily be determined.

5.5.6 Algorithm summary

We summarize the entailment algorithm here for the reader's convenience.

Algorithm to calculate P **æ** Q:

1. **Restate as a logical implication**:

$$\forall \mathcal{I}.(\mathcal{I}[\![P]\!] \rightarrow \mathcal{I}[\![Q]\!])$$

2. **Slice into field predicates**:

$$\forall \mathcal{I}.(\mathcal{I}[\![F]\!]) \rightarrow (\mathcal{I}[\![G]\!])$$

   where $L = \mathcal{L}^{+}(P \cup Q)$, $F = P@L$, and $G = Q@L$.

3. **Simplify the antecedent**:

$$\mathsf{Norm}(F) = (\theta, H)$$

   and put back into the formula:

$$\forall \mathcal{I}.(\mathcal{I}[\![H]\!] \rightarrow \mathcal{I}[\![\theta G]\!])$$

4. **Split**:

$$\bigwedge_{\forall \vec{\phi_Z}, \vec{\phi_N}. \vec{\phi_Z} \uplus \vec{\phi_N} = \vec{\phi}} \forall \vec{\alpha_N}, \vec{\alpha}. [\vec{\phi_Z} \mapsto \bot, \vec{\phi_N} \mapsto \vec{\alpha_N}](\llbracket H \rrbracket \to \llbracket \theta G \rrbracket)$$

where $\vec{\alpha_N}$ are new, $\vec{\phi} = FFV(\theta G)$, and $\mathcal{I}\llbracket \alpha \rrbracket = \alpha$.

5. **Evaluate**:

$$\bigwedge_{\substack{\forall \vec{\phi_Z}, \vec{\phi_N}. \vec{\phi_Z} \uplus \vec{\phi_N} = \vec{\phi} \\ \mathsf{Norm}(\theta_{ZN} H) = (\theta', \mathtt{OK})}} \forall \vec{\alpha_N}, \vec{\alpha}, \vec{\phi}'. \mathcal{I}\llbracket \theta' \circ \theta_{ZN}(\theta G) \rrbracket$$

where $\theta_{ZN} = [\vec{\phi_Z} \mapsto \bot, \vec{\phi_N} \mapsto \vec{\alpha_N}]$, and $\vec{\alpha_N}$ are new.

Proposition 5.5.10. *The algorithm to calculate P æ Q, summarized above, terminates.*

Proof. We show that each step of the algorithm terminates. For step 1, this is clear. For step 2, because the process of constructing a single slice terminates, and there are a finite number of slices, the entire step terminates. Step 3, i.e., simplification, terminates by Proposition 5.5.7. Step 4, "split", terminates because there are a finite number of variables in the formulae. Similarly, step 5 terminates because the formula to be evaluated is finite. □

5.5.7 Incorporating functional dependencies

The addition of functional dependencies requires a modification to the algorithm. Consider the type scheme

$$\forall \rho, \rho' . (l : \mathtt{int}) = (\rho, \rho') \Rightarrow \{\rho\} \to \mathtt{int}$$

As explained in section 5.3, this is not ambiguous, because although ρ' does not appear in $\{\rho\} \rightarrow$ `int`, it is uniquely determined by ρ, which does. In logical terms, ρ' is existentially quantified; its value is a skolem function of ρ. This seems to flatly contradict the universal quantification given to ρ' in the type scheme, but it is actually the case that the two quantifications are equivalent in this particular context, as shown by the following proposition.

Proposition 5.5.11. *Let p and q be propositions. If variable x is not free in q, then $\forall x.(p \rightarrow q)$ is equivalent to $(\exists x.p) \rightarrow q$.*

Proof. We consider two cases for both formulas.

Case $\neg\exists x.p$

Then both $\forall x.(p \rightarrow q)$ and $(\exists x.p) \rightarrow q$ are true. To see this, consider an arbitrary value v. $[x \mapsto v]p$ is not true, and hence the implication $[x \mapsto v]p \rightarrow q$ is true. Since this is true for all values of x, the universally quantified formula is true. $(\exists x.p) \rightarrow q$ is true simply by virtue of the tautology $(\neg P) \rightarrow (P \rightarrow Q)$.

Case $\exists x.p$

In this case, the formula with the existential quantifier reduces to q.

For those x for which p is not true, $p \rightarrow q$ is true. Otherwise, $p \rightarrow q$ is equivalent to q. The universally quantified formula then, also reduces to q.

□

If we can pick either quantification for functionally determined variables, the question is “which is correct?”. To determine this, we must examine how universal and existential quantification of these variables behaves in a context where it matters. We find this situation when the variable in question appears on the right of an implication. For example, application of the expansion of row coercion described in section 4.3.3 to $\rho \blacktriangleright l : \texttt{int}$ yields

$$\{\rho = (l : \texttt{int}, \rho'), l : \texttt{int} \# \rho'\}$$

where ρ' is functionally determined by ρ. Let us write the pair of implications

$$\forall \rho.(\mathcal{I}[\![\rho \blacktriangleright l : \texttt{int}]\!] \rightarrow \forall \rho'.\mathcal{I}[\![\{\rho = (l : \texttt{int}, \rho'), l : \texttt{int} \# \rho'\}]\!])$$

and

$$\forall \rho.(\mathcal{I}[\![\rho \blacktriangleright l : \texttt{int}]\!] \rightarrow \exists \rho'.\mathcal{I}[\![\{\rho = (l : \texttt{int}, \rho'), l : \texttt{int} \# \rho'\}]\!])$$

The implication using the "correct" quantification should be true. It is easy to see that the formula using universal quantification is false. To provide a counterexample, it is sufficient to find one value of ρ' for which the implication

$$\mathcal{I}[\![\rho \blacktriangleright l : \texttt{int}]\!] \rightarrow \mathcal{I}[\![\{\rho = (l : \texttt{int}, \rho'), l : \texttt{int} \# \rho'\}]\!]$$

does not hold. $l : \texttt{int}$ is such a value.

For the other formula, given any value of ρ for which $\mathcal{I}[\![\rho \blacktriangleright l : \texttt{int}]\!]$ holds, we can always determine a value of ρ' which makes $\mathcal{I}[\![\{\rho = (l : \texttt{int}, \rho'), l : \texttt{int} \# \rho'\}]\!]$ true, namely, $\mathcal{I}[\![\rho]\!] - \{\!\cdot(l, \texttt{int})\cdot\!\}$.

In the original problem, $P \Vdash Q$, we were able to universally quantify all free variables. We cannot do so now. Because of proposition 5.5.11, we *can* universally quantify all of the free variables in P, but the functionally dependent variables in Q must be existentially quantified. The formulation becomes

$$\forall Y.([\![P]\!] \rightarrow \exists X.[\![Q]\!])$$

where X is the set of functionally determined free variables in Q, and Y is the union of the set of free variables in P and the set of non-functionally determined free variables in Q . The simplification of P can proceed unchanged. After decomposition, the formula looks like this:

$$\bigwedge_{\forall \vec{\phi_Z}, \vec{\phi_N}. \vec{\phi_Z} \uplus \vec{\phi_N} = \vec{\phi}} [\vec{\phi_Z} \mapsto \bot](\forall \vec{\alpha_N}, \vec{\alpha}. [\vec{\phi_N} \mapsto \vec{\alpha_N}](\llbracket F \rrbracket \rightarrow \exists X. \llbracket G \rrbracket))$$

We can use the equality

$$\begin{aligned} \exists \phi . p &= \exists \phi . (\phi = \bot \rightarrow p) \vee \exists \phi . (\phi \neq \bot \rightarrow p) \\ &= [\phi \mapsto \bot] p \vee \exists \alpha . [\phi \mapsto \alpha] p \quad \alpha \text{ new} \end{aligned}$$

to decompose each *subformula* of the form $\llbracket F \rrbracket \rightarrow \exists \vec{\phi}, \vec{\alpha}. \llbracket G \rrbracket$ into a disjunction of simpler cases, resulting in:

$$\begin{aligned} \llbracket F \rrbracket \rightarrow \exists \vec{\phi'}, \vec{\alpha'}. \llbracket G \rrbracket \quad &= \\ \bigvee\nolimits_{\forall \vec{\phi'_Z}, \vec{\phi'_N}. \vec{\phi'_Z} \uplus \vec{\phi'_N} = \vec{\phi'}} [\vec{\phi'_Z} \mapsto \bot] &(\llbracket F \rrbracket \rightarrow \exists \vec{\alpha'_N}, \vec{\alpha'}. [\vec{\phi'_N} \mapsto \vec{\alpha'_N}] \llbracket G \rrbracket) \end{aligned}$$

As explained in section 5.5.5, we can replace the universally quantified type variables by unique type constants. This leaves the existentially quantified type variables. Because there are no field variables, the predicate forms we can encounter are limited. Some of these contain no variables, and their truth value can immediately be determined. An example of this is `t` $\neq$ `t`. Similarly, others contain type variables, but their truth value can be determined without regard to the specific value of the type variable. An example of this sort of predicate is $\alpha \neq \bot$. The only remaining predicate forms are:

$$\alpha = c$$

$$c = \alpha$$

$$\alpha = \alpha'$$

Determining the truth or falsity of a conjunction of these is straightforward.

We are now ready to state the main result of this chapter:

Theorem 5.5.2. *If P is a well-formed row predicate set, then*

$$\forall \mathcal{I}.(\mathcal{I}[\![P]\!] \rightarrow \mathcal{I}[\![Q]\!]) \quad \leftrightarrow \quad P \text{ æ } Q$$

Proof. The algorithm consists of a series of transformations of the problem. We have shown that, at each step, the formulation of the problem before the transformation is equivalent to the formulation after the transformation. We summarize how we conclude this for each step:

1. **Slice into field predicates**.

   This is correct by Lemma 5.5.2.

2. **Simplify the antecedent**.

   This is correct by Proposition 5.5.8.

3. **Split**.

   This makes use of the logical equality given in section 5.5.4.

4. **Evaluate**.

   This relies on the correctness of substituting unique type constants for each free type variable, as discussed in section 5.5.5.

□

5.6 Satisfiability

If we have a value whose type is inferred to be $\forall \overrightarrow{\alpha}, \overrightarrow{\rho}.P \Rightarrow \tau$, we can only ultimately use this value if we can instantiate the variables $\overrightarrow{\alpha}, \overrightarrow{\rho}$ in such a way that P is satisfied. This assures us that we can never incorrectly instantiate these variables. However, it may be the case that there is no possible instantiation that satisfies P. For example, for the function

```
fun f x = { b = x & { a = 1 },
            c = x \ a }
```

we can infer the type scheme:

$$\forall \alpha, \rho.((\rho, a) \# a), (\rho \# a) \Rightarrow \{\rho, a : \alpha\} \rightarrow \{b : \{\rho, a : int, a : \alpha\}, c : \{\rho\}\}$$

We say that such a type scheme is *unsatisfiable*. The type system would allow us to write such a function, but we will never be able to use it. It is desirable from a language usability viewpoint for the compiler to proactively report this fact, because this is almost certainly an error on the part of the programmer. A warning at the point of definition of such a function would be preferable to an error at the point of use[2].

In this particular case, the type of the result is not well-formed, because field b of the return value has two fields labeled a. In addition, simplification of the first predicate would reveal the unsatisfiability of the type scheme

$$(\rho, a) \# a \quad \rightarrow \quad \{\rho \# a, a \# a\}$$

[2]This is an application of the well-known folk theorem which states that “a stitch in time saves nine”.

However, for more complex sets of predicates, this may not be the case. In the following example, we will posit the existence of a function `id2` whose type scheme is $\forall \alpha \,.\, \alpha \to \alpha \to \alpha$. Our example is:

```
fun unsat a b c d = { x = id2 ((a & b) & (c & d)) { l = 1 },
                      y = id2 (a & b) (c & d) }
```

The inferred type scheme for this function is:

$$\forall \rho_a, \rho_b, \rho_c, \rho_d \,.\, ((\rho_a, \rho_b, \rho_c, \rho_d) = (l : \texttt{int})),\ ((\rho_a, \rho_b) = (\rho_c, \rho_d))$$
$$\Rightarrow \{\rho_a\} \to \{\rho_b\} \to \{\rho_c\} \to \{\rho_d\} \to \{x : \{\rho_a, \rho_b, \rho_c, \rho_d\}, y : \{\rho_a, \rho_b\}\}$$

This type scheme is not satisfiable. To see this, note that according to the first predicate, $(\rho_a, \rho_b, \rho_c, \rho_d) = (l : \texttt{int})$, three of the four row variables must be empty. The remaining row variable has one field. Regardless of which row variable we pick to consist of the one field, the second predicate, $(\rho_a, \rho_b) = (\rho_c, \rho_d)$, cannot be satisfied.

In addition to entailment, it would be desirable to have a way of deciding satisfiability of a given predicate. The formulation of the satisfiability problem is similar to that of entailment. Specifically, if we have a type scheme $\forall Y.Q \Rightarrow \tau$, and a predicate context P, is there any substitution θ, whose domain is limited to Y, such that P satisfies θQ? The algorithm to check for satisfiability is a minor variation of the entailment algorithm. The two algorithms are the same until we do the transformation into first-order logic with equality. Instead of

$$\forall \vec{\phi}.(F_1 \to \forall \vec{\phi}'.F_2)$$

we have

$$\forall \vec{\phi}.(F_1 \to \exists \vec{\phi}'.F_2)$$

We define a solution to a closed formula $\exists \phi_1, \ldots, \phi_n.F$ to be a substitution $[\phi_i \mapsto \tau_i \mid i \in 1..n]$, and a principal solution to be a substitution θ such that each solution can be written $\theta' \circ \theta$ for an appropriate substitution θ'.

Proposition 5.6.1. *Let F be a well-formed field predicate set. Then the closed formula*

$$\exists \vec{\phi_Z}.\exists \vec{\phi_N}.(\bigwedge_{i \in Z} \phi_i = \bot) \wedge (\bigwedge_{i \in N} \phi_i \mathrel{\#} \bot) \wedge F$$

is either unsatisfiable, or has a principal solution.

Proof. Note that, since we have a closed formula, each free variable in F is either a member of $\vec{\phi_Z}$, in which case it is constrained to be $\bot$, or a member of $\vec{\phi_N}$, in which case it is constrained to not be $\bot$. We do a case analysis on the form of each conjunct in F:

1. $(\phi_1, \ldots, \phi_n) = \tau$

   (a) If $\{\phi\} = \{\phi_1, \ldots, \phi_n\} \cap \vec{\phi_N}$ for some ϕ, the predicate becomes $\phi = \tau$.

   (b) Otherwise, `Fail`.

2. $(\phi_1, \ldots, \phi_n) = (\phi'_1, \ldots, \phi'_m)$

   (a) If $\{\phi\} = (\{\phi_1, \ldots, \phi_n\} \cap \vec{\phi_N})$ and $\{\phi'\} = (\{\phi'_1, \ldots, \phi'_m\} \cap \vec{\phi_N})$ for some ϕ, ϕ', the predicate becomes $\phi = \phi'$.

   (b) If $\{\phi_1, \ldots, \phi_n\} \subseteq \vec{\phi_Z}$ and $\{\phi'_1, \ldots, \phi'_m\} \subseteq \vec{\phi_Z}$, then the predicate becomes `OK`.

   (c) Otherwise, `Fail`.

3. $\phi \mathrel{\#} \phi'$

   (a) If $\phi \in \vec{\phi_Z}$ or $\phi' \in \vec{\phi_Z}$, then the predicate reduces to `OK`.

   (b) Otherwise, `Fail`.

4. $(\phi_1, \ldots, \phi_n) \parallel$ `t`

   (a) If $|\{\phi_1, \ldots, \phi_n\} \cap \vec{\phi_N}| = 1$, then `OK`.

   (b) Otherwise, `Fail`.

5. $(\phi_1, \ldots, \phi_n) \parallel (\phi'_1, \ldots, \phi'_m)$

   (a) If $|\{\phi_1, \ldots, \phi_n\} \cap \vec{\phi_N}| = 1$, and $|\{\phi'_1, \ldots, \phi'_m\} \cap \vec{\phi_N}| = 1$, then the predicate reduces to `OK`.

   (b) If $\{\phi_1, \ldots, \phi_n\} \subseteq \vec{\phi_Z}$ and $\{\phi'_1, \ldots, \phi'_m\} \subseteq \vec{\phi_Z}$, then `OK`.

   (c) Otherwise, `Fail`.

After this reduction, we have a set of equations E which are each of one of two forms:

$$\begin{aligned} \phi &= \phi' \\ \phi &= \tau \end{aligned}$$

We can repeatedly simplify equations of these forms to the corresponding substitutions. This may result in equations of the form

$$\tau = \tau'$$

as in the example:

$$\rho_1 = \tau_1, \rho_2 = \tau_2, \rho_1 = \rho_2$$

These equations can be solved by unification. Because we do not allow recursive types, this process is guaranteed to terminate.

For each conjunct in our simplified formula,

$$[\vec{\phi_Z} \mapsto \bot](\forall \vec{\phi_N}.(\bigwedge_{i \in N} \phi_i \mp \bot) \to (F_1 \to \exists \vec{\phi'}.F_2))$$

we solve

$$[\vec{\phi_Z} \mapsto \bot]((\bigwedge_{i \in N} \phi_i \mp \bot) \wedge F_1)$$

If no solution exists, then the conjunct is satisfied. Otherwise, we have a principal solution θ. Let $\vec{\alpha} = \mathrm{TRV}(\theta)$. Then we test the formula

$$\forall \vec{\alpha}.[\phi_i \mapsto \tau_i \mid i \in N] \circ [\vec{\phi_Z} \mapsto \bot](\exists \vec{\phi'}.F_2)$$

If this is satisfiable, then the conjunct is as well. Otherwise, it is not, and the entire problem is not satisfiable. $\square$

5.7 Algorithm Analysis

5.7.1 NP-hardness

Proposition 5.7.1. *The predicate satisfiability problem is NP-hard.*

Proof. We reduce one-in-three SAT to the row predicate satisfiability problem.

One-in-three SAT:

Given: A set X of variables, and collection C of clauses over X such that each clause $c \in C$ has three literals. A literal is either a variable (written x) or a negated variable (written $\bar{x}$. We use z to range over literals.

Question: Is there a truth assignment for X such that each clause in C has exactly one true literal?

The row predicate satisfiability problem:

Given: A set RV of row variables, TV of type variables, and a collection P of predicates.

Question: Is there an assignment for RV and TV such that all predicates in P are satisfied?

The transformation:

$$\begin{aligned}
TV &= \varnothing \\
RV &= \textstyle\bigcup_{x \in X} \{\rho_x, \rho_{\bar{x}}\} \\
P &= \textstyle\bigcup_{x \in X} \{\rho_x \# \rho_{\bar{x}},\ (\rho_x, \rho_{\bar{x}}) \parallel l\} \\
&\cup \textstyle\bigcup_{(z_1 \vee z_2 \vee z_3) \in C} \{(\rho_{z_1}, \rho_{z_2}, \rho_{z_3}) \parallel l,\ \rho_{z_1} \# \rho_{z_2},\ \rho_{z_1} \# \rho_{z_3},\ \rho_{z_2} \# \rho_{z_3}\}
\end{aligned}$$

The idea is that each literal in the original problem becomes a row variable in the transformation. Specifically, the variable x is mapped to the row variable ρ_x, and the literal $\bar{x}$ is mapped to the row variable $\rho_{\bar{x}}$. A truth value of "true" in the original problem corresponds to a row value of $l : \tau$ for any τ, while a truth value of "false" corresponds to a row value of $\varnothing$. We ensure that exactly one of ρ_u and $\rho_{\bar{u}}$ is assigned $l : \tau$, and the other is assigned $\varnothing$ with the $(\rho_u, \rho_{\bar{u}}) \parallel l$ predicate. The corresponding $\rho_u \# \rho_{\bar{u}}$ ensures that the $(\rho_u, \rho_{\bar{u}})$ row is well-formed. The $(\rho_{z_1}, \rho_{z_2}, \rho_{z_3}) \parallel l$ predicate is a translation of the clause $z_1 \vee z_2 \vee z_3$, and the remaining row disjointness predicates ensure that $(\rho_{z_1}, \rho_{z_2}, \rho_{z_3})$ is well-formed. $\square$

5.7.2 Discussion

The NP-hardness is not a result of the number of fields in a row, but the fact that a row can be the union of an arbitrary number of row variables.

If a row cannot contain more than one row variable, the problem is in P. In this case, each simplified predicate would be of one of the forms:

$$\phi \mathrel{\#} \phi' \quad \text{or} \quad \phi \parallel \phi'$$

If each row occurring in each predicate is restricted to contain no more than two row variables, then we will have predicates of the forms:

$$\begin{aligned}(\phi_1, \phi_2) &= \tau \\ (\phi_1, \phi_2) &\parallel \tau \\ \phi &\mathrel{\#} \phi'\end{aligned}$$

Unfortunately, we can encode rows with more than 2 row variables in this framework. To do this, replace each 3-variable row (ϕ_1, ϕ_2, ϕ_3) by $(\phi_{(1,2)}, \phi_3)$, where $\phi_{(1,2)}$ is a fresh row variable, and add the predicate $(\phi_{(1,2)}, \phi'_{(1,2)}) = (\phi_1, \phi_2)$, $t \blacktriangleright \phi'_{(1,2)}$ and $t' \blacktriangleright \phi'_{(1,2)}$, where $t \neq t'$. To prevent this, we need to prohibit predicates of the form $(\phi_1, \phi_2) = (\phi'_1, \phi'_2)$. There seems to be no elegant way of achieving this restriction. One way is to allow record concatenation only when the resulting label set is statically known. This is a severe limitation.

Fortunately, there is hope that this is not a problem in practice, as experience with ML and Haskell has shown. These languages have type inference algorithms which are exponential [31, 33], but this behavior has mainly been observed on contrived examples. In fact, McAllester [35] proves that the running time is actually nearly linear for "reasonable" code.

Chapter 6

Compilation

We will show how to compile the record calculus into Core ML augmented with untyped tuples and tuple selection.

Unlike type systems for records, there has not been a great deal of work done in compilation methods for records. Notable exceptions are Ohori [40, 41], Remy [50], and Gaster and Jones [18].

6.1 Evidence

When compiling a language which uses qualified types, one important choice to be made is whether to ignore the predicates in the qualified types after type checking is complete, or to make use of them in the generated code. It is often the case that the latter choice leads to more efficient code. An example of how predicates can be transformed into run-time entities can be found in Haskell [45] compilers. See Hall, Hammond, Peyton Jones, and Wadler [19] for an exposition of this method. Consider the Haskell function:

```
eq3 a b c = a == b && b == c
```

The process of type inference will yield the type scheme of

$$\forall\alpha\,.\,\texttt{Eq}\;\alpha \Rightarrow \alpha \to \alpha \to \alpha \to \texttt{Bool}$$

This function takes three arguments and compares the first and last two for equality. In Haskell, one may define specialized equality functions for different types, so the system cannot use a single, predetermined equality function. The compilation process transforms the `Eq` α predicate into an additional parameter to the `eq3` function, which represents the correct equality function to be used. Pseudocode for the compiled function would be:

```
eq3 eq a b c = (eq a b) && (eq b c)
```

A call to `eq3` would pass in the correct equality function, depending on the instantiated type of the function. For example,

```
eq3 1 2 1
```

would be translated into

```
eq3 eqInt 1 2 1
```

where `eqInt` is the integer equality function.

A more complex example would be:

```
eq3 [1,2] [1,2] [1,2]
```

which requires the equality for lists of integers. The definition of that is:

```
(==) []     []     = True
(==) (x:xs) (y:ys) = x == y && xs == ys
```

```
(==) _      _      = False
```

This has the type scheme

$$\forall\alpha\,.\,\texttt{Eq}\;\alpha \Rightarrow [\alpha] \rightarrow [\alpha] \rightarrow \texttt{Bool}$$

and is translated into:

```
eqList eq []     []     = True
eqList eq (x:xs) (y:ys) = (eq x y) && eqList eq xs ys
eqList eq _      _      = False
```

Note that `eqList` by itself is not the equality function for lists of, e.g. `Int`s. To get the equality for lists of type `t`, we need to pass `eqList` the equality function for type `t`. Getting back to our last example, where we need the equality function for lists of `Int`s, the translation would be:

```
eq3 (eqList eqInt) [1,2] [1,2] [1,2]
```

Another example, and one closer to the subject of this dissertation, involves the record calculus described in [18]. In this system, a row is represented by a sequence of fields and at most one row variable. The single predicate form is $row\backslash l$, which denotes that row *row* does not contain a field with label l. A proposed compilation method translates records into vectors of values. The fields are arranged according to the lexicographic order of their labels. The labels themselves do not appear at runtime. In this system, the predicate $row\backslash l$ translates into an index which specifies where to insert a field whose label is l into the vector implementing a value of type $\{row\}$. Two of the operations supported are record extension:

$$(l = \_|\_) : \forall\alpha,\rho\,.\,\rho\backslash l \Rightarrow \alpha \times \{\rho\} \rightarrow \{l : \alpha|\rho\}$$

and record restriction:

$$(\_ - l) : \forall \alpha, \rho \,.\, \rho \backslash l \Rightarrow \{l : \alpha | \rho\} \rightarrow \{\rho\}$$

In the translation, we will use the following notations:

Notation	**Meaning**
`[ e`$_1$`, ..., e`$_n$` ]`	a record, represented as a list of elements `e`$_1$ through `e`$_n$
`a ++ b`	concatenation of lists `a` and `b`
`a[i]`	`i`'th element of list[1] `a`
`a[..i]`	list consisting of elements of `a`, from the first to the `i`th, inclusive
`a[i..]`	list consisting of elements of `a`, from the `i`th to the last, inclusive

As an example, consider the function

```
fun f (r1, r2) =
    if r1.old > 0
    then { new = 0 | r1 − old }
    else { new = r1.old | r2 }
```

This function has the type

$$\forall \rho \,.\, \rho \backslash old, \rho \backslash new \Rightarrow \{old : int, \rho\} \times \{\rho\} \rightarrow \{new : int, \rho\}$$

We can write the translation as:

```
fun f $i_{old}$ $i_{new}$ (r1, r2) =
```

[1]The first element is at index 0.

```
if r1[$i_{old}$] >0
then let val r = r1[..$i_{old-1}$] ++ r1[$i_{old+1}$..]
     in  r[..$i_{new-1}$] ++ [0] ++ r[$i_{new}$..]
     end
else r2[..$i_{new-1}$] ++ [r1[$i_{old}$]] ++ r2[$i_{new}$..]
```

The call

```
f { old = 1, oak = "strong", pun = "fore" }
  { oak = "tall", pun = "yore" }
```

would be translated into

```
f 1 0 ["strong", 1, "fore"] ["tall", "yore"]
```

and yield `[0, "strong", "fore"]`.

6.2 Compilation into tuples

Our compilation method translates each record value into *record tuple*. This is simply a tuple, which we will write enclosed in special braces, i.e. $\{\!|\,|\!\}$, to distinguish it from "regular" tuples. Specifically, each field value of the record will compile into an element of the tuple. The fields are arranged according to the lexicographic ordering of their labels. The labels themselves are dropped. For example, the record:

```
{ b = 1, d = "hello", a = 3.2 }
```

will compile into:

$$\{\!|\texttt{3.2}, \texttt{1}, \texttt{"hello"}|\!\}$$

Field selection will compile into tuple element selection, and predicates will compile into evidence parameters.

