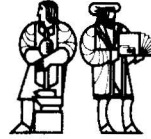


LABORATORY FOR
COMPUTER SCIENCE



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-468

**ML WITH EXTENDED PATTERN
MATCHING AND SUBTYPES**

Lalita A. Jategaonkar

August 1989

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

This blank page was inserted to preserve pagination.

ML with Extended Pattern Matching and Subtypes

by

Lalita A. Jategaonkar

submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment
of the requirements for the degrees of

Bachelor of Science

and

Master of Science in Electrical Engineering and Computer Science

at the

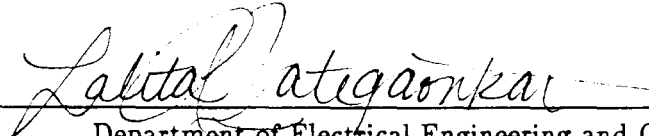
Massachusetts Institute of Technology

August, 1989

© Lalita A. Jategaonkar, 1989

The author hereby grants to MIT permission to reproduce and to
distribute copies of this thesis document in whole or in part.

Signature of Author



Department of Electrical Engineering and Computer Science

August 9, 1989

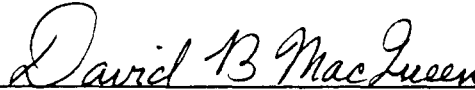
Certified by



Albert R. Meyer, Professor of Computer Science and Engineering

Thesis Supervisor

Certified by



Dr. David B. MacQueen, AT&T Bell Laboratories

Company Supervisor

Accepted by

Arthur C. Smith

Chairman, Department Committee on Graduate Students

ML with Extended Pattern Matching and Subtypes

by

Lalita A. Jategaonkar

Submitted to the Department of Electrical Engineering and Computer Science
on August 24, 1989 in partial fulfillment of the requirements for the degrees of
Bachelor of Science
and
Master of Science in Electrical Engineering and Computer Science

Abstract

We extend a fragment of the programming language ML by incorporating a more general form of record pattern matching and providing for user-declared subtypes. Together, these two enhancements may be used to support a restricted object-oriented programming style. In keeping with the framework of ML, we present typing rules for the language, and develop a type inference algorithm. We prove that the algorithm is sound with respect to the typing rules, and that it infers a most general typing for every typable expression.

Thesis Supervisor: Albert R. Meyer

Title: Professor of Computer Science and Engineering

Keywords: type inference, ML, subtyping, inheritance, object-oriented programming.

The author was supported in part by AT&T Bell Laboratories and by a graduate fellowship from AT&T Bell Laboratories Graduate Research Program for Women.

Acknowledgments

First, I would like to thank John Mitchell, without whom this thesis would not have been possible. I will always be grateful to him for introducing me to this general area of research, and for his continuing guidance and encouragement. I would also like to thank my advisor, Albert Meyer, for many extremely helpful discussions about this research, for pointing out some of the “right” questions to ask, and for helping me to answer those questions. Thanks are also due to Dave MacQueen, my official Bell Labs advisor, for his guidance and support. Bard Bloom, Trevor Jim, and Jon Riecke provided many useful comments, and countless helpful discussions about this research. At various times, Val Breazu-Tannen, Kim Bruce, Luca Cardelli, Didier Remy, and Mitch Wand gave some useful clarifications about their own work. Bell Labs provided both financial support and a conducive environment during the time I spent there working on this thesis. My friends helped out in many little ways. Last, but far far from least, I am indebted to my parents for their constant support and encouragement.

Contents

1	Introduction	1
1.1	Type Inference and Object-Oriented Programming	1
1.2	Background and Results	2
1.3	Related Research and Future Directions	4
1.4	Outline of Thesis	7
2	ML⁺: Types, Syntax, and Notation	8
2.1	Types	8
2.2	The Language Syntax	9
2.3	Meta-notation	10
3	The Typing System	12
3.1	Overview	12
3.2	Proof System for Subtyping Assertions	12
3.3	Syntax of Typing Statements	13
3.4	Proof System for Typing Statements	14
3.5	Substitutions, Instances, and the <i>R</i> -typing System	17
3.6	Properties of the <i>R</i> -Typing System	25
3.7	Most General Typings	33
3.8	Relating the <i>R</i> -Typing System to the Original System	34
4	The Type Inference Algorithm	35
4.1	Overview	35
4.2	Unification	35
4.3	Matching	37
4.4	The Type Inference Algorithm	43
4.5	Soundness	45
4.6	Completeness	47
4.7	Relating GE to the Original Typing System	55
A	Unification: Algorithm and Proof of Correctness	56

Chapter 1

Introduction

1.1 Type Inference and Object-Oriented Programming

During the past decade, a programming style known as “object-oriented” programming has become the basis for several popular programming languages, including Smalltalk [GR83] and C++ [Str86]. In this programming methodology, the basic building blocks are objects, which are grouped together in classes of similar objects. There is a hierarchy of classes, where more specialized objects of a class constitute subclasses; importantly, operations defined on a class can be *inherited*, or automatically used, by objects of any subclass. Furthermore, the representation of objects and the associated class operations can be hidden from the outside world. These features of *inheritance* and *information hiding* are often regarded as central to object-oriented programming.

Another class of programming languages consists of strongly typed languages, in which every expression has a type that can be checked at compile-time. These languages, such as Pascal and CLU, are desirable because, among other things, they guarantee that a certain class of errors known as type errors will not arise at run-time. Some drawbacks of traditional strongly typed languages are that they require the programmer to declare the types of all variables, and require the same function to be redefined over different types. However, in languages that support type inference, the types of all variables and expressions can be *inferred* by the compiler from the surrounding context, and thus can be omitted by the programmer. Moreover, the inferred types are “most general” types, from which all valid types can be easily derived. Languages with type inference thus provide greater flexibility and expressive power than traditional strongly typed languages, while maintaining the same guarantees on run-time behavior. ML [Mil85], a functional language that is based on the simply typed lambda calculus, is the paradigmatic language with type inference.

Combining strong-typing and object-oriented programming clearly has many advantages, and in the past few years, there has been much research on extending strongly typed languages to support an object-oriented programming style [Car84, Mit84, Wan87, Sta88, JM88, Rem89] by incorporating some form of *subtyping* to model subclass relations. However, extending languages with type inference in this manner can pose two serious problems: there may be no single “most general” typings, or worse yet, the type inference problem may become undecidable.

1.2 Background and Results

The viewpoint developed by [Car84] and adopted by many others [Wan87, Sta88, JM88, Rem89] is to think of an object as a record consisting of a finite set of labeled, typed fields. Inheritance is modeled by allowing a function defined on a record to be used automatically on a record with more fields. More specifically, the “subclass” relation is that a record type y is a subtype of record type x if y has at least all of the fields of x , and perhaps more.

Although viewing objects as records captures some of the “object-oriented” behavior that we want, it does not permit information hiding. For this reason, it seems that a more useful perspective is to think of objects as some form of abstract data types. For example, we can think of objects as “ML-style” abstract data types, which have an associated name, a hidden representation, and associated operations whose implementation is hidden. Importantly, the types of these operations do not reveal the underlying representation of the abstract type; they only refer to the abstract type by name. In order to support both inheritance and information hiding in this framework, we want objects of any subtype to be able to use these operations automatically, without revealing the representation of the subtype. Thus, in this scenario, it seems that we need some mechanism to declare explicitly the subtyping relations between the names of abstract types in order to get the sort of subtyping behavior that we desire. Some relevant work, which we believe can be extended to support subtyping between abstract type names, is [Mit84], which develops a type system and algorithm for inferring most general types for pure lambda terms in the context of subtype declarations. In [Mit84], the system of subtyping relations between simple types is referred to as “atomic subtyping,” and can be thought of as a more general form of “bounded quantification” [CW85].

There has been quite a bit of work done recently on developing systems with subtyping of *records*. The seminal work on this topic, [Car84], provides a set of typing rules for a type system with record subtyping and presents a type-checking algorithm for that language. However, the programmer is required to declare the types of all variables, and type inference is not provided.

[Wan87] has the a similar sort of “subtype” relation among record types as [Car84], but a somewhat different set of typing rules, and he does provide an algorithm for type inference. The main technical innovations are the introduction of “row variables”, which allow more flexibility in the typing of records, and an extension of unification [Rob65] that supports row variables. However, due to some technical problems, single most general typings do not exist for some typable terms in the system of [Wan87]; in fact, the typing algorithm has to infer a (finite) *set* of most general typings.

In [JM88], we extend the atomic subtyping system of [Mit84] to support base types and show that it can be merged in a natural manner with a system of rules for deriving record subtyping that is similar to that of [Wan87]. We define our language ML^+ , which is an extension of a kernel of the programming language ML, and a type inference algorithm for ML^+ , which uses an extension of unification similar to that of [Wan87]. We show that our algorithm infers only types derivable by our type system, and generates *single* most general typings. Importantly, by imposing a minor restriction on records, we correct the technical difficulties of [Wan87]; we believe that our paper was the first published work to do so. This thesis revises some of the material presented in [JM88] and develops it in more detail.

The ML^+ language is based on a functional subset of ML that includes built-in constants, records, and function abstraction using pattern matching constructs to decompose records into their constituents. For the sake of simplicity, we do not include the polymorphic

“let” construct of ML, since it can be regarded as syntactic sugar. Although there are some algorithmic issues of dealing efficiently with the “let” construct, we do not address them here. We do not include variant records (tagged unions), although we believe that the subtyping of variants involves issues closely related to the subtyping of records [Car84, Wan87, Sta88, Rem89]. We would expect that incorporating variants into our system would not cause any serious difficulties, but we do anticipate that there would be some technical overhead involved.

We extend this kernel of ML two ways. We develop a form of “extended pattern matching” that allows us to type records in a way that allows a function on records to be applied to every record containing some minimum set of required fields. More specifically, we introduce expression variables denoting elements of Wand’s “rows.” These may be bound by pattern matching to sequences of labeled values (parts of records). Using this extended pattern matching, we may define functions that operate “anonymously” on parts of records without knowing the names of the fields involved. In keeping with the spirit of ML, this form of extended pattern matching eliminates the need for conditional statements to decompose arguments passed to functions, and we have developed some rather elaborate technical machinery in our system in order to support this expressive power.

Our second extension to ML involves subtyping relations between atomic types. For built-in atomic types such as *int* and *bool*, subtyping relationships such as $int \subseteq real$ must be specified as part of the language design. As mentioned earlier, our treatment of subtyping extends the system for atomic subtyping developed in [Mit84]. It is our hope that this form of subtyping combined with record subtyping will provide a framework for subtyping among abstract types.

Some examples will illustrate the flavor of ML^+ . Following Standard ML, function expressions (lambda abstractions) are written $\mathbf{fn} P \Rightarrow M$, where P is a pattern, often resembling a record expression, and M is any expression. For example, a function f incrementing the a field of a record may be written as

$$\mathbf{fn} \{a = x; v\} \Rightarrow \{a = x + 1; v\} \quad (f)$$

Essentially, the pattern $\{a = x; v\}$ matches any record with an a field, binding x to the value of a , and binding v to any finite mapping from labels to values extending $a = x$. One type of this function in ML^+ is

$$\{a: int; \text{NULL}\} \rightarrow \{a: int; \text{NULL}\}$$

where *NULL* indicates that the record contains exactly the fields specified, since f clearly maps records with exactly an $a: int$ field to records with exactly an $a: int$. Another type of f is written

$$\{a: int; \mathcal{X}\} \rightarrow \{a: int; \mathcal{X}\}$$

where \mathcal{X} is a “row” variable denoting a sequence of labeled types. The row variable \mathcal{X} in this type expression is implicitly universally quantified, so this typing “says” that f has type $\{a: int; \mathcal{X}\} \rightarrow \{a: int; \mathcal{X}\}$ for any row \mathcal{X} . In particular, f has type

$$\{a: int, b: bool, c: string\} \rightarrow \{a: int, b: bool, c: string\}$$

since $b: bool, c: string$ is a possible value for \mathcal{X} . Thus, if we apply f to the record $\{a = 1, b = true, c = \text{“extra”}\}$, then the “extension” variable v is bound to $b = true, c = \text{“extra”}$, and the result of the function application is

$$f(\{a = 1, b = true, c = \text{“extra”}\}) = \{a = 2, b = true, c = \text{“extra”}\}$$

As mentioned earlier, the use of row variables to describe the inherent polymorphism of record operations is due to Wand [Wan87]. One technical difficulty is that \mathcal{X} in a type $\{a: \text{int}; \mathcal{X}\} \rightarrow \dots$ should not denote a row giving a type to a , since then the type of a might be multiply-defined. This leads to certain subtle considerations in our typing algorithm, and also the point of departure from Wand’s previous work. While Wand allowed type expressions with multiple occurrences of field names (using order of occurrence to determine precedence), this in fact leads to a set of most general typings. In a sense, the difficulty with Wand’s algorithm begins with his expression language. Wand’s **with** expression has two informal readings: the expression x **with** $a: = 3$ has the effect of either modifying the a field of x if there is one, or adding one if there is not. From a type checking point of view, it is not clear whether we should assume x has an a field or not, and so there are two typings to consider.

We avoid these technical problems associated with Wand’s language by restricting well-formed record types not to contain duplicate labels, and by using extended pattern matching instead of **with**. Together, these two restrictions avoid the ambiguity of Wand’s expressions by compelling the programmer to choose which interpretation he wants his expression to have. For example, the expression $\lambda x. x$ **with** $a: = 3$ in Wand’s system translates to two expressions in ML^+ , namely $\text{fn } \{a = y; u\} \Rightarrow \{a = 3; u\}$ and $\text{fn } \{u\} \Rightarrow \{a = 3; u\}$. The first expression constrains the argument record to have an a field, while the second expression constrains the argument record *not* to have an a field. This contrasts with the expression in Wand’s system, which allows a record either with or without an a field to be passed as an argument. Aside from translating ambiguous expressions in Wand’s language to sets of expressions, ML^+ retains all of the expressive power of Wand’s system, while generating single most general typings. We note that this expressive power is due to pattern matching; making a similar restriction on the records in Wand’s system would greatly restrict the expressive power of that language. For example, the expression $\lambda x. x$ **with** $a: = x.a + 1$ in Wand’s system would not be typable with this restriction, since the result would have two a fields. However, by using extended pattern matching, we can write this expression in ML^+ as $\text{fn } \{a = y; u\} \Rightarrow \{a = y + 1; u\}$.

Furthermore, there are some expressions that we can write in ML^+ that are not expressible in Wand’s language. One such sort of expression that we can write in ML^+ is

$$\text{fn } \{a = y; u\} \Rightarrow \{u\}$$

which allows some fields of a record to be “forgotten”. Another such expression in ML^+ is

$$\text{fn } \{a = y; \text{EMPTY}\} \Rightarrow y$$

where the **EMPTY** indicates that the record has *exactly* an a field. Although Wand’s language can conceivably be extended to have this expressive power, there are some problems. First, it seems difficult, and perhaps even impossible, to define and type a “forget” operation that forgets *all* occurrences of a given field, including those fields that have been overwritten. Second, an operation “exactly” that restricts a record to have exactly a certain set of fields has precisely the same sort of ambiguity with respect to overwritten fields as that of the **with** construct, and thus also leads to a set of most general typings.

1.3 Related Research and Future Directions

Between the time that [JM88] was published and the time that this thesis was completed, Remy [Rem89] has resolved some of the technical difficulties of [Wan87] without imposing

any restrictions on duplicate fields in records, and his system also incorporates variants and recursive types. The main technical innovation is a new perspective on records in the context of a *finite* set of labels. All records are assumed to contain fields corresponding to *all* labels in the (finite) set; however, only some of the fields need to contain values. The fields that do not contain values are considered to be “uninitialized”, but must be written out explicitly. However, in a more recent paper [Wand89], Wand extends Remy’s system to an infinite set of labels using row variables, so that only the fields that play a role in a certain expression need be written explicitly.

Remy’s type system is richer than both the system of [Wan87] and our system without atomic subtyping. In fact, in Remy’s system, *all* of the typable expressions in the language of [Wan87] have *single* most general typings. Moreover, if we omit atomic subtyping from our system, then all expressions that are typable in our system can be translated into Remy’s language and typed in his system. In particular, expressions corresponding to “forgetting” fields of a record can be translated directly, while expressions denoting that a record must have exactly a certain set of (initialized) fields can be translated into a straightforward extension of his language. Furthermore, Remy’s system can type expressions that are not typable in either the system of [Wan87] or our complete system. For example, his approach would assign to the expression ¹

$$\text{if } x \text{ then } \{a = 3, b = \text{true}\} \text{ else } \{a = 5\}$$

the record type such that a may be considered to be an integer field, and all other fields are “uninitialized”, since the records $\{a = 3, b = \text{true}\}$ and $\{a = 5\}$ are unifiable in his system. However, under both our approach and the approach of [Wan87], these records are not unifiable, and thus, this expression would not be typable.