For this compilation method, the evidence for a row disjointness predicate will be a vector, i.e. a sequence of elements. v will range over vectors. $|v|$ is the number of elements in vector v, also known as the length of the vector. We will use the same notation for rows to denote the number of fields in the row. $|v|^s$ is the number of occurrences of element s in vector v. $v_1 \cdot v_2$ is the concatenation of vectors v_1 and v_2. Where clear from context, a vector element x will denote the vector whose single element is x. $v[i]$ denotes the ith element of v, where the first element of vector v is denoted by $v[1]$. $v[i..j]$ denotes a vector which consists of the ith through jth elements of v, inclusive. It must be the case that $1 \leqslant i \leqslant j \leqslant |v|$. We will sometimes write a vector as a sequence of optionally comma-separated elements enclosed by angle brackets (e.g. $\langle\, 1, 2, 3 \,\rangle$) in order to make clear where the vector begins and ends. $v_1 + v_2$ denotes the point-wise sum of vectors v_1 and v_2, which must be of the same length. For example, $\langle\, 3, 1, 0, 4 \,\rangle + \langle\, 1, 2, 1, 2 \,\rangle$ is $\langle\, 4, 3, 1, 6 \,\rangle$. The empty vector is denoted $\langle\,\rangle$. s^n denotes the vector composed of n instances of element s.

The evidence for a predicate of the form $row_1 \mathrel{\#} row_2$ is a vector of length $|row_1| + |row_2|$, where each element of the vector is either $\ltimes$ or $\rtimes$ (pronounced "left" and "right", respectively). This evidence is used to determine how to merge a value of type $\{row_1\}$ with a value of type $\{row_2\}$. The element $\ltimes$ indicates that the corresponding element of $\{row_1, row_2\}$ is taken from the left value, while the element $\rtimes$ indicates that it is taken from the right value.

For example, consider the following two records, along with their types and translations:

Record value	**Type**	**Translation**
`{ b = 1, d = "hello", a = 3.2 }`	$\{row_1\}$	$\{\!\mid \texttt{3.2}, \texttt{1}, \texttt{"hello"} \mid\!\}$
`{ f = "world", c = 42 }`	$\{row_2\}$	$\{\!\mid \texttt{42}, \texttt{"world"} \mid\!\}$

where $row_1 = (a : real, b : int, d : string)$, and $row_2 = (c : int, f : string)$.

The evidence for $row_1 \# row_2$ is the merge value $\langle \ltimes, \ltimes, \rtimes, \ltimes, \rtimes \rangle$. When used to concatenate the two record values, it specifies that the first, second, and fourth elements of the resulting value are taken from row_1, and the third and fifth are taken from row_2:

Merge value:	$\ltimes$	$\ltimes$	$\rtimes$	$\ltimes$	$\rtimes$
First record:	`3.2`	`1`		`"hello"`	
Second record:			`42`		`"world"`
Concatenated record:	`3.2`	`1`	`42`	`"hello"`	`"world"`

Predicates of the form $row_1 = row_2$ do not have evidence values associated with them. To see why, we need to consider what evidence values are used for. Informally, an evidence value v for a predicate π allows us to perform actions in the value domain analogous to what the predicate π allows us to do in the type domain. In the case of a row disjointness predicate, if we know that $row_1 \# row_2$, and that row_1 and row_2 are well-formed, then (row_1, row_2) is also a well-formed row. The corresponding "action" in the value domain is to concatenate a record of type $\{row_1\}$ with a record of type $\{row_2\}$, resulting in a record value of type $\{row_1, row_2\}$.

In the case of a row equality predicate $row_1 = row_2$ on the other hand, the "action" in the type domain is the ability to substitute a type $\{row_1\}$ for $\{row_2\}$ which is identical except that the corresponding sub-term is $\{row_2\}$. In the value domain, the evidence value would make it possible to substitute a value of the type $\{row_1\}$ for a value of type $\{row_2\}$. This could be accomplished by having the evidence value tell us how to perform the translation from $\{row_1\}$ to $\{row_2\}$. However, we already know how to do this; in every case where we know $row_1 = row_2$, this translation is the identity function! To be pedantic, we could construct evidence values which are identity functions, and apply them to values whenever the translation from $\{row_1\}$ to $\{row_2\}$ is necessary. This, however, serves no practical purpose.

For the same reason, predicates of the form $lset_1 \parallel lset_2$ also do not have evidence values associated with them.

The syntax of the target language is given in Figure 6.1.

The language is stratified into two levels: "regular" expressions E, and "evidence" expressions M. Regular expressions may contain evidence expressions as sub-terms, but not the converse.

E	$::=$	x		variable
	$\mid$	$\lambda x.E$		lambda abstraction
	$\mid$	$E\,E$		application
	$\mid$	$\textbf{let}\ x = E\ \textbf{in}\ E$		let expression
	$\mid$	$\{\!\mid E, \ldots, E \mid\!\}$		record tuple
	$\mid$	$\textbf{merge}\, M\, (E, E)$		merge
	$\mid$	$\textbf{extract}^{\ltimes}\, M\, E$	$\mid\ \textbf{extract}^{\rtimes}\, M\, E$	left/right extract
	$\mid$	$\textbf{index}^{\ltimes} M E$	$\mid\ \textbf{index}^{\rtimes} M E$	left/right index
	$\mid$	$\lambda m.E$		merge abstraction
	$\mid$	$E\,M$		merge application
M	$::=$	m		merge variable
	$\mid$	$\langle\, s, \ldots, s\, \rangle$		literal merge
	$\mid$	$\textbf{flip}\, M$		merge flip
	$\mid$	$\textbf{xtrct}^{\ltimes}\, M\, M$	$\mid\ \textbf{xtrct}^{\rtimes}\, M\, M$	left/right merge extract
	$\mid$	$\textbf{mrg}^{\ltimes}\, (M, M)$	$\mid\ \textbf{mrg}^{\rtimes}\, (M, M)$	left/right merge merge
s	$::=$	$\ltimes \mid \rtimes$		merge elements

Figure 6.1: Target language syntax

Simple types and qualified types are merged into the same syntactic entity; type schemes remain separate:

$$\begin{array}{rcl} \tau & ::= & t \\ & | & \alpha \\ & | & \tau \to \tau \\ & | & \{row\} \\ & | & \pi \Rightarrow \tau \end{array}$$

$$\begin{array}{rcl} \sigma & ::= & \tau \\ & | & \forall \alpha \,.\, \sigma \end{array}$$

The type rules for the target language are in Figures 6.2 and 6.3. Note that the target language does not enjoy the principal type property, because of the record rule. Because the record labels are omitted, a record tuple may have one of many types, differing in the labels assigned to each element of the record tuple. For example, the value

$$\{\!|\, \texttt{1}, \texttt{"hi"} \,|\!\}$$

can be assigned any of the types:

$$\{a : \mathit{int}, b : \mathit{string}\}$$
$$\{c : \mathit{int}, d : \mathit{string}\}$$
$$\{a : \mathit{int}, c : \mathit{string}\}$$

It cannot be assigned the type

$$\{b : \mathit{int}, a : \mathit{string}\}$$

however, because of the restriction that the labels must be in ascending order. The restriction on the type of the record tuple cannot be expressed in a single type (or

$$(\mathbf{var}) \quad \frac{x : \sigma \in A \quad (P \Rightarrow \tau) \leqslant \sigma}{P|A \overset{T}{\vdash} x : \tau}$$

$$(\mathbf{app}) \quad \frac{P|A \overset{T}{\vdash} E_1 : \tau' \to \tau \quad P|A \overset{T}{\vdash} E_2 : \tau'}{P|A \overset{T}{\vdash} (E_1\, E_2) : \tau}$$

$$(\mathbf{abs}) \quad \frac{P|A\backslash x, x : \tau_1 \overset{T}{\vdash} E : \tau_2}{P|A \overset{T}{\vdash} (\lambda x.E) : \tau_1 \to \tau_2}$$

$$(\mathbf{let}) \quad \frac{P|A \overset{T}{\vdash} E_1 : \tau' \quad P'|A\backslash x, x : \sigma \overset{T}{\vdash} E_2 : \tau}{P'|A \overset{T}{\vdash} (\mathbf{let}\ x = E_1\ \mathbf{in}\ E_2) : \tau} \quad \sigma = Gen(A, P \Rightarrow \tau')$$

$$(\mathbf{merge\text{-}abs}) \quad \frac{P\backslash m, m : \pi|A \overset{T}{\vdash} E : \tau}{P|A \overset{T}{\vdash} \lambda m.E : \pi \Rightarrow \tau}$$

$$(\mathbf{merge\text{-}app}) \quad \frac{P|A \overset{T}{\vdash} E : \pi \to \tau \quad P \Vdash M : \pi}{P|A \overset{T}{\vdash} E\, M : \tau}$$

Figure 6.2: Target language type rules (non-records)

even any finite disjunction of types) in this system.

$$\textbf{(record)}\quad \frac{P|A \overset{T}{\vdash} E_i : \tau_1 \quad \forall i \in 1..n}{P|A \overset{T}{\vdash} \{\!|E_1, \ldots, E_n|\!\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}} \quad l_1 < l_2 < \ldots < l_n$$

$$\textbf{(merge)}\quad \frac{P|A \overset{T}{\vdash} E_1 : \{row_1\} \quad P|A \overset{T}{\vdash} E_2 : \{row_2\} \quad P \Vdash M : row_1 \# row_2}{P|A \overset{T}{\vdash} \mathbf{merge}\, M\, (E_1, E_2) : \{row_1, row_2\}}$$

$$\textbf{(extract-left)}\quad \frac{P|A \overset{T}{\vdash} E : \{row_1, row_2\} \quad P \Vdash M : row_1 \# row_2}{P|A \overset{T}{\vdash} \mathbf{extract}^{\ltimes}\, M\, E : \{row_1\}}$$

$$\textbf{(extract-right)}\quad \frac{P|A \overset{T}{\vdash} E : \{row_1, row_2\} \quad P \Vdash M : row_1 \# row_2}{P|A \overset{T}{\vdash} \mathbf{extract}^{\rtimes}\, M\, E : \{row_2\}}$$

$$\textbf{(index-left)}\quad \frac{P|A \overset{T}{\vdash} E : \{row, l : \tau\} \quad P \Vdash M : l \# row}{P|A \overset{T}{\vdash} \mathbf{index}^{\ltimes} M E : \tau}$$

$$\textbf{(index-right)}\quad \frac{P|A \overset{T}{\vdash} E : \{row, l : \tau\} \quad P \Vdash M : row \# l}{P|A \overset{T}{\vdash} \mathbf{index}^{\rtimes} M E : \tau}$$

Figure 6.3: Target language type rules for records

We will use V to range over regular values (i.e. completely evaluated expressions E). V is a subset of E. Specifically,

$$\begin{array}{rcll} V & ::= & x & \text{variable} \\ & \mid & \lambda x.E & \text{lambda abstraction} \\ & \mid & \{\!| V, \ldots, V |\!\} & \text{record tuple} \\ & \mid & \lambda m.E & \text{merge abstraction} \end{array}$$

Similarly, W ranges over merge values, which are:

$$\begin{array}{rcll} W & ::= & m & \text{merge variable} \\ & \mid & \langle\, s, \ldots, s \,\rangle & \text{literal merges} \end{array}$$

Semantics of the familiar constructs is standard; we follow the style in Wright and Felleisen [61]. We need to define several auxiliary functions:

We define $\overline{\ltimes}$ and $\overline{\rtimes}$ to be $\rtimes$ and $\ltimes$, respectively, and extend this definition in the natural way to vectors of these elements.

Figures 6.8 and 6.9 defines a pair of relations we call *notions of reduction*. The discerning reader will recall that we previously defined notions of reduction for expressions in the source language in Chapter 4, on page 29. Here, we define the analogous relation on both target language "regular" expressions and target language evidence expressions.

In Chapter 4, we needed to define an evaluation context in order to specify in what order subexpressions were evaluated. For the target language, because the expressions are stratified into two levels, we also need to take this into consideration when defining the evaluation context. Firstly, we split the evaluation context into two contexts: $\mathcal{E}$ and $\mathcal{M}$. We also need to expand the standard context syntax. The target language has two distinct sorts of terms: "ordinary" expressions, and merge

$$
\begin{array}{rcl}
\mathcal{E} & ::= & [\,]^{E} \mid \mathcal{E}\, e \mid V\, \mathcal{E} \mid \mathbf{let}\ x = \mathcal{E}\ \mathbf{in}\ e \mid V\, \mathcal{M} \\
& \mid & \mathbf{merge}\, \mathcal{M}\, (E, E) \mid \mathbf{merge}\, W\, (\mathcal{E}, E) \mid \mathbf{merge}\, W\, (V, \mathcal{E}) \\
& \mid & \mathbf{extract}^{s}\, \mathcal{M}\, E \mid \mathbf{extract}^{s}\, W\, \mathcal{E} \\
& \mid & \mathbf{index}^{s} \mathcal{M} E \mid \mathbf{index}^{s} W \mathcal{E} \\
\mathcal{M} & ::= & [\,]^{M} \\
& \mid & \mathbf{mrg}^{s}\, (\mathcal{M}, M) \mid \mathbf{mrg}^{s}\, (W, \mathcal{M}) \\
& \mid & \mathbf{flip}\, \mathcal{M} \\
& \mid & \mathbf{xtrct}^{s}\, \mathcal{M}\, M \mid \mathbf{xtrct}^{s}\, W\, \mathcal{M}
\end{array}
$$

Figure 6.4: Target language evaluation contexts

expressions. Ordinary expressions may contain both ordinary and merge expressions as subterms, but merge expressions may not contain ordinary expressions as subterms. We distinguish the sort of subterm in an expression evaluation context $\mathcal{E}$ by superscripting the brackets with the sort of the subterm: either E (for ordinary expressions) or M (for merge expressions). We may leave off the superscript where it is obvious from context. The two contexts are defined in Figure 6.4.

As before, we define the relation $\longmapsto$ to be the union of:

$$
\begin{array}{lclccc}
\mathcal{E}[E_1]^{E} & \longmapsto & \mathcal{E}[E_2]^{E} & \text{iff} & E_1 & \longrightarrow & E_2 \\
\mathcal{E}[M_1]^{M} & \longmapsto & \mathcal{E}[M_2]^{M} & \text{iff} & M_1 & \longrightarrow & M_2
\end{array}
$$

The reflexive and transitive closure of $\longmapsto$ is $\longmapsto\!\!\!\!\rightarrow$.

Finally, the evaluation function `eval` is defined for closed expressions:

$$\texttt{eval}(E) = V \quad \text{iff} \quad E \longmapsto\!\!\!\!\rightarrow V$$

We explain the merge, extract, and index operations as follows. $\mathbf{merge}\,_{\_}\,(\_,\_)$ takes a merge value and two record values and merges them according to the merge value. For example,

$$\mathbf{merge}\langle \ltimes \ltimes \rtimes \ltimes \rtimes \rangle(\{\!|1,2,3|\!\},\{\!|\texttt{'a'},\texttt{'b'}|\!\}) \quad\longrightarrow\quad \{\!|1,2,\texttt{'a'},3,\texttt{'b'}|\!\}$$

The left (right) extract operator takes a merge value and a record value, and extracts the elements from the record value that correspond to the $\ltimes$ ($\rtimes$) elements of the merge vector. For example,

$$\mathbf{extract}^{\ltimes}\langle \ltimes \ltimes \rtimes \ltimes \rtimes \rangle\{\!|1,2,3,4,5|\!\} \quad\longrightarrow\quad \{\!|1,2,4|\!\}$$
$$\mathbf{extract}^{\rtimes}\langle \ltimes \ltimes \rtimes \ltimes \rtimes \rangle\{\!|1,2,3,4,5|\!\} \quad\longrightarrow\quad \{\!|3,5|\!\}$$

Finally, the index operator is used to extract a single field from a record. The restriction on the merge argument ensures that only one filed is chosen. One can think of it as an extract (with the same arguments), which would result in a record with a single component, followed by an "unwrapping" of the record constructor to get the field within.

6.3 Type Soundness

The evidence expression in the conclusion of the **#-compatible-**$\|$ rule is the same as the one in one of its premises; only its type changes. We will call any instance of this rule *non-productive*. Instances of all other rules are unsurprisingly called *productive*. A derivation is called productive (non-productive) if its last rule is productive (non-productive). A derivation is *purely productive* if it consists solely of productive rules. Note that a productive derivation may include instances of non-productive rules.

Proposition 6.3.1. *Any derivation of $P \Vdash M : lset_1 \# lset_2$ can be rewritten as a derivation where:*

1. *Each productive rule is followed by exactly one non-productive rule.*

2. *Each non-productive rule is either the last rule in the derivation, or is followed by a productive rule.*

We will make extensive use of this proposition in proving various properties of these type rules.

Lemma 6.3.1 (Subject Reduction for evidence). *If there is a derivation of $P \Vdash M_1 : lset_1 \# lset_2$ where P is satisfiable, and $M_1 \longrightarrow M_2$, then there is a derivation of $P \Vdash M_2 : lset_1 \# lset_2$.*

Lemma 6.3.2 (Subject Reduction – target version). *For a target language expression E_1, where $P|A \overset{\mathcal{T}}{\vdash} E_1 : \tau$, for some P, A, and τ, and $E_1 \longrightarrow E_2$, then $P|A \overset{\mathcal{T}}{\vdash} E_2 : \tau$.*

Just as in Section B.1, we need to define what a faulty expression is, and establish a crucial lemma. For the target language, there are a number of faulty expressions. These are summarized in Figures 6.5 and 6.6.

Lemma 6.3.3 (Uniform evaluation – target version). *For closed target language expression E, if there is no E' such that $E \longmapsto\!\!\!\!\rightarrow E'$ and E' is faulty, then either E diverges, or $E \longmapsto\!\!\!\!\rightarrow V$.*

Theorem 6.3.1 (Type soundness). *If $\overset{\mathcal{T}}{\vdash} E : \tau$, then $E \longmapsto\!\!\!\!\rightarrow V$ and $\overset{\mathcal{T}}{\vdash} V : \tau$.*

The compilation rules are a modification of the type rules given in Figure 4.12. This is done by augmenting the type judgments with a compilation result, and also augmenting predicate judgments with an evidence result. We write

$$P|A \vdash E : \tau \rightsquigarrow E'$$

$c\,V$	where $\delta(c, V)$ is not defined
$V_1\,V_2$	unless V_1 is of the form $\lambda x.E_1$
$\mathbf{merge}\,M\,(V_1, V_2)$	unless V_1 and V_2 are record literals such that $\|V_1\| = \|M\|^{\ltimes}$ and $\|V_2\| = \|M\|^{\rtimes}$
$\mathbf{extract}^{\ltimes}\,W\,V$ *and* $\mathbf{extract}^{\rtimes}\,W\,V$	unless V is a record literal where $\|V\| = \|W\|$
$\mathbf{index}^{\ltimes}\,M E$ *and* $\mathbf{index}^{\rtimes}\,M E$	unless V is a record literal where $\|V\| = \|W\|$

Figure 6.5: Faulty target expressions

$\mathbf{xtrct}^{s}\,W_1\,W_2$	unless $W_1 = \langle\, s_{11}, \ldots, s_{1n}\,\rangle$, $W_2 = \langle\, s_{21}, \ldots, s_{2m}\,\rangle$, and $\|W_2\|^{s} = \|W_1\|$
$\mathbf{mrg}^{s}\,(W_1, W_2)$	unless $W_1 = \langle\, s_{11}, \ldots, s_{1n}\,\rangle$, $W_2 = \langle\, s_{21}, \ldots, s_{2m}\,\rangle$, and $\|\,\mathit{spans}^{s}(W_1)\| = \|\,\mathit{spans}^{s}(W_2)\|$

Figure 6.6: Faulty evidence expressions

to mean that expression E has type τ under the type assumption set A and predicate set P, and compiles to expression E' in the target language. Figure 6.7 has the compilation rules.

Predicate sets and predicate entailment judgments are also augmented. Each element of a predicate set will now have the form:

$$m : \pi$$

where m is the evidence variable for π.

Even though we do not use any evidence values for row or labelset equality, we will, where convenient, use the same notational form for uniformity. Conversely,

$$(\mathbf{var})\qquad \frac{x : \forall\vec{\alpha},\vec{\rho}\,.\,Q \Rightarrow \tau' \in A \quad P \Vdash M : [\vec{\alpha} \mapsto \vec{\tau}, \vec{\rho} \mapsto \vec{row}]Q}{P|A \vdash x : [\vec{\alpha} \mapsto \vec{\tau}, \vec{\rho} \mapsto \vec{row}]\tau' \rightsquigarrow x\, M}$$

$$(\mathbf{app})\qquad \frac{P|A \vdash E_1 : \tau' \to \tau \rightsquigarrow E_1' \quad P|A \vdash E_2 : \tau' \rightsquigarrow E_2'}{P|A \vdash (E_1\; E_2) : \tau \rightsquigarrow (E_1'\; E_2')}$$

$$(\mathbf{abs})\qquad \frac{P|A\backslash x, x : \tau_1 \vdash E : \tau_2 \rightsquigarrow E'}{P|A \vdash (\lambda x.E) : \tau_1 \to \tau_2 \rightsquigarrow (\lambda x.E')}$$

$$(\mathbf{let})\qquad \frac{\begin{array}{c} m : P'|A \vdash E_1 : \tau' \rightsquigarrow E_1' \\ P|A\backslash x, x : \sigma' \vdash E_2 : \tau \rightsquigarrow E_2' \end{array} \quad \sigma' = \mathit{Gen}(A, P' \Rightarrow \tau')}{P|A \vdash (\mathbf{let}\ x = E_1\ \mathbf{in}\ E_2) : \tau \rightsquigarrow (\mathbf{let}\ x = \lambda m.E_1'\ \mathbf{in}\ E_2')}$$

$$(\mathbf{record})\qquad \frac{P|A \vdash E_i : \tau_i \rightsquigarrow E_1' \quad \forall i \in 1..n \qquad \forall i, j \in 1..n\,.\, i < j \to l_i < l_j}{P|A \vdash \{l_1 = E_1, \ldots, l_n = E_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \rightsquigarrow \{\!|E_1', \ldots, E_n'|\!\}}$$

$$(\mathbf{sel})\qquad \frac{P|A \vdash E : \{l : \tau, row\} \rightsquigarrow E' \quad P \Vdash M : row \mathrel{\#} l}{P|A \vdash E.l : \tau \rightsquigarrow \mathbf{index}^{\rtimes} M E'}$$

$$(\mathbf{extract})\qquad \frac{P|A \vdash E : \{l : \tau, row\} \rightsquigarrow E' \quad P \Vdash M : row \mathrel{\#} l}{P|A \vdash E\backslash l : \{row\} \rightsquigarrow \mathbf{extract}^{\ltimes}\; M\, E'}$$

Figure 6.7: Compilation rules

when we are not interested in the evidence for a predicate, we may omit it and use the old form instead.

A predicate entailment judgment is written:

$$P \Vdash M : \pi$$

which denotes that P entails predicate π with evidence M.

Theorem 6.3.2 (Type Preservation). *If $P|A \vdash E : \eta \rightsquigarrow E'$, then using the target language type rules, $P|A \overset{T}{\vdash} E' : \eta$.*

Proof. By induction on the structure of derivations. For each rule, the result follows immediately from the induction assumptions. □

We wish to also show that translation preserves the semantics of terms as well as their types. For base types, we would like the source and target values to be identical. For records, the essential difference between source and target is that the target value has no labels and the fields are ordered. We need to lift this equivalence to all types, qualified types, and type schemes. To do this, we use the notion of *logical relations*; the development is similar to that of Ohori's in [40, 41].

We define a family of relations $\mathcal{R}$, indexed by type, which relate source and target terms of that type. We denote the set of source terms of type τ by $\mathit{Term}^{\tau}_{\mathit{Src}}$ and the set of target terms by $\mathit{Term}^{\tau}_{\mathit{Tgt}}$. These are defined:

$$\begin{aligned}\mathit{Term}^{\tau}_{\mathit{Src}} &= \{E \mid \ \vdash E : \tau\} \\ \mathit{Term}^{\tau}_{\mathit{Tgt}} &= \{E \mid \ \vdash E : \tau\}\end{aligned}$$

We cannot use this definition for qualified types or type schemes, because there are no source judgements which state that an expression is of some qualified type or type scheme. Therefore, we will define these terms somewhat differently. For qualified types, we define

$$\begin{aligned} \mathit{Term}_{\mathit{Src}}^{\pi \Rightarrow \tau} &= \{E \mid \pi | \varnothing \vdash E : \tau\} \\ \mathit{Term}_{\mathit{Tgt}}^{\pi \Rightarrow \tau} &= \{E \mid \ \vdash E : \pi \Rightarrow \tau\} \end{aligned}$$

and for type schemes

$$\begin{aligned} \mathit{Term}_{\mathit{Src}}^{\forall \alpha . \sigma} &= \{E \mid \forall \tau . E \in \mathit{Term}_{\mathit{Src}}^{[\alpha \mapsto \tau]\sigma}\} \\ \mathit{Term}_{\mathit{Tgt}}^{\forall \alpha . \sigma} &= \{E \mid \forall \tau . E \in \mathit{Term}_{\mathit{Src}}^{[\alpha \mapsto \tau]\sigma}\} \end{aligned}$$

Let $\mathcal{R}$ be a family of binary relations $\{\mathcal{R}^{ty} \subseteq \mathit{Term}_{\mathit{Src}}^{ty} \times \mathit{Term}_{\mathit{Tgt}}^{ty}\}$, where ty is either a type, qualified type, or a type scheme. We define $\mathcal{R}$ by induction on ty:

$$\begin{aligned} (E, E') \in \mathcal{R}^{ty} \quad & \text{where } E \longmapsto\!\!\!\!\!\rightarrow V \text{ and } E' \longmapsto\!\!\!\!\!\rightarrow V', \\ & \text{if one of the following holds:} \end{aligned}$$

1. $ty = t$, then $V = V'$.

2. $ty = \tau_1 \rightarrow \tau_2$, and, for any $(E_0, E_0') \in \mathcal{R}^{\tau_1}$, $(V\, E_0, V'\, E_0') \in \mathcal{R}^{\tau_2}$.

3. $ty = \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$, $V = \{l_1 = V_1, \ldots, l_n = V_n\}$, $V' = \{\!|V_1', \ldots, V_n'|\!\}$, and $(V_i, V_i') \in \mathcal{R}^{\tau_i}$ for $1 \leqslant i \leqslant n$.

4. $ty = \pi \Rightarrow \tau$, and, for all M such that $\Vdash M : \pi$, $(V, V'\, M) \in \mathcal{R}^{\tau}$.

5. $ty = \forall \alpha \,.\, \eta$, and for all τ, $(V, V') \in \mathcal{R}^{[\alpha \mapsto \tau]\eta}$.

A *value environment* is a mapping from variables to values. Specifically, a source value environment E^S maps variables to source values. A target value environment E^T maps variables to target values. We extend target value environments to a mapping from both regular and evidence variables to regular and evidence values, respectively. We write E^T_P for the target value environment whose domain is extended to the evidence variables in predicate set P and where $\Vdash \mathsf{E}^T_P(m) : \pi$ iff $m : \pi \in P$. We extend $\mathcal{R}$ to pairs of value environments, where $(\mathsf{E}^S, \mathsf{E}^T) \in \mathcal{R}^A$ iff for any $x : \sigma \in A$, $(\mathsf{E}^S(x), \mathsf{E}^T(x)) \in \mathcal{R}^\sigma$. We use $\mathsf{E}^S(E)$ to indicate the result of replacing each free variable x in E by $\mathsf{E}^S(x)$, and similarly for target environments.

We can now state our theorem:

Theorem 6.3.3 (Semantics Preservation). *If $P|A \vdash E : \tau \rightsquigarrow E'$, then for any ground substitution θ such that $\Vdash \theta P$, and for any pair of value environments $(\mathsf{E}^S, \mathsf{E}^T_P) \in \mathcal{R}^{\theta A}$, $(E, E') \in \mathcal{R}^\tau$.*

We cannot easily augment the entailment algorithm described in section 5.5 with evidence. Instead, we take a different approach, described in section 6.4.

Figure 6.10 gives the implementation of various possible record operations.

6.4 Predicate entailment with evidence

In chapter 5, we presented an algorithm which determines, given sets of predicates P and Q, whether $P \Vdash Q$. Unfortunately, we cannot use this algorithm to generate evidence values, because the algorithm uses the methods of classical as opposed to intuitionistic logic. Intuitionistic logic grew out out the philosophy of intuitionism first espoused by Brouwer [5, 6].