The system of [Wan87] is a special case of Remy’s system, but the relationship is more complicated for ML^+ . Although any ML^+ expression that is typable without atomic subtyping can be translated into a typable expression in Remy’s language, our system without atomic subtyping is *not* a special case of Remy’s system. The typing that Remy’s system generates on a translated ML^+ expressions does not translate *back* into the typing that our system yields; specifically, Remy’s typings cannot capture the constraint that records cannot have duplicate labels anywhere in the typing derivation of an expression. (As will become apparent in following chapters of this thesis, our typings capture this constraint by explicitly stating the set of types that appear in the derivation of an expression, but do not appear in the final typing statement.)

In fact, our system without atomic subtyping distinguishes a larger class of type errors than that of Remy’s system. For example, our type inference algorithm generates a type error if the function $\text{fn } \{u\} \Rightarrow \{a = 3; u\}$ is applied to a record with an a field, thus avoiding the situation in which a programmer unwittingly overwrites a field. Since at this moment it is unclear what sorts of languages and type systems are desirable for object-oriented programming, it may turn out later on that the larger class of type errors in ML^+ is advantageous to programmers. The above example shows that there is some question as to whether the ambiguous expressions in Wand’s system should be typable.

On the other hand, language designers may consider the system of [Rem89] more desirable than our system, since Remy’s system types more expressions than ours, his system

¹Although an “if” statement is not directly expressible in some of the languages discussed here, it is a simple matter to type such a statement. Namely, the if-clause must be a boolean, and both arms of the statement must have a “least” type in common.

incorporates variants and recursive types, and his type inference algorithm uses the usual unification algorithm rather than the extension to unification developed in [Wan87]. Thus, anyone wishing to implement a language with type inference that supports automatic subtyping between records should probably not consider the type system and type inference algorithm presented in this thesis, but instead, should adopt the system of [Rem89] extended to an infinite set of labels as in [Wand89]. It is important to point out, though, that Remy has not incorporated atomic subtyping into his system. Although it is quite possible that atomic subtyping can be merged naturally into Remy’s system, a serious student of the subject may wish to read our work in order to get some insights into how it can be done.

In addition to the three papers discussed in detail above, a host of other papers on this topic have been published in the past few years [Sta88, OB88, Wand89, FM88, CCHMO89], presenting different languages, type systems, and type inference algorithms that support various “object-oriented” features. We give a brief comparison of these systems here.

If we only consider the core language consisting of variables, records, lambda abstraction, and function application in all the systems, then the system of [Car84] is incomparable with the system of [Wan87] and our system. For example, the [Car84] approach would assign the expression

$$\text{if } x \text{ then } \{a: = 3, b: = \text{true}\} \text{ else } \{a: = 5, b: = \lambda x. x\}$$

the type $\{a: \text{int}\}$, since this is the least upper bound of the record types $\{a: = 3, b: = \text{true}\}$ and $\{a: = 5, b: = \lambda x. x\}$. However, under both our approach and the approach of [Wan87], these records are not unifiable, and thus, this expression is not typable. On the other hand, in Cardelli’s system, one cannot translate into a typable expression an expression like

$$((\lambda x. x \text{ with } a: = x.a + 1)\{a: = 3, l: = y\}).l$$

where l is an *arbitrary* label. In Cardelli’s system, the type of a lambda-bound variable must be declared, and since there is no mechanism to express the “rest of a record”, the type of a lambda-bound record must have a certain *fixed* set of fields. Thus, the argument record must be coerced to have that same fixed set of fields, and, in general, applying a function to a record with more fields results in the loss of the “extra” fields.

Although the basic system of [Rem89] discussed earlier in this section is incomparable to that of [Car84] for the same reasons discussed above, it seems that an extension to Remy’s basic system can type all the expressions typable in the system of [Car84], as well as all the expressions typable in the system of [Wan87]. However, in this system, one cannot express the notion of restricting a record to have exactly a certain set of fields, and thus, this extended system cannot type all the expressions typable in ML^+ (without atomic subtyping). At any rate, this extension to Remy’s system is merely sketched, and not worked out in detail, in [Rem89].

A type inference algorithm for a language based on [Car84] is presented in detail in [Sta88], which generates a set of subtype relations between records that are satisfied whenever a typing for an expression is derivable. He does not, however, present an algorithm that checks the satisfiability of sets of such subtype relations, and his principal types may be empty. As a consequence, given an untypable expression, his algorithm does not necessarily indicate that no typing exists for that expression. Furthermore, like that of [Car84], Stansifer’s language does not contain any general mechanism for record extension or modification, and so his system is also incomparable to that of [Wan87] and [JM88].

[OB88] presents a somewhat different core language that introduces sets, joins, and projections in the context of subtyping of records, and provides type inference for this

language. Because of the restrictions upon the join operation, this system is incomparable to that of [Wan87], [JM88], and [Rem89], since one cannot translate into the language expressions like $\lambda x. x \text{ with } a: = x.a + 1$. Using the join operation, however, it is possible to translate expressions like $\lambda x. x \text{ with } a: = 3$ into this language.

[Wand89] extends the core language of [Wan87] to express record concatenation, and gives a treatment of classes and multiple inheritance by using syntactic sugar for this underlying language. Using the system of [Rem89] extended to an infinite set of labels, a type inference algorithm is given that generates a set of most general typings for this language. If we consider the core language of [Wand89] without record concatenation, this system is a special case of that of [Rem89].

Leaving type inference aside, [CCHMO89] presents an extended form of “bounded quantification” [CW85] that seems useful when recursive type definitions and subtyping are used. Leaving subtyping of records aside, [FM88] presents a type inference algorithm based on that of [Mit84], and develops procedures for simplifying the set of subtyping assertions that are inferred by the algorithm.

As is apparent from the preceding discussion, there exist a fair number of languages and type systems that embody some “object-oriented” features, and these systems differ from one another in some non-trivial technical ways. Although such a comparison is useful, it is important to ask some broader questions. First, which of the different features is it feasible to merge together with the aim of developing a more powerful “object-oriented” language? Second, what further extensions should we aim to develop?

It seems to us that it is feasible to merge most of the approaches discussed above into one system with type inference. However, we feel that abstract data types capture an important property of object-oriented programming and should be incorporated into such typed languages. Other important features that should be considered are multiple inheritance, “self”, and “method specialization.”

In this entire discussion, we have only considered type systems and type inference algorithms, and have not discussed whether these type systems are themselves “reasonable”. It would be interesting and worthwhile to look at the semantics of these systems, both from an operational and denotational point of view. Such an investigation does not appear in this thesis, and the interested reader is directed towards [Kam88, BL88, Red88, Coo89, BCGS89].

1.4 Outline of Thesis

Chapter 2 presents the syntax of ML^+ . The typing system of the language, which is specified by a set of typing axioms and inference rules, is presented in Chapter 3. The notion of substitutions, instances, and “most general” typings is also developed in that chapter. Finally, Chapter 4 presents the algorithm for inferring a most general typing for any expression in ML^+ . In that chapter, we prove that the algorithm is sound with respect to the type system, in the sense that whenever the algorithm gives term M type σ , the assertion that M has type σ is provable from the typing rules. We also show that the algorithm infers the most general type for any typable term. Specifically, if we can prove M has type σ using the typing rules, then the algorithm succeeds in finding a typing for M which is “more general” than σ in a precise sense developed in earlier chapters.

Chapter 2

ML⁺: Types, Syntax, and Notation

2.1 Types

We begin with an infinite set of type variables and some fixed set of base types. We fix this set to be **{int, bool, real, string}**; however, our type system and algorithm can be easily extended to handle a larger set of base types.

There are two forms of structured types: function types and record types. Function types are written using \rightarrow as usual, so that $\sigma \rightarrow \tau$ is the type of functions from σ to τ . As mentioned in Chapter 1, we believe that variants (tagged unions) involve issues closely related to records [Car84, Wan87, Sta88, Rem89], but we do not consider them here.

Record types are written in a slightly unusual way. Intuitively, record types are finite functions from labels to types. In our system, part of this finite function can be named but unspecified, so that it can be passed around and referred to without being fully specified until a later time. To support this naming of parts of record types, we follow [Wan87] and introduce an infinite set of *row variables*, which denote finite functions from labels to types, and the *row constant* NULL which denotes the empty function.

In order to make our type system and algorithm more understandable in a technical sense, while continuing to write our examples in ML⁺, we use two different notations for record types. In the formal notation, summarized in Table 2.3, record types are pairs $\langle h, \mathcal{Z} \rangle$ where h is a finite function from labels to types and \mathcal{Z} is either a row variable or NULL. We call a record type in which \mathcal{Z} is a row variable an *extended* record type, and a record type in which \mathcal{Z} is NULL *fixed*.

In ML⁺, record types are written $\{l_1: \tau_1, \dots, l_n: \tau_n; \mathcal{Z}\}$. For example, the record type $\{a: \sigma, b: \tau; \mathcal{X}\}$ is intuitively the finite function that maps a to σ and b to τ , combined with the as yet unspecified function denoted by the row variable \mathcal{X} . On the other hand, the record type $\{a: \sigma, b: \tau; \text{NULL}\}$ is intuitively the finite function that maps exactly a to σ and b to τ .

Since we use finite functions to write records, *all of the labels in a record must be distinct*. This distinguishes our type expressions from the expressions used in [Wan87], and leads to an important and slightly subtle complication in our system. To avoid the algorithmic inefficiencies of Wand's algorithm [Wan87], we must assume that the domains of h and \mathcal{Z} (as finite functions) are disjoint. This complicates substitution of record expressions for row variables, since it only makes sense to replace a row variable by a record type which has

c	base types
t	type variables
$\sigma \rightarrow \tau$	function types
$\{\text{NULL}\}$	empty fixed records
$\{\mathcal{X}\}$	empty extended records
$\{l_1:\tau_1, \dots, l_n:\tau_n; \text{NULL}\}$	fixed records
$\{l_1:\tau_1, \dots, l_n:\tau_n; \mathcal{X}\}$	extended records
<i>where $l_i \neq l_j$ for all $i \neq j$</i>	

Table 2.1: Summary of Types

an appropriately limited domain. To avoid incorrect substitutions, our typing algorithm therefore maintains restrictions on row variables.

The types are summarized in Table 2.1.

2.2 The Language Syntax

ML^+ is derived from a subset of ML with pattern matching, function abstraction, records, ground constants, and built-in (possibly higher-order) constants. For the ground constants, we assume the set \mathbf{Z} of integers, the set \mathfrak{R} of reals, the set $\{\mathbf{true}, \mathbf{false}\}$ of booleans, and an infinite set of strings. Any closed term may be incorporated into ML^+ as a built-in constant. We note here that although there is no explicit construct for declaring recursive functions, it is possible to define, as a built-in constant, an operator *fix* that returns the fixed-point of a function. Thus, using *fix*, we can define recursive functions.

The two new features of ML^+ are a more powerful (extended) form of pattern matching, which permits subtyping of records, and structural subtyping, which allows subtyping between base types.

Extended pattern matching is achieved using an infinite set of *extension variables*, which play a role in the language similar to that of row variables in the type system. Formally, extension variables denote finite functions from labels to expressions, while the *extension constant* EMPTY denotes the empty function. Analogous to record types, record expressions are pairs whose first component is a finite function from labels to expressions and whose second component is an extension variable or EMPTY . As with record types, we call a record expression whose second element is an extension variable an *extended* record expression, and one whose second element is EMPTY *fixed*. We do not allow record expressions with duplicate fields, thus imposing the same sort of restrictions on record expressions as on record types.

Patterns are simply a proper subset of expressions, consisting recursively of ground constants, variables, fixed record expressions, and extended record expressions. However, all variables and extension variables within a pattern must be distinct.

In ML^+ , as in ML, pattern matching provides a limited form of equality testing over the structure of types, while pattern matching over constants provides an equality test for ground constants. For the sake of simplicity in this thesis, we do not provide a polymorphic equality function since such an extension, while fairly straightforward, would add a layer of technical complication to our system.

<i>Expressions</i>	
b	ground constants
q	built-in constants
x	variables
$\mathbf{fn} P \Rightarrow M$	abstraction
$M N$	application
$\{\text{EMPTY}\}$	empty fixed records
$\{u\}$	empty extended records
$\{l_1 = M_1, \dots, l_k = M_k; \text{EMPTY}\}$	fixed records
$\{l_1 = M_1, \dots, l_k = M_k; u\}$	extended records
	<i>where $l_i \neq l_j$ for all $i \neq j$</i>

Table 2.2: Summary of Expressions and Patterns

The language is summarized in Table 2.2.

2.3 Meta-notation

Since there are quite a few syntactic categories to ML^+ , the meta-notation we use is summarized in Table 2.3. This notation will be used extensively in our proof rules and algorithm, while the examples will be written using ML^+ syntax.

M, N	expressions
b	ground constants
q	built-in constants
x, y, z	variables
u, v	extension variables
w	variables and extension variables
E	extension variables and EMPTY
f	finite functions from labels to expressions
$\langle f, E \rangle$	record expressions
P	patterns, namely, abstraction-free, application-free expressions with no repeated variables or extension variables
σ, τ	type expressions
c	base types
s, t	type variables
\mathcal{X}, \mathcal{Y}	row variables
Z	row variables and NULL
h	finite functions from labels to types
$\langle h, Z \rangle$	record types

Table 2.3: Summary of Notational Conventions

Chapter 3

The Typing System

3.1 Overview

As mentioned in the introduction, the main purpose of this thesis is twofold. First, we would like to define a typing system for ML^+ that supports the form of subtyping discussed in Chapter 1 and Chapter 2. Since the legal terms in ML^+ are exactly the pre-terms in ML^+ syntax that have a typing derivable through this typing system, the typing system serves to define the language. Secondly, we would like to develop a type inference algorithm that is “equivalent” to the typing system in the sense that it generates a typing for exactly those terms that have a typing derivable through the typing system. Moreover, for any typable expression M , we want the typing algorithm to generate a “most general” typing, whose set of “instances” is exactly the set of provable typings for M .

In this chapter, we present the typing system for ML^+ and give a precise definition of instances and most general typings in the context of this typing system. Our typing system consists of two separate but related sets of typing rules, one for deriving subtype assertions, which are subtype relations between types, and the other for typing expressions.

The chapter is organized as follows. First, in Section 3.2, we present the derivation rules for subtype assertions. Section 3.3 defines the form of typing statements, which are formulas that capture the information we need in order to derive typings for expressions, and Section 3.4 presents a set of seemingly natural derivation rules for typing statements. In Section 3.5, we develop the “instance” relation in some detail and show that, in the typing system of Section 3.4, provable typings are not closed under the instance relation. As we show, this implies that “most general” typings do not exist for some expressions for which a type is derivable in the typing system. We examine the difficulty, and then modify the typing system a bit in order to get the R -typing system, which has the technical properties that we need. Section 3.6 is devoted to proving Theorem 3.6.7, the main theorem of the chapter, which shows that, in the R -typing system, all instances of a provable typing are provable. Section 3.7 develops the notion of most general typings in the context of the R -typing system, and Section 3.8 examines how the the R -typing system relates to the original typing system.

3.2 Proof System for Subtyping Assertions

As mentioned in the introduction, subtype assertions are of the form $\sigma \subseteq \tau$. We require that the subtype relation, \subseteq , act like a pre-order on types. In particular, we follow [Mit84]

and introduce into our system for deriving subtype assertions the following three axioms and rules.

$$\begin{array}{l}
 \text{ref} \qquad \qquad \qquad \sigma \subseteq \sigma \\
 \\
 \text{trans} \qquad \qquad \qquad \frac{\sigma \subseteq \tau, \tau \subseteq \gamma}{\sigma \subseteq \gamma} \\
 \\
 \text{arrow} \qquad \qquad \qquad \frac{\sigma \subseteq \sigma', \tau' \subseteq \tau}{\sigma' \rightarrow \tau' \subseteq \sigma \rightarrow \tau}
 \end{array}$$

We note that the function type constructor \rightarrow is antimonotonic in the first argument and monotonic in the second.

We have a similar rule for the “subtype” relation on records.

$$\text{record} \quad \frac{\langle \phi, \mathcal{Z} \rangle \subseteq \langle \phi, \mathcal{Z}' \rangle, \forall l \in \text{dom}(h). h(l) \subseteq h'(l)}{\langle h, \mathcal{Z} \rangle \subseteq \langle h', \mathcal{Z}' \rangle} \text{dom}(h) = \text{dom}(h') \neq \phi$$

We say that a subtype assertion $\sigma \subseteq \tau$ is provable from a set C of subtype assertions, written $C \vdash \sigma \subseteq \tau$, if $\sigma \subseteq \tau$ can be derived from assertions in C using only the typing rules above. We will write $C \vdash C'$ if $C \vdash \sigma \subseteq \tau$ for every $\sigma \subseteq \tau \in C'$. It is easy to show that \vdash is a transitive relation on sets.

In our typing system, we restrict C to consist only of *atomic* subtype assertions, which are subtype assertions of a certain simple form. We say that a subtype assertion $\sigma \subseteq \tau$ is *atomic* if

- σ and τ are both type variables
- σ and τ are both ground types
- Each of σ and τ is a ground type or a type variable
- Each of σ and τ is of the form $\langle \phi, \mathcal{Z} \rangle$

We say that C is an *atomic set* or that C is *atomic* if it consists only of atomic subtype assertions.

3.3 Syntax of Typing Statements

In the typing system for expressions, the typing statements have the form

$$C, A \supset M : \sigma$$

where

- C is an *atomic* set of subtype assertions.
- A is a finite set of associations $x : \sigma$ between variables and types, and associations $u : \langle h, \mathcal{Z} \rangle$ between extension variables and record types.
- M is an expression and σ is a type expression.

The typing $C, A \supset M: \sigma$ may be read as, “Given the subtype assertions in set C and the assignment of types to variables and extension variables described by A , the expression M has the type σ .”

To allow a slightly modified form of “ML-style” polymorphism, we consider all type and row variables that occur in C and σ but do not occur in A to be implicitly universally quantified. For example, the typing

$$\{s \subseteq t\}, \phi \supset \mathbf{fn} \ x \Rightarrow x: s \rightarrow t$$

should be read as, “For all types s and t such that $s \subseteq t$, the identity function has the type $s \rightarrow t$.” We note that this form of polymorphism does *not* correspond to the sort of polymorphism that the ML “let” construct provides.

3.4 Proof System for Typing Statements

In this section, we present a seemingly natural set of typing rules for expressions. We begin with the typing axioms for ground constants, which are as follows.

int $C, A \supset b: \mathbf{int}$ whenever b is an integer

The axioms for reals, booleans, and strings are analogous, with **int** replaced by **real**, **bool**, and **string**, respectively.

The typing rules for variables and application are standard.

var $C, A \supset x: \sigma$ whenever $x: \sigma \in A$

app
$$\frac{C, A \supset M: \sigma \rightarrow \tau, C, A \supset N: \sigma}{C, A \supset MN: \tau}$$

Due to pattern matching, the typing rule for lambda abstraction is somewhat more complicated than usual, since lambda abstraction may bind arbitrary patterns, in addition to variables. The typing rule for function expressions must therefore take the typing of patterns into account. There is a subtle but important *reversal* of subtype assertions in the abstraction rule which seems best illustrated by a simplified example. The pattern $\{a = x, b = y\}$ has typing

$$\{s \subseteq t\}, \{x: s, y: s\} \supset \{a = x, b = y\}: \{a: t, b: t\},$$

which means that if we give x and y values of type s , and $s \subseteq t$, then the pattern has type $\{a: t, b: t\}$. However, when we lambda abstract over the pattern, we actually bind a value to $\{a = x, b = y\}$ and access its components using variables x and y . So, in effect, we use the typing statement about the pattern “backwards.” The simplest technical adjustment seems to be to reverse the set of subtype assertions before combining them with the typing statement for the function body. If the function body is simply x , for example, then this gives us the typing

$$\{t \subseteq s\}, \emptyset \supset \mathbf{fn} \ \{a = x, b = y\} \Rightarrow x: \{a: t, b: t\} \rightarrow s$$

while writing $s \subseteq t$ instead would give us an incorrect typing. (An alternative is to reformulate the typing rules for patterns in a way that more accurately reflects their use. However, the current formulation has algorithmic advantages due to the similarities with expressions.)

In order to state the typing rule for lambda abstraction, we need to develop some notation and some definitions. Since the notion of *free variables* is important in the typing rule for abstraction, we define $vars(M)$ inductively as:

- $vars(x) = x$
- $vars(\langle f, EMPTY \rangle) = \bigcup_{l \in dom(f)} vars(f(l))$
- $vars(\langle f, u \rangle) = \bigcup_{l \in dom(f)} vars(f(l)) \cup \{u\}$
- $vars(MN) = vars(M) \cup vars(N)$
- $vars(\mathbf{fn} P \Rightarrow M) = vars(M) - vars(P)$, where “ $-$ ” is set difference

We define $vars(A) = \{w \mid w: \sigma \in A\}$, where w ranges over variables and extension variables.

We use the notation C^{op} for the *opposite* set of subtype relations,

$$C^{op} = \{\sigma \subseteq \tau \mid \tau \subseteq \sigma \in C\}$$

and $A[A']$ for the result of modifying A so that every variable and extension variable mentioned in A' has the type specified by A' rather than A . More precisely, $A[A'] = A_1 \cup A'$, where $A = A_1 \cup A_2$ and $A_2 = \{w: \sigma \mid w: \sigma \in A \text{ and } w \in vars(A')\}$.

The typing rule for abstraction is then as follows.

$$abs \quad \frac{C^{op}, A' \supset P: \sigma, \quad C, A[A'] \supset M: \tau}{C, A \supset \mathbf{fn} P \Rightarrow M : \sigma \rightarrow \tau} vars(A') = vars(P)$$

The condition $vars(A') = vars(P)$ ensures that every variable occurring in A' must occur in P . This is important, since we want $A[A']$ to modify A only on variables that become bound by \mathbf{fn} .

Proceeding to the typing rules for records, we first introduce the two axioms for records whose first component is the empty function.

$$rec1 \quad C, A \supset \langle \phi, EMPTY \rangle: \langle \phi, NULL \rangle$$

$$rec2 \quad C, A \supset \langle \phi, u \rangle: \langle h, \mathcal{Z} \rangle \text{ whenever } u: \langle h, \mathcal{Z} \rangle \in A$$

The third typing rule is for records whose first component is a non-empty function. Since extension variables may be assigned arbitrary record types, we have a minor complication since we do not want duplicate field names. To express the rule succinctly, we define the partial operation $+$ on finite functions from labels to types as

$$h_1 + h_2 = h \text{ s.t. } h(l) = \begin{cases} h_1(l) & \text{if } l \in dom(h_1) \\ h_2(l) & \text{if } l \in dom(h_2) \\ \text{undefined} & \text{otherwise} \end{cases} \quad \text{if } dom(h_1) \cap dom(h_2) = \emptyset$$

If $dom(h_1)$ and $dom(h_2)$ are not disjoint, then $h_1 + h_2$ is undefined.

The third rule for records is then as follows.

$$rec3 \quad \frac{C, A \supset \langle \phi, E \rangle: \langle h_2, \mathcal{Z} \rangle, \forall l \in dom(f). C, A \supset f(l): h_1(l)}{C, A \supset \langle f, E \rangle: \langle h_1 + h_2, \mathcal{Z} \rangle} \text{all typings well-formed}$$

where $\text{dom}(f) = \text{dom}(h_1) \neq \phi$

It is worth saying a few words about the form of $\text{rec}\beta$. Generally, axioms and inference rules are written as schemes, which are “abbreviations” for all of their substitution instances. However, not all substitution instances of rule $\text{rec}\beta$ are well formed, since $+$ may be undefined. Therefore, we interpret this rule “scheme” as indicating that any meaningful (well-formed) substitution instance may be used in deriving types for terms. As a consequence of the partiality of $+$, typing derivations that use $\text{rec}\beta$ may not be preserved by arbitrary substitutions, but only those which, throughout the proof, yield well-formed types (*i.e.*, types that do not anywhere contain record types that have duplicate labels).

Our typing system also allows any set of closed expressions to be built in as term constants, as long as each is given a proper type. For example, the type for the language construct if can be built into the typing system as a term constant. A reasonable typing for if is

$$\phi, \phi \supset \text{if}: \mathbf{bool} \rightarrow t \rightarrow t \rightarrow t$$

It may seem at first glance that this typing does not capture any of the power that subtyping provides us; however, as we will see in Section 3.5, we can derive from the above typing that

$$\vdash C, A \supset \text{if}: s \rightarrow t_1 \rightarrow t_2 \rightarrow t',$$

$$\text{where } C = \{s \subseteq \mathbf{bool}, t_1 \subseteq t, t_2 \subseteq t, t \subseteq t'\},$$

and A is any type environment. Furthermore, as we shall see in Section 3.7, this second typing is the “most general” typing of if .

It is useful to point out that an operator fix that returns the fixed-point of a function can also be defined as a built-in constant. A reasonable typing for fix is

$$\phi, \phi \supset \text{fix}: (t \rightarrow t) \rightarrow t$$

Again, we can derive the “most general” typing of fix from this above typing.

The axiom for term constants is

$$\text{const} \quad C, A \supset q: S\tau_q \text{ whenever } S \text{ is defined on } \tau_q$$

where τ_q is the built-in type for the constant q , and the application of the substitution S on τ_q results in a well-formed type. The idea here is that since all type variables in τ_q are considered to be implicitly universally quantified, we allow different substitution instances of the type of a built-in constant to be used within the typing of any expression.

Although this typing axiom gives us the desired typing for if , we may want more flexibility in building in the types of other built-in constants. More specifically, we may wish to incorporate into our system a *typing* for a (closed) term q ; that is, we wish to build in a set C_q as well as a type τ_q for q . Unfortunately, our system, as exists, does not seem to be powerful enough to handle these sorts of typings in general without sacrificing the property of most general typings. However, it seems that we can build in a large class of such typings into our system as axioms, while maintaining most general typings; we omit further discussion here.

Finally, in order to make use of subtyping, we have the rule

$$\text{coerce} \quad \frac{C, A \supset M: \sigma, C \vdash \sigma \subseteq \tau}{C, A \supset M: \tau}$$

By this rule, an expression may be considered to be of the type of any “supertype” of its actual type.

3.5 Substitutions, Instances, and the R -typing System

Although the typing system in Section 3.4 seems to capture the sort of subtyping that we want to achieve, it turns out that, for some typable expressions, most general typings do not exist. As mentioned in the overview to this chapter, the problem with the typing system arises because, for some expressions, provable typings are not closed under the “instance” relation. In this section, we develop the notion of instance in some detail and define most general typings using this instance relation. We discuss the difficulty in the typing system of Section 3.4, and then modify the typing system in order to correct this difficulty. We call the modified typing system the R -typing system.

We would like to begin by straightaway discussing the notion of “instance”, but as the basic part of the instance relation consists of a *substitution*, which is a finite function from type variables to types and from row variables to record types, we need to first discuss substitutions in some detail. Because a substitution may map a row variable to a record type $\langle h, \mathcal{Z} \rangle$ where h is a non-empty function, applying a substitution naively may result in a record type containing duplicate labels. To prevent this, we will treat the process of applying a substitution to a type as a partial operation.

Using the operation $+$ developed in the previous section, we define the partial operation $S\sigma$, the *action of a substitution S on a type σ* , where we write $S[t] = \sigma$ if S maps t to σ , and $S[\mathcal{X}] = \sigma$ if S maps \mathcal{X} to σ . $S\sigma$ is defined inductively as:

- $S c = c$
- $S t = \begin{cases} S[t] & \text{if } t \in \text{dom}(S) \\ t & \text{otherwise} \end{cases}$
- $S \langle \phi, \mathcal{X} \rangle = \begin{cases} S[\mathcal{X}] & \text{if } \mathcal{X} \in \text{dom}(S) \\ \langle \phi, \mathcal{X} \rangle & \text{otherwise} \end{cases}$
- $S \langle \phi, \text{NULL} \rangle = \langle \phi, \text{NULL} \rangle$
- $S \langle h, \mathcal{Z} \rangle = \begin{cases} \langle h; S + \text{left}(S \langle \phi, \mathcal{Z} \rangle), \text{right}(S \langle \phi, \mathcal{Z} \rangle) \rangle & \text{if } h; S \text{ is defined and} \\ & \text{the } + \text{ operation is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$
- $S(\sigma \rightarrow \tau) = S\sigma \rightarrow S\tau$

It is important to note that, due to the partiality of $+$, the action of S on a record type is undefined if an ill-formed function from labels to types is created *anywhere* within the resulting record type. Since h may (recursively) map a label to a record type, the composition of a finite function h from labels to types with a substitution S , written $h; S$, is also a partial operation. We define $h; S$ as:

$$h; S = \begin{cases} h' \text{ s.t. } h'(l) = \begin{cases} S(h(l)) & \text{if } l \in \text{dom}(h) \\ \text{undefined} & \text{otherwise} \end{cases} & \text{if } \forall l \in \text{dom}(h). S(h(l)) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Similarly, composition of a substitution S with another substitution T , written $S; T$, is a partial operation. We define $S; T$ as:

$$S;T = \begin{cases} U & \text{if } \forall t \in \text{dom}(S). T(S[t]) \text{ is defined,} \\ & \forall \mathcal{X} \in \text{dom}(S). T(S[\mathcal{X}]) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

where $\text{dom}(U) = \text{dom}(S) \cup \text{dom}(T)$, and

$$U[t] = \begin{cases} T(S[t]) & \text{if } t \in \text{dom}(S) \\ T[t] & \text{if } t \in \text{dom}(T) \text{ and } t \notin \text{dom}(S) \end{cases}$$

and

$$U[\mathcal{X}] = \begin{cases} T(S[\mathcal{X}]) & \text{if } \mathcal{X} \in \text{dom}(S) \\ T[\mathcal{X}] & \text{if } \mathcal{X} \in \text{dom}(T) \text{ and } \mathcal{X} \notin \text{dom}(S) \end{cases}$$

We note that if $S;T$ is defined, then $(S;T)\sigma = T(S\sigma)$, or both are undefined. We also note that $h;(S;T) = (h;S);T$, or both are undefined. Likewise, composition of substitutions is associative.

We also define the action of a substitution S on type environments and on sets of possibly non-atomic subtype assertions. A substitution S applied to A is the assignment $SA = \{w:S\sigma \mid w:\sigma \in A\}$, where w ranges over variables and extension variables, if S is defined on all such σ . We consider SA undefined otherwise. Similarly, the application of substitution S to a possibly non-atomic C is the set $SC = \{S\tau_1 \subseteq S\tau_2 \mid \tau_1 \subseteq \tau_2 \in C\}$, provided $S\tau_i$ is defined for all such τ_i . Again, we consider SC undefined otherwise.

In defining our instance relation, we could follow [Mit84] and say that a typing $C', A' \supset M:\sigma'$ is an instance of a typing $C, A \supset M:\sigma$ by a substitution S if S is defined on C, A , and σ , and

$$C' \vdash SC, \quad A' \supseteq SA, \quad \text{and} \quad \sigma' = S\sigma$$

However, since in the above definition, SC may be a non-atomic set of subtype assertions, we cannot use the same sort of reasoning about SC as we do for C and C' . Thus, in order to simplify the proofs of our main theorems, we use a seemingly more complex notion of instance that uses, in place of SC , an *atomic* set $S \bullet C$ that is computable and that is closely related to SC . We show that this definition is, in fact, “equivalent” to the one above in the sense that $C', A' \supset M:\sigma'$ is an instance of $C, A \supset M:\sigma$ by a substitution S under the above definition iff it is an instance under this definition.

The basic idea is as follows. We want to define $S \bullet C$ as a “least” atomic set that implies $S \bullet C \vdash SC$; that is, any atomic set C' that implies SC should also imply $S \bullet C$. It then follows that, for any atomic set C' , $C' \vdash SC$ iff $C' \vdash S \bullet C$, and thus, the two above definitions of instance are equivalent.

The reasoning we will use in defining the \bullet operation is as follows. We will first show that, in order for an atomic set C' to imply SC , SC must contain only subtype assertions that have a certain form. We will then show how to compute, from any subtype assertion $\sigma \subseteq \tau$ of this form, a “least” atomic set that implies $\sigma \subseteq \tau$. Thus, any atomic set C' that implies $\sigma \subseteq \tau$ must also imply this computed atomic set. Using these ideas, we will define $S \bullet C$ so that it has the properties outlined above.

We begin by showing that an atomic set can imply only *matching* subtype assertions $\sigma \subseteq \tau$, where σ and τ have the same syntactic form. More precisely, we define the relation *match* on types recursively as:

- t_1 matches t_2

- c_1 matches c_2
- t matches c , and c matches t
- $\langle \phi, \mathcal{Z} \rangle$ matches $\langle \phi, \mathcal{Z}' \rangle$
- σ matches σ' and τ matches τ' iff $\sigma \rightarrow \tau$ matches $\sigma' \rightarrow \tau'$.
- $\text{dom}(h) = \text{dom}(h')$ and $\forall l \in \text{dom}(h). h(l)$ matches $h'(l)$
iff $\langle h, \mathcal{Z} \rangle$ matches $\langle h', \mathcal{Z}' \rangle$

We say that a subtype assertion $\sigma \subseteq \tau$ is matching if σ and τ match. It is easy to see that match is an equivalence relation on types.

Lemma 3.5.1 *If $C \vdash \sigma \subseteq \tau$, where C is atomic, then σ matches τ .*

Proof. We prove the lemma by induction on the length of the derivation of $\sigma \subseteq \tau$ from C .

If the proof of $\sigma \subseteq \tau$ from C requires no steps, then $\sigma \subseteq \tau \in C$ and is thus an atomic subtype assertion. Therefore, σ matches τ .

Otherwise, the proof must have ended in an application of either *(trans)*, *(arrow)*, or *(record)*. If the final step was *(trans)*, then $C \vdash \sigma \subseteq \tau$ must have followed by the antecedents $\sigma \subseteq \gamma$ and $\gamma \subseteq \tau$ for some γ . Since the proofs of $\sigma \subseteq \gamma$ and $\gamma \subseteq \tau$ are shorter, we may assume that σ matches γ and γ matches τ . Thus, by the properties of match, σ matches τ .