An exemplar of the difference between classical and intuitionistic logic is the law of the excluded middle:

$$A \vdash p \vee \neg p$$

$$
\begin{array}{lcl}
(\lambda x.E)V & \longrightarrow & [x \mapsto V]E \\
\textbf{let}\ x = V\ \textbf{in}\ E & \longrightarrow & [x \mapsto V]E \\
(\lambda m.E)W & \longrightarrow & [m \mapsto W]E \\
\textbf{extract}^{s}\, W\, \{\!|V_1, \ldots, V_n|\!\} & \longrightarrow & \{\!|V_{i_1}, \ldots, V_{i_m}|\!\} \\
\quad \text{where } W[j] = s \text{ iff } j \in \{i_1, \ldots, i_m\} & & \\
\quad \text{if } |W| = n & & \\
\textbf{merge}\, W\, (\{\!|V_1, \ldots, V_n|\!\}, \{\!|V'_1, \ldots, V'_m|\!\}) & \longrightarrow & \{\!|V''_1, \ldots, V''_{n+m}|\!\} \\
\quad \text{where } V''_i = \begin{cases} V_{|W[1..i]|^{\ltimes}} & \text{if } W[i] = \ltimes \\ V'_{|W[1..i]|^{\rtimes}} & \text{if } W[i] = \rtimes \end{cases} & & \\
\quad \text{if } |W| = n + m & & \\
\textbf{index}^{s} W \{\!|V_1, \ldots, V_n|\!\} & \longrightarrow & V_i \\
\quad \text{where } W[i] = s \text{ and } W[j] = \overline{s} \text{ for all } j \neq i & & \\
\quad \text{if } |W| = n & &
\end{array}
$$

Figure 6.8: Notions of reduction for "regular" expressions

This rule is present in classical logic, but not in intuitionistic logic. The philosophical difference between the two camps is that in classical logic, the raison d'être[2] of a proof is to establish the truth of a formula, while in constructive logic, its purpose is to construct a mathematical object that exemplifies, or bears witness to, the formula. Specifically, in classical logic, in order to prove that a formula is always true, we need to show that it is true for all possible instantiations of the free variables. In constructive logic, we construct a proof object that computes a *witness* value that shows the formula is true. For example, to constructively prove

[2]Or "reason deeter", as I like to pronounce it.

$$
\begin{array}{lcl}
\mathbf{flip}(W) & \longrightarrow & \overline{W} \\[1ex]
\mathbf{mrg}^s\,(\ltimes^{i_1}, \ltimes^{i_2}) & \longrightarrow & \ltimes^{i_1+i_2} \\
\mathbf{mrg}^s\,(\rtimes^{i_1}, \rtimes^{i_2}) & \longrightarrow & \rtimes^{i_1+i_2} \\
\mathbf{mrg}^s\,(W_1, W_2) & \longrightarrow & \mathit{spans}^{s^{-1}}(\mathit{spans}^s(W_1) + \mathit{spans}^s(W_2)) \\
 & & \text{if } |\,\mathit{spans}^s(W_1)| = |\,\mathit{spans}^s(W_2)| \\[1ex]
\mathbf{xtrct}^s\,W_1\,W_2 & \longrightarrow & \mathit{merge\text{-}extract}^s\,(W_1, W_2) \\
 & & \text{if } |W_2|^s = |W_1|
\end{array}
$$

where we define *spans*^{s} and *merge-extract*^{s} as follows:

$$
\begin{array}{lcl}
\mathit{spans}^s(s^n \cdot \overline{s} \cdot v) & = & n \cdot \mathit{spans}^s(v) \\
\mathit{spans}^s(s^n) & = & n
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{merge\text{-}extract}^s\,(W,\, \overline{s} \cdot W') & = & \overline{s} \cdot (\mathit{merge\text{-}extract}^s\,(W,\, W')) \\
\mathit{merge\text{-}extract}^s\,(s \cdot W,\, s \cdot W') & = & s \cdot (\mathit{merge\text{-}extract}^s\,(W,\, W')) \\
\mathit{merge\text{-}extract}^s\,(\overline{s} \cdot W,\, s \cdot W') & = & \mathit{merge\text{-}extract}^s\,(W,\, W') \\
\mathit{merge\text{-}extract}^s\,(\langle\,\rangle, \langle\,\rangle) & = & \langle\,\rangle
\end{array}
$$

Figure 6.9: Notions of reduction for evidence expressions

Operation	:	**Type**
	=	**Implementation**
$\_.l$	:	$\forall\alpha,\rho\,.\,\rho \# l \Rightarrow \{\rho, l : \tau\} \to \alpha$
	=	$\mathbf{index}^{\rtimes}$
$\_\backslash l$	:	$\forall\alpha,\rho\,.\,\rho \# l \Rightarrow \{\rho, l : \tau\} \to \{\rho\}$
	=	$\mathbf{extract}^{\ltimes}$
$\_\,\&\,\_$	:	$\forall\rho_1,\rho_2\,.\,\rho_1 \# \rho_2 \Rightarrow \{\rho_1\} \times \{\rho_2\} \to \{\rho_1,\rho_2\}$
	=	$\mathbf{merge}$
$\_\,\lvert\&\rvert\,\_$	:	$\forall\rho_1,\rho_2,\rho_3\,.\,(\rho_1 \# \rho_2,\ \rho_2 \# \rho_3,\ \rho_1 \# \rho_3)$ $\quad\Rightarrow \{\rho_1,\rho_2\} \times \{\rho_2,\rho_3\} \to \{\rho_1,\rho_2,\rho_3\}$
	=	$\lambda m_{12}.\lambda m_{23}.\lambda m_{13}.$ $\quad\lambda(x_{12},x_{23})\,.\,\mathbf{merge}\,(\mathbf{mrg}^{\rtimes}\,(m_{12},m_{13}))\,(\mathbf{extract}^{\rtimes}\,m_{12}\,x_{12},x_{23})$
$\_\,\&\&\,\_$	:	$\forall\rho_1,\rho_2,\rho_2',\rho_3\,.\,(\rho_1 \# \rho_2, \rho_2 \parallel \rho_2', \rho_2 \# \rho_3, \rho_1 \# \rho_3)$ $\quad\Rightarrow \{\rho_1,\rho_2'\} \times \{\rho_2,\rho_3\} \to \{\rho_1,\rho_2,\rho_3\}$
	=	$\lambda m_{12}.\lambda m_{23}.\lambda m_{13}.$ $\quad\lambda(x_{12},x_{23})\,.\,\mathbf{merge}\,(\mathbf{mrg}^{\rtimes}\,(m_{12},m_{13}))\,(\mathbf{extract}^{\rtimes}\,m_{12}\,x_{12},x_{23})$
`thin(_)`	:	$\forall\rho,\rho'\,.\,\rho' \blacktriangleright \rho \Rightarrow \{\rho'\} \to \{\rho\}$
	=	$\mathbf{extract}^{\rtimes}$

Figure 6.10: Record operations

$p \wedge q$, we need a proof of p and a proof of q. For $p \vee q$, we need a proof of either p or q, and an indication of which sub-proposition it is a proof of. Clearly, the rule of excluded middle cannot be proven in this way, because we do not know which of two alternatives is in reality true. The $\rightarrow$ connective is particularly interesting. To prove $p \rightarrow q$, we need a procedure that transforms a proof of p into a proof of q.

Another example of the difference between classical and constructive logic involves the treatment of double negation. In classical logic, both of the following are axioms,

$$p \implies \neg\neg p$$
$$\neg\neg p \implies p$$

whereas only the first is valid in constructive logic. To see this, it is necessary to understand that negation in constructive logic has a different meaning than the familiar classical one. In classical logic, $\neg p$ is true if the formula is false; in constructive logic, $\neg p$ is true if it is not possible to prove p. A double negative applied to a formula p then, means that it is impossible to prove that a proof of p is impossible. The first rule above makes sense in this interpretation: if we have a proof of p, it is definitely impossible to prove that a proof of p is impossible. However, the second rule is not justified. Just because we have a proof that it is impossible to prove that a proof of p is impossible does not automatically mean that we have a proof of p.

Strictly more formulas are provable in classical logic than in constructive logic. In exchange for this, however, constructive proofs have more information than classical proofs, because they show not only that a formula is true, but also how to construct a *witness* of it, i.e., a mathematical object that exemplifies the truth of the formula. It is precisely this additional information that we will require for the compilation of our record calculus.

In our case, one step of our entailment algorithm which is not constructive is the transformation from row predicates to field predicates. We first show an example where the transformation is not a problem.

If we know, a priori, that there are exactly three labels in a program, denoted $\{\mathbf{l}(1), \mathbf{l}(2), \mathbf{l}(3)\}$, and we have a predicate $lset_1 \# lset_2$, this predicate will have three slices, with an evidence value associated with each slice. The evidence for each slice will either be an empty vector or a single $\ltimes$ or $\rtimes$ element. We use the same notation for evidence at a slice as for other sorts of slices, i.e., $M@l$ denotes the slice of M at l. We can merge the evidence slices in a straightforward manner, namely:

$$M = M@\mathbf{l}(1) \cdot M@\mathbf{l}(2) \cdot M@\mathbf{l}(3)$$

We can do this because we know, by definition, that $\mathbf{l}(1) < \mathbf{l}(2) < \mathbf{l}(3)$.

However, in general, we do not know the complete set of labels, and hence we need the “extra” label $\mathbf{l}(0)$, which we use to stand in for the set of labels which do not explicit appear. Because we have no way of knowing what the labels of these fields are, and how they are ordered with respect to the known labels, we cannot determine how to merge the evidence value at this slice with the others.

This difficulty does not exist for the other predicate forms, because we do not require evidence values for those.

Our strategy is to use the algorithm described in the previous chapter for entailment of labelset and row equality predicates. This is summarized in Figure 6.11.

We also modify the well-formedness judgment to incorporate evidence values:

$$\textbf{(parallel)} \quad \frac{P \text{ æ } lset_1 \parallel lset_2}{P \Vdash lset_1 \parallel lset_2}$$

$$\textbf{(equal)} \quad \frac{P \text{ æ } row_1 = row_2}{P \Vdash row_1 = row_2}$$

Figure 6.11: Equality predicate inference rules

for each premise of the form $P \Vdash row_1 \# row_2$ in the well-formedness rules, we substitute the judgment $P \Vdash M : row_1 \# row_2$.

For disjointness predicates, we use entailment inference rules augmented with evidence. We start by giving a set of inference rules for all merge expressions in Figure 6.12.

As is typically the case, these rules cannot be used directly, because they do not specify an algorithm. We can, however, restrict the order in which the rules are applied in such a way as to guarantee termination. To this end, we first describe a naive entailment algorithm for disjointness predicates, and then describe refinements.

For this, we need to have a notion of equivalent evidence expressions. We say that two evidence expressions M_1 and M_2 are equivalent, written $M_1 \approx M_2$, if there exists an evidence value W that both M_1 and M_2 reduce to.

Using the classical algorithm for the labelset equality predicate enables us to make inferences which would otherwise be impossible. For example, let P be the

$$(\textbf{elem}) \qquad \frac{m : \pi \in P}{P \Vdash m : \pi}$$

$$(\#\textbf{-null-null}) \qquad P \Vdash \langle\,\rangle : \varnothing \mathrel{\#} \varnothing$$

$$(\#\textbf{-labels-left}) \qquad \frac{P \Vdash W : (l_2, \ldots, l_n) \mathrel{\#} (l'_1, \ldots, l'_m) \quad l_1 < (l_2, \ldots, l_n, l'_1, \ldots, l'_m)}{P \Vdash \ltimes \cdot W : (l_1, \ldots, l_n) \mathrel{\#} (l'_1, \ldots, l'_m)}$$

$$(\#\textbf{-labels-right}) \qquad \frac{P \Vdash W : (l_1, \ldots, l_n) \mathrel{\#} (l'_2, \ldots, l'_m) \quad l'_1 < (l_1, \ldots, l_n, l'_2, \ldots, l'_m)}{P \Vdash \rtimes \cdot W : (l_1, \ldots, l_n) \mathrel{\#} (l'_1, \ldots, l'_m)}$$

$$(\#\textbf{-flip}) \qquad \frac{P \Vdash M : \mathit{lset}_1 \mathrel{\#} \mathit{lset}_2}{P \Vdash (\textbf{flip}\, M) : \mathit{lset}_2 \mathrel{\#} \mathit{lset}_1}$$

$$(\#\textbf{-merge-left}) \qquad \frac{P \Vdash M_1 : \mathit{lset}_1 \mathrel{\#} \mathit{lset} \quad P \Vdash M_2 : \mathit{lset}_2 \mathrel{\#} \mathit{lset} \quad P \Vdash \mathit{lset}_1 \mathrel{\#} \mathit{lset}_2}{P \Vdash (\textbf{mrg}^{\ltimes}\, (M_1, M_2)) : (\mathit{lset}_1, \mathit{lset}_2) \mathrel{\#} \mathit{lset}}$$

$$(\#\textbf{-merge-right}) \qquad \frac{P \Vdash M_1 : \mathit{lset} \mathrel{\#} \mathit{lset}_1 \quad P \Vdash M_2 : \mathit{lset} \mathrel{\#} \mathit{lset}_2 \quad P \Vdash \mathit{lset}_1 \mathrel{\#} \mathit{lset}_2}{P \Vdash (\textbf{mrg}^{\rtimes}\, (M_1, M_2)) : \mathit{lset} \mathrel{\#} (\mathit{lset}_1, \mathit{lset}_2)}$$

$$(\#\textbf{-extract-left}) \qquad \frac{P \Vdash M_{12} : (\mathit{lset}_1, \mathit{lset}_2) \mathrel{\#} \mathit{lset} \quad P \Vdash M : \mathit{lset}_1 \mathrel{\#} \mathit{lset}_2}{P \Vdash (\textbf{xtrct}^{\ltimes}\, M\, M_{12}) : \mathit{lset}_1 \mathrel{\#} \mathit{lset}}$$

$$(\#\textbf{-extract-right}) \qquad \frac{P \Vdash M_{12} : \mathit{lset} \mathrel{\#} (\mathit{lset}_1, \mathit{lset}_2) \quad P \Vdash M : \mathit{lset}_1 \mathrel{\#} \mathit{lset}_2}{P \Vdash (\textbf{xtrct}^{\rtimes}\, M\, M_{12}) : \mathit{lset} \mathrel{\#} \mathit{lset}_1}$$

$$(\#\textbf{-compatible-}\|) \qquad \frac{P \Vdash M : \mathit{lset}'_1 \mathrel{\#} \mathit{lset}'_2 \quad P \Vdash \mathit{lset}'_1 \parallel \mathit{lset}_1 \quad P \Vdash \mathit{lset}'_2 \parallel \mathit{lset}_2}{P \Vdash M : \mathit{lset}_1 \mathrel{\#} \mathit{lset}_2}$$

Figure 6.12: Disjointness predicate inference rules with evidence

set of predicates

$$M_{12} : \rho_1 \# \rho_2,\ M_{13} : \rho_1 \# \rho_3,\ M_{34} : \rho_3 \# \rho_4,$$
$$M_{24} : \rho_2 \# \rho_4,\ M_{35} : \rho_3 \# \rho_5,\ M_{46} : \rho_4 \# \rho_6,$$
$$((\rho_1, \rho_3) \parallel (\rho_2, \rho_4)),\ ((\rho_1, \rho_2) \parallel (\rho_3, \rho_4)),\ ((\rho_3, \rho_5) \parallel (\rho_4, \rho_6))$$

Then $\rho_2 \# \rho_5$ cannot be derived using the inference rules given in a previous chapter. It can, however, be derived using classical entailment for $\parallel$ by the derivation:

$$\frac{P \Vdash M : \rho_3 \# \rho_5 \quad P\ \textbf{æ}\ \rho_3 \parallel \rho_2}{P \Vdash M : \rho_2 \# \rho_5}$$

6.5 Predicate entailment

Entailment of row equality and labelset equality predicates is handled by the algorithm given in the previous chapter. The remaining predicate type is labelset disjointness.

We first outline a naive algorithm, and then give a refinement. The naive algorithm is an exhaustive search among all possible derivations of a disjointness predicate. We show how to organize this search so that termination is guaranteed.

Given a set of context reduced predicates P, and a disjointness predicate $lset_1 \# lset_2$, the algorithm first applies the context reduction rules to the predicate until none is applicable. This yields a set of *normalized* $\#$ predicates of the form $le_1 \# le_2$, where at least one of le_1, le_2 is a row variable. A derivation of this predicate, if it exists, can be written so that it has one of the forms:

- **elem**,
- **elem** followed by **$\#$-flip**,

- a derivation ending with **$\#$-compatible-$\|$**, **$\#$-flip**, and **$\#$-extract-left**, or
- a derivation ending with **$\#$-compatible** and **$\#$-extract-right**.

We can eliminate the first two cases from consideration by checking if either $M : le_1 \# le_2$ or $M : le_2 \# le_1$ are in P. If they are, the algorithm reports success, with an evidence value of M or $\mathbf{flip}\, M$, respectively.

We can construct a single derived rule to cover the remaining two cases:

$$(\textbf{\#-disjoint}) \quad \frac{\begin{array}{c} P \Vdash lset'_1 \parallel (lset_1, lset''_1) \quad P \Vdash M_1 : lset_1 \# lset''_1 \\ P \Vdash lset'_2 \parallel (lset_2, lset''_2) \quad P \Vdash M_2 : lset_2 \# lset''_2 \\ P \Vdash M : lset'_1 \# lset'_2 \end{array}}{P \Vdash (\mathbf{xtrct}^{\ltimes}\, M_1\, (\mathbf{xtrct}^{\rtimes}\, M_2\, M)) : lset_1 \# lset_2}$$

The algorithm now does a depth-first search, applying the disjoint rule to the predicate $le_1 \# le_2$ for each possible value of $lset'_1$, $lset''_1$, $lset'_2$, and $lset''_2$, which we call the parameters of the search. The values are taken from the powerset of $L(P) \cup RV(P)$. Each attempt generates two labelset equality constraints and three disjointness constraints. The former are handled by the entailment algorithm in the previous chapter, while the disjointness predicates cause a recursive application of the current algorithm. The algorithm returns failure if all branches of the search return failure. A branch returns failure if any of the predicates result in failure, or if a generated predicate is the same as one previously generated along the same branch of the search. Figures 6.13 and 6.14 give pseudocode for the algorithm.

Since the universe of possible values for the parameters is finite, and the recursive

```
$\mathrm{A}(P, row_1 = row_2) = P\ \boldsymbol{\ae}\ row_1 = row_2$
$\mathrm{A}(P, lset_1 \parallel lset_2) = P\ \boldsymbol{\ae}\ lset_1 \parallel lset_2$
$\mathrm{A}(P, lset_1 \# lset_2) = \mathrm{Reduce}(P, \varnothing, lset_1 \# lset_2)$

$\mathrm{Reduce}(P, D, l \# \varnothing)$          $= \textbf{return} \ltimes$
$\mathrm{Reduce}(P, D, \varnothing \# l)$          $= \textbf{return} \rtimes$
$\mathrm{Reduce}(P, D, l_1 \# l_2)$
    $\mid l_1 = l_2 = \textbf{return } \mathit{FAIL}$
    $\mid l_1 < l_2 = \textbf{return } \ltimes \rtimes$
    $\mid l_1 > l_2 = \textbf{return } \rtimes \ltimes$
$\mathrm{Reduce}(P, D, (le, row) \# row') =$
    $\textbf{case } (\mathrm{B}(P, D, le \# row'),$
            $\mathrm{B}(P, D, row \# row'),$
            $\mathrm{B}(P, D, le \# row)) \textbf{ of}$
        $(M_1, M_2, M) \rightarrow \textbf{return mrg}^{\ltimes} (M_1, M_2)$
        $\mathit{otherwise} \rightarrow \textbf{return } \mathit{FAIL}$
$\mathrm{Reduce}(P, D, row' \# (le, row)) =$
    $\textbf{case } (\mathrm{B}(P, D, row' \# le),$
            $\mathrm{B}(P, D, row' \# row),$
            $\mathrm{B}(P, D, le \# row)) \textbf{ of}$
        $(M_1, M_2, M) \rightarrow \textbf{return mrg}^{\rtimes} (M_1, M_2)$
        $\mathit{otherwise} \rightarrow \textbf{return } \mathit{FAIL}$
$\mathrm{Reduce}(P, D, le_1 \# le_2)$          $= \mathrm{B}(P, D, le_1 \# le_2)$
```

Figure 6.13: Entailment algorithm, part 1

```
$\mathrm{B}(P, D, le_1 \# le_2)$
    $| \; M : le_1 \# le_2 \in P$ = **return** $M$
    $| \; M : le_2 \# le_1 \in P$ = **return flip** $M$
    $| \; le_1 \# le_2 \in D$       = **return** $FAIL$
    $| \; otherwise$               =
        **for** $(lset'_1, lset''_1, lset'_2, lset''_2) \in \mathrm{SearchSpace}(P, le_1, le_2)$
            **if** $\mathrm{A}(P, lset'_1 \parallel (le_1, lset''_1))$ **and**
               $\mathrm{A}(P, lset'_2 \parallel (le_2, lset''_2))$
            **then let** $D' = D \cup le_1 \# le_2$
                **in**  **case** $(\mathrm{B}(P, D', le_1 \# lset''_1),$
                            $\mathrm{B}(P, D', le_2 \# lset''_2),$
                            $\mathrm{B}(P, D', lset''_1 \# lset''_2))$ **of**
                        $(M_1, M_2, M) \to$ **return xtrct**$^{\ltimes}$ $M_1$ (**xtrct**$^{\rtimes}$ $M_2\,M$)
                        $otherwise$      $\to$ **continue**
                **end let**
            **end if**
        **end for**
        **return** $FAIL$

$\mathrm{SearchSpace}(P, le_1, le_2) = LE \times LE \times LE \times LE$
    **where** $LE = \mathrm{Powerset}(L(P) \cup RV(P) - \{le_1, le_2\})$
```

Figure 6.14: Entailment algorithm, part 2

```
$\mathrm{B}(P, D, le_1 \# le_2)$
    $| \; M : le_1 \# le_2 \in P$ = **return** $M$
    $| \; M : le_2 \# le_1 \in P$ = **return flip** $M$
    $| \; le_1 \# le_2 \in D$     = **return** $FAIL$
    $| \; \neg(P$ æ $le_1 \# le_2)$ = **return** $FAIL$
    $| \; otherwise$              =
        **for** $(lset'_1, lset''_1, lset'_2, lset''_2) \in \mathrm{SearchSpace}(P, le_1, le_2)$
            **if** $\mathrm{A}(P, lset'_1 \parallel (le_1, lset''_1))$ **and**
               $\mathrm{A}(P, lset'_2 \parallel (le_2, lset''_2))$
            **then let** $D' = D \cup le_1 \# le_2$
                  **in**  **case** $(\mathrm{B}(P, D', le_1 \# lset''_1),$
                                 $\mathrm{B}(P, D', le_2 \# lset''_2),$
                                 $\mathrm{B}(P, D', lset''_1 \# lset''_2))$ **of**
                            $(M_1, M_2, M) \rightarrow$ **return** $\mathbf{xtrct}^{\ltimes} \; M_1 \; (\mathbf{xtrct}^{\rtimes} \; M_2 \; M)$
                            $otherwise$      $\rightarrow$ **continue**
                  **end let**
            **end if**
        **end for**
        **return** $FAIL$


$\mathrm{SearchSpace}(P, le_1, le_2)$
        $= \{(lset'_1, lset''_1, lset'_2, lset''_2) \mid lset'_1 \in \mathrm{DPS}(LE), lset''_1 \in \mathrm{DPS}(LE),$
    $lset'_2 \in \mathrm{DPS}(LE), lset''_2 \in \mathrm{DPS}(LE),$
    $lset'_1 \cap lset''_1 = \varnothing, lset'_2 \cap lset''_2 = \varnothing,$
    $lset'_1 \cap lset''_2 = \varnothing, lset'_2 \cap lset''_1 = \varnothing\}$
    **where** $LE = L(P) \cup RV(P) - \{le_1, le_2\}$
          $\mathrm{DPS}(S) = \{lset \mid lset \in \mathrm{Powerset}(S), \bigwedge_{\forall le_1, le_2 \in lset, le_1 \neq le_2} P$ æ $le_1 \# le_2\}$
```

Figure 6.15: Improved entailment algorithm, part 2

applications of the algorithm use the same universe of values, the search must ultimately terminate.

The search space, however, is huge. If we do not restrict the chosen values in some way, the number of possible values for the four parameters is $(2^N)^4$, where $N = |L(P)| + |RV(P)|$.

Fortunately, a few observations can restrict the search space and prune the search tree substantially.

For both the original and generated disjointness predicates, we can prune the search tree if the non-constructive algorithm reports that the predicate is not entailed, because **æ** is complete, and therefore, a negative result there precludes the constructive algorithm from being successful. We can also reduce the search space from the powerset of labelset elements to the set of well-formed labelsets, i.e. sets of labelset elements which are mutually disjoint. We can approximate disjointness by using the non-constructive algorithm. This is the purpose of precomputing P **æ** $le \# le'$ for all distinct (le, le') pairs. In addition, the following restrictions can be enforced:

$$\begin{gathered} le_1 \notin lset_1'' \cup lset_1'' \cup lset_2' \cup lset_2'' \\ le_2 \notin lset_1'' \cup lset_1'' \cup lset_2' \cup lset_2'' \\ lset_1' \cap lset_1'' = \varnothing \\ lset_2' \cap lset_2'' = \varnothing \end{gathered}$$

6.6 Constant folding

This compilation procedure will generate $\mathbf{index}^{\rtimes} ME$ code even for cases where all the fields of E are known. However, in all such cases, the value of M is statically known, and a straightforward constant-folding pass can replace the expression $\mathbf{index}^{\rtimes} ME$ with a fixed index operation.

6.7 Concatenation with the empty record

The compilation system described has an unexpected blind spot: evidence for $\rho \# \varnothing$ cannot be determined statically. It is clear that the correct evidence in this case is a vector of length $|\rho|$, consisting solely of $\ltimes$ elements. The problem is that we statically do not know the length of this vector. To rectify this shortcoming, we can extend our system so that in all cases where the length of the row in question is not known, but the intention is that all elements of the evidence vector have the same value (e.g., $\ltimes$), we use a distinct evidence literal, which we can write such evidence values as the intended element value with a superscript of $*$. In the case above, the evidence would be written $\ltimes^*$.

We need to extend the definition of concatenation of two evidence literals to incorporate the two new literals. It can easily be shown that we will never encounter a case where $\ltimes^*$ is concatenated with $\rtimes^n$ or $\rtimes^*$. If we think of the superscript $*$ as denoting an unknown length, it is clear that the concatenation of s^* with any literal is s^*. Similarly, the operation $\bar{\cdot}$ is extended in the obvious way. Reduction rules and type rules need to be extended, and various proofs need to have additional cases covered.

An alternative way of handling this situation is to simply have an extra pass which rewrites expressions of the form $e \,\&\, \{\}$ to e.

6.8 Lazy merging

The compilation scheme we have developed merges record values eagerly. That is, a new record value is created at the point where a concatenation is invoked. We may instead choose to delay the actual concatenation until the result is needed. This is known as *lazy* evaluation [23] and is used as a general evaluation scheme in Haskell [45] and earlier, in Miranda [54, 55].

An interesting effect of using lazy evaluation is that various algorithms that are difficult to express in an eager language can be elegantly expressed in a lazy language. Laziness makes compilation more difficult, but on the other hand, in some cases it can lead to more efficient programs, because it avoids computing results which are not needed.

One pitfall of implementing lazy evaluation is the possibility that an expression whose evaluation is delayed may inadvertently be evaluated multiple times, negating any advantage lazy evaluation may have otherwise had. We first give a naive implementation of non-eager record merging that suffers from this problem, and then show how this can be corrected.