If the final step was *(arrow)*, then σ must be of the form $\sigma = \sigma_1 \rightarrow \sigma_2$ and τ must be of the form $\tau = \tau_1 \rightarrow \tau_2$, and $C \vdash \sigma \subseteq \tau$ must have followed by the antecedents $\tau_1 \subseteq \sigma_1$ and $\sigma_2 \subseteq \tau_2$. Since the proofs of $\tau_1 \subseteq \sigma_1$ and $\sigma_2 \subseteq \tau_2$ are shorter, we may assume that τ_1 matches σ_1 and σ_2 matches τ_2 . Thus, by the properties of of match, σ matches τ .

If the final step was *(record)*, then σ must be of the form $\sigma = \langle h, \mathcal{Z} \rangle$ and τ must be of the form $\tau = \langle h', \mathcal{Z}' \rangle$, where $\text{dom}(h) = \text{dom}(h') \neq \phi$. Moreover, $C \vdash \sigma \subseteq \tau$ must have followed by the antecedents $\langle \phi, \mathcal{Z} \rangle \subseteq \langle \phi, \mathcal{Z}' \rangle$, and $\forall l \in \text{dom}(h). h(l) \subseteq h'(l)$. Since the proofs of the $h(l) \subseteq h'(l)$ are shorter, we may assume that $\forall l \in \text{dom}(h). h(l)$ matches $h'(l)$. Thus, by the properties of match, σ matches τ , proving the lemma. ■

We now wish to show that for any matching subtype assertion $\sigma \subseteq \tau$, we can compute an atomic set C that implies $\sigma \subseteq \tau$, and such that for any atomic set C' that implies $\sigma \subseteq \tau$, C' also implies C . However, in order to justify “decomposing” $\sigma \subseteq \tau$ into atomic subtype assertions, we will need to use the following lemmas about the form of derivable subtype assertions.

Lemma 3.5.2 *For any atomic set C , $C \vdash \sigma_1 \rightarrow \sigma_2 \subseteq \tau_1 \rightarrow \tau_2$ iff $C \vdash \tau_1 \subseteq \sigma_1$ and $C \vdash \sigma_2 \subseteq \tau_2$.*

Proof. The proof of this lemma is exactly that of [Mit84].

One direction is a direct consequence of rule *(arrow)*: if $C \vdash \tau_1 \subseteq \sigma_1$ and $C \vdash \sigma_2 \subseteq \tau_2$, then $C \vdash \sigma_1 \rightarrow \sigma_2 \subseteq \tau_1 \rightarrow \tau_2$. It remains to prove the converse.

We show that if $C \vdash \sigma \subseteq \tau$ for any σ and τ of the form $\sigma = \sigma_1 \rightarrow \sigma_2$ and $\tau = \tau_1 \rightarrow \tau_2$, then there is a proof of $\sigma \subseteq \tau$ from C that ends with an application of the rule *(arrow)*. We argue by induction on the length of the proof of $\sigma \subseteq \tau$ from C . If the proof is one step, then this step must be an application of rule *(arrow)* and so, trivially, there must be a proof ending in an application of *(arrow)*.

For the inductive step, assume that we have a proof whose final step is a use of rule *(trans)* from antecedents $\sigma \subseteq \gamma$ and $\gamma \subseteq \tau$. By Lemma 3.5.1, we know that γ has the form $\gamma = \gamma_1 \rightarrow \gamma_2$. Since the proofs of $\sigma \subseteq \gamma$ and $\gamma \subseteq \tau$ are shorter, we may assume that we have proofs of these inclusions ending in applications of rule *(arrow)*. Thus,

$$C \vdash \gamma_1 \subseteq \sigma_1, \sigma_2 \subseteq \gamma_2, \tau_1 \subseteq \gamma_1, \gamma_2 \subseteq \tau_2.$$

By rule *(trans)*, we have $C \vdash \tau_1 \subseteq \sigma_1$ and $C \vdash \sigma_2 \subseteq \tau_2$, which proves the lemma. ■

We have a similar lemma for records.

Lemma 3.5.3 *For any atomic set C , $C \vdash \langle h, \mathcal{Z} \rangle \subseteq \langle h', \mathcal{Z}' \rangle$, where h is non-empty, iff $\text{dom}(h) = \text{dom}(h') \neq \phi$ and $C \vdash \langle \phi, \mathcal{Z} \rangle \subseteq \langle \phi, \mathcal{Z}' \rangle$ and $\forall l \in \text{dom}(h). C \vdash h(l) \subseteq h'(l)$.*

Proof.

The proof of this lemma is analogous to the proof of Lemma 3.5.2.

One direction is a direct consequence of rule *(record)*: If $\text{dom}(h) = \text{dom}(h') \neq \phi$ and $C \vdash \langle \phi, \mathcal{Z} \rangle \subseteq \langle \phi, \mathcal{Z}' \rangle$ and $\forall l \in \text{dom}(h). C \vdash h(l) \subseteq h'(l)$, then $C \vdash \langle h, \mathcal{Z} \rangle \subseteq \langle h', \mathcal{Z}' \rangle$, where h is non-empty. It remains to prove the converse.

We show that if $C \vdash \sigma \subseteq \tau$ for any σ and τ of the form $\sigma = \langle h, \mathcal{Z} \rangle$ and $\tau = \langle h', \mathcal{Z}' \rangle$, where h is non-empty, then there is a proof of $\sigma \subseteq \tau$ from C that ends with an application of the rule *(record)*. We argue by induction on the length of the proof of $\sigma \subseteq \tau$ from C . If the proof is one step, then this step must be an application of rule *(record)* and so, trivially, there must be a proof ending in an application of *(record)*.

For the inductive step, assume that we have a proof whose final step is a use of rule *(trans)* from antecedents $\sigma \subseteq \gamma$ and $\gamma \subseteq \tau$. By Lemma 3.5.1, we know that σ , γ , and τ match, and that γ has the form $\gamma = \langle h'', \mathcal{Z}'' \rangle$, where $\text{dom}(h) = \text{dom}(h'') = \text{dom}(h')$. Since the proofs of $\sigma \subseteq \gamma$ and $\gamma \subseteq \tau$ are shorter than that of $\sigma \subseteq \tau$, we may assume that we have proofs of these inclusions ending in applications of rule *(record)*. Thus, $C \vdash \langle \phi, \mathcal{Z} \rangle \subseteq \langle \phi, \mathcal{Z}'' \rangle$, $C \vdash \langle \phi, \mathcal{Z}'' \rangle \subseteq \langle \phi, \mathcal{Z}' \rangle$, and

$$\text{for all } l \in \text{dom}(h), C \vdash h(l) \subseteq h''(l) \text{ and } C \vdash h''(l) \subseteq h'(l).$$

By rule *(trans)*, $C \vdash \langle \phi, \mathcal{Z} \rangle \subseteq \langle \phi, \mathcal{Z}' \rangle$ and $\forall l \in \text{dom}(h). C \vdash h(l) \subseteq h'(l)$, which proves the lemma. ■

Using the above lemmas, we can now prove the property we desire.

Lemma 3.5.4 *Let σ and τ be matching type expressions. There is an atomic set $C = \text{ATOMIC}(\sigma \subseteq \tau)$ with $C \vdash \sigma \subseteq \tau$ and such that if C' is any atomic set of subtype assertions with $C' \vdash \sigma \subseteq \tau$, then $C' \vdash C$.*

Proof.

We define $\text{ATOMIC}(\sigma \subseteq \tau)$ as:

- If $\sigma \subseteq \tau$ is an atomic subtype assertion, then $\text{ATOMIC}(\sigma \subseteq \tau) = \{\sigma \subseteq \tau\}$.
- If $\sigma \subseteq \tau$ is of the form $\sigma_1 \rightarrow \sigma_2 \subseteq \tau_1 \rightarrow \tau_2$ then

$$\text{ATOMIC}(\sigma \subseteq \tau) = \text{ATOMIC}(\tau_1 \subseteq \sigma_1) \cup \text{ATOMIC}(\sigma_2 \subseteq \tau_2).$$

- If $\sigma \subseteq \tau$ is of the form $\langle h, \mathcal{Z} \rangle \subseteq \langle h', \mathcal{Z}' \rangle$, where h and h' are non-empty, then

$$\text{ATOMIC}(\sigma \subseteq \tau) = \bigcup_{l \in \text{dom}(h)} \text{ATOMIC}(h(l) \subseteq h'(l)) \cup \text{ATOMIC}(\langle \phi, \mathcal{Z} \rangle \subseteq \langle \phi, \mathcal{Z}' \rangle).$$

Let $C = \text{ATOMIC}(\sigma \subseteq \tau)$. We first prove, by induction on the structure of $\sigma \subseteq \tau$, that $C \vdash \sigma \subseteq \tau$ and that C is atomic.

If $\sigma \subseteq \tau$ is atomic, then, trivially, $C \vdash \sigma \subseteq \tau$ and C is atomic.

If σ is of the form $\sigma = \sigma_1 \rightarrow \sigma_2$ and τ is of the form $\tau = \tau_1 \rightarrow \tau_2$, then we can assume inductively that $\text{ATOMIC}(\tau_1 \subseteq \sigma_1) \vdash \tau_1 \subseteq \sigma_1$ and $\text{ATOMIC}(\sigma_2 \subseteq \tau_2) \vdash \sigma_2 \subseteq \tau_2$. Thus, $C \vdash \tau_1 \subseteq \sigma_1$ and $C \vdash \sigma_2 \subseteq \tau_2$, and so, by rule (*arrow*), $C \vdash \sigma \subseteq \tau$. To show that C is atomic, we can assume inductively that both $\text{ATOMIC}(\tau_1 \subseteq \sigma_1)$ and $\text{ATOMIC}(\sigma_2 \subseteq \tau_2)$ are atomic; thus, so is C .

If σ is of the form $\sigma = \langle h, \mathcal{Z} \rangle$ and τ is of the form $\tau = \langle h', \mathcal{Z}' \rangle$, where h and h' are non-empty, then, since $\sigma \subseteq \tau$ is matching, $\text{dom}(h) = \text{dom}(h')$. Since $\text{dom}(h) \neq \phi$, we can assume inductively that

$$\text{ATOMIC}(\langle \phi, \mathcal{Z} \rangle \subseteq \langle \phi, \mathcal{Z}' \rangle) \vdash \langle \phi, \mathcal{Z} \rangle \subseteq \langle \phi, \mathcal{Z}' \rangle$$

and that

$$\text{for all } l \in \text{dom}(h), \text{ATOMIC}(h(l) \subseteq h'(l)) \vdash h(l) \subseteq h'(l).$$

Thus,

$$C \vdash \langle \phi, \mathcal{Z} \rangle \subseteq \langle \phi, \mathcal{Z}' \rangle \text{ and } \forall l \in \text{dom}(h). C \vdash h(l) \subseteq h'(l).$$

Therefore, by rule (*record*), $C \vdash \sigma \subseteq \tau$. To show that C is atomic, we can assume inductively that $\text{ATOMIC}(\langle \phi, \mathcal{Z} \rangle \subseteq \langle \phi, \mathcal{Z}' \rangle)$ is atomic, and that $\forall l \in \text{dom}(h)$, $\text{ATOMIC}(h(l) \subseteq h'(l))$ is atomic. Therefore, so is C .

Since $\sigma \subseteq \tau$ is matching, one of the above cases must hold, thus proving the first part of the lemma.

To show that this set is minimal, we again proceed by induction on the structure of subtype assertions. We wish to prove that, for any atomic set C' , if $C' \vdash \sigma \subseteq \tau$, then $C' \vdash \text{ATOMIC}(\sigma \subseteq \tau)$.

If $\sigma \subseteq \tau$ is atomic and $C' \vdash \sigma \subseteq \tau$, then $C' \vdash \text{ATOMIC}(\sigma \subseteq \tau)$ trivially.

If $\sigma \subseteq \tau$ is of the form $\sigma_1 \rightarrow \sigma_2 \subseteq \tau_1 \rightarrow \tau_2$, then, by Lemma 3.5.2, $C' \vdash \tau_1 \subseteq \sigma_1$ and $C' \vdash \sigma_2 \subseteq \tau_2$. We can thus assume inductively that $C' \vdash \text{ATOMIC}(\tau_1 \subseteq \sigma_1)$ and $C' \vdash \text{ATOMIC}(\sigma_2 \subseteq \tau_2)$. Therefore, $C' \vdash C$.

If σ is of the form $\sigma = \langle h, \mathcal{Z} \rangle$ and τ is of the form $\tau = \langle h', \mathcal{Z}' \rangle$, where h and h' are non-empty, then, by Lemma 3.5.3, $\text{dom}(h) = \text{dom}(h') \neq \phi$, $C' \vdash \langle \phi, \mathcal{Z} \rangle \subseteq \langle \phi, \mathcal{Z}' \rangle$, and $\forall l \in \text{dom}(h). C' \vdash h(l) \subseteq h'(l)$. Since $\text{dom}(h) \neq \phi$, we can assume inductively that

$$C' \vdash \text{ATOMIC}(\langle \phi, \mathcal{Z} \rangle \subseteq \langle \phi, \mathcal{Z}' \rangle) \text{ and } \forall l \in \text{dom}(h). C' \vdash \text{ATOMIC}(h(l) \subseteq h'(l)).$$

Thus, $C' \vdash C$.

Since $\sigma \subseteq \tau$ is matching, one of the above cases must hold, thus proving the lemma. ■

Before proceeding to develop our notion of instance, we need some definitions. We say that a substitution S is a *matching substitution* for a set C of possibly nonmatching assertions $\tau_1 \subseteq \tau_2$ if SC is defined and every $S\tau_1 \subseteq S\tau_2 \in SC$ is matching. Furthermore, we say that a substitution S *respects* a set C of atomic subtype assertions $\tau_1 \subseteq \tau_2$ if SC is defined and every $S\tau_1 \subseteq S\tau_2 \in SC$ is matching.

Using the results of Lemma 3.5.4, we now define

$$S \bullet C = \bigcup_{\sigma \subseteq \tau \in C} \text{ATOMIC}(S\sigma \subseteq S\tau)$$

for any S that respects an atomic set C . The following lemma states that this definition of \bullet gives us precisely the desired behavior.

Lemma 3.5.5 *If a substitution S respects an atomic set C , then there is an atomic set $S \bullet C$ with $S \bullet C \vdash SC$ and such that if C' is any set of atomic subtype assertions with $C' \vdash SC$, then $C' \vdash S \bullet C$.*

Proof. Let $S \bullet C = \bigcup_{\sigma \subseteq \tau \in C} \text{ATOMIC}(S\sigma \subseteq S\tau)$. The proof of the lemma follows easily by Lemma 3.5.4.

Since S respects C , we know that $\forall \sigma \subseteq \tau \in C$. $S\sigma$ matches $S\tau$. Then, by Lemma 3.5.4, we know that $\forall \sigma \subseteq \tau \in C$. $\text{ATOMIC}(S\sigma \subseteq S\tau)$ is atomic and that $\forall \sigma \subseteq \tau \in C$. $\text{ATOMIC}(S\sigma \subseteq S\tau) \vdash S\sigma \subseteq S\tau$. Thus, $S \bullet C$ is atomic. Moreover, since $SC = \bigcup_{\sigma \subseteq \tau} (S\sigma \subseteq S\tau)$,

$$\bigcup_{\sigma \subseteq \tau \in C} \text{ATOMIC}(S\sigma \subseteq S\tau) \vdash SC$$

proving the first part of the lemma.

To prove the second part of the lemma, we note $C' \vdash SC$ implies that $\forall \sigma \subseteq \tau \in C$. $C' \vdash S\sigma \subseteq S\tau$. By Lemma 3.5.4, we know that $\forall \sigma \subseteq \tau \in C$, $C' \vdash \text{ATOMIC}(S\sigma \subseteq S\tau)$. Thus,

$$C' \vdash \bigcup_{\sigma \subseteq \tau \in C} \text{ATOMIC}(S\sigma \subseteq S\tau),$$

which proves the lemma. ■

We note that, in our original definition of instance, if $C' \vdash SC$ then, by Lemma 3.5.1, S must respect C . Thus, putting it all together, we say that

Definition 3.5.6 *A typing $C', A' \supset M : \sigma'$ is an instance of the typing $C, A \supset M : \sigma$ by a substitution S if S respects C , is defined on A and σ , and*

$$C' \vdash S \bullet C, \quad A' \supseteq SA, \quad \text{and} \quad \sigma' = S\sigma$$

We often simply say one statement is an instance of another, without mentioning the substitution involved.

We wish to prove ultimately that, for any typable expression M , our type inference algorithm infers a “most general” typing; that is, a provable typing for M whose set of instances is exactly the set of provable typings for M . To this end, we would like to follow [Mit84] and show that every instance of a provable typing is provable. Then, simply showing that the typing inferred by the algorithm is provable would give the proof of one direction. However, it turns out that the partiality of substitution complicates the situation.

Since a derivation of a typing $C, A \supset M : \sigma$ may involve type expressions that do not appear explicitly in this statement, a substitution which is defined on $C, A \supset M : \sigma$ may not be defined on all typings used in its proof. To see why this is a problem, consider the typing

$$\emptyset, v : \{\mathcal{X}\} \supset (\text{fn } \{a = x; u\} \Rightarrow \{u\})\{a = 3; v\} : \{\mathcal{X}\}$$

which follows from the typing

$$\emptyset, v: \{\mathcal{X}\} \supset \mathbf{fn} \{a = x; u\} \Rightarrow \{u\} : \{a: \mathit{int}; \mathcal{X}\} \rightarrow \{\mathcal{X}\}$$

by rule *app*. For the sake of simplicity, we do not consider the set of subtype assertions, since they are not relevant to this discussion. The typing statement for the first term is syntactically well-formed if we substitute *any* record type for \mathcal{X} . However, the second typing, which is needed to prove the first, becomes ill-formed if we replace \mathcal{X} by $a: \mathit{bool}; \mathcal{Y}$, say, since this gives a two types. Thus, for the expression $(\mathbf{fn} \{a = x; u\} \Rightarrow \{u\})\{a = 3; v\} : \{\mathcal{X}\}$, there is an instance of a provable typing that is NOT provable. It is easy to show by contradiction that this implies that no most general typing exists for the above expression.

This example illustrates the fact that in order to have every instance of a provable typing be provable, we must impose additional conditions on substitutions. Specifically, we must keep track of type expressions that appear in the derivations, but not in the final typing statement.

Fortunately, the required bookkeeping is not as complicated as it might appear at first glance. A careful analysis of the typing rules reveals that for all but one rule, any substitution defined on the consequent of the rule (the typing statement below the horizontal line) will necessarily be defined on all assertions in the antecedent. For rule *coerce*, this is a consequence of the following lemma.

Lemma 3.5.7 *Suppose that a substitution S respects an atomic set C , and $C \vdash \sigma \subseteq \tau$ or $C \vdash \tau \subseteq \sigma$.*

1. *If S is defined on σ , then S is defined on τ .*
2. *If S is defined on σ and τ , then $S\sigma$ matches $S\tau$.*

Proof. We note that by Lemma 3.5.1, σ and τ must match. We also note that, for any S and \mathcal{Z} , $S\langle\phi, \mathcal{Z}\rangle$ is always defined. We prove the lemma by induction on the length of the derivation of $\sigma \subseteq \tau$ or $\tau \subseteq \sigma$ from C .

If the derivation of $\sigma \subseteq \tau$ from C requires no steps, then $\sigma \subseteq \tau$ is of the form $t_1 \subseteq t_2$, $c \subseteq t$, $t \subseteq c$, $c_1 \subseteq c_2$, or $\langle\phi, \mathcal{Z}\rangle \subseteq \langle\phi, \mathcal{Z}'\rangle$. To prove (1), we note that the application of S to a type variable, ground type, or a record type whose first component is the empty function is always defined. To prove (2), we note that since the derivation requires no steps, $\sigma \subseteq \tau \in C$. Since S respects C , $S\sigma$ matches $S\tau$ trivially. The proof of (1) and (2) for the case where $C \vdash \tau \subseteq \sigma$ is analogous.

If the final step of the derivation of $\sigma \subseteq \tau$ is (*trans*), then, $C \vdash \sigma \subseteq \tau$ must have followed by the antecedents $\sigma \subseteq \gamma$ and $\gamma \subseteq \tau$ for some γ . To prove (1), we note that since the proof of $\sigma \subseteq \gamma$ is shorter than that of $\sigma \subseteq \tau$, we can assume inductively that S is defined on γ . Then, since the proof of $\gamma \subseteq \tau$ is also shorter than that of $\sigma \subseteq \tau$, we can assume that S is defined on τ . To prove (2), we note that since the proof of $\sigma \subseteq \gamma$ is shorter than that of $\sigma \subseteq \tau$, we can assume inductively by (1) that S is defined on γ . We can then assume inductively by (2) that $S\sigma$ matches $S\gamma$, and that $S\gamma$ matches $S\tau$. Therefore, by transitivity, $S\sigma$ matches $S\tau$. The proof of (1) and (2) for the case where $C \vdash \tau \subseteq \sigma$ is analogous.

If the final step of the derivation of $\sigma \subseteq \tau$ is (*arrow*), then σ must be of the form $\sigma = \sigma_1 \rightarrow \sigma_2$ and τ must be of the form $\tau = \tau_1 \rightarrow \tau_2$, and $C \vdash \sigma \subseteq \tau$ must have followed by the antecedents $\tau_1 \subseteq \sigma_1$ and $\sigma_2 \subseteq \tau_2$. Since, by assumption, $S\sigma$ is defined, so are $S\sigma_1$ and

$S\sigma_2$. Since the proofs of $\tau_1 \subseteq \sigma_1$ and $\sigma_2 \subseteq \tau_2$ are shorter than that of $\sigma \subseteq \tau$, we can assume that $S\tau_1$ and $S\tau_2$ are defined and that $S\tau_1$ matches $S\sigma_1$ and $S\sigma_2$ matches $S\tau_2$. Thus, τ is defined, and $S\sigma$ matches $S\tau$, proving (1) and (2). The proof of (1) and (2) for the case where $C \vdash \tau \subseteq \sigma$ is analogous.

If the final step of the derivation of $\sigma \subseteq \tau$ is (*record*), then σ must be of the form $\sigma = \langle h, \mathcal{Z} \rangle$ and τ must be of the form $\tau = \langle h', \mathcal{Z}' \rangle$, where $\text{dom}(h) = \text{dom}(h') \neq \phi$. Also, $C \vdash \sigma \subseteq \tau$ must have followed by the antecedents $\langle \phi, \mathcal{Z} \rangle \subseteq \langle \phi, \mathcal{Z}' \rangle$ and $\forall l \in \text{dom}(h). h(l) \subseteq h'(l)$.

To prove (1), it suffices to show that $\forall l \in \text{dom}(h'). S(h'(l))$ is defined and that $\text{dom}(h') \cap \text{dom}(\text{left}(S\langle \phi, \mathcal{Z}' \rangle)) = \phi$. Since, by assumption, $S\sigma$ is defined, then

$$\text{dom}(h) \cap \text{dom}(\text{left}(S\langle \phi, \mathcal{Z} \rangle)) = \phi,$$

and $\forall l \in \text{dom}(h). S(h(l))$ is defined. Since, for all $l \in \text{dom}(h)$, the proofs of $h(l) \subseteq h'(l)$ are shorter than that of $\sigma \subseteq \tau$, we can assume inductively that $\forall l \in \text{dom}(h). S(h'(l))$ is defined. Since the proof of $\langle \phi, \mathcal{Z} \rangle \subseteq \langle \phi, \mathcal{Z}' \rangle$ is shorter than that of $\sigma \subseteq \tau$, we can assume inductively by (2) that $S\langle \phi, \mathcal{Z} \rangle$ matches $S\langle \phi, \mathcal{Z}' \rangle$. Thus,

$$\text{dom}(\text{left}(S\langle \phi, \mathcal{Z} \rangle)) = \text{dom}(\text{left}(S\langle \phi, \mathcal{Z}' \rangle)),$$

which proves (1).

To prove (2), we note that we can assume inductively that for all $l \in \text{dom}(h)$, $S(h(l))$ matches $S(h'(l))$, and that $S\langle \phi, \mathcal{Z} \rangle$ matches $S\langle \phi, \mathcal{Z}' \rangle$. Therefore, if we let $h_1 = \text{left}(S\langle \phi, \mathcal{Z} \rangle)$ and $h_2 = \text{left}(S\langle \phi, \mathcal{Z}' \rangle)$, then by the definition of match, $\text{dom}(h_1) = \text{dom}(h_2)$, and $\forall l \in \text{dom}(h_1). h_1(l)$ matches $h_2(l)$. Careful inspection of the definition of match shows that this implies that $S\sigma$ matches $S\tau$, as desired. The proof of (1) and (2) for the case where $C \vdash \tau \subseteq \sigma$ is analogous, which proves the lemma. ■

In most other typing rules, every type expression appearing in the antecedent occurs (possibly as a subexpression) somewhere in the consequent. The only exception is in the application rule *app*, where the so-called *cut formula* (σ , as the rule is written) is eliminated. (This will come as no surprise to proof theorists.) Thus, in order to determine which substitutions preserve provability of typing statements, we only need to keep track of the cut formulas used in proofs.

Since the set of cut formulas will be used as a restriction on allowable substitutions, we adopt a notation that combines typing statements with restrictions. We will call a formula $R \mid C, A \supset M : \sigma$ combining a typing statement with a set R of type expressions a *restricted typing statement*. Moreover, we say that

Definition 3.5.8 $C, A \supset M : \sigma$ is R -provable if $C, A \supset M : \sigma$ is provable by a derivation using only types from R as cut formulas, and we write $\vdash R \mid C, A \supset M : \sigma$.

Notice that since there are no function applications (and therefore no cut formulas) in patterns, there is no need to compute any restriction on substitutions for patterns – every well-formed substitution instance of a provable pattern typing is provable. However, due to the algorithmic advantages that arise from the similarities between patterns and expressions, we use restricted typing statements for patterns as well, noting here that R will always be empty.

Using restrictions, we may now define a useful instance relation on restricted typing statements.

Definition 3.5.9 A restricted typing $R' | C', A' \supset M : \sigma'$ is an instance of $R | C, A \supset M : \sigma$ by substitution S if S respects C , is defined on R, A , and σ , and

$$R' \supseteq SR, \quad C' \vdash S \bullet C, \quad A' \supseteq SA, \quad \text{and} \quad \sigma' = S\sigma$$

Again, we often simply say one restricted typing statement is an instance of another, without mentioning the substitution involved.

While we have defined “provability” of a restricted typing statement $R | C, A \supset M : \sigma$ using the typing system for statements without restrictions, the typing system of Section 3.4 can be reformulated to prove restricted statements directly. The changes are relatively minor. Essentially, restrictions are added to each statement in each rule, with the cut type (formula) added to the restriction set in the consequent of rule *app*. It is easy to show that a restricted statement is provable in this augmented system iff it is R -provable. Through the rest of the paper, we refer to this augmented system, which is summarized in Table 3.1, as the R -typing system.

3.6 Properties of the R -Typing System

This section is devoted to proving Theorem 3.6.7, the main theorem of this chapter, which shows that, in R -typing system, all instances of a provable typing are provable. In order to prove this theorem, we will need to use a number of technical properties of the R -typing system. Thus, before presenting the proof of Theorem 3.6.7, we first discuss these technical properties in some detail.

First, we want to show that, if a restricted typing $R | C, A \supset M : \sigma$ is provable, then if we add “extra” type restrictions to R , “extra” subtype assertions to C , or “extra” variable-to-type or extension variable-to-record type associations to A , the resulting restricted typing is also provable. As a consequence of these properties, we can prove Theorem 3.6.7 by simply showing that $\vdash R | C, A \supset M : \sigma$ implies that $\vdash SR | S \bullet C, SA \supset M : S\sigma$ for any substitution S that respects C and is defined on R, A , and σ .

The following three lemmas formally state the properties mentioned above.

Lemma 3.6.1 *If $\vdash R | C, A \supset M : \sigma$ and $R' \supseteq R$, then $\vdash R' | C, A \supset M : \sigma$.*

Proof. We prove this lemma by induction on the length of the derivation of $\vdash R | C, A \supset M : \sigma$.

If the derivation requires no steps, then $\vdash R | C, A \supset M : \sigma$ follows from either (*int*), (*real*), (*bool*), (*string*), (*var*), (*rec1*), (*rec2*), or (*const*). The lemma holds trivially since, for any R' , $\vdash R' | C, A \supset M : \sigma$.

If the final step of the derivation is (*coerce*), then $\vdash R | C, A \supset M : \sigma$ must have followed by the antecedents $\vdash R | C, A \supset M : \gamma$ and $C \vdash \gamma \subseteq \sigma$ for some γ . Since the proof of this is shorter, we can assume inductively that $\vdash R' | C, A \supset M : \gamma$. By (*coerce*), we have that $\vdash R' | C, A \supset M : \sigma$.

If the final step of the derivation is (*app*), then M is of the form $M = M'N'$. Thus, for some γ , $R = R_1 \cup \{\gamma\}$, and $\vdash R | C, A \supset M : \sigma$ must have followed by the antecedents $\vdash R_1 | C, A \supset M' : \gamma \rightarrow \sigma$ and $\vdash R_1 | C, A \supset N' : \gamma$. Since the proofs of these are shorter, and since $R' \supseteq R_1$, we can assume that $\vdash R' | C, A \supset M' : \gamma \rightarrow \sigma$ and $\vdash R' | C, A \supset N' : \gamma$. Since $\gamma \in R'$, we can infer that $\vdash R' | C, A \supset M : \sigma$ by (*app*).

If the final step of the derivation is (*abs*), then M is of the form $M = \mathbf{fn} P \Rightarrow M'$ and σ is of the form $\sigma = \sigma_1 \rightarrow \sigma_2$. Moreover, $\vdash R | C, A \supset M : \sigma$ must have followed by the antecedents

<i>int</i>	$R \mid C, A \supset b: \mathbf{int}$ whenever b is an integer	
<i>real</i>	$R \mid C, A \supset b: \mathbf{real}$ whenever b is a real	
<i>bool</i>	$R \mid C, A \supset b: \mathbf{bool}$ whenever b is a boolean	
<i>string</i>	$R \mid C, A \supset b: \mathbf{string}$ whenever b is a string	
<i>const</i>	$R \mid C, A \supset q: S\tau_q$ whenever S is defined on τ_q , where τ_q is the built-in type for q	
<i>var</i>	$R \mid C, A \supset x: \sigma$ whenever $x: \sigma \in A$	
<i>app</i>	$\frac{R \mid C, A \supset M: \sigma \rightarrow \tau, R \mid C, A \supset N: \sigma}{R \cup \{\sigma\} \mid C, A \supset MN: \tau}$	
<i>abs</i>	$\frac{R \mid C^{op}, A' \supset P: \sigma, R \mid C, A[A'] \supset M: \tau}{R \mid C, A \supset \mathbf{fn} P \Rightarrow M: \sigma \rightarrow \tau}$	$vars(A') = vars(P)$
<i>coerce</i>	$\frac{R \mid C, A \supset M: \sigma, C \vdash \sigma \subseteq \tau}{R \mid C, A \supset M: \tau}$	
<i>rec1</i>	$R \mid C, A \supset \langle \phi, EMPTY \rangle: \langle \phi, NULL \rangle$	
<i>rec2</i>	$R \mid C, A \supset \langle \phi, u \rangle: \langle h, \mathcal{Z} \rangle$ whenever $u: \langle h, \mathcal{Z} \rangle \in A$	
<i>rec3</i>	$\frac{R \mid C, A \supset \langle \phi, E \rangle: \langle h_2, \mathcal{Z} \rangle, \forall l \in dom(f). R \mid C, A \supset f(l): h_1(l)}{R \mid C, A \supset \langle f, E \rangle: \langle h_1 + h_2, \mathcal{Z} \rangle}$	all typings well-forme
	where $dom(f) = dom(h_1) \neq \phi$	

Table 3.1: The R -Typing System

$\vdash R \mid C^{op}, A' \supset P: \sigma_1$ and $\vdash R \mid C, A[A'] \supset M': \sigma_2$ for some A' such that $vars(A') = vars(P)$. Since the proofs of these are shorter, we can assume that $\vdash R' \mid C^{op}, A' \supset P: \sigma_1$ and $\vdash R' \mid C, A[A'] \supset M': \sigma_2$. By (*abs*), we have that $\vdash R' \mid C, A \supset M: \sigma$.

If the final step of the derivation is (*rec3*), then M is the form $M = \langle f, E \rangle$ and σ is of the form $\sigma = \langle h_1 + h_2, \mathcal{Z} \rangle$. Moreover, $\vdash R \mid C, A \supset M: \sigma$ must have followed by the antecedents $dom(f) = dom(h_1) \neq \phi$, $\vdash R \mid C, A \supset \langle \phi, E \rangle: \langle h_2, \mathcal{Z} \rangle$, and $\forall l \in dom(f). \vdash R \mid C, A \supset f(l): h_1(l)$. Since the proofs of these are shorter, we can assume that $\vdash R' \mid C, A \supset \langle \phi, E \rangle: \langle h_2, \mathcal{Z} \rangle$, and that $\forall l \in dom(f). \vdash R' \mid C, A \supset f(l): h_1(l)$. By (*rec3*), we can infer that $\vdash R' \mid C, A \supset M: \sigma$, which proves the lemma. ■

Lemma 3.6.2 *If $\vdash R \mid C_1, A \supset M: \sigma$ and $C_2 \vdash C_1$, where C_2 is an atomic set, then $\vdash R \mid C_2, A \supset M: \sigma$.*

Proof. We prove this lemma by induction on the length of the derivation of $\vdash R \mid C_1, A \supset M: \sigma$.

If the derivation requires no steps, then $\vdash R \mid C_1, A \supset M: \sigma$ follows from either (*int*), (*real*), (*bool*), (*string*), (*var*), (*rec1*), (*rec2*), or (*const*). The lemma holds trivially since, for any C_2 , $\vdash R \mid C_2, A \supset M: \sigma$.