6.8.1 A naive lazy implementation

The merge operation takes three operands: the two records to be merged, and the merge vector. A simple lazy implementation of the merge operation would be to simply package these operands into a tuple. We will call this the merge tuple. A record, then, would be either a traditional vector of field values, or a merge tuple. The field extraction operation takes a record and an index, and returns the field value at that index in the vector implementing the record. If the record is represented as a merge tuple, the operation needs to determine which of the two records in the tuple the desired field is in, and what its index in that record is.

The following is ML pseudocode which implements the data structure and operations:

```
type value = (* Universal—value—type *)
type merge = bool vector
datatype record = SIMPLE of value vector
                | MERGE  of merge * record * record

fun merge (m, r1, r2) = MERGE (m, r1, r2)

fun extract (SIMPLE v,        i) = sub (v, i)
  | extract (MERGE (m,r1,r2), i) =
    case splitIndex (m, i) of
        (Left,  i') ⇒ extract (r1, i')
      | (Right, i') ⇒ extract (r2, i')

fun splitIndex (m, i) = if sub (m, i)
                        then (Right, countOnes (m, i))
                        else (Left,  i − countOnes (m, i))

fun countOnes (m, i) =
    let exception Done of int
  in  foldli (fn (j,b,count) ⇒ if j ⩾ i
                               then raise (Done count)
                               else if b
                                    then count + 1
                                    else count) 0 m
  end
```

countOnes (m, i) counts the number of 1 bits in the slice of vector m from indices 0 up to but not including i. This is a naive implementation. A more practical implementation would use bit-twiddling tricks known in the folklore such as the ones described by Manku in [34] or Anderson in [1].

6.8.2 An improved lazy implementation

A problem with the implementation in the previous section is that if we need to extract a given field from a record more than once, we need to duplicate the work. It would be better if we could remember the result of the first query so that subsequent extractions of the same field were fast. This can be done by augmenting the data structure with an extra datum, which is a mutable vector of fields. Each field is either not initialized, or contains the correct value for the index it is at. We assume that there is some way to determine whether a value is initialized. The implementation is now:

```
type value = (* Universal value type *)
type merge = bool vector
datatype record = SIMPLE of value vector
```

```
                | MERGE  of (value option array *
                                merge * record * record)

fun merge (m, r1, r2) =
    let val initVals = Array.array (NONE, Vector.length m)
        (*  initVals is an array with all elements initialized
            to NONE  *)
    in  MERGE (initVals, m, r1, r2)
    end

fun extract (SIMPLE v,             i) = v[i]
  | extract (MERGE (a, m, r1, r2), i) =
    (case a[i] of
         NONE   ⇒ let val e = extract' (m, r1, r2, i)
                  in  a[i] := e;
                      e
                  end
       | SOME e ⇒ e)
and extract' (m, r1, r2, i) =
    case splitIndex (m, i) of
        (Left,  i') ⇒ extract (r1, i')
      | (Right, i') ⇒ extract (r2, i')

fun splitIndex (m, i) = if sub (m, i)
```

```
                            then (Right, countOnes (m, i))
                            else (Left,  i − countOnes (m, i))


fun countOnes (m, i) =
    let exception Done of int
    in  foldli (fn (j,b,count) ⇒ if j ⩾ i
                                 then raise (Done count)
                                 else if b
                                      then count + 1
                                      else count) 0 m
    end
```

Each concatenation creates a new merged record node, which has as children two other merged record nodes, which may have other records as children themselves. An important point is that an update anywhere within this tree reduces the amount of work necessary to extract that field for any ancestor record.

6.9 Compilation into maps

An alternative to compiling into tuples is to discard the type information in the compilation, and compile each record into a map, where the keys are the record labels. There are a number of map implementations to choose from, with different trade-offs among the time required to access a field, extend a record, and concatenate two records. Another consideration is the typical usage pattern and sizes of records in an application. Several reasonable candidates are:

- **association list** This may be the best choice if most records contain only a handful of fields, in which case the overhead of creating and traversing a balanced tree would not be overcome by its improved asymptotic performance.

- **mergeable maps** Described in [43], this is a persistent [14] balanced tree implementation of a map, based on Patricia trees [39], which has the characteristic that merges of two maps are fast.

In any case, type predicates would be discarded during the compilation process. They would only serve the purpose of ensuring type correctness.

6.9.1 A second look

There is yet a third alternative: to compile into maps, but compile the type predicates into some sort of information that helps speed up various record operations. I explore this possibility for the case of maps implemented as balanced trees. For this approach to be possible, knowing the type of the record, or, more precisely, the labels that occur in the record fields must be enough to completely determine the "shape" of the tree. A splay tree [53], for example, would not be suitable, because the path from the root to the leaf where the value associated with a particular field is located can vary, even among splay trees that represent the same exact record value.

Chapter 7

Related work

A fair amount of research has been done into type systems for records. A significantly smaller amount has been done on compilation of such systems. We start by reviewing the support for records provided by a number of general purpose programming languages.

7.1 Programming Language Implementations

7.1.1 C

Most programming languages provide support for records. Virtually all statically-typed traditional imperative languages such as C, C++, and Ada have a very similar design for records. In these languages, record types are *generative.* That is, each definition of a record type generates an entirely new type. Two record type definitions are considered distinct, even if they describe record types that have exactly the same fields. Because of the similarities, we will consider records in C.

If we have the declarations

```
typedef struct {
  int i;
  char c;
  double z;
} A;

typedef struct {
  int i;
  char c;
  double z;
} B;

A a1, a2;
B b;
```

then the assignment

```
a2 = a1;
```

is valid, but

```
b = a1;
```

is not. Another way to describe this is to say that record types obey *name equivalence* in these languages. The only record operations provided are field access and imperative field update. There is no polymorphism, except for those languages that have object oriented features. In such languages, records can be modeled as objects with all public fields, and subtype polymorphism is available.

Records in such languages are laid out in memory as a sequence of field values, modulo padding to ensure proper alignment of each field. There is no reason to keep field names at runtime. For objects, there may be additional information to keep track of associate methods and inheritance hierarchies.

7.1.2 ML

In ML, record types are not generative, and therefore record types do not need to be defined in order to create record values. This significantly increases the ease of using records. The declaration

```
val p = { name = "Joe", age = 28, height = 71 }
```

is a record with fields `name`, `age`, and `height`. Its type is

$$\{\texttt{name} : \texttt{string}, \texttt{age} : \texttt{int}, \texttt{height} : \texttt{int}\}$$

To access the field `age`, we use the appropriately named field selector function:

```
#age p
```

We can also use pattern matching to match on only a subset of a record's fields. For example,

```
fun canDrink { age = a, ... } = a ⩾ 21
```

The pattern `{ age =a, ... }` will bind to any record with an `age` field, and bind `a` to the value of that field. Unfortunately, despite the promise of the syntax, `canDrink` is not polymorphic. The ML type system is not expressive enough to give a properly polymorphic type to this function. In fact, the way it is written, it will not typecheck due to ambiguity. This is one of the few instances in ML where a type declaration is needed to disambiguate the type:

```
fun canDrink { age = a, ... }
    : { age : int, name : string, height : int}
    = a ⩾ 21
```

Another limitation of ML is the lack of support for record update. In imperative languages, fields of records can be updated in place. In functional languages, of course, in-place update is not the modus operandi. Instead, a new record value is constructed which has the same fields and values as the old one, except for the field whose value we want to change. Because ML does not have any polymorphic record operations, this is not a limitation of the type system, but simply a matter of syntax. Instead of a succinct syntax for this operation, to construct the new record, we need to explicitly construct it by specifying the values of each field.

7.1.3 Haskell

In Haskell, records are inextricably tied to individual datatypes. An example of the use of records is:

```
data Person = P { name :: String, age :: Int, height :: Int }
```

This defines a new type called `Person`, a data constructor called `P`, and field accessor functions:

```
name   :: Person → String
age    :: Person → Int
height :: Person → Int
```

A value of type `Person` can be constructed by using either positional or record syntax. For example

```
P { age = 28, name = "Joe", height = 71 }
P "Joe" 28 71
```

both construct the same value. Similarly, the same choices are available for pattern matching:

```
case e of P { name = n, height = h, age = a } ⇒ a+1
case e of P a n h                             ⇒ a+1
```

The selector functions and pattern matching deconstructors are specific to the type `Person`; they cannot be reused for other types in the same namespace. In summary, records in Haskell are essentially a convenient alternative syntax for tuples.

Unlike ML, however, Haskell does provide a convenient syntax for functional record update. If `p` is a value of type `Person`, then

```
p { age = age p + 1 }
```

constructs a new `Person` value with the `age` field incremented. Mark Jones and Simon Peyton Jones propose a change to Haskell records in [28]. This proposal outlines a system that is essentially the one presented in [18], with various details, largely of a syntactic nature, that make it smoothly integrate with Haskell's type system. This proposal is not compatible with Haskell's current support for records; this seems unavoidable.

7.1.4 OCaml

Records in OCaml [32] are generative, just as in Haskell. Unlike the case in Haskell, however, records are not associated with specific datatypes.

In the cases where records are generative, records are implemented as sequences of field values, in the same order as they are declared. For Haskell, because records are associated with a particular variant of a datatype, the “record” value may also need room for a tag. For ML, which has non-generative record types on the other hand, we need to have an order for the fields which is independent of the order they are declared in; an obvious choice is to order the fields by sorting them by their labels. This is necessary to ensure that different record values which have the same type, which may be defined with their labels in differing orders, have the same memory layout.

7.2 Research

A great deal of research has been done on type checking and inference for type systems incorporating records. The systems support a wide variety of capabilities, ranging from basic field selection and record extension to more advanced operations including concatenation and natural join, to esoteric capabilities such as first-class labels. Several approaches have been studied, including bounded polymorphism, rows, and predicates.

7.2.1 Bounded polymorphism

A number of systems [41, 44, 62] support record polymorphism via a bound on the type variables. In most of these, the bound on the type variable specifies that the type is any record that includes a particular field or fields. These systems also typically include the subsumption rule:

$$\textbf{(sub)}\quad \frac{A \vdash E : \tau' \quad A \vdash \tau' \leqslant \tau}{A \vdash E : \tau}$$

where $\tau' \leqslant \tau$ denotes that τ' is a subtype of τ. In terms of records, there are two distinct forms of subtyping:

- *width*: If record A has all the fields of record B, and possibly others, A is a subtype of B.
- *depth*: If all the fields of record A are either subtypes of the fields of B or not present in B, then A is a subtype of B. (Any system that has depth-subtyping must also support width-subtyping.)

Subsumption has advantages and disadvantages. The advantage is that a limited form of polymorphism is available "for free", without even mentioning it in the type. The disadvantage is that, like King Midas' touch [52], it is impossible to turn it off[1].

Furthermore, there are implementation issues. Subsumption can either be implemented by coercing the value of the subtype to the supertype, or by passing

[1]Unlike in the myth, there is no river Pactolus where subsumption can be washed off.

the value as is and "forgetting" about the existence of some of its fields. The latter sounds more attractive, except that now we cannot efficiently access fields of a record, even if the type of the record is statically known. This is because the actual value may be a larger record where some of the fields have been forgotten. Accessing the specified field now incurs a non-trivial run-time cost.

Cardelli and Mitchell [8] describe one of the earliest record type systems, called $\lambda^{\parallel}$. This system supports symmetric concatenation, but, unlike our system, requires explicit type abstraction and application. Bounded quantification is used, but rather than specifying subtyping bounds, the bound is disjointness.

7.2.2 Rows

Remy introduced the concept of rows in [47] and also used them to good effect in [48]. A row represents the set of fields in a record (or variant, for those systems that support variants as well). This is a significant improvement over subtype-based bounded polymorphism. In a system with subtype-based bounded polymorphism, we can specify that an argument must be a record which contains a particular field:

$$\forall \alpha \leqslant \{a : \tau\}. \ldots \alpha \ldots$$

In a system with rows, we can express the same constraint:

$$\forall \rho. \ldots \{a : \tau, \rho\} \ldots$$

but we have an additional piece of information; namely ρ, which represents the rest of the fields in the record. We can use ρ in other parameters or in the result type, thus expressing richer constraints than are possible with bounded polymorphism.

For example, using rows, we can give a type to an operation that renames the field with label a to a field with label b for an arbitrary record:

$$\forall \alpha, \rho \,.\, \{a : \alpha, \rho\} \to \{b : \alpha, \rho\}$$

Using only bounded polymorphism, the best we can do is write:

$$\forall \alpha, \beta \leqslant \{a : \alpha\}, \gamma \leqslant \{b : \alpha\} \,.\, \beta \to \gamma$$

This type is not quite correct, however, because there is no guarantee that the fields in β and γ are in any way related.

In [47] Remy defines a row to be a finite map from labels to one of PRE(τ) or ABS. The former indicates that the label is present and has a type of τ, while the latter indicates that the field is absent. Furthermore, the system offers abstraction (i.e., variables that range) over PRE(τ) and ABS:

$$\begin{array}{rcl} \theta & ::= & \text{PRE}(\tau) \\ & | & \text{ABS} \end{array}$$

This has the result that Remy's system can support an operation that we cannot:

$$\mathit{exchange}^{a \leftrightarrow b} : \{a : \theta_a, b : \theta_b, \rho\} \to \{a : \theta_b, b : \theta_a, \rho\}$$

This exchanges two fields in a record, *whether or not they exist*. In addition, just as in our system, both strict and non-strict record extension and field removal are supported. On the other hand, Remy's system does not support concatenation. In [48], Remy presents a slightly different system, where for all but a finite set of fields, each field of a record may have a default value.

7.2.3 Predicates

Predicates generalize bounded polymorphism by separating the bound on a type variable from its binding. This has two distinct advantages:

1. One can specify multiple-type relations.
2. If one has a type scheme of the form $\forall\alpha.\forall\beta.\ldots$, then one can express constraints on β in terms of α, and constraints on α in terms of β. With bounded polymorphism, only the former is possible, because variables need to be bound before we can use them.

In [58,59], Wand and Mitchell anticipate predicates by describing a system which supports concatenation by generating sets of (C, σ) pairs for each expression, where C is a set of constraints, and σ is a familiar type scheme. This is the forerunner to Jones' $(P|\sigma)$ notation. An example Wand and Mitchell give for the necessity of generating a set of (C, σ) pairs, rather than just one is in [58]:

Consider the term

$$\lambda xy\,.\,((x||y).a + 1)$$

... This term does not have a principal type in any known system ... We shall show its types are generated by two type schemes.

Note that our system is able to give this term a single most general type. In fact, we used a similar example to motivate our need for the row equality predicate on page 50. This predicate allows us to infer a single most general type for the example above, namely

$$\forall \rho_1, \rho_2, \rho \,.\, (\rho_1, \rho_2) = (\rho, a : \mathit{int}) \Rightarrow \{\rho_1\} \to \{\rho_2\} \to \mathit{int}$$

In [21], Harper and Pierce present a system that uses the disjointness predicate and supports strict concatenation. This system, however, does not support type inference; unlike ours, it is explicitly typed.

One system which supports natural join and several other database operations is described by Ohori and Buneman in [7]. Rather that full predicates, a system of "kinds" is used to restrict the instantiation of type variables. For example,

```
fun name x = x.Name
```

is assigned the type

$$\forall \alpha :: [\mathit{Name} : \beta] \,.\, \alpha \to \beta$$

α is limited to record types which contain an *Name* field. This sort of kind can be seen as a restricted form of predicate; namely a unary relation on types. This is also similar to systems which support record polymorphism via subtyping, except that the subsumption rule is not present in this system. This system supports type inference, but the database operations use special inference rules rather than simply having some particular polymorphic type. In our system, there are no special inference rules for these operations; they can be assigned a type scheme just like other functions.

Ohori and Buneman describe a similar system in [42] where recursive concatenation is supported. This is a form of non-strict concatenation where, instead of picking the value of the second operand's field for those fields which exist in both operands, both fields are themselves recursively concatenated. For example, given

$$x = \{a = 1, b = \{i = 2, k = 3\}, c = 4\}$$
$$y = \{b = \{j = 5\}, d = 6\}$$

the concatenation of x and y would result in

$$\{a = 1,\ b = \{i = 2, j = 5, k = 3\},\ c = 4,\ d = 6\}$$

Unfortunately, this complex capability comes at a price: types in record fields cannot include function types. Our system has no such limitation.

In [62], Zwanenburg describes a system that supports the equivalent of our compatible concatenation. Unlike our system, Zwanenburg's system includes the subsumption rule, and deep record subtyping. Type variables in type schemes are given both subtyping and *compatibility* bounds. A compatibility bound, written $R \# S$, means that R and S have the same type for each label they have in common. (Note that this is different from the meaning of $\#$ in our system.) However, there is no type inference; type application and abstraction is explicit.

In [44], Palsberg and Zhao describe a system with a fast ($O(n^5)$) type inference algorithm. It has two forms of record type parameters: one for which exact field information is available, and another which is known only to be a subtype of a specific record type. Only width subtyping is supported. The former record type can be concatenated; the latter cannot. Types may also be recursive, in order to support OO programming.

Gaster and Jones describe polymorphic record system in [18] that supports polymorphic record update and extension, but unlike our system, no concatenation.

Not surprisingly, predicates are used to constrain type variables. Like our system, Gaster and Jones' system offers full type inference.

Predicates are orthogonal to rows and can be combined to good effect (as in our system).

We can divide the record type systems into those that support some sort of concatenation operation and those that do not. Because concatenation is considered the most difficult of the basic record operations, the earlier systems either do not have it or have other serious limitations.

One of the earliest descriptions of an advanced record type system is given in [9], which defines a second-order language with records, based on System F. This language has subtyping and bounded quantification, which is a subtyping bound on a type variable. It is explicitly typed, and does not solve the polymorphic record update problem.

In [49], Remy gives an encoding of records that emulates record concatenation in any language that supports record extension. Records are encoded as functions that extend their argument with the fields in the record. For example, the record

$$\{\mathtt{a} = 5, \mathtt{b} = "hi"\}$$

would be encoded as:

$$\lambda r \,.\, r\ \mathtt{with}\ \{\mathtt{a} = 5\}\ \mathtt{with}\ \{\mathtt{b} = "hi"\}$$

which has the polymorphic type:

$$\forall t.\{a : \text{ABS}, b : \text{ABS}, \rho\} \rightarrow \{a : \text{PRE(int)}, b : \text{PRE(string)}, \rho\}$$

System	TI	RU	RE	SC	CC	UC	NJ	CM
Remy [47]	✓	✓	✓					
Wand [58]	*	✓		✓	✓			
Harper and Pierce [21]		✓	✓	✓				
Ohori and Buneman [7]	✓					*	*	
Zwanenburg [62]		✓	✓		✓			
Palsberg and Zhao [44]	✓	✓	✓	✓				
Cardelli and Mitchell [8]		✓	✓	✓				
Gaster and Jones [18]	✓	✓	✓					✓
Our system	✓	✓	✓	✓	✓	✓	✓	✓

Key:

TI	Type Inference	
RU	Polymorphic Record Update	`(r update { l =v })`
RE	Polymorphic Record Extension	`(r with { l =v })`
SC	Strict Concatenation	`(r1 & r2)`
CC	Compatible Concatenation	`(r1 |&| r2)`
UC	Unrestricted Concatenation	`(r1 && r2)`
NJ	Natural Join	
CM	Compilation Method	

✓	System has feature
(blank)	System does not have feature
*	System has a restricted form of the feature

Figure 7.1: Record system comparison

Unfortunately, this "free" capability has a significant limitation[2]. Because the type of a record's encoding is a polymorphic function, when passed as an argument, it becomes monomorphic, and therefore cannot be applied to two different record types. For example, If we pass the above record to the function

```
fun f a = (a || { x = 5 },
           a || { y = 6 })
```

where `||` is Remy's concatenation function, we would get a type error, because, within the function `f`, `a` has a monomorphic type of the form

$$\{a : \text{ABS}, b : \text{ABS}, \rho\} \rightarrow \{a : \text{PRE}(\text{int}), b : \text{PRE}(\text{string}), \rho\}$$

for some particular ρ. Unfortunately, `f` demands that `a` be used in a context where ρ is `x` : `int`, as well as a context where ρ is `y` : `int`.

Significantly less research has been performed on compilation methods for record type systems or record representation. In [50], Remy explores how extensible records can be represented. He suggests a scheme where each record value is an array of values; the first is a pointer to a header which is used to determine the position of a field value given its label, and the rest are the field values. The header can be shared among all records of the same type, so it need take up only one "slot" per record value. The header is essentially a hash table that maps integers representing field names to indices into the rest of the record array.

Gaster and Jones [18] describe how to translate the predicates in their system into evidence values. Records are represented as sequences of field values, sorted

[2]One of many examples of the notion of TANSTAAFL, i.e. "There ain't no such thing as a free lunch". See [22] for details and examples of this principle

by an order on labels. An example is the type for the field selection operation for label l:

$$\forall \alpha, \rho \,.\, \rho \backslash l \Rightarrow \{l : \alpha, \rho\} \rightarrow \alpha$$

The predicate $\rho \backslash l$ specifies that row ρ is prohibited from having a field with label l. This predicate is translated into the index into the sequence of fields represented by ρ where a field with label l would be inserted. The correctness of the translation is not proven. Our system can be seen as an extension of this work, where evidence values are essentially bit vectors rather than indices, and allow for concatenation of record values in addition to field selection.

The most complete and thorough presentation of a compilation method for records is given in [41], where Ohori defines a record calculus that supports polymorphic update and field selection, and translates this implicitly typed language into an explicitly typed, Church-style implementation calculus. Just as in ours, in Ohori's implementation calculus, records are represented as sequences of field values. Ohori's system does not allow polymorphic record update, but does allow polymorphic field selection. Type variable bounds, which specify which field a record is required to have are translated into index values. The correctness of the translation is proven. Our system can be seen as an extension of this system, as it is somewhat more limited than the system described by Gaster and Jones.

Chapter 8

Conclusion

In conclusion, we have developed and investigated a powerful polymorphic record calculus, and presented both a type inference scheme and a compilation method. The central problem in the type inference algorithm was predicate entailment; we have shown two methods of computing entailment, one of which was suited for the compilation method shown.

8.1 Future work

There are a number of directions in which these results could be extended. An alternative compilation method may use a representation of records different from a simple sequence of field values, and therefore also a different representation of predicate evidence. In particular, perhaps there is a way to support a faster concatenation operation while not compromising on the speed of more basic operations such as field access. Our compilation method assumed that each field value occupied a slot in an array. This corresponds to a simple implementation of polymorphic values, known as *boxing*. It would be useful to be able to apply unboxing optimizations to this representation, which presents challenges to representing predicate evidence.

With respect to the type system itself, there are several small extensions that we conjecture are possible:

- Extending the system to variants, which can be considered the dual of records
- Providing advanced pattern matching to match the available record operations. For example, allowing record extension and/or concatenation patterns would be useful.
- Incorporating recursive types, to provide true OOP capabilities.

In Chapter 5, we discussed simplification of field predicates and gave a number of simplification rules. All of the rules simplified individual predicates. One topic that can be explored is the possibility of rules which consider multiple predicates during simplification, using the ideas in [24]. For example, we can simplify the pair of predicates:

$$((\rho, \rho_1) = l : \tau),\ ((\rho, \rho_2) = l : \tau)$$

by applying the substitution $[\rho_2 \mapsto \rho_1]$, yielding the singleton predicate set $(\rho, \rho_1) = l : \tau$.

It remains to be seen whether it is possible to combine Remy's PRE(τ)/ABS fields with our row variables. This would allow us to type the exchange operation that our system cannot. Other powerful extensions would be first-class labels and higher level operations on rows, both from [18].

Appendix A

Summary of Notation

$[a_1 \mapsto b_1, \ldots, a_n \mapsto b_n]$	Substitution mapping a_i to b_i for all $i \in 1..n$, and any other variable to itself
α, β	Type variables
ρ	Row variable
row	Row metavariable
l	Label metavariable
$\mathbf{l}(i)$	Label
$\mathcal{L}(x)$	Set of labels in entity x
$\mathcal{L}^+(x)$	Set of labels in entity x and an extra label
θ	Substitution metavariable
lp	Label presence
lpe	Label presence element

fld	Field
fe	Field element
LP	Set of label presences
ϕ	Field metavariable
$\boldsymbol{\phi}(\rho, l)$	Field variable, indexed by row variable ρ and label l
fld/LP	Restricted field
lp/LP	Restricted field
$\mathcal{I}$	An interpretation, i.e., a mapping of variables to values
$\mathcal{I}[\![K]\!]$	Semantics of entity K under interpretation $\mathcal{I}$
E	Term metavariable
V	Value metavariable
$\lvert v\rvert$	Length of vector v
$\rtimes, \ltimes$	Vector evidence elements for $\#$
s	Merge metavariable
$\langle\, s, \ldots, s \,\rangle$	Literal merge value
$\overline{s}, \overline{W}$	Negated merge element and value, respectively
$v[i]$	ith element of vector v
$v[i..j]$	Vector consisting of ith through jth elements of v
$v_1 \cdot v_2$	Vector concatenation
$v_1 + v_2$	Pointwise vector sum
π	A single predicate

F, G, H	Sets of field predicates
P, Q, R	Sets of row predicates
A	Sets of type assignments, i.e., finite maps from variables to types or type schemes
$A \backslash x$	A without x
K	Metavariable for any entity
X, Y	Sets of variables
$\overrightarrow{K}$	A sequence or set of entity K. Often used as a shorthand for $K_1, \ldots, K_n$
$\mathcal{C}, \mathcal{E}$	Expression contexts
m	Evidence variable
M	Evidence expression
W	Evidence value
$\{\!\vert E, \ldots, E \vert\!\}$	Translation of record expression in target language
$\mathrm{TV}(x)$	Free type variables in x
$\mathrm{RV}(x)$	Free row variables in x
$\mathrm{FV}(x)$	Free field variables in x
$\mathrm{TRV}(x)$	$\mathrm{TV}(x) \cup \mathrm{RV}(x)$
$\mathrm{TFV}(x)$	$\mathrm{TV}(x) \cup \mathrm{FV}(x)$
$X \rightsquigarrow Y$	Functional dependency: Y is functionally dependent on X

$E_1 \longrightarrow E_2$	Notion of reduction; can also be used with evidence expressions
$E_1 \longmapsto E_2$	Single evaluation step
$E_1 \longmapsto\!\!\!\!\!\rightarrow E_2$	Zero or more evaluation steps
$eval(E)$	Value that E evaluates to
E^S, E^T	Source and target value environments

Appendix B

Proofs

This chapter contains detailed proofs of various theorems in the dissertation.

To aid readability, the theorems are repeated in their entirety before each proof.

B.1 A Second-Order Record Calculus

Proposition 4.4.1 (Well-formedness). *For any well-formed predicate set P,*

1. *if (row_1, row_2) occurs in P, then $P \Vdash row_1 \mathbin{\#} row_2$.*

2. *if $(lset_1, lset_2)$ occurs in P, then $P \Vdash lset_1 \mathbin{\#} lset_2$.*

Proof. Considering conclusion (1) above, if (row_1, row_2) occurs in P, then $P = P' \cup \{row = row'\}$, or $P = P' \cup \{row \blacktriangleright row'\}$, where one of row, row' is (row_1, row_2). Without loss of generality, let $row = (row_1, row_2)$. Then, by **WF-pset-=** or **WH-pset-▸**, we must have $P' \vdash row\ \mathtt{ROW}$ and $P' \vdash row'\ \mathtt{ROW}$, implying that $P' \vdash (row_1, row_2)\ \mathtt{ROW}$. By **WF-row-union**, $P' \Vdash row_1 \mathbin{\#} row_2$, and hence that $P \Vdash row_1 \mathbin{\#} row_2$.

We can prove conclusion (2) using the same reasoning, but by using one of the rules **WF-pset-$\#$** or **WF-pset-$\|$** in the first step, and **WF-lset-union** in the second. □

Theorem 4.6.1 (Preservation of Well-Formedness). *The type rules preserve well-formedness. That is, for every type rule which concludes $P|A \vdash E : \tau$, if any predicates, type assumptions, and types appearing in the premises or introduced in the conclusion are well-formed, then P, $A|_X$, and τ are well-formed, where X is the set of free variables in E.*

Proof. For each of the rules, this follows immediately from the well-formedness of the premises and the WF-type-$*$ rules.

□

Lemma B.1.1. *If τ, row and τ' are well-formed, then so are $[\alpha \mapsto \tau]\tau'$ and $[\rho \mapsto row]\tau'$.*

Proof. This follows by induction on the structure of well-formedness type and row derivations. □

Observation. We will make use of the following observations, which follow from inspection of the type equality rules:

1. If $P \Vdash \tau_1 \to \tau_2 = \tau$, then τ is of the form $\tau_1' \to \tau_2'$, for some τ_1' and τ_2'.
2. If $P \Vdash \{row\} = \tau$, then τ is of the form $\{row'\}$ for some row'.

Lemma 4.5.1 (Type Substitution). *If $P|A \vdash E : \tau$, and θ is an arbitrary substitution, then $\theta P|\theta A \vdash E : \theta\tau$.*

Proof. This proof is an adaptation of the proof of Proposition 3.10 in [29]. It proceeds by induction of the structure of a derivation of $P|A \vdash E : \tau$, and case analysis of the last rule applied.

Most of the cases are straightforward. Several involve the use of the closure property of predicates. We demonstrate one such case: **sel**.

In this case, the last rule applied is:

$$\frac{P|A \vdash E : \{l : \tau, row\} \quad P \Vdash row \# l}{P|A \vdash E.l : \tau} \textbf{ sel}$$

We would like to justify the derivation:

$$\frac{\theta P|\theta A \vdash E : \{l : \theta\tau, \theta row\} \quad \theta P \Vdash \theta row \# \theta l}{\theta P|\theta A \vdash E.l : \theta\tau} \textbf{ sel}$$

The first antecedent, $\theta P|\theta A \vdash E : \{l : \theta\tau, \theta row\}$, justified by the induction hypothesis. The second, $\theta P \Vdash \theta row \# \theta l$, is justified by the closure property of predicates.

The cases for **app**, **abs** and **extract** proceed similarly.

The cases for **var** and **let** proceed exactly as in the proof of Proposition 5.20 in [26].

$\square$

Lemma 4.5.2 (Value Substitution). *If* $P|A, x : (\forall\alpha_1, \ldots, \alpha_n.Q \Rightarrow \tau) \vdash E : \tau'$, *and* $x \notin \mathcal{D}(A)$, $P|A \vdash V : \tau$ *and* $\{\alpha_1, \ldots, \alpha_n\} \cap (\mathrm{TRV}(A) \cup \mathrm{TRV}(P)) = \varnothing$, *then* $P|A \vdash [x \mapsto V]E : \tau'$.

Proof. The proof is an adaptation of the proof of Lemma 4.4 in [61], which is by induction on the structure of a derivation of $P|A\backslash x, x : \forall\alpha_1, \ldots, \alpha_n.Q \Rightarrow \tau \vdash E : \tau'$, and case analysis of the last step.

The main adaptations involve the fact that our judgements have the form $P|A \vdash E : \tau$, whereas the proof in [61] has judgements of the form $A \vdash E : \tau$. This requires that references to Lemma 4.5 in [61] be replaced by references to the analogous Lemma for our system, namely, the Type Substitution Lemma.

With this in mind, the cases for **abs** and **let** are the same as in the aforementioned proof. The cases for **app**, **record**, **sel**, and **extract** are straightforward – they follow directly from the rule antecedents and the induction hypothesis, because for each of those rules, the same P and A are used in all antecedents and the consequent.

This leaves **var**, which is also a slightly different adaptation of the proof in [61]:

$$\frac{x' : \sigma \in A \quad (P \Rightarrow \tau') \leqslant \sigma}{P|A \vdash x' : \tau'}\ \textbf{var}$$

If $x \neq x'$, then $[x \mapsto V]x' = x'$.

If $x = x'$, then, $\sigma = \forall\alpha_1, \ldots, \alpha_n.Q \Rightarrow \tau$. By **var**, $(P \Rightarrow \tau' \leqslant \forall\alpha_1, \ldots, \alpha_n.Q \Rightarrow \tau)$. By Proposition 3.4 in [26], we can find a substitution θ whose domain is $\{\alpha_1, \ldots, \alpha_n\}$ such that $\theta\tau = \tau'$ and $P \Vdash \theta Q$. If $P|A \vdash V : \tau$, and $\{\alpha_1, \ldots, \alpha_n\} \cap (\mathrm{TRV}(A) \cup \mathrm{TRV}(P)) = \varnothing$, then $P|A \vdash V : \theta\tau$ and therefore $P|A \vdash [x \mapsto V]x : \tau'$. □

Lemma 4.5.3. *Subject reduction If* $P|A \vdash E : \tau$, *and* $E \longrightarrow E'$, *then* $P|A \vdash E' : \tau$.

Proof. This proof follows the steps of the proof of Main Lemma 4.3 in [61]. The differences are that we have more notions of reduction. The proof proceeds by induction and case analysis on $\longrightarrow$.

CASE $c\,V \longrightarrow \delta(c, V)$

$P|A \vdash c : \tau' \to \tau$ and $P|A \vdash V : \tau'$ by **app** and the structure of the type equality rules.

CASE $(\lambda x.E)\,V \longrightarrow [x \mapsto V]E$

This step is just as in Main Lemma 4.3 in [61].

CASE $\textbf{let}\ x = V\ \textbf{in}\ E \longrightarrow [x \mapsto V]E$

This step is just as in Main Lemma 4.3 in [61].

CASE $\{l_1 = V_1, \ldots, l_n = V_n\}.l_k \longrightarrow V_k$

We can derive this $P|A \vdash \{l_1 = V_1, \ldots, l_n = V_n\} : \{l_k : \tau, row\}$.

We can write the derivation of this as:

$$\dfrac{\dfrac{P|A \vdash V_i : \tau_i \quad \forall i \in 1..n}{P|A \vdash \{\overrightarrow{l = V}\} : \{\overrightarrow{l : \tau}\}}\,\textbf{record} \qquad P \Vdash (l_1, \ldots, l_{k-1}, l_{k+1}, \ldots, l_n) \# l_k}{P|A \vdash \{\overrightarrow{l = V}\}.l_k : \tau_k}\,\textbf{sel}$$

We can see that one of the antecedents of this derivation is exactly what we want, namely, that $P|A \vdash V_k : \tau_k$.

CASE $\{l_1 = V_1, \ldots, l_n = V_n\}\backslash l_i \longrightarrow \begin{array}{l}\{l_1 = V_1, \ldots, l_{i-1} = V_{i-1},\\ l_{i+1} = V_{i+1}, \ldots, l_n = V_n\}\end{array}$

The proof of this case is analogous to that of the preceding.

$\square$

Lemma 4.5.4 (Uniform Evaluation). *For closed E, if there is no E' such that $E \longmapsto\!\!\!\!\rightarrow E'$ and E' is faulty, then either E diverges, or $E \longmapsto\!\!\!\!\rightarrow V$.*

Proof. The proof of this is an adaptation of the proof of Lemma 4.10 in [61]. The main difference is that we have more cases.

By induction on the length of the reduction sequence, we need only show that one of the following cases must hold:

1. E is faulty,
2. $E \longmapsto E'$ and E' is closed, or
3. E is a value.

Note that $E \longmapsto E'$ iff $E = \mathcal{E}[E_1]$, $E' = \mathcal{E}[E_1']$, and $E_1 \longrightarrow E_1'$.

The proof proceeds by induction on the structure of E. We need only consider the cases not covered in [61].

Case $\{l_1 = E_1, \ldots, l_n = E_n\}$

Either all E_i's are values, in which E is a value, or there exists k such that all E_i for $i < k$ are values, and E_k is not a value. If E_k is faulty, then so is E. Otherwise, let $E_k \longrightarrow E_k'$. Then $E_k = \mathcal{E}_\infty[e_k'']$, $E_k' = \mathcal{E}_\infty[e_k'''']$, and $E_k'' \longrightarrow E_k''''$. Then $E = \mathcal{E}[E_k'']$, where $E = \{l_1 = E_1, \ldots, l_{k-1} = E_{k-1}, l_k = \mathcal{E}_\infty, l_{k+1} = E_{k+1}, \ldots, l_n = E_n\}$. Therefore, $E \longmapsto \mathcal{E}[E_k'''']$.

Case $E'.l$

If E' is faulty, then so is E. If E' is a value, then by the type rules, the only sort of value for which we can derive a type for E is a record value: $\{l_1 = V_1, \ldots, l_n = V_n\}$, and

Case $E' \backslash l$

If E' is faulty, then so is E. If E' is a value, then by the type rules, the only sort of value for which we can derive a type for E is a record value: $\{l_1 = V_1, \ldots, l_n = V_n\}$, where $l = l_k$ for some $k \in 1..n$. Then $E \longrightarrow V_k$. Otherwise, $E' \longrightarrow E''$, and $E' = \mathcal{E}_\infty E'_1$, and $E'' = \mathcal{E}_\infty E''_1$. Then $\mathcal{E} = \{l_1 = V_1, \ldots, l_{k-1} = V_{k-1}, l_k = \mathcal{E}_\infty, l_{k+1} = E_{k+1}, \ldots, l_n = E_n\}$, and $E \longrightarrow \mathcal{E}[E''_1]$.

Case $\{l_1 = V_1, \ldots, l_n = V_n\}\backslash l_i \longrightarrow \begin{array}{l}\{l_1 = V_1, \ldots, l_{i-1} = V_{i-1},\\ l_{i+1} = V_{i+1}, \ldots, l_n = V_n\}\end{array}$

This follows the same reasoning as the previous case.

□

Theorem 4.5.1 (Syntactic Type Soundness). *If* $\vdash E : \tau$, *then* $E \longmapsto\!\!\!\!\rightarrow V$ *and* $\vdash V : \tau$.

Proof. The proof is the same as the one in [61], except that references to the Uniform Evaluation Lemma are replaced by references to our version, i.e., Lemma 4.5.4. □

B.2 Predicates

Proposition B.2.1.

$$\begin{array}{rcl}\mathcal{I}[\![le@l]\!]_{LP} & = & l \,\dot{\in}\, \mathcal{I}[\![le]\!]_{LS} \\ \mathcal{I}[\![re@l]\!]_{FLD} & = & \mathcal{I}[\![re]\!](l)\end{array}$$

Proof. We consider row and labelset elements separately, and do case analysis on the form of the element.

First, the labelset elements:

Case $le = l'$

Expanding the left side:

$$\mathcal{I}[\![l'@l]\!]_{LP} = \begin{cases} \texttt{t} & \text{if } l = l' \\ \texttt{f} & \text{otherwise} \end{cases}$$

Expanding the right side:

$$l \,\dot{\in}\, \mathcal{I}[\![le]\!]_{LS} = l \,\dot{\in}\, \{\cdot l' \cdot\} = \begin{cases} \texttt{t} & \text{if } l = l' \\ \texttt{f} & \text{otherwise} \end{cases}$$

Case $le = \rho$

Expanding the left side:

$$\mathcal{I}[\![\rho@l]\!]_{LS} = \mathcal{I}[\![\phi(\rho, l)]\!] = l \,\dot{\in}\, \mathcal{D}(\mathcal{I}[\![\rho]\!])$$

Expanding the right side:

$$l \,\dot{\in}\, \mathcal{I}[\![\rho]\!]_{LS} = l \,\dot{\in}\, \mathcal{D}(\mathcal{I}[\![\rho]\!])$$

Secondly, row elements:

Case $re = l' : \tau$

Expanding the left side:

$$\mathcal{I}[\![(l' : \tau)@l]\!]_{FLD} = \begin{cases} \tau & \text{if } l = l' \\ \bot & \text{otherwise} \end{cases}$$

Expanding the right side:

$$\mathcal{I}[\![(l' : \tau)]\!]_{ROW} = \{\!\cdot(l', \tau)\cdot\!\}$$

Clearly, both sides of the equation evaluate to τ if $l = l'$, and to $\perp$ otherwise.

CASE $re = \rho$

Expanding the left side:

$$\mathcal{I}[\![\rho @ l]\!]_{FLD} = \mathcal{I}[\![\phi(\rho, l)]\!] = \mathcal{I}[\![\rho]\!](l)$$

The right side is already in the same form.

$\square$

Proposition B.2.2.

$$\mathcal{I}[\![lset @ l]\!] = l \mathrel{\dot\in} \mathcal{I}[\![lset]\!]$$
$$\mathcal{I}[\![row @ l]\!] = \mathcal{I}[\![row]\!](l)$$

Proof. We consider each of the two equations separately.

We start with $\mathcal{I}[\![lset @ l]\!] = l \mathrel{\dot\in} \mathcal{I}[\![lset]\!]$:

Let $lset = (le_1, \ldots, le_n)$. Expanding the left side,

$$\begin{array}{rcl} \mathcal{I}[\![(le_1, \ldots, le_n) @ l]\!] & = & \mathcal{I}[\![(le_1 @ l, \ldots, le_n @ l)]\!] \\ & = & \bigvee_{i \in 1..n} \mathcal{I}[\![le_i @ l]\!] \end{array}$$

By Proposition B.2.1, this is equivalent to

$$\bigvee_{i \in 1..n} l \mathrel{\dot\in} \mathcal{I}[\![le_i]\!]$$

Expanding the right side,

$$l \mathrel{\dot\in} \mathcal{I}[\![(le_1, \ldots, le_n)]\!] = l \mathrel{\dot\in} \dot{\bigcup_{i \in 1..n}} \mathcal{I}[\![le_i]\!]$$

By the definition of $\dot\cup$, this is the same as

$$\bigvee_{i \in 1..n} \mathcal{I}[\![le_i@l]\!]$$

Now, we consider $\mathcal{I}[\![row@l]\!] = \mathcal{I}[\![row]\!](l)$:

Let $row = (re_1, \ldots, re_n)$.

Expanding the left side,

$$\begin{array}{rcl} \mathcal{I}[\![(re_1, \ldots, re_n)@l]\!] & = & \mathcal{I}[\![(re_1@l, \ldots, re_n@l)]\!] \\ & = & \dot\bigcup_{i \in 1..n} \mathcal{I}[\![re_i@l]\!] \end{array}$$

Expanding the right side,

$$\mathcal{I}[\![(re_1, \ldots, re_n)]\!](l) = \dot{\bigcup_{i \in 1..n}} \mathcal{I}[\![re_i]\!](l)$$

By Proposition B.2.1, these expansions are equal.

□

Proposition 5.5.1.

$$\mathcal{I}[\![\pi]\!] = \forall l.\mathcal{I}[\![\pi@l]\!]$$

Proof. We prove this for each form of predicate separately.

Case $\pi = lset_1 \# lset_2$

We need to show that

$$\mathcal{I}[\![lset_1 \# lset_2]\!] = \forall l.\mathcal{I}[\![(lset_1 \# lset_2)@l]\!]$$

Expanding the left side:

$$\mathcal{I}[\![lset_1 \# lset_2]\!] \quad = \quad (\mathcal{I}[\![lset_1]\!] \,\dot\cap\, \mathcal{I}[\![lset_2]\!] = \varnothing)$$

By the definitions of $\dot{\cap}$, $=$, and $\varnothing$, we have:

$$\forall l.\min(\mathcal{I}[\![lset_1]\!](l), \mathcal{I}[\![lset_2]\!](l)) = 0$$

Therefore,

$$\forall l.l \dot{\notin} \mathcal{I}[\![lset_1]\!] \vee l \dot{\notin} \mathcal{I}[\![lset_2]\!]$$

Expanding the right side:

$$\begin{aligned}\forall l.\mathcal{I}[\![(lset_1 \mathrel{\#} lset_2)@l]\!] &= \forall l.\mathcal{I}[\![(lset_1@l) \mathrel{\#} (lset_2@l)]\!] \\ &= \forall l.\neg\mathcal{I}[\![lset_1@l]\!] \vee \neg\mathcal{I}[\![lset_2@l]\!]\end{aligned}$$

By Proposition B.2.2, this is equivalent to

$$\forall l.l \dot{\notin} \mathcal{I}[\![lset_1]\!] \vee l \dot{\notin} \mathcal{I}[\![lset_2]\!]$$

Case $\pi = lset_1 \parallel lset_2$

Expanding the left side,

$$\mathcal{I}[\![lset_1]\!]_{LS} = \mathcal{I}[\![lset_2]\!]_{LS}$$

By definition of $=$, this is equivalent to

$$\forall l.(l \dot{\in} \mathcal{I}[\![lset_1]\!]) \leftrightarrow (l \dot{\in} \mathcal{I}[\![lset_2]\!])$$

Expanding the right side,

$$\forall l.\mathcal{I}[\![(lset_1@l) \parallel (lset_2@l)]\!]$$
$$\forall l.\mathcal{I}[\![lset_1@l]\!]_{LS} \leftrightarrow \mathcal{I}[\![lset_2@l]\!]_{LS}$$

By Proposition B.2.2, this is equivalent to

$$\forall l.(l \dot{\in} \mathcal{I}[\![lset_1]\!]) \leftrightarrow (l \dot{\in} \mathcal{I}[\![lset_2]\!])$$

CASE $\pi = row_1 = row_2$

Expanding the left side,

$$\mathcal{I}[\![row_1]\!] = \mathcal{I}[\![row_2]\!]$$

By definition of $=$,

$$\forall l.\mathcal{I}[\![row_1]\!](l) = \mathcal{I}[\![row_2]\!](l)$$

Expanding the right side,

$$\forall l.\mathcal{I}[\![(row_1@l) = (row_2@l)]\!]$$
$$\forall l.\mathcal{I}[\![row_1@l]\!] = \mathcal{I}[\![row_2@l]\!]$$

By Proposition B.2.2, this is equivalent to

$$\forall l.\mathcal{I}[\![row_1]\!](l) = \mathcal{I}[\![row_2]\!](l)$$

□

Proposition 5.5.2 (Slice well-formedness preservation). *If row predicate set P is well-formed, then so is $P@l$, for any label l.*

Proof. Each $(fe_1, \ldots, fe_n)$ occurring in $P@l$ corresponds to a $(re_1, \ldots, re_n)$ occurring in P, and is in fact $(re_1@l, \ldots, re_n@l)$.

Let $Q = \{re_i \# re_j | 1 \leqslant i, j \leqslant n, i \neq j\}$. Then, by the definition of well-formedness of row predicate sets, $P \Vdash Q$.

By definition,

$$(P \Vdash Q) \quad \rightarrow \quad (\forall \mathcal{I}.\mathcal{I}[\![P]\!] \rightarrow \mathcal{I}[\![Q]\!]).$$

By Proposition 5.5.1, we also have that

$$(\mathcal{I}[\![P]\!] \rightarrow \mathcal{I}[\![Q]\!]) \quad \leftrightarrow \quad \forall l.\mathcal{I}[\![P@l]\!] \rightarrow \mathcal{I}[\![Q@l]\!]$$

Note that $Q@l$ is $\{re_i@l \ne re_j@l | 1 \leqslant i, j \leqslant n, i \ne j\}$, which is simply $\{fe_i \ne fe_j | 1 \leqslant i, j \leqslant n, i \ne j\}$. □

Lemma 5.5.2.

$$\begin{array}{rl} & \models \forall \mathcal{I}.(\forall l.\mathcal{I}[\![P@l]\!]) \rightarrow (\forall l.\mathcal{I}[\![Q@l]\!]) \\ \leftrightarrow & \models \forall \mathcal{I}.(\bigwedge_{l \in \mathcal{L}^+(P \cup Q)} \mathcal{I}[\![P@l]\!]) \rightarrow (\bigwedge_{l \in \mathcal{L}^+(P \cup Q)} \mathcal{I}[\![Q@l]\!]) \end{array}$$

Proof. By Lemma 5.5.1, we can justify the equivalences

$$(\forall l.\mathcal{I}[\![P@l]\!]) \quad \leftrightarrow \quad (\bigwedge_{l \in \mathcal{L}^+(P \cup Q)} \mathcal{I}[\![P@l]\!])$$

and

$$(\forall l.\mathcal{I}[\![Q@l]\!]) \quad \leftrightarrow \quad (\bigwedge_{l \in \mathcal{L}^+(P \cup Q)} \mathcal{I}[\![Q@l]\!])$$

We need to have a conjunction over $\mathcal{L}^+(P \cup Q)$ so that the "extra" label represents the same infinite set in both antecedent and consequent. □

Proposition B.2.3. $\forall \mathcal{I}.\mathcal{I}[\![\pi]\!] = \forall \mathcal{I}.\mathcal{I}[\![\bigcup_{l \in L} \pi@l]\!]$, *where* $L = \mathcal{L}^+(\pi)$.

Proof. By Proposition 5.5.1, we have that

$$\mathcal{I}[\![\pi]\!] = \forall l.\mathcal{I}[\![\pi@l]\!]$$

Rewriting the left hand side slightly, we have the following which needs to be proven:

$$\bigcup_{l} \mathcal{I}[\![\pi@l]\!] \quad = \quad \bigcup_{l \in L} \mathcal{I}[\![\pi@l]\!]$$

We can rewrite the two sides as:

$$(\bigcup_{l \in \mathcal{L}(\pi)} \mathcal{I}[\![\pi@l]\!]) \wedge (\bigcup_{l \notin \mathcal{L}(\pi)} \mathcal{I}[\![\pi@l]\!]) \quad = \quad (\bigcup_{l \in \mathcal{L}(\pi)} \mathcal{I}[\![\pi@l]\!]) \wedge (\mathcal{I}[\![\pi@l_0]\!])$$

where $\{l_0\} = \mathcal{L}^+(\pi) - \mathcal{L}(\pi)$.

Reducing, we have:

$$(\bigcup_{l \notin \mathcal{L}(\pi)} \mathcal{I}[\![\pi @ l]\!]) \quad = \quad \mathcal{I}[\![\pi @ l_0]\!]$$

This is true by Proposition 5.5.3, which states that $\mathcal{I}[\![\pi @ l]\!] = \mathcal{I}[\![\pi @ l']\!]$ for any $l \notin \mathcal{L}(\pi)$. □

Lemma B.2.1. *If F is a well-formed field predicate set, $\forall \mathcal{I}. \mathcal{I}[\![F']\!] \rightarrow \mathcal{I}[\![F]\!]$, and* $\mathrm{WFC}[F'] \subseteq \mathrm{WFC}[F]$*, then F' is well-formed.*

Proof. Let $\mathrm{WFC}[F] = \mathrm{WFC}[F'] \uplus G$. Then, we have the tautology

$$(\mathcal{I}[\![F']\!] \rightarrow \mathcal{I}[\![F]\!]) \wedge (\mathcal{I}[\![F]\!] \rightarrow (\mathcal{I}[\![\mathrm{WFC}[F'] \uplus G]\!])) \quad \rightarrow \quad (\mathcal{I}[\![F']\!] \rightarrow \mathcal{I}[\![\mathrm{WFC}[F']]\!])$$

The conclusion of the implication is simply the criteria for well-formedness of F'. □

Lemma B.2.2. *For any field predicate set F,*

$$\forall \mathcal{I}. \mathcal{I}[\![\mathrm{WFC}[F]]\!] \rightarrow \mathcal{I}[\![\mathrm{WFC}[[\phi \mapsto \bot]F]]\!]$$

Proof. We show a stronger result, namely, that

$$\forall \mathcal{I}. \mathcal{I}[\![lp_1 \# lp_2]\!] \rightarrow \mathcal{I}[\![[\phi \mapsto \bot] lp_1 \# lp_2]\!]$$

From this, the lemma immediately follows. We consider two cases for each lp: it is either ϕ, or something else.

Case neither lp_1 nor lp_2 are ϕ.

Clearly, $[\phi \mapsto \bot](lp_1 \# lp_2) = lp_1 \# lp_2$, and the implication is trivially true.

Case one of lp_1, lp_2 is ϕ.

Because $\#$ is symmetric, we need only consider one case (pick $lp_1 = \phi$). Then we have that $[\phi \mapsto \bot](lp_1 \mathrel{\#} lp_2) = \bot \mathrel{\#} lp_2$. Because $\bot \mathrel{\#} lp_2$ is always true, so is the implication.

Case both of lp_1, lp_2 are ϕ.

Therefore, $[\phi \mapsto \bot](lp_1 \mathrel{\#} lp_2) = \bot \mathrel{\#} \bot$, and, for the same reason as the previous case, the implication is true.

$\square$

Lemma B.2.3. *If* $\forall\mathcal{I}.\mathcal{I}[\![F]\!] \rightarrow (\mathcal{I}[\![\phi]\!] = \mathcal{I}[\![\hat{\tau}]\!])$*, then*

$$(\forall\mathcal{I}.\mathcal{I}[\![F]\!] \rightarrow \mathcal{I}[\![G]\!]) \quad \leftrightarrow \quad (\forall\mathcal{I}.\mathcal{I}[\![[\phi \mapsto \hat{\tau}]F]\!] \rightarrow \mathcal{I}[\![[\phi \mapsto \hat{\tau}]G]\!])$$

Proof. First, we do the forward direction. We proceed to prove that $(\forall\mathcal{I}.\mathcal{I}[\![[\phi \mapsto \hat{\tau}]F]\!] \rightarrow \mathcal{I}[\![[\phi \mapsto \hat{\tau}]G]\!])$ holds, assuming that $(\forall\mathcal{I}.\mathcal{I}[\![F]\!] \rightarrow \mathcal{I}[\![G]\!])$ does. We can also assume that $\forall\mathcal{I}.\mathcal{I}[\![F]\!] \rightarrow (\mathcal{I}[\![\phi]\!] = \mathcal{I}[\![\hat{\tau}]\!])$. We can combine and rearrange these two assumptions:

$$\begin{aligned} &(\mathcal{I}.\mathcal{I}[\![F]\!] \rightarrow \mathcal{I}[\![G]\!]) \wedge (\mathcal{I}[\![F]\!] \rightarrow (\mathcal{I}[\![\phi]\!] = \mathcal{I}[\![\hat{\tau}]\!])) \\ \rightarrow \quad &(\mathcal{I}[\![\phi]\!] = \mathcal{I}[\![\hat{\tau}]\!]) \rightarrow (\mathcal{I}.\mathcal{I}[\![F]\!] \rightarrow \mathcal{I}[\![G]\!]) \end{aligned}$$

Given any $\mathcal{I}$, we can construct $\mathcal{I}' = [\phi \mapsto \hat{\tau}]\mathcal{I}$. Note that $\mathcal{I}'[\![\phi]\!] = \mathcal{I}'[\![\bot]\!] = \mathcal{I}[\![\bot]\!]$. By the assumption, we have $\mathcal{I}'[\![F]\!] \rightarrow \mathcal{I}'[\![G]\!]$. But $\mathcal{I}'[\![F]\!] = \mathcal{I}[\![[\phi \mapsto \hat{\tau}]F]\!]$ and similarly for G.

Now, we do the backward direction. We need to prove $\forall\mathcal{I}.\mathcal{I}[\![F]\!] \rightarrow \mathcal{I}[\![G]\!]$, assuming $(\forall\mathcal{I}.\mathcal{I}[\![[\phi \mapsto \hat{\tau}]F]\!] \rightarrow \mathcal{I}[\![[\phi \mapsto \hat{\tau}]G]\!])$ and $\forall\mathcal{I}.\mathcal{I}[\![F]\!] \rightarrow (\mathcal{I}[\![\phi]\!] = \mathcal{I}[\![\hat{\tau}]\!])$.

Pick an $\mathcal{I}$ such that $\mathcal{I}[\![F]\!]$. By the second assumption, we have that $\mathcal{I}[\![\phi]\!] = \mathcal{I}[\![\hat{\tau}]\!]$, which means that $\mathcal{I}[\![F]\!]$ is the same as $\mathcal{I}[\![[\phi \mapsto \hat{\tau}]F]\!]$. It therefore follows that $\mathcal{I}[\![[\phi \mapsto \bot]G]\!]$, and by the previous argument, is the same as $\mathcal{I}[\![G]\!]$.