If the final step of the derivation is (*coerce*), then, $\vdash R \mid C_1, A \supset M: \sigma$ must have followed by the antecedents $\vdash R \mid C_1, A \supset M: \gamma$ and $C_1 \vdash \gamma \subseteq \sigma$ for some γ . Since the proof of $\vdash R \mid C_1, A \supset M: \gamma$ is shorter, we can assume inductively that $\vdash R \mid C_2, A \supset M: \gamma$. Since $C_2 \vdash C_1$, we know that $C_2 \vdash \gamma \subseteq \sigma$. By (*coerce*), we have that $\vdash R \mid C_2, A \supset M: \sigma$.

If the final step of the derivation is (*app*), then M is of the form $M = M'N'$, and $\vdash R \mid C_1, A \supset M: \sigma$ must have followed by the antecedents $\vdash R \mid C_1, A \supset M': \gamma \rightarrow \sigma$ and $\vdash R \mid C_1, A \supset N': \gamma$ for some γ . Since the proofs of these restricted typing statements are shorter, we can assume that $\vdash R \mid C_2, A \supset M': \gamma \rightarrow \sigma$ and $\vdash R \mid C_2, A \supset N': \gamma$. By (*app*), we have that $\vdash R \mid C_2, A \supset M: \sigma$.

If the final step of the derivation is (*abs*), then M is of the form $M = \mathbf{fn} P \Rightarrow M'$ and σ is of the form $\sigma = \sigma_1 \rightarrow \sigma_2$. Moreover, $\vdash R \mid C_1, A \supset M: \sigma$ must have followed by the antecedents $\vdash R \mid C_1^{op}, A' \supset P: \sigma_1$ and $\vdash R \mid C_1, A[A'] \supset M': \sigma_2$ for some A' such that $vars(A') = vars(P)$. Since the proofs of these restricted typing statements are shorter, we can assume that $\vdash R \mid C_2^{op}, A' \supset P: \sigma_1$ and $\vdash R \mid C_2, A[A'] \supset M': \sigma_2$. By (*abs*), we can infer that $\vdash R \mid C_2, A \supset M: \sigma$.

If the final step of the derivation is (*rec3*), then M is the form $M = \langle f, E \rangle$ and σ is of the form $\sigma = \langle h_1 + h_2, \mathcal{Z} \rangle$. Moreover, $\vdash R \mid C_1, A \supset M: \sigma$ must have followed by the antecedents $dom(f) = dom(h_1) \neq \phi$, $\vdash R \mid C_1, A \supset \langle \phi, E \rangle: \langle h_2, \mathcal{Z} \rangle$, and $\forall l \in dom(f). \vdash R \mid C_1, A \supset f(l): h_1(l)$. Since the proofs of these restricted typing statements are shorter, we can assume that $\vdash R \mid C_2, A \supset \langle \phi, E \rangle: \langle h_2, \mathcal{Z} \rangle$, and that

$$\forall l \in dom(f). \vdash R \mid C_2, A \supset f(l): h_1(l).$$

Thus, by (*rec3*), $\vdash R \mid C_2, A \supset M: \sigma$. ■

We prove a slightly more complicated lemma for type environments. In our proof of Theorem 3.6.7, we only need to use the second property outlined below; however, in the proofs of our main theorems in Chapter 4, we will need to use all three of these properties.

Lemma 3.6.3 *Suppose that $\vdash R \mid C, A \supset M: \sigma$.*

1. *If w occurs free in M , then $w: \tau \in A$ for some τ .*

2. If $w:\tau \in B$ for every $w:\tau \in A$ with w free in M , then $\vdash R \mid C, B \supset M:\sigma$.
3. There is a proof of $\vdash R \mid C, B \supset M:\sigma$ that is of the same length as that of $\vdash R \mid C, A \supset M:\sigma$.

Proof. We note that proving part (1) is equivalent to proving that if $\vdash R \mid C, A \supset M:\sigma$, then $\text{vars}(M) \subseteq \text{vars}(A)$. For part (2), it is sufficient to prove that if $\vdash R \mid C, A \supset M:\sigma$ and if $A_M \subseteq B$, where $A_M = \{w:\sigma \mid w:\sigma \in A \text{ and } w \in \text{vars}(M)\}$ then $\vdash R \mid C, B \supset M:\sigma$. We prove the lemma by induction on the length of the derivation of $\vdash R \mid C, A \supset M:\sigma$.

If the derivation requires no steps, then the axiom used is either *(int)*, *(real)*, *(bool)*, *(string)*, *(var)*, *(rec1)*, *(rec2)*, or *(const)*. Since $\text{vars}(b) = \text{vars}(\langle \phi, \text{EMPTY} \rangle) = \text{vars}(q) = \phi$, part (1) is trivially true for all the cases other than *(var)* and *(rec2)*. To prove (1), we note that if the rule applied is *(var)* or *(rec2)*, then $\text{vars}(M) \subseteq \text{vars}(A)$ by the axiom itself. Similarly, part (2) is trivially true if the rule applied is any other than *(var)* or *(rec2)*, since $\vdash R \mid C, B \supset M:\sigma$ for any B . To prove (2), we note that if the axiom used is *(var)*, where $M = x$, then $A_M = \{x:\sigma\}$. Since, by assumption, $A_M \subseteq B$, we can infer that $\vdash R \mid C, B \supset M:\sigma$ by *(var)*. If the axiom used is *(rec2)*, where $M = \langle \phi, u \rangle$, then $A_M = \{u:\sigma\}$. Since, by assumption, $A_M \subseteq B$, we can infer that $\vdash R \mid C, B \supset M:\sigma$ by *(rec2)*. For part (3), we note that $\vdash R \mid C, B \supset M:\sigma$ holds by the same axiom as $\vdash R \mid C, A \supset M:\sigma$.

If the final step of the derivation is *(coerce)*, then $\vdash R \mid C, A \supset M:\sigma$ must have followed by the antecedents $\vdash R \mid C, A \supset M:\gamma$ and $C \vdash \gamma \subseteq \sigma$ for some γ . Since the proof of $\vdash R \mid C, A \supset M:\gamma$ is shorter, we can assume inductively that (1) holds true for M . To prove (2), we can assume inductively that $\vdash R \mid C, B \supset M:\gamma$. Since $C \vdash \gamma \subseteq \sigma$, we can derive by rule *(coerce)* that $\vdash R \mid C, B \supset M:\sigma$. To prove (3), we can assume inductively that the proof of $\vdash R \mid C, B \supset M:\gamma$ is of the same length as that of $\vdash R \mid C, A \supset M:\gamma$. By *(coerce)*, we can infer that (3) holds for $\vdash R \mid C, B \supset M:\sigma$.

If the final step of the derivation is *(app)*, then M is of the form $M = M'N'$, and $\vdash R \mid C, A \supset M:\sigma$ must have followed by the antecedents $\vdash R \mid C, A \supset M':\gamma \rightarrow \sigma$ and $\vdash R \mid C, A \supset N':\gamma$ for some γ . Since the derivations of these are shorter, we can assume that

$$\text{vars}(M') \subseteq \text{vars}(A) \text{ and } \text{vars}(N') \subseteq \text{vars}(A).$$

Thus, $\text{vars}(M) \subseteq \text{vars}(A)$. To prove (2), we note that $A_M \subseteq B$ implies that $A_{M'} \subseteq B$ and $A_{N'} \subseteq B$. We can thus assume inductively that $\vdash R \mid C, B \supset M':\gamma \rightarrow \sigma$ and $\vdash R \mid C, B \supset N':\gamma$. By rule *(app)*, we can infer that $\vdash R \mid C, B \supset M:\sigma$. To prove (3), we can assume inductively that (3) holds for $\vdash R \mid C, B \supset M':\gamma \rightarrow \sigma$ and for $\vdash R \mid C, B \supset N':\gamma$. By rule *(app)*, we have that (3) holds for $\vdash R \mid C, B \supset M:\sigma$.

If the final step of the derivation is *(abs)*, then M is of the form $M = \mathbf{fn} P \Rightarrow M'$ and σ is of the form $\sigma = \sigma_1 \rightarrow \sigma_2$. Moreover, $\vdash R \mid C, A \supset M:\sigma$ must have followed by the antecedents $\vdash R \mid C^{op}, A' \supset P:\sigma_1$ and $\vdash R \mid C, A[A'] \supset M':\sigma_2$ for some A' such that $\text{vars}(A') = \text{vars}(P)$. Thus, we can assume inductively that $\text{vars}(M') \subseteq \text{vars}(A[A'])$. We note that by definition, $A[A'] = A_1 \cup A'$,

$$\text{where } A = A_1 \cup A_2, A_2 = \{w:\sigma \mid w:\sigma \in A \text{ and } w \in \text{vars}(A')\},$$

which implies that $A[A'] - A' = A_1$. Therefore,

$$\text{vars}(M') - \text{vars}(P) \subseteq \text{vars}(A_1) \subseteq \text{vars}(A),$$

as desired. To prove (2), we note that $A_M \subseteq B$ implies that $(A[A'])_M \subseteq B[A']$, and thus we can assume inductively that $\vdash R \mid C, B[A'] \supset M':\sigma_2$. By *(abs)*, we can infer that $\vdash R \mid C, B \supset M:\sigma$. To prove (3), we can assume inductively that (3) holds for $\vdash R \mid C, B[A'] \supset M':\sigma_2$. By *(abs)*, we have that (3) holds for $\vdash R \mid C, B \supset M:\sigma$.

If the final step of the derivation is *(rec3)*, then M is the form $M = \langle f, E \rangle$ and σ is of the form $\sigma = \langle h_1 + h_2, \mathcal{Z} \rangle$. Moreover, $\vdash R \mid C, A \supset M:\sigma$ must have followed by the antecedents $\text{dom}(f) = \text{dom}(h_1) \neq \phi$, $\vdash R \mid C, A \supset \langle \phi, E \rangle: \langle h_2, \mathcal{Z} \rangle$, and $\forall l \in \text{dom}(f)$. $\vdash R \mid C, A \supset f(l):h_1(l)$. We can assume inductively that

$$\text{vars}(\langle \phi, E \rangle) \subseteq \text{vars}(A) \text{ and that } \forall l \in \text{dom}(h). \text{vars}(f(l)) \subseteq \text{vars}(A).$$

Thus, $\text{vars}(M) \subseteq \text{vars}(A)$, proving (1). To prove (2), we note that $A_M \subseteq B$ implies that $A_{\langle \phi, E \rangle} \subseteq B$ and that $\forall l \in \text{dom}(h)$. $A_{f(l)} \subseteq B$. We can thus assume inductively that $\vdash R \mid C, B \supset \langle \phi, E \rangle: \langle h_2, \mathcal{Z} \rangle$, and that

$$\forall l \in \text{dom}(f). \vdash R \mid C, B \supset f(l):h_1(l).$$

By rule *(rec3)*, we can infer that $\vdash R \mid C, B \supset M:\sigma$. To prove (3), we can assume inductively that (3) holds for $\vdash R \mid C, B \supset \langle \phi, E \rangle: \langle h_2, \mathcal{Z} \rangle$ and for $\forall l \in \text{dom}(f)$. $\vdash R \mid C, B \supset f(l):h_1(l)$. By *(rec3)*, we have that (3) holds for $\vdash R \mid C, B \supset M:\sigma$, proving the lemma. \blacksquare

The next three lemmas present some properties about substitutions that we will need in the proof of Theorem 3.6.7 for technical reasons.

Lemma 3.6.4 *Suppose that σ , γ , and τ match. If C is an atomic set, and $C \vdash \text{ATOMIC}(\sigma \subseteq \gamma)$ and $C \vdash \text{ATOMIC}(\gamma \subseteq \tau)$, then $C \vdash \text{ATOMIC}(\sigma \subseteq \tau)$.*

Proof. The proof proceeds by induction on the structure of σ , γ , and τ . If $\sigma \subseteq \gamma$ and $\gamma \subseteq \tau$ are atomic, then $\text{ATOMIC}(\sigma \subseteq \gamma) = \{\sigma \subseteq \gamma\}$, and $\text{ATOMIC}(\gamma \subseteq \tau) = \{\gamma \subseteq \tau\}$. Thus, $C \vdash \{\sigma \subseteq \tau\}$, as desired.

For the remaining cases, the proof follows by simple induction. \blacksquare

Using the above lemma, we show the following property.

Lemma 3.6.5 *Suppose that a substitution S respects an atomic set C and is defined on σ and τ . If $C \vdash \sigma \subseteq \tau$, then $S \bullet C \vdash \text{ATOMIC}(S\sigma \subseteq S\tau)$.*

Proof. We prove the lemma by induction on the length of the derivation of $\sigma \subseteq \tau$ from C .

If the derivation of $\sigma \subseteq \tau$ from C requires no steps, then $\sigma \subseteq \tau \in C$. Thus, by definition,

$$\text{ATOMIC}(S\sigma \subseteq S\tau) \in S \bullet C.$$

If the final step of the derivation of $\sigma \subseteq \tau$ is *(trans)*, then $C \vdash \sigma \subseteq \tau$ must have followed by the antecedents $\sigma \subseteq \gamma$ and $\gamma \subseteq \tau$ for some γ . By Lemma 3.5.7, we have that S is defined on γ . Since the proofs of $\sigma \subseteq \gamma$ and $\gamma \subseteq \tau$ are shorter than that of $\sigma \subseteq \tau$, we can assume that

$$S \bullet C \vdash \text{ATOMIC}(S\sigma \subseteq S\gamma) \text{ and that } S \bullet C \vdash \text{ATOMIC}(S\gamma \subseteq S\tau).$$

By Lemma 3.6.4, we can infer that $\text{ATOMIC}(S \bullet C \vdash S\sigma \subseteq S\tau)$.

If the final step of the derivation of $\sigma \subseteq \tau$ is *(arrow)*, then σ must be of the form $\sigma = \sigma_1 \rightarrow \sigma_2$ and τ must be of the form $\tau = \tau_1 \rightarrow \tau_2$, and $C \vdash \sigma \subseteq \tau$ must have followed by the antecedents $\tau_1 \subseteq \sigma_1$ and $\sigma_2 \subseteq \tau_2$. Since, by assumption, $S\sigma$ and $S\tau$ are defined, so are

$S\sigma_1, S\sigma_2, S\tau_1$, and $S\tau_2$. Since the proofs of $\tau_1 \subseteq \sigma_1$ and $\sigma_2 \subseteq \tau_2$ are shorter than that of $\sigma \subseteq \tau$, we can assume that

$$S \bullet C \vdash \text{ATOMIC}(S\tau_1 \subseteq S\sigma_1) \text{ and } S \bullet C \vdash \text{ATOMIC}(S\sigma_2 \subseteq S\tau_2).$$

Thus, $S \bullet C \vdash \text{ATOMIC}(S\sigma \subseteq S\tau)$.

If the final step of the derivation of $\sigma \subseteq \tau$ is (*record*), then σ must be of the form $\sigma = \langle h, \mathcal{Z} \rangle$ and τ must be of the form $\tau = \langle h', \mathcal{Z}' \rangle$, where $\text{dom}(h) = \text{dom}(h') \neq \phi$. Since the final step of the derivation is (*record*), we can assume that $C \vdash \sigma \subseteq \tau$ must have followed by the antecedents $\langle \phi, \mathcal{Z} \rangle \subseteq \langle \phi, \mathcal{Z}' \rangle$ and $\forall l \in \text{dom}(h). h(l) \subseteq h'(l)$. Since the proofs of these inclusions are shorter, we can assume inductively that

$$\forall l \in \text{dom}(h). S \bullet C \vdash \text{ATOMIC}(S(h(l)) \subseteq S(h'(l)))$$

and

$$S \bullet C \vdash \text{ATOMIC}(S\langle \phi, \mathcal{Z} \rangle \subseteq S\langle \phi, \mathcal{Z}' \rangle)$$

Let $\langle h_s, Z_s \rangle = S\langle \phi, \mathcal{Z} \rangle$, and let $\langle h_s', Z_s' \rangle = S\langle \phi, \mathcal{Z}' \rangle$. Since $S\sigma$ and $S\tau$ match, we can infer that $\text{dom}(h_s) = \text{dom}(h_s')$. Then,

$$\forall l \in \text{dom}(h_s). S \bullet C \vdash \text{ATOMIC}(h_s(l) \subseteq h_s'(l)),$$

and $S \bullet C \vdash \text{ATOMIC}(\langle \phi, Z_s \rangle \subseteq \langle \phi, Z_s' \rangle)$. Thus, $S \bullet C \vdash \text{ATOMIC}(S\sigma \subseteq S\tau)$. \blacksquare

The third property is as follows.

Lemma 3.6.6 *Suppose that $\vdash R \mid C, A \supset P: \sigma$ where $\text{vars}(A) = \text{vars}(P)$. If S is a substitution that respects C and is defined on σ , then S is defined on A .*

Proof. The proof is by induction on the length of the derivation of $\vdash R \mid C, A \supset P: \sigma$. We write A_P for the type environment A restricted to the free variables of P ; more precisely, $A_P = \{w: \sigma \mid w: \sigma \in A \text{ and } w \in \text{vars}(P)\}$.