□

Lemma B.2.4. *If $\mathcal{I}[\![lp_1 \# lp_2]\!]$, then $\mathcal{I}[\![lp_1/lp \# lp_2/lp]\!]$.*

Proof. We repeat the semantic rule for lp/LP for easy reference:

$$\mathcal{I}[\![lp/LP]\!] \equiv \begin{cases} \mathcal{I}[\![lp]\!] & \text{if } \wedge_{lp' \in LP} \mathcal{I}[\![lp']\!] \\ \bot & \text{otherwise} \end{cases}$$

If $\mathcal{I}[\![lp_1 \# lp_2]\!]$, by the semantics of $\#$, the interpretation of at least one of lp_1, lp_2 must be false. Without loss of generality, we assume that $\neg\mathcal{I}[\![lp_1]\!]$. Then $\neg\mathcal{I}[\![lp_1/lp]\!]$ for any lp. □

Lemma B.2.5. *For any field predicate sets F and G, and any field variable ϕ and field type $\hat{\tau}$,*

$$(\forall\mathcal{I}.\mathcal{I}[\![F]\!] \rightarrow \mathcal{I}[\![G]\!]) \quad \rightarrow \quad (\forall\mathcal{I}.\mathcal{I}[\![[\phi \mapsto \hat{\tau}]F]\!] \rightarrow \mathcal{I}[\![[\phi \mapsto \hat{\tau}]G]\!])$$

Proof. We need to prove

$$\forall\mathcal{I}.\mathcal{I}[\![[\phi \mapsto \hat{\tau}]F]\!] \rightarrow \mathcal{I}[\![[\phi \mapsto \hat{\tau}]G]\!]$$

assuming

$$\forall\mathcal{I}.\mathcal{I}[\![F]\!] \rightarrow \mathcal{I}[\![G]\!]$$

Let $\mathcal{I}' = [\phi \mapsto \hat{\tau}]\mathcal{I}$ for an arbitrary $\mathcal{I}$. Then $\mathcal{I}'[\![[\phi \mapsto \hat{\tau}]F]\!] = \mathcal{I}[\![[\phi \mapsto \hat{\tau}]F]\!] = \mathcal{I}'[\![F]\!]$. By the assumption, $\mathcal{I}'[\![G]\!]$. But $\mathcal{I}'[\![G]\!] = \mathcal{I}[\phi \mapsto \hat{\tau}]G$, and so we have that $\mathcal{I}[\![[\phi \mapsto \hat{\tau}]F]\!] \rightarrow \mathcal{I}[\![[\phi \mapsto \hat{\tau}]G]\!]$ for an arbitrary $\mathcal{I}$. □

Lemma 5.5.3 (Single Step Simplification Equivalence). *If $F = F' \uplus \{\pi\}$ is a well-formed field predicate set, and $\pi \longrightarrow (\theta, H)$, then*

1. $(\forall \mathcal{I}.\mathcal{I}[\![F]\!] \rightarrow \mathcal{I}[\![G]\!]) \quad \leftrightarrow \quad (\forall \mathcal{I}.\mathcal{I}[\![\theta F \cup H]\!] \rightarrow \mathcal{I}[\![\theta G]\!])$

2. $\theta F \cup H$ *is well-formed*

Proof. We prove this by a case analysis of each rule.

Case Consider the first of these two rules:

$$\begin{array}{lcll} (fe_1, \ldots, fe_n, \bot) & \longrightarrow & (fe_1, \ldots, fe_n) & \text{where } n \geqslant 1 \\ (lpe_1, \ldots, lpe_n, \mathtt{f}) & \longrightarrow & (lpe_1, \ldots, lpe_n) & \text{where } n \geqslant 1 \end{array}$$

It is the case that $\forall \mathcal{I}.\mathcal{I}[\![(fe_1, \ldots, fe_n, \bot)]\!] = \mathcal{I}[\![(fe_1, \ldots, fe_n)]\!]$ where $n \geqslant 1$, and similarly for the second rule. This follows directly from the semantic rules for fields and label presences. Let π' be the result of substituting $(fe_1, \ldots, fe_n)$ for $(fe_1, \ldots, fe_n, \bot)$ in π. Therefore, $\forall \mathcal{I}.\mathcal{I}[\![F' \uplus \pi]\!] = \mathcal{I}[\![F' \cup \pi']\!]$. Also, $\text{WFC}[\pi] \supseteq \text{WFC}[\pi']$. By Lemma B.2.1, $F' \cup \pi'$ is well-formed. The proof for the second rule follows in the same manner.

Case Consider the first of these two rules:

$$\begin{array}{lcl} & & ([\phi_1 \mapsto \bot, \ldots, \phi_n \mapsto \bot], \\ (\phi_1, \ldots, \phi_n, \mathtt{t}, lpe_1, \ldots, lpe_m) & \longrightarrow & (\mathtt{t}, lpe_1, \ldots, lpe_m)) \\ & & \text{where } n \geqslant 1, m \geqslant 0 \\ & & ([\phi_1 \mapsto \bot, \ldots, \phi_n \mapsto \bot], \\ (\phi_1, \ldots, \phi_n, \tau, fe_1, \ldots, fe_m) & \longrightarrow & (\tau, fe_1, \ldots, fe_m)) \\ & & \text{where } n \geqslant 1, m \geqslant 0 \end{array}$$

The expression in the result of the simplification contains a strict subset of the one in the source, just as in the first two rules. However, we also have a substitution. For each $i \in 1..n$, $\phi_i \mathrel{\#} \mathtt{t} \in \text{WFC}[F]$. This is true only when

$\phi_i = \bot$. Let π' be the result of substituting $(fe_1, \ldots, fe_n)$ for $(fe_1, \ldots, fe_n, \bot)$ in π. By Lemma B.2.3,

$$(\forall \mathcal{I}. \mathcal{I}[\![F']\!] \rightarrow \mathcal{I}[\![G]\!]) \quad \leftrightarrow \quad (\forall \mathcal{I}. \mathcal{I}[\![\theta F' \cup \{\pi'\}]\!] \rightarrow \mathcal{I}[\![\theta G]\!])$$

where $\theta = [\phi_1 \mapsto \bot, \ldots, \phi_n \mapsto \bot]$. Because the substitution only maps field variables to $\bot$, by repeated application of Lemma B.2.2 we can deduce that $\theta F' \cup \{\pi'\}$ is well-formed.

The second rule can be proved correct in the same manner.

Case Consider any rule of the form $\pi \longrightarrow \texttt{OK}$

The rules in this group are:

$$\begin{array}{ccccc}
\texttt{f} & \# & lp & \longrightarrow & \texttt{OK} \\
\texttt{f} & \parallel & \texttt{f} & \longrightarrow & \texttt{OK} \\
\texttt{t} & \parallel & \texttt{t} & \longrightarrow & \texttt{OK} \\
\bot & = & \bot & \longrightarrow & \texttt{OK} \\
t & = & t & \longrightarrow & \texttt{OK}
\end{array}$$

By a simple application of the semantic rules, we can verify that for all these rules, $\forall \mathcal{I}. \mathcal{I}[\![\pi]\!]$. We can therefore always remove such a rule from our set of predicates. Well-formedness is maintained as well, by Lemma B.2.1.

Case Consider any rule of the form $\pi \longrightarrow \texttt{Fail}$

The rules in this group are:

$$\begin{array}{rclcll}
\texttt{t} & \# & \texttt{t} & \longrightarrow & \texttt{Fail} & \\
\texttt{t} & \| & \texttt{f} & \longrightarrow & \texttt{Fail} & \\
t & = & \tau & \longrightarrow & \texttt{Fail} & \text{if } \tau \not\# t \\
\tau_1 \to \tau_2 & = & \tau & \longrightarrow & \texttt{Fail} & \text{if } \tau \text{ is not of the form } \tau_1' \to \tau_2' \\
\{row\} & = & \tau & \longrightarrow & \texttt{Fail} & \text{if } \tau \text{ is not of the form } \{row'\} \\
\tau & = & \bot & \longrightarrow & \texttt{Fail} &
\end{array}$$

By a simple application of the semantic rules, we can verify that for all these rules, $\forall \mathcal{I}.\neg\mathcal{I}[\![\pi]\!]$. Therefore, $\mathcal{I}[\![F' \uplus \{\pi\}]\!]$ is always false, and $\forall \mathcal{I}.\mathcal{I}[\![F' \uplus \{\pi\}]\!] \to \mathcal{I}[\![G]\!]$ is trivially true. Similarly, $\forall \mathcal{I}.\mathcal{I}[\![F' \uplus \{\texttt{Fail}\}]\!] \to \mathcal{I}[\![G]\!]$ is trivially true. The same argument holds for Point 2; namely, that $\forall \mathcal{I}.\mathcal{I}[\![F' \uplus \{\texttt{Fail}\}]\!] \to \mathcal{I}[\![\mathrm{WFC}[F' \uplus \{\texttt{Fail}\}]]\!]$.

Case Consider the rule $(\phi_1, \ldots, \phi_n) \# lp \longrightarrow \{\phi_i \# lp \mid i \in 1..n\}$

To prove the first point, it is enough to show that $\forall \mathcal{I}.\mathcal{I}[\![(\phi_1, \ldots, \phi_n) \# lp]\!] \leftrightarrow \mathcal{I}[\![\{\phi_i \# lp \mid i \in 1..n\}]\!]$.

Expanding the left side:

$$\begin{array}{rcl}
\mathcal{I}[\![(\phi_1, \ldots, \phi_n) \# lp]\!] & = & \neg\mathcal{I}[\![(\phi_1, \ldots, \phi_n)]\!]_{LP} \vee \neg\mathcal{I}[\![lp]\!] \\
& = & \neg(\bigvee_{i \in 1..n} \mathcal{I}[\![\phi_i]\!]_{LS}) \vee \neg\mathcal{I}[\![lp]\!]
\end{array}$$

Expanding the right side:

$$\begin{array}{rcll}
\mathcal{I}[\![\{\phi_i \# lp \mid i \in 1..n\}]\!] & = & \bigwedge_{i \in 1..n} \mathcal{I}[\![\phi_i \# lp]\!] & \\
& = & \bigwedge_{i \in 1..n} (\neg\mathcal{I}[\![\phi_i]\!]_{LS} \vee \neg\mathcal{I}[\![lp]\!]) & \\
& = & \bigwedge_{i \in 1..n} (\neg\mathcal{I}[\![\phi_i]\!]_{LS}) \vee \neg\mathcal{I}[\![lp]\!] & \\
& = & \neg(\bigvee_{i \in 1..n} \mathcal{I}[\![\phi_i]\!]_{LS}) \vee \neg\mathcal{I}[\![lp]\!] & \text{DeMorgan's law}
\end{array}$$

The second point follows from the fact that the well-formedness constraints of the left side is a superset of the ones on the right.

CASE Consider any rule of the form $\pi \longrightarrow \theta$:

$$\begin{array}{rclcll}
\phi & \# & \mathtt{t} & \longrightarrow & [\phi \mapsto \bot] & \\
\phi & \| & \mathtt{t} & \longrightarrow & [\phi \mapsto \alpha] & \alpha \text{ new} \\
(\phi_1, \ldots, \phi_n) & \| & \mathtt{f} & \longrightarrow & [\phi_1 \mapsto \bot, \ldots, \phi_n \mapsto \bot] & \\[1ex]
\alpha & = & \tau & \longrightarrow & [\alpha \mapsto \tau] & \text{if } \alpha \text{ does not occur in } \tau \\
\phi & = & \hat{\tau} & \longrightarrow & [\phi \mapsto \hat{\tau}] & \text{if } \phi \text{ does not occur in } \hat{\tau} \\
(\phi_1, \ldots, \phi_n) & = & \bot & \longrightarrow & [\phi_1 \mapsto \bot, \ldots, \phi_n \mapsto \bot] &
\end{array}$$

In each of these cases, it is the case that the predicate implies that one or more field variables is either $\bot$, or is anything other than $\bot$. The substitution for the first case is obvious. In the second case, we can substitute a new type variable α for the field variable, as this covers all values other than $\bot$. In each case, we can apply Lemma B.2.3 to prove point 1. For point 2, we have two cases:

1. the values substituted for each field variable are $\bot$
2. the value substituted for the field variable is not $\bot$

To prove point 2, we start by restating what needs to be proven: given that $F = F' \uplus \{\pi\}$ is well-formed, we need to prove that $\theta F'$ is as well. By Lemma B.2.5, we have that $\theta(F' \uplus \{\pi\})$ is well-formed. By the semantic equivalence of $F' \uplus \{\pi\}$ and $\theta F'$,

$$(\mathcal{I}[\![F' \uplus \{\pi\}]\!] \rightarrow \mathcal{I}[\![\mathrm{WFC}[F' \uplus \{\pi\}]]\!]) \quad \rightarrow \quad (\mathcal{I}[\![\theta F']\!] \rightarrow \mathcal{I}[\![\mathrm{WFC}[F' \uplus \{\pi\}]]\!])$$

We also have that $\mathrm{WFC}[[\phi \mapsto \hat{\tau}]F'] \subseteq \mathrm{WFC}[F'] \cup \mathrm{WFC}[\hat{\tau}]$. For all of the rules considered in this case, $\hat{\tau}$ is either $\bot$, a new type variable, or a type which exists in the predicate π. In the first two cases, $\mathrm{WFC}[\hat{\tau}] = \varnothing$. In the last case, we have $\mathrm{WFC}[\hat{\tau}] = \mathrm{WFC}[\pi]$. We can combine these to get:

$$\mathrm{WFC}[\hat{\tau}] \subseteq \mathrm{WFC}[\pi]$$

Therefore, we have

$$\mathrm{WFC}[[\phi \mapsto \hat{\tau}]F'] \subseteq \mathrm{WFC}[F'] \cup \mathrm{WFC}[\pi]$$

and therefore, the resulting $\theta F'$ is well-formed.

Case $(\phi, \hat{\tau}) = (\phi, \hat{\tau}') \longrightarrow \hat{\tau} = \hat{\tau}'$

Point 1 can be proven for this rule by expanding the semantics of both sides. Expanding the left side:

$$\begin{aligned}
\mathcal{I}[\![(\phi, \hat{\tau}) = (\phi, \hat{\tau}')]\!] &= \mathcal{I}[\![(\phi, \hat{\tau})]\!] = \mathcal{I}[\![(\phi, \hat{\tau}')]\!] \\
&= (\mathcal{I}[\![\phi]\!] \,\dot{\cup}\, \mathcal{I}[\![\hat{\tau}]\!]) = (\mathcal{I}[\![\phi]\!] \,\dot{\cup}\, \mathcal{I}[\![\hat{\tau}']\!]) \\
&= \mathcal{I}[\![\hat{\tau}]\!] = \mathcal{I}[\![\hat{\tau}']\!]
\end{aligned}$$

The right side:

$$\mathcal{I}[\![\hat{\tau} = \hat{\tau}']\!] = \mathcal{I}[\![\hat{\tau}]\!] = \mathcal{I}[\![\hat{\tau}']\!]$$

Point 2: Using the conclusion for point 1, we can prove point 2 by a direct application of Lemma B.2.1.

Case $(\phi, lp) \parallel (\phi, lp') \longrightarrow lp \parallel lp'$

We start by expanding the definition of $\mathcal{I}[\![\cdot]\!]$ applied to both sides of the rule:

$$\begin{array}{lrl} \text{Left side:} & & \mathcal{I}[\![(\phi, \mathit{lp}) \parallel (\phi, \mathit{lp}')]\!] \\ & = & (\mathcal{I}[\![\phi]\!]_{LP} \vee \mathcal{I}[\![\mathit{lp}]\!]) \leftrightarrow (\mathcal{I}[\![\phi]\!]_{LP} \vee \mathcal{I}[\![\mathit{lp}']\!]) \\ \text{Right side:} & & \mathcal{I}[\![\mathit{lp} \parallel \mathit{lp}']\!] \\ & = & \mathcal{I}[\![\mathit{lp}]\!] \leftrightarrow \mathcal{I}[\![\mathit{lp}']\!] \end{array}$$

By the well-formedness of F, we have

$$\mathcal{I}[\![F]\!] \to (\mathcal{I}[\![\phi \# \mathit{lp}]\!] \wedge \mathcal{I}[\![\phi \# \mathit{lp}']\!])$$

Therefore, at least one of $\mathcal{I}[\![\phi]\!]_{LP}$, $\mathcal{I}[\![\mathit{lp}]\!]$ are false, and likewise for $\mathcal{I}[\![\phi]\!]_{LP}$, $\mathcal{I}[\![\mathit{lp}']\!]$.

There are two cases:

– $\mathcal{I}[\![\phi]\!]_{LP} = \mathit{false}$

Then the left hand side of the rule reduces to

$$\mathcal{I}[\![\mathit{lp}]\!] \leftrightarrow \mathcal{I}[\![\mathit{lp}']\!]$$

– $\mathcal{I}[\![\phi]\!]_{LP} \neq \mathit{false}$

Then both $\mathcal{I}[\![\mathit{lp}]\!]$ and $\mathcal{I}[\![\mathit{lp}']\!]$ must be *false*; the left hand side of the rule reduces to

$$\mathcal{I}[\![\phi]\!]_{LP} \leftrightarrow \mathcal{I}[\![\phi]\!]_{LP}$$

and the right hand side of the rule also reduces to *true*.

Case $\tau_1 \to \tau_2 = \tau_1' \to \tau_2' \longrightarrow \{\tau_1 = \tau_1', \tau_2 = \tau_2'\}$, and

$\{\mathit{row}\} = \{\mathit{row}'\} \longrightarrow \{\mathit{row}@\mathbf{l}(0) = \mathit{row}'@\mathbf{l}(0), \ldots, \mathit{row}@\mathbf{l}(n) = \mathit{row}'@\mathbf{l}(n)\}$

The correctness of the first rule follows from the structure of the type equality rules. The second rule follows from the composition of Proposition 5.5.1 and Lemma 5.5.2.

The well-formedness of the transformed predicate set follows from the fact that the WFC of both sides of the rules are equal.

CASE Consider the rule

$$\begin{aligned}
&(fe_1, \ldots, fe_n) = (fe'_1, \ldots, fe'_m) \longrightarrow H \\
&\text{where } H = \Big(\bigcup_{i \in 1..n} \{ fe_i = (fe_i/fe'_1, \ldots, fe_i/fe'_m) \} \cup \\
&\qquad\qquad \bigcup_{j \in 1..m} \{ fe'_j = (fe_1/fe'_j, \ldots, fe_n/fe'_j) \} \Big)
\end{aligned}$$

By the well-formedness of F, the interpretation of at most one of the fe_i's is not $\perp$, and similarly for the fe'_j's.

By Lemma B.2.4, the same can be said for each set of the restricted label presences

$$\{fe_i/fe'_j | j \in 1..m\} \quad \text{where } i \in 1..n$$

and also

$$\{fe'_j/fe_i | i \in 1..n\} \quad \text{where } j \in 1..m$$

The interpretations of each of $(fe_1, \ldots, fe_n)$ and $(fe'_1, \ldots, fe'_m)$ may be $\perp$ or τ, giving us four possibilities. Because $\#$ is symmetric, we need only consider three. Let *lhs* be the predicate on the left side of the rule, and *rhs* be the set of predicates on the right side of the rule.

– $\mathcal{I}[\![(fe_1, \ldots, fe_n)]\!] = \perp$ and $\mathcal{I}[\![(fe'_1, \ldots, fe'_m)]\!] = \perp$

Then the interpretation of each fe_i and fe'_j is $\bot$, and clearly $\mathcal{I}[\![lhs]\!] \leftrightarrow \mathcal{I}[\![rhs]\!]$.

– $\mathcal{I}[\![(fe_1, \ldots, fe_n)]\!] = \tau$ and $\mathcal{I}[\![(fe'_1, \ldots, fe'_m)]\!] = \bot$

We have that $\neg\mathcal{I}[\![lhs]\!]$. Let $\mathcal{I}[\![fe_k]\!] = \tau$. Then $\mathcal{I}[\![fe_i/fe'_j]\!] = \bot$ and $\mathcal{I}[\![fe'_j/fe_i]\!] = \bot$ for all i, j, and the interpretation of

$$fe_i = (fe_i/fe'_1, \ldots, fe_i/fe'_m)$$

is true for $i \neq k$, and false for $i = k$. Therefore, $\neg\mathcal{I}[\![fe_k]\!]$.

– $\mathcal{I}[\![(fe_1, \ldots, fe_n)]\!] = \tau$ and $\mathcal{I}[\![(fe'_1, \ldots, fe'_m)]\!] = \tau'$

We have that $\mathcal{I}[\![lhs]\!] \leftrightarrow \mathcal{I}[\![\tau = \tau']\!]$. Let $\mathcal{I}[\![fe_k]\!] = \tau$, and $\mathcal{I}[\![fe'_l]\!] = \tau'$. Then $\mathcal{I}[\![fe_i/fe'_j]\!] = \bot$ for all i, j, except $i = k, j = l$. Because $\mathcal{I}[\![fe_k/fe'_l]\!] = \mathcal{I}[\![fe_k]\!]$, $\mathcal{I}[\![fe_k = (fe_k/fe'_1, \ldots, fe_k/fe'_m)]\!]$. We also have that $\mathcal{I}[\![fe_i = (fe_i/fe'_1, \ldots, fe_i/fe'_m)]\!]$ for all $i \neq k$. Similarly, $\mathcal{I}[\![fe'_j = (fe_1/fe'_j, \ldots, fe_1/fe'_j)]\!]$ for all $j \neq l$. At $j = l$, we have that $fe'_l = (fe_1/fe'_l, \ldots, fe_1/fe'_l)$ can be written as $\tau' = \tau$, and therefore $\mathcal{I}[\![rhs]\!] \leftrightarrow \mathcal{I}[\![\tau = \tau']\!]$.

Case Consider the rule

$$\begin{aligned}
&(lpe_1, \ldots, lpe_n) \parallel (lpe'_1, \ldots, lpe'_m) \longrightarrow H \\
&\text{where } H = \Big(\bigcup_{i \in 1..n} \{\, lpe_i = (lpe_i/lpe'_1, \ldots, lpe_i/lpe'_m) \,\}\Big) \cup \\
&\qquad\qquad \Big(\bigcup_{j \in 1..m} \{\, lpe'_j = (lpe'_j/lpe_1, \ldots, lpe'_j/lpe_n) \,\}\Big)
\end{aligned}$$

This can be proven using the same reasoning as the previous case.

$\square$

B.3 Compilation

Proposition 6.3.1. *Any derivation of $P \Vdash M : lset_1 \# lset_2$ can be rewritten as a derivation where:*

1. *Each productive rule is followed by exactly one non-productive rule.*
2. *Each non-productive rule is either the last rule in the derivation, or is followed by a productive rule.*

All derivations of a judgement of the form $P \Vdash M : lset_1 \# lset_2$ must include one or more productive rules, because the non-productive rule relies on the result of a productive rule in its premise. In any derivation, each productive rule is followed by a sequence of zero or more non-productive rules. In the case where a productive rule is not followed by a non-productive rule, we can insert one because the $\parallel$ relation is symmetric:

$$\frac{P \Vdash M : lset_1 \# lset_2 \quad P \Vdash lset_1 \parallel lset_1 \quad P \Vdash lset_2 \parallel lset_2}{P \Vdash M : lset_1 \# lset_2}\ \textbf{\#-compatible-}\parallel$$

Similarly, a sequence of non-productive rules can be collapsed into one due to the transitivity of $\parallel$.

Proposition B.3.1. *If $P \Vdash W : lset_1 \# lset_2$ for some P, W, $lset_1$, and $lset_2$, then $P \Vdash \overline{W} : lset_2 \# lset_1$.*

Proof. Let W be $\langle s_1, \ldots, s_n \rangle$

Then all derivations of $P \Vdash W : lset_1 \# lset_2$ consist of an application of **#-null-null**, followed by a sequence of applications of the rules **#-labels-left**,

$\#$**-labels-right**, and $\#$**-compatible-**$\|$. Occurrences of $\ltimes$ ($\rtimes$) in W correspond to applications of $\#$**-labels-left** ($\#$**-labels-right**). There can be an arbitrary number of occurrences of $\#$**-compatible-**$\|$ interspersed before, between, and after the other rules.

Without loss of generality, we will assume that there is always exactly one application of this rule wherever a sequence may occur. This is justified by the transitivity of $\|$, which allows us to replace multiple applications of this rule by one, and by the identity of $\|$, which allows us to insert an application of this rule where none occurred, without changing the result of the derivation. We will make use of this convention in some of the proofs that follow.

We can take any such derivation, and replace each application of $\#$**-labels-left** with an application of $\#$**-labels-right** and vice versa and simultaneously swap each occurrence of $lset_1 \# lset_2$ with $lset_2 \# lset_1$ at each step in the derivation. The result is a valid derivation of $P \Vdash \overline{W} : lset_2 \# lset_1$. $\square$

Lemma B.3.1 (Subject Reduction for evidence (A)). *If there is a productive derivation of $P \Vdash M_1 : lset_1 \# lset_2$ where P is satisfiable, and $M_1 \longrightarrow M_2$, then there is a productive derivation of $P \Vdash M_2 : lset_1 \# lset_2$.*

Proof. The proof proceeds by considering all the cases for the reduction and applying induction on the sizes of the subterms. The induction hypothesis in each case is that, for some particular form of M and W, there is a reduction $M \longrightarrow W$, and if $P \Vdash M : \pi$, then $P \Vdash W : \pi$. For each such form of M, we have one or more extensions of M, which we call M^+, which reduce to W^+, which we show is some particular extension of W. We show that any judgement $P \Vdash M^+ : \pi$ can be

$$\begin{tikzcd}
M \arrow[r] & W \\
& \cdots \\
M^+ \arrow[r] & W^+
\end{tikzcd}$$

Figure B.1: Structure of the induction step in the proof of subject reduction for evidence

derived in a particular way, and likewise for M, W, and W^+. Finally, we show that the premises and side conditions in the derivations of M^+ and W^+ are implied by the derivations of M and W. This is described in Figure B.1.

Case $\mathbf{flip}(W) \longrightarrow \overline{W}$

By 6.3.1, the derivation of $P \Vdash \mathbf{flip}\, W : \mathit{lset}_1 \# \mathit{lset}_2$ can be decomposed into four parts:

1. A derivation of $P \Vdash W : \mathit{lset}''_1 \# \mathit{lset}''_2$
2. An application of **$\#$-compatible-$\|$**.
3. An application of **$\#$-flip**.
4. Another application of **$\#$-compatible-$\|$**.

We can therefore construct a derivation of $P \Vdash \overline{W} : \mathit{lset}_1 \# \mathit{lset}_2$ by substituting a derivation of $P \Vdash \overline{W} : \mathit{lset}''_2 \# \mathit{lset}''_1$ for the first derivation in the list above, and then collapsing the two applications of **$\#$-compatible-$\|$** into one.

Case $\mathbf{mrg}^s\,(W_1, W_2) \longrightarrow \mathit{spans}^{s^{-1}}(\mathit{spans}^s(W_1) + \mathit{spans}^s(W_2))$

We prove this by induction on the lengths of W_1 and W_2.

The base case is $\mathbf{mrg}^{\ltimes}\,(\langle\,\rangle,\langle\,\rangle)$. There is one productive derivation of this:

$$\frac{P \Vdash \langle\,\rangle : \varnothing \# \varnothing \quad P \Vdash \langle\,\rangle : \varnothing \# \varnothing \quad P \Vdash \varnothing \# \varnothing}{P \Vdash (\mathbf{mrg}^{\ltimes}\,(\langle\,\rangle,\langle\,\rangle)) : \varnothing \# \varnothing}$$

This reduces to $\mathit{spans}^{s^{-1}}(\mathit{spans}^{s}(\langle\,\rangle) + \mathit{spans}^{s}(\langle\,\rangle))$, which is $\langle\,\rangle$. There is also only one productive derivation of this:

$$\frac{}{P \Vdash \langle\,\rangle : \varnothing \# \varnothing}$$

By induction, we are given that the proposition is true for $\mathbf{mrg}^{s}\,(W_1, W_2)$, where $|W_1| = n$ and $|W_2| = m$. Let $W_{12} = \mathit{spans}^{s^{-1}}(\mathit{spans}^{s}(W_1) + \mathit{spans}^{s}(W_2))$. We can extend the result in three ways:

1. $\mathbf{mrg}^{s}\,(s \cdot W_1, W_2) \quad \longrightarrow \quad s \cdot W_{12}$
2. $\mathbf{mrg}^{s}\,(W_1, s \cdot W_2) \quad \longrightarrow \quad s \cdot W_{12}$
3. $\mathbf{mrg}^{s}\,(\overline{s} \cdot W_1, \overline{s} \cdot W_2) \quad \longrightarrow \quad \overline{s} \cdot W_{12}$

The structure of the proof for the first two cases is essentially the same. Likewise, the two possibilities for s lead to essentially the same proof. We therefore will give details for the first and third cases, where $s = \ltimes$.

The first case: $\mathbf{mrg}^{\ltimes}\,(\ltimes \cdot W_1, W_2) \quad \longrightarrow \quad \ltimes \cdot W_{12}$

Using Proposition 6.3.1, we can write any productive derivation of

$$P \Vdash (\mathbf{mrg}^{\ltimes}\,(\ltimes \cdot W_1, W_2)) : \mathit{lset}_1 \# \mathit{lset}_2$$

in the form

$$\frac{P \Vdash \ltimes \cdot W_1 : \mathit{lset}_{1a} \# \mathit{lset}'_2 \qquad \begin{array}{c} P \Vdash W_2 : \mathit{lset}_{1b} \# \mathit{lset}'_2 \\ P \Vdash \mathit{lset}_{1a} \# \mathit{lset}'_{1b} \end{array}}{P \Vdash (\mathbf{mrg}^{\ltimes}\,(\ltimes \cdot W_1, W_2)) : (\mathit{lset}_{1a}, \mathit{lset}_{1b}) \# \mathit{lset}'_2}$$

where $P \Vdash \ltimes \cdot W_1 : lset_{1a} \# lset'_2$ is derived

$$\dfrac{\dfrac{P \Vdash W_1 : lset'_{1a} \# (l_1, \ldots, l_n) \quad P \Vdash (l) \# lset'_{1a}}{P \Vdash \ltimes \cdot W_1 : (l, lset'_{1a}) \# (l_1, \ldots, l_n)} \quad \begin{array}{c} P \Vdash (l, lset'_{1a}) \parallel lset_{1a} \\ P \Vdash (l_1, \ldots, l_n) \parallel lset'_2 \end{array}}{P \Vdash \ltimes \cdot W_1 : lset_{1a} \# lset'_2}$$

Using this, we can construct the derivation:

$$\dfrac{\dfrac{\begin{array}{c} P \Vdash W_1 : lset'_{1a} \# (l_1, \ldots, l_n) \\ P \Vdash (l_1, \ldots, l_n) \parallel lset'_2 \end{array}}{P \Vdash W_1 : lset'_{1a} \# lset'_2} \quad \begin{array}{c} P \Vdash W_2 : lset_{1b} \# lset'_2 \\ P \Vdash lset'_{1a} \# lset_{1b} \end{array}}{P \Vdash (\mathbf{mrg}^{\ltimes}(W_1, W_2)) : (lset'_{1a}, lset_{1b}) \# lset'_2}$$

Each of the leaves of this derivation occurs in the derivation above, except for $P \Vdash lset'_{1a} \# lset_{1b}$. This can be derived:

$$\dfrac{\dfrac{\begin{array}{c} P \Vdash lset_{1a} \# lset_{1b} \\ P \Vdash (l, lset'_{1a}) \parallel lset_{1a} \end{array}}{P \Vdash (l, lset'_{1a}) \# lset_{1b}} \quad P \Vdash (l) \# lset'_{1a}}{P \Vdash lset'_{1a} \# lset_{1b}} \; \textbf{\#-compose-left}$$

By the induction hypothesis, there exists a productive derivation of

$$P \Vdash W_{12} : (lset'_{1a}, lset_{1b}) \# lset_2$$

From this, we can construct the productive derivation:

$$\dfrac{P \Vdash \ltimes \cdot W_{12} : (l, lset'_{1a}, lset_{1b}) \# (l_1, \ldots, l_n) \quad \begin{array}{c} P \Vdash (l, lset'_{1a}, lset_{1b}) \parallel lset_1 \\ P \Vdash (l_1, \ldots, l_n) \parallel lset_2 \end{array}}{P \Vdash \ltimes \cdot W_{12} : lset_1 \# lset_2}$$

where $P \Vdash \ltimes \cdot W_{12} : (l, lset'_{1a}, lset_{1b}) \# (l_1, \ldots, l_n)$ is derived:

$$\cfrac{\cfrac{P \Vdash W_{12} : (lset'_{1a}, lset_{1b}) \# lset_2 \quad P \Vdash lset_2 \parallel (l_1, \ldots, l_n)}{P \Vdash W_{12} : (lset'_{1a}, lset_{1b}) \# (l_1, \ldots, l_n)}}{P \Vdash \ltimes \cdot W_{12} : (l, lset'_{1a}, lset_{1b}) \# (l_1, \ldots, l_n)}$$

The third case: $\mathbf{mrg}^{\ltimes}(\rtimes \cdot W_1, \rtimes \cdot W_2) \quad \longrightarrow \quad \rtimes \cdot W_{12}$

By Proposition 6.3.1, the last rule of any productive derivation of the left hand side is:

$$\frac{P \Vdash \rtimes \cdot W_1 : lset_{1a} \# lset_2 \quad P \Vdash \rtimes \cdot W_2 : lset_{1b} \# lset_2 \quad P \Vdash lset_{1a} \# lset_{1b}}{P \Vdash (\mathbf{mrg}^{\ltimes}(\rtimes \cdot W_1, \rtimes \cdot W_2)) : (lset_{1a}, lset_{1b}) \# lset_2}$$

By Proposition 6.3.1, we can rewrite any derivation of

$$P \Vdash \rtimes \cdot W_1 : lset_{1a} \# lset_2$$

into the form:

$$\frac{\cfrac{P \Vdash W_1 : \overrightarrow{l_{1a}} \# \overrightarrow{l_{2a}} \quad l_A < (\overrightarrow{l_{1a}}, \overrightarrow{l_{2a}})}{P \Vdash \rtimes \cdot W_1 : \overrightarrow{l_{1a}} \# (l_a, \overrightarrow{l_{2a}})} \qquad \begin{array}{c} P \Vdash \overrightarrow{l_{1a}} \parallel lset_{1a} \\ P \Vdash (l_a, \overrightarrow{l_{2a}}) \parallel lset_2 \end{array}}{P \Vdash \rtimes \cdot W_1 : lset_{1a} \# lset_2}$$

and similarly for $P \Vdash \rtimes \cdot W_2 : lset_{1b} \# lset_2$

$$\frac{\cfrac{P \Vdash W_1 : \overrightarrow{l_{1b}} \# \overrightarrow{l_{2b}} \quad l_b < (\overrightarrow{l_{1b}}, \overrightarrow{l_{2b}})}{P \Vdash \rtimes \cdot W_1 : \overrightarrow{l_{1b}} \# (l_b, \overrightarrow{l_{2b}})} \qquad \begin{array}{c} P \Vdash \overrightarrow{l_{1b}} \parallel lset_{1b} \\ P \Vdash (l_b, \overrightarrow{l_{2b}}) \parallel lset_2 \end{array}}{P \Vdash \rtimes \cdot W_1 : lset_{1b} \# lset_2}$$

From the judgements $P \Vdash (l_a, \overrightarrow{l_{2a}}) \parallel lset_2$ and $P \Vdash (l_b, \overrightarrow{l_{2b}}) \parallel lset_2$, the side conditions $l_a < \overrightarrow{l_{2a}}$ and $l_b < \overrightarrow{l_{2b}}$, and the fact that P is satisfiable, we can conclude that $l_a \equiv l_b$, and $\overrightarrow{l_{2a}} \equiv \overrightarrow{l_{2b}}$. In the sequel, we will use l and $\overrightarrow{l_2}$ in place of these labels and labelsets, respectively. We can now construct a productive derivation of $P \Vdash \mathbf{mrg}^{\ltimes}(W_1, W_2) : (\overrightarrow{l_{1a}}, \overrightarrow{l_{1b}}) \mathrel{\#} \overrightarrow{l_2}$:

$$\frac{P \Vdash W_1 : \overrightarrow{l_{1a}} \mathrel{\#} \overrightarrow{l_2} \quad P \Vdash W_2 : \overrightarrow{l_{1b}} \mathrel{\#} \overrightarrow{l_2} \quad P \Vdash \overrightarrow{l_{1a}} \mathrel{\#} \overrightarrow{l_{1b}}}{P \Vdash (\mathbf{mrg}^{\ltimes}(W_1, W_2)) : (\overrightarrow{l_{1a}}, \overrightarrow{l_{1b}}) \mathrel{\#} \overrightarrow{l_2}}$$

By the induction hypothesis, there exists a productive derivation of $P \Vdash W_{12} : (\overrightarrow{l_{1a}}, \overrightarrow{l_{1b}}) \mathrel{\#} \overrightarrow{l_2}$. With this, we can construct a productive derivation of $P \Vdash \rtimes \cdot W_{12} : (\overrightarrow{l_{1a}}, \overrightarrow{l_{1b}}) \mathrel{\#} \overrightarrow{l_2}$:

$$\frac{\dfrac{P \Vdash W_{12} : (\overrightarrow{l_{1a}}, \overrightarrow{l_{1b}}) \mathrel{\#} \overrightarrow{l_2} \quad l < (\overrightarrow{l_{1a}}, \overrightarrow{l_{1b}}, \overrightarrow{l_2})}{P \Vdash \rtimes \cdot W_{12} : (\overrightarrow{l_{1a}}, \overrightarrow{l_{1b}}) \mathrel{\#} (l, \overrightarrow{l_2})} \quad \begin{array}{c} P \Vdash (\overrightarrow{l_{1a}}, \overrightarrow{l_{1b}}) \parallel (lset_{1a}, lset_{1b}) \\ P \Vdash (l, \overrightarrow{l_2}) \parallel lset_2 \end{array}}{P \Vdash \rtimes \cdot W_{12} : (lset_{1a}, lset_{1b}) \mathrel{\#} lset_2}$$

Case $\mathbf{xtrct}^s\, W_1\, W_2 \longrightarrow \textit{merge-extract}^s(W_1, W_2)$

Just as for $\mathbf{mrg}^s(W_1, W_2)$, we use induction on the lengths of the arguments. The two choices for s are symmetric; as in the last subproblem, we show the proof for $s = \ltimes$. However, we do not have symmetry in the arguments to $\mathbf{xtrct}^s$, so more combinations need to be dealt with explicitly.

Base case: $\mathbf{xtrct}^{\ltimes} \langle\rangle\langle\rangle \longrightarrow \langle\rangle$

There is only one productive derivation of the left hand side, namely:

$$\frac{P \Vdash \langle\rangle : (\varnothing, \varnothing) \mathrel{\#} \varnothing \quad P \Vdash \langle\rangle : \varnothing \mathrel{\#} \varnothing}{P \Vdash (\mathbf{xtrct}^{\ltimes} \langle\rangle\langle\rangle) : \varnothing \mathrel{\#} \varnothing}\ \textbf{\#-compose-left}$$

Likewise, there is is only one productive derivation of the right hand side; the **#-null-null** axiom.

Induction cases: If the lemma holds for $|W_1| = n$, and $|W_2| = m$, then it holds for various extensions of W_1 and W_2. There are three possible extensions of W_1 and W_2, due to the constraints placed on the operands of $\mathbf{xtrct}^s\,\_\,\_$. These follow directly from the definition of *merge-extract*$^{\ltimes}$. In the following, let $W_{12} = \textit{merge-extract}^s\,(W_1, W_2)$.

1. $\mathbf{xtrct}^s\, W_1\, (\overline{s} \cdot W_2) \longrightarrow \overline{s} \cdot W_{12}$
2. $\mathbf{xtrct}^s\, (s \cdot W_1)\, (s \cdot W_2) \longrightarrow s \cdot W_{12}$
3. $\mathbf{xtrct}^s\, (\overline{s} \cdot W_1)\, (s \cdot W_2) \longrightarrow W_{12}$

As promised, we will only consider the situation where $s = \ltimes$.

Firstly, any productive derivation of the left hand side of the induction hypothesis has the form:

$$\frac{\begin{array}{c} P \Vdash W_2 : (l_{11}, \ldots, l_{1n}, l_{21}, \ldots, l_{2n}) \mathbin{\#} (l_1, \ldots, l_p) \\ P \Vdash W_1 : (l_{11}, \ldots, l_{1n}) \mathbin{\#} (l_{21}, \ldots, l_{2n}) \end{array}}{P \Vdash (\mathbf{xtrct}^{\ltimes}\, W_1\, W_2) : (l_{11}, \ldots, l_{1n}) \mathbin{\#} (l_1, \ldots, l_p)}$$

We now proceed by subcases:

– $\mathbf{xtrct}^s\, W_1\, (\overline{s} \cdot W_2) \longrightarrow \overline{s} \cdot W_{12}$

By Proposition 6.3.1, any productive derivation of the left hand side can be written in the form

$$\frac{P \Vdash \rtimes \cdot W_2 : (lset_1, lset_2) \mathbin{\#} lset \qquad \dfrac{P \Vdash W_1 : \overrightarrow{l_1} \mathbin{\#} \overrightarrow{l_2} \qquad \begin{array}{c} P \Vdash \overrightarrow{l_1} \parallel lset_1 \\ P \Vdash \overrightarrow{l_2} \parallel lset_2 \end{array}}{P \Vdash W_1 : lset_1 \mathbin{\#} lset_2}}{P \Vdash (\mathbf{xtrct}^{\ltimes}\, W_1\, (\rtimes \cdot W_2)) : lset_1 \mathbin{\#} lset}$$

where $P \Vdash \rtimes \cdot W_2 : (\mathit{lset}_1, \mathit{lset}_2) \# \mathit{lset}$ is derived:

$$\dfrac{\dfrac{P \Vdash W_2 : \overrightarrow{l_{12}} \# \overrightarrow{l} \qquad \begin{array}{c} P \Vdash l \# \overrightarrow{l} \\ l < \overrightarrow{l_{12}} \end{array}}{P \Vdash \rtimes \cdot W_2 : \overrightarrow{l_{12}} \# (l, \overrightarrow{l})} \qquad \begin{array}{c} P \Vdash \overrightarrow{l_{12}} \parallel (\mathit{lset}_1, \mathit{lset}_2) \\ P \Vdash (l, \overrightarrow{l}) \parallel \mathit{lset} \end{array}}{P \Vdash \rtimes \cdot W_2 : (\mathit{lset}_1, \mathit{lset}_2) \# \mathit{lset}}$$

Because P is satisfiable, it is the case that $(\overrightarrow{l_1}, \overrightarrow{l_2}) \equiv \overrightarrow{l_{12}}$.

We can construct the productive derivation

$$\dfrac{P \Vdash W_2 : (\overrightarrow{l_1}, \overrightarrow{l_2}) \# \overrightarrow{l} \quad P \Vdash W_1 : \overrightarrow{l_1} \# \overrightarrow{l_2}}{P \Vdash (\mathbf{xtrct}^{\ltimes}\, W_1\, W_2) : \overrightarrow{l_1} \# \overrightarrow{l}}$$

By the induction hypothesis, there exists a productive derivation of $P \Vdash W_{12} : \overrightarrow{l_1} \# \overrightarrow{l}$. We can use this derivation to construct a productive derivation of $P \Vdash (\rtimes \cdot W_{12}) : \mathit{lset}_1 \# \mathit{lset}$:

$$\dfrac{\dfrac{P \Vdash W_{12} : \overrightarrow{l_1} \# \overrightarrow{l} \quad P \Vdash l \# \overrightarrow{l} \quad l < \overrightarrow{l_1}}{P \Vdash (\rtimes \cdot W_{12}) : \overrightarrow{l_1} \# (l, \overrightarrow{l})} \qquad \begin{array}{c} P \Vdash \overrightarrow{l_1} \parallel \mathit{lset}_1 \\ P \Vdash (l, \overrightarrow{l}) \parallel \mathit{lset} \end{array}}{P \Vdash (\rtimes \cdot W_{12}) : \mathit{lset}_1 \# \mathit{lset}}$$

– $\mathbf{xtrct}^{s}\, (s \cdot W_1)\, (s \cdot W_2) \longrightarrow s \cdot W_{12}$

The last rule of any productive derivation of the left hand size is:

$$\dfrac{P \Vdash \ltimes \cdot W_2 : (\mathit{lset}_1, \mathit{lset}_2) \# \mathit{lset} \quad P \Vdash \ltimes \cdot W_1 : \mathit{lset}_1 \# \mathit{lset}_2}{P \Vdash (\mathbf{xtrct}^{\ltimes}\, (\ltimes \cdot W_1)\, (\ltimes \cdot W_2)) : \mathit{lset}_1 \# \mathit{lset}}$$

By Proposition 6.3.1, we can rewrite any derivation of $P \Vdash \ltimes \cdot W_2 :$

$(\mathit{lset}_1, \mathit{lset}_2) \# \mathit{lset}$ into the form:

$$\dfrac{\dfrac{P \Vdash W_2 : \overrightarrow{l_{12}} \# \overrightarrow{l} \quad l_A < (\overrightarrow{l}, \overrightarrow{l_{12}})}{P \Vdash \ltimes \cdot W_2 : (l_A, \overrightarrow{l_{12}}) \# \overrightarrow{l}} \qquad \begin{array}{c} P \Vdash (l_A, \overrightarrow{l_{12}}) \parallel (\mathit{lset}_1, \mathit{lset}_2) \\ P \Vdash \overrightarrow{l} \parallel \mathit{lset} \end{array}}{P \Vdash \ltimes \cdot W_2 : (\mathit{lset}_1, \mathit{lset}_2) \# \mathit{lset}}$$

and similarly for $P \Vdash \ltimes \cdot W_1 : \mathit{lset}_1 \# \mathit{lset}_2$:

$$\dfrac{\dfrac{P \Vdash W_1 : \overrightarrow{l_1} \# \overrightarrow{l_2} \quad l_B < (\overrightarrow{l_1}, \overrightarrow{l_2})}{P \Vdash \ltimes \cdot W_1 : (l_B, \overrightarrow{l_1}) \# \overrightarrow{l_2}} \qquad \begin{array}{c} P \Vdash (l_B, \overrightarrow{l_1}) \parallel \mathit{lset}_1 \\ P \Vdash \overrightarrow{l_2} \parallel \mathit{lset}_2 \end{array}}{P \Vdash \ltimes \cdot W_1 : \mathit{lset}_1 \# \mathit{lset}_2}$$

We can infer that $(l_A, \overrightarrow{l_{12}}) \equiv (l_B, \overrightarrow{l_1}, \overrightarrow{l_2})$ from the $\parallel$ predicates in the two derivations. Because $l_A < \overrightarrow{l_{12}}$ and $l_B < (\overrightarrow{l_1}, \overrightarrow{l_2})$, l_A is the minimum element of $(l_A, \overrightarrow{l_{12}})$, and l_B is the minimum element of $(l_B, \overrightarrow{l_1}, \overrightarrow{l_2})$. Since these two sets are equal, it follows that $l_A \equiv l_B$. With this in mind, we can now construct a productive derivation of $P \Vdash (\mathbf{xtrct}^{\ltimes}\, W_1\, W_2) : \overrightarrow{l_1} \# \overrightarrow{l}$:

$$\dfrac{P \Vdash W_1 : \overrightarrow{l_1} \# \overrightarrow{l_2} \qquad \dfrac{P \Vdash W_2 : \overrightarrow{l_{12}} \# \overrightarrow{l} \quad P \Vdash \overrightarrow{l_{12}} \parallel (\overrightarrow{l_1}, \overrightarrow{l_2})}{P \Vdash W_2 : (\overrightarrow{l_1}, \overrightarrow{l_2}) \# \overrightarrow{l}}}{P \Vdash (\mathbf{xtrct}^{\ltimes}\, W_1\, W_2) : \overrightarrow{l_1} \# \overrightarrow{l}}$$

By the induction hypothesis, we have a productive derivation of $P \Vdash W_{12} : \overrightarrow{l_1} \# \overrightarrow{l}$, which we can use to construct a productive derivation of $P \Vdash \ltimes \cdot W_{12} : \mathit{lset}_1 \# \mathit{lset}$:

$$\dfrac{\dfrac{P \Vdash W_{12} : \overrightarrow{l_1} \# \overrightarrow{l} \quad l_A < (\overrightarrow{l_1}, \overrightarrow{l})}{P \Vdash \ltimes \cdot W_{12} : (l_a, \overrightarrow{l_1}) \# \overrightarrow{l}} \qquad \begin{array}{c} P \Vdash (l_a, \overrightarrow{l_1}) \parallel \mathit{lset}_1 \\ P \Vdash \overrightarrow{l} \parallel \mathit{lset} \end{array}}{P \Vdash \ltimes \cdot W_{12} : \mathit{lset}_1 \# \mathit{lset}}$$

– $\mathbf{xtrct}^{s}\,(\overline{s}\cdot W_1)\,(s\cdot W_2) \longrightarrow W_{12}$

The last rule of any productive derivation of the left-hand size is:

$$\frac{P \Vdash \ltimes \cdot W_2 : (\mathit{lset}_1, \mathit{lset}_2) \# \mathit{lset} \quad P \Vdash \rtimes \cdot W_1 : \mathit{lset}_1 \# \mathit{lset}_2}{P \Vdash (\mathbf{xtrct}^{\ltimes}\,(\rtimes \cdot W_1)\,(\ltimes \cdot W_2)) : \mathit{lset}_1 \# \mathit{lset}}$$

By Proposition 6.3.1, we can rewrite any derivation of $P \Vdash \ltimes \cdot W_2 : (\mathit{lset}_1, \mathit{lset}_2) \# \mathit{lset}$ into the form:

$$\frac{\dfrac{P \Vdash W_2 : \overrightarrow{l_{12}} \# \overrightarrow{l} \quad l_A < (\overrightarrow{l}, \overrightarrow{l_{12}})}{P \Vdash \ltimes \cdot W_2 : (l_A, \overrightarrow{l_{12}}) \# \overrightarrow{l}} \quad \begin{array}{c} P \Vdash (l_A, \overrightarrow{l_{12}}) \parallel (\mathit{lset}_1, \mathit{lset}_2) \\ P \Vdash \overrightarrow{l} \parallel \mathit{lset} \end{array}}{P \Vdash \ltimes \cdot W_2 : (\mathit{lset}_1, \mathit{lset}_2) \# \mathit{lset}}$$

and similarly for $P \Vdash \rtimes \cdot W_1 : \mathit{lset}_1 \# \mathit{lset}_2$:

$$\frac{\dfrac{P \Vdash W_1 : \overrightarrow{l_1} \# \overrightarrow{l_2} \quad l_B < (\overrightarrow{l_1}, \overrightarrow{l_2})}{P \Vdash \rtimes \cdot W_1 : \overrightarrow{l_1} \# (l_B, \overrightarrow{l_2})} \quad \begin{array}{c} P \Vdash \overrightarrow{l_1} \parallel \mathit{lset}_1 \\ P \Vdash (l_B, \overrightarrow{l_2}) \parallel \mathit{lset}_2 \end{array}}{P \Vdash \rtimes \cdot W_1 : \mathit{lset}_1 \# \mathit{lset}_2}$$

By the same reasoning as in the last case, we can conclude that $l_A \equiv l_B$, and construct a productive derivation of $P \Vdash (\mathbf{xtrct}^{\ltimes}\,W_1\,W_2) : \overrightarrow{l_1} \# \overrightarrow{l}$:

$$\frac{P \Vdash W_1 : \overrightarrow{l_1} \# \overrightarrow{l_2} \quad \dfrac{P \Vdash W_2 : \overrightarrow{l_{12}} \# \overrightarrow{l} \quad P \Vdash \overrightarrow{l_{12}} \parallel (\overrightarrow{l_1}, \overrightarrow{l_2})}{P \Vdash W_2 : (\overrightarrow{l_1}, \overrightarrow{l_2}) \# \overrightarrow{l}}}{P \Vdash (\mathbf{xtrct}^{\ltimes}\,W_1\,W_2) : \overrightarrow{l_1} \# \overrightarrow{l}}$$

We can construct the productive derivation

$$\frac{P \Vdash W_2 : (\overrightarrow{l_1}, \overrightarrow{l_2}) \# \overrightarrow{l} \quad P \Vdash W_1 : \overrightarrow{l_1} \# \overrightarrow{l_2}}{P \Vdash (\mathbf{xtrct}^{\ltimes}\,W_1\,W_2) : \overrightarrow{l_1} \# \overrightarrow{l}}$$

By the induction hypothesis, there exists a productive derivation of $P \Vdash W_{12} : \overrightarrow{l_1} \# \overrightarrow{l}$. We can use this derivation to construct a productive derivation of $P \Vdash W_{12} : \mathit{lset}_1 \# \mathit{lset}$:

$$\frac{P \Vdash W_{12} : \overrightarrow{l_1} \# \overrightarrow{l} \qquad \begin{array}{c} P \Vdash \overrightarrow{l_1} \parallel \mathit{lset}_1 \\ P \Vdash \overrightarrow{l} \parallel \mathit{lset} \end{array}}{P \Vdash W_{12} : \mathit{lset}_1 \# \mathit{lset}}$$

□

Lemma 6.3.1 (Subject Reduction for evidence). *If $P \Vdash M_1 : \mathit{lset}_1 \# \mathit{lset}_2$ where P is satisfiable, and $M_1 \longrightarrow M_2$, then $P \Vdash M_2 : \mathit{lset}_1 \# \mathit{lset}_2$.*

Proof. If there is a productive derivation of $P \Vdash M_1 : \mathit{lset}_1 \# \mathit{lset}_2$, the result is immediate, by the previous Lemma. If the only derivation of $P \Vdash M_1 : \mathit{lset}_1 \# \mathit{lset}_2$ is non-productive, it can be written as:

$$\frac{P \Vdash M_1 : \mathit{lset}'_1 \# \mathit{lset}'_2 \quad P \Vdash \mathit{lset}'_1 \parallel \mathit{lset}_1 \quad P \Vdash \mathit{lset}'_2 \parallel \mathit{lset}_2}{P \Vdash M_1 : \mathit{lset}_1 \# \mathit{lset}_2}$$

where $P \Vdash M_1 : \mathit{lset}'_1 \# \mathit{lset}'_2$ is productive. From the previous lemma, it follows that we have a productive derivation of $P \Vdash M_2 : \mathit{lset}'_1 \# \mathit{lset}'_2$, with which we can derive:

$$\frac{P \Vdash M_2 : \mathit{lset}'_1 \# \mathit{lset}'_2 \quad P \Vdash \mathit{lset}'_1 \parallel \mathit{lset}_1 \quad P \Vdash \mathit{lset}'_2 \parallel \mathit{lset}_2}{P \Vdash M_2 : \mathit{lset}_1 \# \mathit{lset}_2}$$

□

Let L range over labelsets, possibly empty, which consist solely of labels (i.e., which, unlike *lset*, contain no variables). Let $L[i]$ be the ith smallest label in L. That is, there are exactly i labels in L that are smaller than or equal to $L[i]$.

Lemma B.3.2. *If $P \Vdash W : L_a \# L_b$, then*

$$W[i] = \begin{cases} \ltimes & \textit{if } (L_a \cup L_b)[i] \in L_a \\ \rtimes & \textit{if } (L_a \cup L_b)[i] \in L_b \end{cases}$$

for $1 \leqslant i \leqslant |L_a \cup L_b|$

Proof. The proof proceeds by induction on the length of a derivation.