If the derivation requires no steps, then $\vdash R \mid C, A \supset P: \sigma$ follows from either (*int*), (*real*), (*bool*), (*string*), (*var*), (*rec1*), or (*rec2*). For all of the cases except (*var*) and (*rec2*), the lemma holds trivially since $A = \phi$. If the axiom used is (*var*), where P is of the form $P = x$, then $A = \{x: \sigma\}$, and so, by assumption, S is defined on A . If the axiom used is (*rec2*), where P is of the form $P = \langle \phi, u \rangle$ then $A = \{u: \sigma\}$, and again by assumption, S is defined on A .

If the final step of the derivation is (*coerce*), then $\vdash R \mid C, A \supset P: \sigma$ must have followed by the antecedents $\vdash R \mid C, A \supset P: \gamma$ and $C \vdash \gamma \subseteq \sigma$ for some γ . By Lemma 3.5.7, S is defined on γ . Since the proof of $\vdash R \mid C, A \supset P: \gamma$ is shorter than that of $\vdash R \mid C, A \supset P: \sigma$, we can assume inductively that S is defined on A .

If the final step of the derivation is (*rec3*), then P is the form $P = \langle f, E \rangle$ and σ is of the form $\sigma = \langle h_1 + h_2, \mathcal{Z} \rangle$. Moreover, $\vdash R \mid C, A \supset P: \sigma$ must have followed by the antecedents $\text{dom}(f) = \text{dom}(h_1) \neq \phi$, $\vdash R \mid C, A \supset \langle \phi, E \rangle: \langle h_2, \mathcal{Z} \rangle$, and $\forall l \in \text{dom}(f). \vdash R \mid C, A \supset f(l): h_1(l)$. By Lemma 3.6.3, we know that

$$\vdash R \mid C, A_{\langle \phi, E \rangle} \supset \langle \phi, E \rangle: \langle h_1, \mathcal{Z} \rangle,$$

and that

$$\forall l \in \text{dom}(f). \vdash R \mid C, A_{f(l)} \supset f(l): h_1(l).$$

Moreover, by Lemma 3.6.3, we can infer that the proofs of these are shorter than that of $\vdash R \mid C, A \supset P: \sigma$.

We can thus assume inductively that $\forall l \in \text{dom}(f)$. S is defined on $A_{f(l)}$ and that S is defined on $A_{\langle \phi, E \rangle}$. Since

$$A = \bigcup_{l \in \text{dom}(f)} A_{f(l)} \cup A_{\langle \phi, E \rangle},$$

S is defined on A , which proves the lemma. \blacksquare

Having developed all the necessary technical machinery, we now prove the main theorem of the chapter. This theorem shows that the R -typing system has the important property that the original typing system lacked: namely, that all instances of a provable typing are provable. In Chapter 4, we will make use of this property to show that the algorithm infers most general typings.

Theorem 3.6.7 *Suppose that $R' \mid C', A' \supset M: \sigma'$ is an instance of $R \mid C, A \supset M: \sigma$. If $\vdash R \mid C, A \supset M: \sigma$, then $\vdash R' \mid C', A' \supset M: \sigma'$.*

Proof. We prove the lemma by induction on the length of the derivation of $\vdash R \mid C, A \supset M: \sigma$.

We note that if $R' \mid C', A' \supset M: \sigma'$ is an instance of $R \mid C, A \supset M: \sigma$, then there exists a substitution S such that S respects C and is defined on R, A and σ , and such that

$$R' \supseteq SR, \quad C' \vdash S \bullet C, \quad A' \supseteq SA, \quad \text{and} \quad \sigma' = S\sigma.$$

If the derivation requires no steps, then $R \mid C, A \supset M: \sigma$ must have followed by a typing axiom. For the purposes of our proof, it suffices to show that $\vdash SR \mid S \bullet C, SA \supset M: S\sigma$, since we can then conclude by Lemma 3.6.3, Lemma 3.6.2, and Lemma 3.6.1 that $\vdash R' \mid C', A' \supset M: \sigma'$. If the typing axiom used is *(int)*, *(real)*, *(bool)*, *(string)*, or *(rec1)* then, trivially, $\vdash SR \mid S \bullet C, SA \supset M: S\sigma$ by that same axiom. If the axiom used is *(var)* or *(rec2)* $\vdash SR \mid S \bullet C, SA \supset M: S\sigma$ follows by that same axiom, since $S\sigma \in SA$.

Otherwise, the axiom used was *(const)*. Thus, for some substitution T that is defined on τ_q , we have that $\sigma = T\tau_q$, and so, $S\sigma = S(T\tau_q)$. Since, by assumption, $S\sigma$ is defined, we can assume that $S(T\tau_q)$ is defined as well. We would now like to show that $S\sigma = (S; T)\tau_q$; however, since T may map type or row variables other than τ_q to types, $S; T$ may not be defined. We thus consider instead the substitution T' , where $\text{dom}(T') = \{\tau_q\}$, and $T'[\tau_q] = T[\tau_q]$, and note that since $S\sigma$ is defined, so is $S; T'$. Thus, we can infer that $\vdash SR \mid S \bullet C, SA \supset M: (S; T')\sigma$ by *(const)*.

If the final step of the derivation is *(coerce)*, then $\vdash R \mid C, A \supset M: \sigma$ must have followed by the antecedents $\vdash R \mid C, A \supset M: \gamma$ and $C \vdash \gamma \subseteq \sigma$ for some γ . We note that by Lemma 3.5.7, S is defined on γ . Moreover, by Lemma 3.6.5, we have that $S \bullet C \vdash \text{ATOMIC}(S\gamma \subseteq S\sigma)$; thus, by transitivity,

$$C' \vdash S\gamma \subseteq S\sigma.$$

Since $R' \mid C', A' \supset M: S\gamma$ is an instance of $R \mid C, A \supset M: \gamma$, and the proof of $\vdash R \mid C, A \supset M: \gamma$ is shorter than that of $\vdash R \mid C, A \supset M: \sigma$, we can assume that $\vdash R' \mid C', A' \supset M: S\gamma$. By *(coerce)*, we can infer that $\vdash R' \mid C', A' \supset M: \sigma'$.

If the final step of the derivation is *(app)*, then M is of the form $M = M'N'$, and $\vdash R \mid C, A \supset M: \sigma$ must have followed by the antecedents $\vdash R_1 \mid C, A \supset M': \gamma \rightarrow \sigma$ and

$\vdash R_1 | C, A \supset N': \gamma$ for some γ and R such that $R = R_1 \cup \{\gamma\}$. Since by assumption S is defined on R , we can assume that

S is defined on γ .

It is worthwhile to note here that *this is precisely the reason that restriction sets are needed in our typing system*. Without restriction sets, it is impossible to guarantee that S is defined on γ .

Proceeding with the proof, we can now assume that $R' | C', A' \supset M': S\gamma \rightarrow S\sigma$ is an instance of $R_1 | C, A \supset M': \gamma \rightarrow \sigma$ and $R' | C', A' \supset N': S\gamma$ is an instance of $R_1 | C, A \supset N': \gamma$. Since the proofs of $\vdash R_1 | C, A \supset M': \gamma \rightarrow \sigma$ and $\vdash R_1 | C, A \supset N': \gamma$ are shorter than that of $\vdash R | C, A \supset M: \sigma$, we can assume inductively that $\vdash R' | C', A' \supset M': S\gamma \rightarrow S\sigma$ and $\vdash R' | C', A' \supset N': S\gamma$. Thus, since $S\gamma \in R'$, we can infer by (*app*) that $\vdash R' | C', A' \supset M: \sigma'$.

If the final step of the derivation is (*abs*), then M is of the form $M = \mathbf{fn} P \Rightarrow M'$ and σ is of the form $\sigma = \sigma_1 \rightarrow \sigma_2$. Moreover, $\vdash R | C, A \supset M: \sigma$ must have followed by the antecedents $\vdash R | C^{op}, A_P \supset P: \sigma_1$ and $\vdash R | C, A[A_P] \supset M': \sigma_2$ for some A_P such that $\mathit{vars}(A_P) = \mathit{vars}(P)$. Since, by assumption, S is defined on σ , we can assume that S is defined on σ_1 and σ_2 . Thus, by Lemma 3.6.6,

S is defined on A_P .

Since $A[A_P] = A_1 \cup A_P$, where $A_1 \subseteq A$,

S is defined on $A[A_P]$.

We can thus assume that $SR | S \bullet C^{op}, SA_P \supset P: S\sigma_1$ is an instance of $R | C^{op}, A_P \supset P: \sigma_1$, and that $SR | S \bullet C, S(A[A_P]) \supset M': S\sigma_2$ is an instance of $R | C, A[A_P] \supset M': \sigma_2$. Since the proofs of $\vdash R | C^{op}, A_P \supset P: \sigma_1$ and $\vdash R | C, A[A_P] \supset M': \sigma_2$ are shorter than that of $\vdash R | C, A \supset M: \sigma$, we can assume inductively that $\vdash SR | S \bullet C^{op}, SA_P \supset P: S\sigma_1$ and $\vdash SR | S \bullet C, S(A[A_P]) \supset M': S\sigma_2$. By (*abs*), we can infer that $\vdash SR | SC, SA \supset \mathbf{fn} P \Rightarrow M': S\sigma_1 \rightarrow S\sigma_2$. Using the results of Lemma 3.6.3, Lemma 3.6.2, and Lemma 3.6.1, we can conclude that $\vdash R' | C', A' \supset M: \sigma'$.

If the final step of the derivation is (*rec3*), then M is the form $M = \langle f, E \rangle$ and σ is of the form $\sigma = \langle h, \mathcal{Z} \rangle$, where, for some h_1 and h_2 , $h = h_1 + h_2$ and $\mathit{dom}(h_1) \cap \mathit{dom}(h_2) = \phi$. Moreover, $\vdash R | C, A \supset M: \sigma$ must have followed by the antecedents $\mathit{dom}(f) = \mathit{dom}(h_1) \neq \phi$, $\vdash R | C, A \supset \langle \phi, E \rangle: \langle h_2, \mathcal{Z} \rangle$, and $\forall l \in \mathit{dom}(f)$. $\vdash R | C, A \supset f(l): h_1(l)$. Since, by assumption, S is defined on σ , we can assume that $\forall l \in \mathit{dom}(h)$. $S(h(l))$ is defined and that $\mathit{dom}(h) \cap \mathit{dom}(\mathit{left}(S\langle \phi, \mathcal{Z} \rangle)) = \phi$. Since $h_2 \subseteq h$, we can assume that $S\langle h_2, \mathcal{Z} \rangle$ is defined as well. Thus, we can assume inductively that $\vdash R' | C', A' \supset \langle \phi, E \rangle: S\langle h_2, \mathcal{Z} \rangle$, and $\forall l \in \mathit{dom}(f)$. $\vdash R' | C', A' \supset f(l): S(h_1(l))$. It remains to show that the antecedents needed to derive $\vdash R' | C', A' \supset \langle f, E \rangle: S\langle h, \mathcal{Z} \rangle$ by (*rec3*) are true.

We first note that $S\langle h_2, \mathcal{Z} \rangle = \langle h_2; S + \mathit{left}(S\langle \phi, \mathcal{Z} \rangle), \mathit{right}(S\langle \phi, \mathcal{Z} \rangle) \rangle$. Since $+$ is associative, we have that

$$S\langle h, \mathcal{Z} \rangle == \langle (h_1 + h_2); S + \mathit{left}(S\langle \phi, \mathcal{Z} \rangle), \mathit{right}(S\langle \phi, \mathcal{Z} \rangle) \rangle$$

which is equal to

$$\langle h_1; S + (h_2; S + \mathit{left}(S\langle \phi, \mathcal{Z} \rangle)), \mathit{right}(S\langle \phi, \mathcal{Z} \rangle) \rangle.$$

Thus, we can infer by (*rec3*) that $\vdash R' | C', A' \supset \langle f, E \rangle: \sigma'$, which proves the theorem. \blacksquare

3.7 Most General Typings

In this section, we discuss the notion of most general typings in the context of the R -typing system. We extend the definition of most general typings from [Mit84] a bit and say that a restricted typing $R|C, A \supset M:\sigma$ is a *most general restricted typing* for M iff the set of its instances is exactly the set of provable restricted typings for M .

It is worth noting here that most general restricted typings in our system are essentially unique in the sense that they form a natural equivalence class. This is also true of ML, where the equivalence relation is that all most general typings for a given expression are unique up to variable renaming. Interestingly, in ML, two typing statements are unique up to variable renaming iff they are instances of each other. With the introduction of subtyping assertions, however, the notions of instance and variable renaming are no longer equivalent, and it turns out that the correct equivalence relation for this system is that all most general restricted typings for a given expression are instances of one another.

For example,

$$\phi | \{s \subseteq t\}, \emptyset \supset \mathbf{fn} \ x \Rightarrow x : s \rightarrow t$$

and

$$\phi | \{a \subseteq b, c \subseteq d\}, \emptyset \supset \mathbf{fn} \ x \Rightarrow x : c \rightarrow d$$

are both most general restricted typings of the identity function, since every typing of $\mathbf{fn} \ x \Rightarrow x$ is an instance of both of them. These restricted typings are not unique up to variable renaming, but they are instances of one another.

It is useful to point out here that the presence of an equivalence class of most general restricted typings versus a single most general restricted typing does not affect the decidability of the type inference problem. As we will prove in the next chapter, the algorithm, given a typable expression in the language, yields a most general typing for that expression. It *does not matter* which most general restricted typing is yielded, since it is possible to check whether one restricted typing is an instance of another.

Lemma 3.7.1 *For any restricted typing $R|C, A \supset M:\sigma$, the set of instances of $R|C, A \supset M:\sigma$ is recursive.*

We omit the proof here, stating only that the obvious algorithm takes time exponential in the size of the restricted typing. The difficulty in this algorithm seems to lie in checking the instance conditions on the R and C , and we haven't as yet had a chance to determine whether a polynomial time algorithm exists. We do know that if we assume that all expressions in ML^+ are renamed (*i.e.*, alpha-converted) so that all lambda-bound variables and extension variables in the expression are distinct, and we associate the corresponding expressions with the cut types in R , then checking the instance condition on this augmented set R can be done in polynomial time. It is unclear to us at the present time whether the condition on C can be checked in polynomial time; however, we haven't as yet had a chance to think about this issue in much detail.

We note here that, as a consequence of Lemma 3.7.1, the decision problem for determining whether a restricted typing for an expression M is provable is reducible to the problem of computing a most general typings for M .

3.8 Relating the R -Typing System to the Original System

It may seem peculiar to the reader that we first developed a typing system that captured the form of subtyping discussed in Chapter 1, proceeded to discuss the technical difficulties in the typing system, and then modified the typing system so that it has the technical properties that we desire. The obvious question arises as to which typing system – the original typing system or the R -typing system – is the “real” typing system that defines ML^+ ?

It turns out that the original typing system and the R -typing system are, in fact, intimately related. More precisely,

Lemma 3.8.1 *A typing $C, A \supset M : \sigma$ is provable in the original typing system iff there exists a set R such that $R | C, A \supset M : \sigma$ is provable in the R -typing system.*

Proof. To prove one direction, suppose that $\vdash C, A \supset M : \sigma$, and let R be the set of “cut” types in the derivation of $C, A \supset M : \sigma$. Then there is a derivation of $R | C, A \supset M : \sigma$ that uses the sequence of typing rules in the R -typing system that correspond exactly to the sequence of typing rules used in the derivation of $C, A \supset M : \sigma$.

To prove the other direction, suppose that $\vdash R | C, A \supset M : \sigma$ for some set R . Then $\vdash C, A \supset M : \sigma$ by a derivation that uses the sequence of typing rules in the original typing system that correspond exactly to the sequence of typing rules used in the derivation of $\vdash R | C, A \supset M : \sigma$. ■

Furthermore, we can state the following property of the original typing system that is very similar to Theorem 3.6.7.

Lemma 3.8.2 *Suppose $\vdash C, A \supset M : \sigma$. If $C', A' \supset M : \sigma'$ is an instance of $C, A \supset M : \sigma$ by a substitution that is defined on the set of cut types in the derivation of $C, A \supset M : \sigma$, then $\vdash C', A' \supset M : \sigma'$.*

The proof is very similar to that of Theorem 3.6.7.

Using this restricted notion of instance, we could formulate a definition of most general typings for the original typing system that “implicitly” referred to the set of cut types in the derivation, and proceed to show that the algorithm is “equivalent” to the original typing system. However, the R -typing system provides a nice syntactic mechanism to explicitly keep track of these cut types, and therefore we will prove in Chapter 4 that the algorithm is “equivalent” to the R -typing system. However, using the above lemmas as justification, we consider the *original* typing system to be the typing system that defines ML^+ .