The base case is $w = \langle\,\rangle$, which is derived by the rule

$$(\textbf{\#-null-null}) \quad P \Vdash \langle\,\rangle : \varnothing \# \varnothing$$

The consequent of the lemma is trivially true.

There are two induction cases, but they are symmetric, so we need only consider one:

Let $W = \ltimes \cdot W'$. Then we have the derivation:

$$(\textbf{\#-labels-left}) \quad \frac{P \Vdash W' : (L'_a) \# (L_b) \quad l_1 < (L'_a \cup L_b)[1]}{P \Vdash \ltimes \cdot W' : L_a \# L_b}$$

where $L_a = L'_a \cup \{l_1\}$, for some L'_a, L_b, and l_1. We have that $W'[i] = W[i+1]$, so by the induction hypothesis, we have that

$$W'[i] = W[i+1] = \begin{cases} \ltimes & \text{if } (L'_a \cup L_b)[i] \in L'_a \\ \rtimes & \text{if } (L'_a \cup L_b)[i] \in L_b \end{cases}$$

for $1 \leqslant i \leqslant |L'_a \cup L_b|$.

We have that $W[1] = \ltimes$ by definition; it is also the case that $(L_a \cup L_b)[1] = l_1 \in L_a$. Making use of this, and also the fact that $(L'_a \cup L_b)[i] = (L_a \cup L_b)[i+1]$, we can conclude that

$$W[i] = \begin{cases} \ltimes & \text{if } (L_a \cup L_b)[i] \in L_a \\ \rtimes & \text{if } (L_a \cup L_b)[i] \in L_b \end{cases}$$

for $1 \leqslant i \leqslant |L_a \cup L_b|$.

$\square$

Lemma 6.3.2 (Subject Reduction – target version). *For a target language expression E_1, where $P|A \overset{T}{\vdash} E_1 : \tau$, for some P, A, and τ, and $E_1 \longrightarrow E_2$, then $P|A \overset{T}{\vdash} E_2 : \tau$.*

Proof. The proof proceeds by considering all the cases for the reduction $E_1 \longrightarrow E_2$.

Case $(\lambda x.E)V \longrightarrow [x \mapsto V]E$

See [61].

Case **let** $x = V$ **in** $E \longrightarrow [x \mapsto V]E$

See [61].

Case $(\lambda m.E)W \longrightarrow [m \mapsto W]E$

This case is exactly analogous to the "regular" app expression.

Case $\mathbf{extract}^{\ltimes}\ W\ \{\!|V_1, \ldots, V_n|\!\} \longrightarrow \{\!|V_{i_1}, \ldots, V_{i_m}|\!\}$

We have that

$$\frac{P|A \overset{T}{\vdash} \{\!|V_1, \ldots, V_n|\!\} : \{\!|\overrightarrow{l_a : \tau_a}, \overrightarrow{l_b : \tau_b}|\!\} \quad P \Vdash W : \overrightarrow{l_a} \# \overrightarrow{l_b}}{P|A \overset{T}{\vdash} \mathbf{extract}^{\ltimes}\ W\ \{\!|V_1, \ldots, V_n|\!\} : \{\!|\overrightarrow{l_a : \tau_a}|\!\}}\ \textbf{extract-left}$$

By Lemma B.3.2, the field values $V_{i_1}, \ldots, V_{i_m}$ extracted are exactly those corresponding to the type $\{\!|\overrightarrow{l_a : \tau_a}|\!\}$, and so we can conclude that

$$P|A \overset{T}{\vdash} \{\!|V_{i_1}, \ldots, V_{i_m}|\!\} : \{\!|\overrightarrow{l_a : \tau_a}|\!\}.$$

Case **extract**$^{\rtimes}\, W\, \{\!|V_1, \ldots, V_n|\!\} \longrightarrow \{\!|V_{i_1}, \ldots, V_{i_m}|\!\}$

Analogous to previous case.

Case **merge**$\, W\, (\{\!|V_1, \ldots, V_n|\!\}, \{\!|V'_1, \ldots, V'_m|\!\}) \longrightarrow \{\!|V''_1, \ldots, V''_{n+m}|\!\}$

$$\text{where } V''_i = \begin{cases} V_{|W[1..i]|^{\ltimes}} & \text{if } W[i] = \ltimes \\ V'_{|W[1..i]|^{\rtimes}} & \text{if } W[i] = \rtimes \end{cases}$$

The proof of this case proceeds by induction on the structure of a derivation.

– $W = \langle\,\rangle$ (base case)

We have the derivation

$$\frac{P|A \overset{\mathcal{T}}{\vDash} V : \{\mathit{row}\} \quad P|A \overset{\mathcal{T}}{\vDash} V' : \{\mathit{row}'\} \quad P \Vdash \langle\,\rangle : \mathit{row} \# \mathit{row}'}{P|A \overset{\mathcal{T}}{\vDash} \textbf{merge}\langle\,\rangle(V, V') : \{\mathit{row}, \mathit{row}'\}}$$

The only possible derivation of $P \Vdash \langle\,\rangle : \mathit{row} \# \mathit{row}'$ is:

$$(\#\textbf{-null-null}) \quad P \Vdash \langle\,\rangle : \varnothing \# \varnothing$$

Therefore, row and row' are both $\varnothing$, and therefore, V and V' are both $\{\!|\,|\!\}$. Summarizing, we have:

$$\frac{P|A \overset{\mathcal{T}}{\vDash} \{\!|\,|\!\} : \{\} \quad P|A \overset{\mathcal{T}}{\vDash} \{\!|\,|\!\} : \{\} \quad P \Vdash \langle\,\rangle : \varnothing \# \varnothing}{P|A \overset{\mathcal{T}}{\vDash} \textbf{merge}\langle\,\rangle(\{\!|\,|\!\}, \{\!|\,|\!\}) : \{\}}$$

This merge expression reduces to $\{\!|\,|\!\}$, which has the derivation

$$\frac{}{P|A \overset{\mathcal{T}}{\vDash} \{\!|\,|\!\} : \{\}}\ \textbf{record}$$

– $W = \ltimes \cdot W'$ (induction case)

We have the derivations

$$\dfrac{\begin{array}{c} P|A \overset{T}{\models} \{\!|\overrightarrow{V}|\!\} : \{\overrightarrow{l_a : \tau_a}\} \\ P|A \overset{T}{\models} \{\!|\overrightarrow{V'}|\!\} : \{\overrightarrow{l_b : \tau_b}\} \end{array} \quad P \Vdash \ltimes \cdot W' : \overrightarrow{l_a} \mathbin{\#} \overrightarrow{l_b}}{P|A \overset{T}{\models} \textbf{merge}\, \ltimes \cdot W'\, (\{\!|\overrightarrow{V}|\!\}, \{\!|\overrightarrow{V'}|\!\}) : \{\overrightarrow{l_a : \tau_a}, \overrightarrow{l_b : \tau_b}\}}\ \textbf{merge}$$

and

$$\dfrac{P \Vdash W' : (l_{a2}, \ldots, l_{an}) \mathbin{\#} \overrightarrow{l_b} \quad l_{a1} < (l_{a2}, \ldots, l_{an}, \overrightarrow{l_b})}{P \Vdash \ltimes \cdot W' : \overrightarrow{l_a} \mathbin{\#} \overrightarrow{l_b}}\ \textbf{\#-labels-left}$$

Because l_{a1} is strictly smaller than any of the other labels $l_{a2}, \ldots, l_{an}$, $\overrightarrow{l_b}$, $V''_1 = V_1$. We can also construct the derivation

$$\dfrac{\begin{array}{c} P|A \overset{T}{\models} \{\!|V_2, \ldots, V_n|\!\} : \{row\} \\ P|A \overset{T}{\models} \{\!|V'_1, \ldots, V'_n|\!\} : \{\overrightarrow{l_b : \tau_b}\} \end{array} \quad P \Vdash W' : row \mathbin{\#} \overrightarrow{l_b}}{P|A \overset{T}{\models} \textbf{merge}\, W'\, (\{\!|V_2, \ldots, V_n|\!\}, \{\!|V'_1, \ldots, V'_m|\!\}) : \{row, \overrightarrow{l_b : \tau_b}\}}\ \textbf{merge}$$

where row is $(l_{a2} : \tau_{a2}, \ldots, l_{an} : \tau_{an})$. This merge expression reduces to $\{\!|V''_2, \ldots, V''_{n+m}|\!\}$. By the induction hypothesis,

$$P|A \overset{T}{\models} \{\!|V''_2, \ldots, V''_{n+m}|\!\} : \{l_{a2} : \tau_{a2}, \ldots, l_{an} : \tau_{an}, \overrightarrow{l_b : \tau_b}\}.$$

– $W = \rtimes \cdot W'$.

This case proceeds the same way as the preceding.

CASE $\textbf{index}^s W \{\!|V_1, \ldots, V_n|\!\}$

The proof of this case also proceeds by induction on the structure of a derivation.

– $W = \ltimes \cdot \overrightarrow{\rtimes}$

We have the derivations

$$\frac{P \Vdash W' : \varnothing \mathrel{\#} \overrightarrow{l} \quad l < \overrightarrow{l}}{P \Vdash \ltimes \cdot \overrightarrow{\rtimes} : l \mathrel{\#} \overrightarrow{l}} \ \textbf{\#-labels-left}$$

and

$$\frac{P|A \overset{\mathcal{T}}{\vDash} \{\!|V_1, \ldots, V_n|\!\} : \{row, l : \tau\} \quad P \Vdash (\ltimes \cdot \overrightarrow{\rtimes}) : l \mathrel{\#} \overrightarrow{l}}{P|A \overset{\mathcal{T}}{\vDash} \mathbf{index}^{\ltimes}(\ltimes \cdot \overrightarrow{\rtimes})\{\!|V_1, \ldots, V_n|\!\} : \tau} \ \textbf{index-left}$$

Because l is smaller than all the other labels in $\overrightarrow{l}$, the index expression reduces to V_1, which, by the premises of the second derivation, is of type τ.

– $W = \rtimes \cdot W'$ where W' contains exactly one occurrence of $\ltimes$.

We have the derivations

$$\frac{P \Vdash W' : l_k \mathrel{\#} (l_2, \ldots, l_{k-1}, l_{k+1}, \ldots, l_n)}{P \Vdash \rtimes \cdot W' : l_k \mathrel{\#} (l_1, \ldots, l_{k-1}, l_{k+1}, \ldots, l_n)} \ \textbf{\#-labels-right}$$

where $l_1 < l_2 < \ldots < l_n$, and

$$\frac{\begin{array}{c} P|A \overset{\mathcal{T}}{\vDash} \{\!|V_1, \ldots, V_n|\!\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \\ P \Vdash (\rtimes \cdot W') : l_k \mathrel{\#} (l_1, \ldots, l_{k-1}, l_{k+1}, \ldots, l_n) \end{array}}{P|A \overset{\mathcal{T}}{\vDash} \mathbf{index}^{\ltimes}(\rtimes \cdot W')\{\!|V_1, \ldots, V_n|\!\} : \tau_k} \ \textbf{index-left}$$

We can construct the derivation

$$\frac{\begin{array}{c} P|A \overset{\mathcal{T}}{\vDash} \{\!|V_2, \ldots, V_n|\!\} : \{l_2 : \tau_2, \ldots, l_n : \tau_n\} \\ P \Vdash W' : l_k \mathrel{\#} (l_2, \ldots, l_{k-1}, l_{k+1}, \ldots, l_n) \end{array}}{P|A \overset{\mathcal{T}}{\vDash} \mathbf{index}^{\ltimes} W'\{\!|V_2, \ldots, V_n|\!\} : \tau_k} \ \textbf{index-left}$$

By the inductive hypothesis, $\mathbf{index}^{\ltimes} W' \{\!| V_2, \ldots, V_n |\!\}$ reduces to V_k, and $P|A \overset{T}{\vdash} V_k : \tau_k$. But $\mathbf{index}^{\ltimes}(\rtimes \cdot W')\{\!| V_1, \ldots, V_n |\!\}$ also reduces to V_k.

$\square$

Lemma B.3.3 (Uniform Evaluation for evidence). *For closed E, if there is no E' such that $E \longmapsto\!\!\!\!\rightarrow E'$ and E' is faulty, then either E diverges, or $E \longmapsto\!\!\!\!\rightarrow V$.*

Proof. The proof of this is an adaptation of the proof of Lemma 4.10 in [61]. The difference is that we have a number of different forms.

By induction on the length of the reduction sequence, we need only show that one of the following cases must hold:

1. E is faulty,
2. $E \longmapsto E'$ and E' is closed, or
3. E is a value.

Note that $E \longmapsto E'$ iff $E = \mathcal{E}[E_1]$, $E' = \mathcal{E}[E_1']$, and $E_1 \longrightarrow E_1'$.

The proof proceeds by induction on the structure of E. We need only consider the cases not covered in [61].

Case m

This is not a closed expression

Case $\langle s, \ldots, s \rangle$

This is a value.

Case $\mathbf{flip}\, M'$

If M is a value W, then we have $E \longmapsto \overline{W}$. Otherwise, if $M' \longmapsto M''$, then $M' = \mathcal{M}_1[M_1']$ and $M'' = \mathcal{M}_1[M_1'']$. Therefore, $M \longmapsto \mathcal{M}[M_1'']$, where $\mathcal{M} = \mathbf{flip}\,\mathcal{M}_1$.

CASE $\mathbf{xtrct}^s\, M_1\, M_2$

If M_1 and M_2 are values W_1 and W_2, respectively, and $|W_2|^s = |W_1|$, then we have $M \longmapsto$ *merge-extract*$^s(W_1, W_2)$. If, on the other hand, $|W_2|^s \neq |W_1|$, then E is faulty. Otherwise, one or both of M_1, M_2 are not values. If M_1 is not a value, then $M_1 = \mathcal{M}_1[M_a]$, $M_1' = \mathcal{M}_1[M_a']$, and $M_a \longrightarrow M_a'$. Then $E \longmapsto \mathcal{M}[M_a']$, where $\mathcal{M} = \mathbf{xtrct}^s\,\mathcal{M}_1\, M$, and similarly if M_2 is not a value.

CASE $\mathbf{mrg}^s\,(M, M)$

If M_1 and M_2 are values W_1 and W_2, respectively, and one of the three notions of reduction for $\mathbf{mrg}^s\,(W_1, W_2)$ can be applied, then we have $M \longmapsto M'$ for an appropriate M'. If none of the three notions apply, then M is faulty. Otherwise, one or both of M_1, M_2 are not values. If M_1 is not a value, then $M_1 = \mathcal{M}_1[M_a]$, $M_1' = \mathcal{M}_1[M_a']$, and $M_a \longrightarrow M_a'$. Then $E \longmapsto \mathcal{M}[M_a']$, where $\mathcal{M} = \mathbf{mrg}^s\,(\mathcal{M}_1, M)$, and similarly if M_2 is not a value.

$\square$

Lemma 6.3.3 (Uniform evaluation – target version). *For closed target language expression E, if there is no E' such that $E \longmapsto\!\!\!\!\!\rightarrow E'$ and E' is faulty, then either E diverges, or $E \longmapsto\!\!\!\!\!\rightarrow V$.*

Proof. The proof of this is an adaptation of the proof of Lemma 4.10 in [61]. The main difference is that we have more cases.

By induction on the length of the reduction sequence, we need only show that one of the following cases must hold:

1. E is faulty,
2. $E \longmapsto E'$ and E' is closed, or
3. E is a value.

Note that $E \longmapsto E'$ iff $E = \mathcal{E}[E_1]$, $E' = \mathcal{E}[E_1']$, and $E_1 \longrightarrow E_1'$. Also, by the definition of faultiness, if any subexpression is faulty, then the expression is faulty, so we need not consider any cases where any subexpression is faulty.

The proof proceeds by induction on the structure of E. We need only consider the cases not covered in [61].

Case $\{\!|E, \dots, E|\!\}$

This uses the same reasoning as in the analogous case in Lemma 4.5.4.

Case $\mathbf{merge}\, M\, (E, E)$

If $M = W$, $E_1 = \{\!|V_{11}, \dots, V_{1n}|\!\}$, $E_2 = \{\!|V_{21}, \dots, V_{2m}|\!\}$ (i.e., the subexpressions are all values of the appropriate sort), and $|W| = n+m$, then we can apply the appropriate notion of reduction. If M, E_1, E_2 are all values but do not satisfy the conditions in the previous sentence, then E is faulty. Otherwise, at least one of M, E_1, E_2 are not values, in which case we reduce the first non-value subexpression using one of the three contexts

$$\mathbf{merge}\, \mathcal{M}\, (E, E) \qquad \mathbf{merge}\, W\, (\mathcal{E}, E) \qquad \mathbf{merge}\, W\, (V, \mathcal{E}).$$

Case $\mathbf{extract}^{\ltimes}\, M\, E$, $\mathbf{index}^{\ltimes} M E$

This uses the same line of reasoning as the previous case.

Case $\lambda m.E$

This is a value.

Case $E\,M$

This uses the same reasoning as in the analogous case in Lemma 4.5.4, but makes use of Lemma B.3.3 for the case where M is not a value.

□

Theorem 6.3.1 (Type soundness). *If* $\overset{T}{\models} E : \tau$*, then* $E \longmapsto\!\!\!\!\rightarrow V$ *and* $\overset{T}{\models} V : \tau$*.*

Proof. The proof is the same as the one in [61], except that the appropriate Uniform Evaluation Lemma is used, namely, Lemma 6.3.3. □

Theorem 6.3.3 (Semantics Preservation). *If* $P|A \vdash E : \tau \rightsquigarrow E'$*, then for any ground substitution* θ *such that* $\Vdash \theta P$*, and for any pair of value environments* $(\mathsf{E}^S, \mathsf{E}^T_P) \in \mathcal{R}^{\theta A}$*,* $(E, E') \in \mathcal{R}^{\tau}$*.*

Proof. This proof is a slight adaptation of Ohori's, in [41]. It proceeds by induction on the structure of a translation derivation, and case analysis of the last step. The proofs for most of the cases are essentially the same as in [41]. We do the ones that are different:

Case

$$(\textbf{sel})\ \frac{P|A \vdash E : \{l : \tau, row\} \rightsquigarrow E' \quad P \Vdash M : row \# l}{P|A \vdash E.l : \tau \rightsquigarrow \textbf{index}^{\ltimes} M E'}$$

From the induction hypotheses, we have that $E \longmapsto\!\!\!\!\rightarrow V$ where $V = \{l_1 = V_1, \ldots, l_n = V_n\}$, and $E' \longmapsto\!\!\!\!\rightarrow V'$ where $V' = \{\!|V'_1, \ldots, V'_n|\!\}$. From the Subject Reduction for Evidence Lemma, we have that $M \longmapsto\!\!\!\!\rightarrow W$ where $W =$

$\langle\, s_1, \ldots, s_{n+1} \,\rangle$. By Lemma B.3.2, $W[i] = \rtimes$ only if $l = l_i$, i.e., the ith field is the one labeled l. Therefore, $\mathbf{index}^{\rtimes} W V' \longmapsto V_i'$, and $V \longmapsto V_i$, as desired.

Case

$$(\mathbf{extract})\quad \frac{P|A \vdash E : \{l : \tau, row\} \rightsquigarrow E' \qquad P \Vdash M : row \# l}{P|A \vdash E \backslash l : \{row\} \rightsquigarrow \mathbf{extract}^{\ltimes}\, M\, E'}$$

This is similar to the previous case, except for the last step, where we have: $\mathbf{extract}^{\ltimes}\, W\, V' \longmapsto \{\!|V_1', \ldots, V_{i-1}', V_{i+1}', \ldots, V_n'|\!\}$, and
$V \longmapsto \{l_1 = V_1, \ldots, l_{i-1} = V_{i-1}, l_{i+1} = V_{i+1}, \ldots, l_n = V_n\}$, as desired.

$\square$

Bibliography

[1] Sean Eron Anderson. Bit twiddling hacks. `http://graphics.stanford.edu/~seander/bithacks.html`.

[2] Andrew W. Appel. A critique of standard ML. *Journal of Functional Programming*, 3(4):391–429, 1993.

[3] H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.

[4] Hendrik P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North Holland, revised edition, 1984.

[5] L. E. J. Brouwer. *On the Foundations of Mathematics*. PhD thesis, Amsterdam, 1907.

[6] L. E. J. Brouwer. *Brouwer's Cambidge Lectures on Intuitionism*. Cambridge University Press, 1981.

[7] Peter Buneman and Atsushi Ohori. Polymorphism and type inference in

database programming. *ACM Transactions on Database Systems*, 21(1):30–76, 1996.

[8] Luca Cardelli and John C. Mitchell. Operations on records. In *Proceedings of the Fifth Conference on the Mathematical Foundations of Programming Semantics*. Springer Verlag, 1989.

[9] Luca Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[10] Alonzo Church. A formulation of a simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[11] Alonzo Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941.

[12] Mario Coppo and Mariangiola Dezani-Ciancaglini. A new type-assignment for λ-terms. *Archiv Math. Logik*, 19:139–156, 1978.

[13] H. B. Curry and R. Feys. *Combinatory Logic*. North-Holland, 1958.

[14] J R Driscoll, N Sarnak, D D Sleator, and R E Tarjan. Making data structures persistent. In *STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 109–121, New York, NY, USA, 1986. ACM Press.

[15] Gregory J. Duck, Simon Peyton-Jones, Peter J. Stuckey, and Martin Sulzmann. Sound and decidable type inference for functional dependencies. Submitted to ESOP '04, 2003.

[16] Dominic Duggan and John Ophel. Type checking multi-parameter type classes. *Journal of Functional Programming*, 12(2):133–158, March 2002.

[17] Gottlob Frege. *Grundgesetze der Arithmetik, begriffsschriftlich abgeleitet, vol 1.* Hildesheim, Olms, 1893. Translation by Montgomery Furth: *The Basic Laws of Arithmetic: Exposition of the system*, University of California Press, Berkeley (1967).

[18] Benedict R. Gaster and Mark P. Jones. A Polymorphic Type System for Extensible Records and Variants. Technical Report NOTTCS-TR-96-3, University of Nottingham, Department of Computer Science, Languages and Programming Group, Department of Computer Science, Nottingham NG7 2RD, UK, November 1996.

[19] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip Wadler. Type classes in haskell. In *ESOP*, Jan 94.

[20] Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 131–142, Orlando, Florida, 1991.

[21] Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *ACM Symposium on Principles of Programming Languages (POPL), Orlando, Florida*, pages 131–142, January 1991. Extended version available as Carnegie Mellon Technical Report CMU-CS-90-157.

[22] Robert A. Heinlein. *The Moon Is a Harsh Mistress.* G. P. Putnam's Sons, 1966.

[23] Peter Henderson and James H. Morris Jr. A lazy evaluator. In *POPL*, pages 95–103, 1976.

[24] M. P. Jones. Simplifying and improving qualified types. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 160–169, New York, NY, USA, June 1995. ACM Press.

[25] Mark. P. Jones. A theory of qualified types. In *ESOP'92, Proc. European Symposium on Programming, Rennes*, volume 582 of *Lecture Notes in Computer Science*, pages 287–306. Springer Verlag, 1992.

[26] Mark P. Jones. A theory of qualified types. In Bernd Krieg-Bruckner, editor, *ESOP '92, 4th European Symposium on Programming, Rennes, France, February 1992, Proceedings*, volume 582, pages 287–306. Springer-Verlag, New York, N.Y., 1992.

[27] Mark P. Jones. Type classes with functional dependencies. In *9th European Symposium on Programming, ESOP*, number 1782 in Lecture Notes in Computer Science, pages 230–244, Berlin, Germany, March 2000. Springer-Verlag.

[28] Mark P. Jones and Simon Peyton Jones. Lightweight extensible records for Haskell. In *ACM Haskell Workshop*, informal proceedings, October 1999.

[29] Mark Philip Jones. *Qualified Types: Theory and Practice.* PhD thesis, Oxford University, July 1992. Also available as Programming Research Group technical report 106.

[30] P. C. Kanellakis and J. C. Mitchell. Polymorphic unification by sharing analysis. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 54–74. ACM, ACM, January 1989.

[31] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. MLtypability is DEXPTIME-complete. In *15th Colloquium on Trees in Algebra and Programming, CAAP '90*, Lecture Notes in Computer Science431, pages 206–220, 1990. To appear in *J. Assoc. Comput. Machinery* under the title, "An Analysis of ML Typability".

[32] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rmy, and Jrme Vouillon. *Objective Caml system: Release 3.09.* Institut National de Recherche en Informatique et en Automatique, 2004. `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`.

[33] H. G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 382–401. ACM, January 1990.

[34] Gurmeet Singh Manku. Fast bit counting routines. `http://www-db.stanford.edu/~manku/bitcount/bitcount.html`.

[35] David McAllester. A logical algorithm for ML type inference. In *International Conference on Rewriting Techniques and Applications (RTA), Valencia, Spain*, volume 2706 of *Lecture Notes in Computer Science*, pages 436–451. Springer-Verlag, June 2003.

[36] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, December 1978.

[37] Robin Milner. A proposal for standard ML. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 184–197. ACM, ACM, August 1984. Invited paper.

[38] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT-Press, 1997.

[39] D. R. Morrison. PATRICIA–practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, October 1968.

[40] Atsushi Ohori. A compilation method for ML-style polymorphic record calculi. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 154–165, Albequerque, New Mexico, 1992.

[41] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, November 1995.

[42] Atsushi Ohori and Peter Buneman. Type inference in a database programming language. In *ACM Symposium on Lisp and Functional Programming (LFP), Snowbird, Utah*, pages 174–183, July 1988.

[43] Chris Okasaki and Andy Gill. Fast mergeable integer maps, 1998.

[44] Jens Palsberg and Tian Zhao. Type inference for record concatenation and subtyping. *Information and Computation*, 189:54–86, 2004.

[45] Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised

report. *Journal of Functional Programming*, 13(1):i–xii,1–255, Jan 2003. `http://www.haskell.org/definition/`.

[46] Didier Rémy. Typechecking records and variants in a natural extension of ML. In *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*, pages 242–249, January 1989. Long version in C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, MIT Press, 1994.

[47] Didier Rémy. Type inference for records in a natural extension of ML. Technical Report RR-1431, INRIA, 1991.

[48] Didier Rémy. Projective ML. In *ACM Conference on LISP and Functional Programming*, pages 66–75, 1992.

[49] Didier Rémy. Typing record concatenations for free. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*. ACM, ACM, January 1992.

[50] Didier Rémy. Efficient representation of extensible records. In *Proc. ACM SIGPLAN Workshop on ML and its applications*, pages 12–16, 1994.

[51] J. A. Robinson. A machine-oriented logic based on resolution principle. *Journal of the ACM*, 12(1):23–49, January 1965.

[52] Lynn E. Roller. The legend of midas. *Classical Antiquity*, 2(1):299–313, Apr 1983.

[53] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, June 1983.

[54] Simon J. Thompson. *Miranda: The Craft of Functional Programming*. Addison Wesley, 1995.

[55] D.A. Turner. An overview of Miranda. In D.A. Turner, editor, *Research Topics in Functional Programming*. Addison Wesley, 1990.

[56] Mitchell Wand. Complete type inference for simple objects. In *IEEE Symposium on Logic in Computer Science (LICS), Ithaca, New York*, pages 37–44, June 1987.

[57] Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 92–97, Pacific Grove, CA, June 1989.

[58] Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *Fourth Annual Symposium on Logic in Computer Science*, pages 92–97, Asilomar Conference Center, Pacific Grove, CA, 1989. IEEE Computer Society Press.

[59] Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, pages 1–15, 93.

[60] Joe B. Wells. Typability and type checking in the second-order λ-calculus are equivalent and undecidable. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 176–185, 1994.

[61] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[62] Jan Zwanenburg. A type system for record concatenation and subtyping. In Kim Bruce and Giuseppe Longo, editors, *Informal proceedings of Third Workshop on Foundations of Object-Oriented Languages (FOOL 3)*, 1996.