Chapter 4

The Type Inference Algorithm

4.1 Overview

In this chapter, we present the type inference algorithm, GE, for ML^+ . We develop Theorems 4.5.3 and 4.6.4, the main theorems of this thesis, which show that, for every expression M of ML^+ , GE derives a most general restricted typing statement for M if M is typable in the R -typing system of Chapter 3 and “fails” otherwise. More specifically, Theorem 4.5.3 shows that GE is *sound* with respect to the R -typing system; that is, for any expression M , if the algorithm succeeds with a restricted typing statement, then every instance of the typing is provable. Conversely, Theorem 4.6.4 shows that GE is *complete* with respect to the R -typing system; that is, for any expression M , if there is a provable restricted typing for M , then the algorithm will succeed with a restricted typing statement of which the provable typing is an instance.

This chapter is organized as follows. Sections 4.2 and 4.3 develop two algorithms, UNIFY and MATCH, respectively, that form the foundation of the type inference algorithm. The algorithm UNIFY for *unification* is used to combine most general restricted typings of subexpressions, while the algorithm MATCH for *matching* is used to guarantee that the set of subtype assertions in a most general restricted typing is atomic. The type inference algorithm, GE, is presented in Section 4.4, while Sections 4.5 and 4.6 prove that GE is sound and complete with respect to the R -typing system. Finally, Section 4.7 relates the type inference algorithm to the original typing system.

4.2 Unification

As in ML, we will use unification to combine (restricted) typing statements about subexpressions in the process of inferring a typing for an expression in ML^+ . The standard notion of unification is to produce a substitution that makes a pair of type expressions syntactically identical. However, because the first component of our record types is a function, we need to use a slightly modified definition of syntactic identity. To this end, we define *syntactic likeness* between type expressions as follows:

- b_1 is syntactically like b_2 if b_1 and b_2 are the same ground constant
- q_1 is syntactically like q_2 if q_1 and q_2 are the same built-in constant
- x is syntactically like x

- $\sigma \rightarrow \tau$ is syntactically like $\sigma' \rightarrow \tau'$ if σ is syntactically like σ' and τ is syntactically like τ'
- $\langle h, \mathcal{Z} \rangle$ is syntactically like $\langle h', \mathcal{Z}' \rangle$ if $\text{dom}(h) = \text{dom}(h')$ and $\forall l \in \text{dom}(h). h(l)$ is syntactically like $h'(l)$

We say that a substitution S *unifies* a set E of equations between type expressions, or equivalently, that S is a *unifying substitution for E* , if, for all equations $\sigma = \tau \in E$, S is defined on σ and τ , and $S\sigma$ is syntactically like $S\tau$.

In the process of computing most general typings for expressions in ML^+ , we would like to compute “most general unifiers” in order to combine the most general typings of subexpressions. Following [Rob65], we would like to define a substitution S that unifies a set E of equations between type expressions as a *most general unifier for E* if S unifies E and, for all substitutions T that unify E , there exists a substitution U such that $S;U$ is defined and $T = S;U$. However, due to extended record types, it turns out that we will need to modify this definition somewhat.

We illustrate the difficulty with the standard definition through an example. Consider the equation

$$\{a: \mathbf{int}; \mathcal{X}\} = \{b: \mathbf{bool}; \mathcal{X}'\}$$

Clearly, any substitution that unifies this equation must map \mathcal{X}' to a record type that contains at least an $a: \mathbf{int}$ field and must map \mathcal{X} to a record type that contains at least a $b: \mathbf{bool}$ field. Furthermore, the “rest” of \mathcal{X} and \mathcal{X}' must be identical. Thus, the most general substitution S that unifies this equation is

$$S = [\{b: \mathbf{bool}; \mathcal{Y}\}/\mathcal{X}, \{a: \mathbf{int}; \mathcal{Y}\}/\mathcal{X}'],$$

for some “fresh” \mathcal{Y} , where \mathcal{Y} is a row variable that does not occur anywhere in the set of equations that we are unifying.

It turns out that the difficulty with the standard definition lies precisely in the introduction of such fresh row variables. Consider the substitution

$$T = [\{b: \mathbf{bool}; \mathit{NULL}\}/\mathcal{X}, \{a: \mathbf{int}; \mathit{NULL}\}/\mathcal{X}', \{a: \mathbf{real}; \mathit{NULL}\}/\mathcal{Y}]$$

Clearly, T unifies the above equation. However, since any substitution U such that $T = S;U$ must map \mathcal{Y} to $\{\mathit{NULL}\}$, the action of $S;U$ on \mathcal{Y} differs from that of T . Therefore, there is no such substitution U , and thus, S cannot be the most general unifier for E .

In order to overcome this difficulty, we introduce a “restricted” form of equality over substitutions. Let \mathcal{Q} be a set of type variables and row variables. We say that S is *equal to T with respect to \mathcal{Q}* , written $S =_{\mathcal{Q}} T$, if $\forall t \in \mathcal{Q}. St = Tt$ and $\forall \mathcal{X} \in \mathcal{Q}. S\mathcal{X} = T\mathcal{X}$. We also are more precise in our notion of “fresh” variables, and say that a row variable \mathcal{X} is *fresh with respect to \mathcal{Q}* if $\mathcal{X} \notin \mathcal{Q}$. An analogous definition, which we will need in the next section, holds for type variables.

Using these definitions, we restrict our notion of most general unifiers as follows. Let E be any set of equations between type expressions, and let \mathcal{Q} again be any set of type variables and row variables. We say that a substitution S is a *most general unifying substitution for E with respect to \mathcal{Q}* if S unifies E and, for all substitutions T that unify E , there exists a substitution U such that $S;U$ is defined and $T =_{\mathcal{Q} \cup \mathcal{Q}_E} S;U$, where \mathcal{Q}_E is the set of type and row variables occurring in E . The idea behind this restriction is that, while computing S , we choose the “fresh” row variables to be fresh with respect to $\mathcal{Q} \cup \mathcal{Q}_E$. If we ensure

that \mathcal{Q} is a co-infinite set, then we will always be able to choose a “fresh” row variable, thus getting the behavior we desire. (Recall that we assume an infinite set of type variables and row variables in our system.)

Using these ideas, our algorithm UNIFY takes a set of equations E between type expressions and a co-infinite set \mathcal{Q} of type variables and row variables and produces a most general unifying substitution for E with respect to $\mathcal{Q} \cup \mathcal{Q}_E$, where \mathcal{Q}_E is the set of type variables and row variables that occur in E . If no unifying substitution for E exists, then UNIFY fails. UNIFY is a modification of Robinson’s unification [Rob65], extended to solve equations between extended record types, and is quite similar to the unification algorithm described in [Wan87]. However, it corrects a bug in Wand’s algorithm for unifying two extended record types; we postpone discussion of this bug to the proof of Lemma 4.2.1.

Quite importantly, another subtle bug in Wand’s algorithm is avoided by the syntactic restriction in ML^+ prohibiting duplicate field names within a record. Because the language of [Wan87] does not make this syntactic restriction, the algorithm given in [Wan87] for equations between row expressions is *incorrect* for his system; however, it *is correct* for our system. His corrected algorithm, which appears in [Wand88], needs to compute a *set* of most general unifiers, while our algorithm computes a *single* most general unifier.

Lemma 4.2.1 *Let E be a set of equations of the form $\sigma = \tau$ and let \mathcal{Q} be a co-infinite set of type variables and row variables. Let \mathcal{Q}_E be the set of type variables and row variables that occur in E . There exists an algorithm UNIFY such that whenever there is a unifying substitution for E , $UNIFY(E, \mathcal{Q})$ produces a most general unifying substitution with respect to $\mathcal{Q} \cup \mathcal{Q}_E$. Otherwise, $UNIFY(E, \mathcal{Q})$ fails.*

The proof appears in the Appendix.

4.3 Matching

In the process of computing a most general restricted typing for an expression M , we will use unification to combine the most general restricted typings of the subexpressions of M . However, the most general unifier may not respect the sets of subtyping assertions in these typing statements, thus violating our requirement that typing statements contain only atomic subtype assertions. Therefore, we need to be able to compute a “most general matching substitution” for this set of possibly non-matching subtype assertions, so that we can apply the operation \bullet developed in Chapter 3 to get a “most general” set of atomic subtype assertions.

We would like to follow the standard notion of most general substitutions and say that a substitution S is a *most general matching substitution* for C if S is a matching substitution for C and, for all matching substitutions T for C , there exists a substitution U such that $S;U$ is defined and $T = S;U$. However, a problem arises due to fresh type and row variables which is similar to the one of Section 4.2, and so we restrict our notion of “most general” in a similar manner. Let C be a set of possibly non-matching subtype assertions and let \mathcal{Q} be a set of type variables and row variables. We say that a substitution S is a *most general matching substitution for C with respect to \mathcal{Q}* if S is a matching substitution for C and, for all matching substitutions T for C , there exists a substitution U such that $S;U$ is defined and $T =_{\mathcal{Q} \cup \mathcal{Q}_C} S;U$, where \mathcal{Q}_C is the set of type and row variables occurring in C . Again, the idea behind this restriction is that the fresh type and row variables are chosen from the complement of $\mathcal{Q} \cup \mathcal{Q}_C$.

An extremely useful property of matching is that the problem of finding a most general matching substitution for a set C of possibly non-matching subtype assertions can be reduced to the problem of finding a most general unifying substitution for a set E that is closely related to the set C . The basic idea here is that we treat the set C as a set of equations between types (rather than inequalities) and unify this set of equations. There is a minor complication, however, due to base types, since two distinct base types match but do not unify. In order to get around this difficulty, we replace all base types (and NULL) in C by special type variables (and row variable), and check that the most general unifier merely renames these special type variables and row variable. We then derive the most general matching substitution from this most general unifier S by “factoring” S into a “most general” substitution S_1 composed with a “simple” substitution S_2 such that all type variables and row variables occurring in the range of S_1 are distinct and such that S_2 merely renames type variables and row variables. If we incorporate the set \mathcal{Q} appropriately into the above procedures, we can use the properties of most general unifiers to show that the substitution S_1 is the most general matching substitution for C with respect to \mathcal{Q} .

Using these ideas, our algorithm MATCH takes a set C of possibly non-matching subtype assertions and a co-infinite set \mathcal{Q} of type variables and row variables and produces a most general matching substitution for C with respect to $\mathcal{Q} \cup \mathcal{Q}_C$, where \mathcal{Q}_C is the set of type variables and row variables occurring in C . If no matching substitution for C exists, then MATCH fails. Our algorithm is quite similar to the algorithm of [Mit84], extended to support base types and record types.

This section is organized as follows. We will first show in more detail how the problem of finding matching substitutions reduces to the problem of finding unifying substitutions. Being careful about the set \mathcal{Q} , we will then precisely define what it means for all type and row variables in the range of a substitution to be distinct, and then show how to “factor” *any* substitution S into two substitutions S_1 and S_2 with the properties outlined above. Finally, using these results, we will present the actual algorithm MATCH and give the proof of its correctness.

In order to relate matching and unification, we first need to define precisely what it means for a substitution to merely rename type variables and row variables. Formally, we say that a substitution S is *simple* if, for all $t \in \text{dom}(S)$, there exists a type variable t' such that $St = t'$, and for all $\mathcal{X} \in \text{dom}(S)$, there exists a row variable \mathcal{X}' such that $S\mathcal{X} = \langle \phi, \mathcal{X}' \rangle$. Similarly, a *c-simple* substitution corresponds to either renaming type variables and row variables or replacing them by base types or NULL, respectively. Formally, we say that a substitution S is *c-simple* if, for all $t \in \text{dom}(S)$, there exists either a type variable t' such that $St = t'$ or a base type c such that $St = c$, and for all $\mathcal{X} \in \text{dom}(S)$, either there exists a row variable \mathcal{X}' such that $S\mathcal{X} = \langle \phi, \mathcal{X}' \rangle$ or $S\mathcal{X} = \langle \phi, \text{NULL} \rangle$. It is easy to show that if T is a simple substitution or a *c-simple* substitution, then, for any substitution S , $S;T$ is defined.

Just to keep our definitions explicit, we formally define the *range* of any substitution S to be the set of type variables and row variables that occur in $S[s]$ or $S[\mathcal{Y}]$ for some $s, \mathcal{Y} \in \text{dom}(S)$.

We now show precisely how the problem of finding a matching substitution for a set C (with respect to a set \mathcal{Q}) reduces to the problem of finding a unifying substitution for a related set E (with respect to \mathcal{Q}).

Lemma 4.3.1 *Let C be a set of subtype assertions $\sigma \subseteq \tau$, where σ and τ may not necessarily match, and let \mathcal{Q} be a finite set of type variables and row variables. Let \mathcal{Q}_C be the set of*

type variables and row variables that occur in C , and let t_{int} , t_{real} , t_{bool} , t_{string} , and \mathcal{X}_{NULL} be fresh with respect to $\mathcal{Q} \cup \mathcal{Q}_C$. Let C' be the set C with all occurrences of **int** replaced with t_{int} , all occurrences of **real** replaced with t_{real} , all occurrences of **bool** replaced with t_{bool} , all occurrences of **string** replaced with t_{string} and all occurrences of **NULL** replaced with \mathcal{X}_{NULL} . Let $E = \{\sigma' = \tau' \mid \sigma' \subseteq \tau' \in C'\}$.

T is a matching substitution for C iff there exist substitutions T_1 , T_2 , and T_3 , such that T_2 is c -simple, T_3 is simple,

$$T =_{\mathcal{Q} \cup \mathcal{Q}_C} T_1; T_2, \text{ and } T_1'; T_3 \text{ unifies } E,$$

where $T_1' = T_1 \upharpoonright (\mathcal{Q} \cup \mathcal{Q}_C)$,

and, for all $i \in \{\mathbf{int}, \mathbf{real}, \mathbf{bool}, \mathbf{string}\}$,

1. $(T_1'; T_3)t_i \neq \sigma \rightarrow \tau$, for any σ, τ

2. $(T_1'; T_3)t_i \neq \langle h, \mathcal{Z} \rangle$, for any h, \mathcal{Z}

and $(T_1'; T_3)\mathcal{X}_{NULL} \neq \langle h, \mathcal{Z} \rangle$, for any $h \neq \phi$.

Proof. To prove one direction, suppose that T is a matching substitution for C . Let $\mathcal{Q}_{C'} = \mathcal{Q}_C \cup \{t_i \mid i \in \{\mathbf{int}, \mathbf{real}, \mathbf{bool}, \mathbf{string}\}\} \cup \{\mathcal{X}_{NULL}\}$. Let s_{int} , s_{real} , s_{bool} , s_{string} , and \mathcal{Y}_{NULL} be fresh with respect to $\mathcal{Q} \cup \mathcal{Q}_{C'}$.

Let T_1 be a substitution whose domain is $(\mathcal{Q} \cup \mathcal{Q}_C)$. Furthermore, T_1 maps all type variables t in its domain to Tt with all occurrences of **int** replaced with s_{int} , all occurrences of **real** replaced with s_{real} , all occurrences of **bool** replaced with s_{bool} , all occurrences of **string** replaced with s_{string} and all occurrences of **NULL** replaced with \mathcal{Y}_{NULL} . (The action of T_1 on row variables in its domain is analogous.)

Let $T_2 = [\mathbf{int}/s_{int}, \mathbf{real}/s_{real}, \mathbf{bool}/s_{bool}, \mathbf{string}/s_{string}, \langle \phi, \mathbf{NULL} \rangle / \mathcal{Y}_{NULL}]$. Clearly, $T_1; T_2$ is defined, and

$$T =_{\mathcal{Q} \cup \mathcal{Q}_C} T_1; T_2.$$

It is easy to see that $T_1' = T_1 \upharpoonright (\mathcal{Q} \cup \mathcal{Q}_C) = T_1$. Since T respects C and T_2 is c -simple, it follows that T_1 respects C as well.

Let T_3 be the substitution whose domain is $\mathcal{Q} \cup \mathcal{Q}_{C'} \cup \text{range}(T_1)$ and such that T_3 maps all type variables in its domain to some type variable t' and maps all row variables in its domain to $\langle \phi, \mathcal{X}' \rangle$, for some row variable \mathcal{X}' . Clearly, T_3 is simple, and so $T_1; T_3$ is defined. Furthermore, since the t_i and \mathcal{X}_{NULL} are fresh with respect to $\mathcal{Q} \cup \mathcal{Q}_C$, we have that $T_1 t_i = t_i$ for all the t_i , and that $T_1 \mathcal{X}_{NULL} = \langle \phi, \mathcal{X}_{NULL} \rangle$. Thus, $(T_1; T_3)t_i = t'$ and $(T_1; T_3)\mathcal{X}_{NULL} = \langle \phi, \mathcal{X}' \rangle$, satisfying conditions (1), (2) and (3) above.

Since $T_1 C$ is matching, we have that for all $\sigma \subseteq \tau \in C$, $T_1 \sigma$ and $T_1 \tau$ differ only in the names of type variables and/or ground types and in the names on row variables and/or **NULL**. Thus, by the construction of T_1 and C' , we have that for all $\sigma' \subseteq \tau' \in C'$, $T_1 \sigma'$ and $T_1 \tau'$ differ only in the names of type variables and in the names of row variables. Thus, $T_1; T_3$ unifies E , proving one direction.

To prove the converse, we note that T_3 is simple implies that for all $\sigma' \subseteq \tau' \in C'$, $T_1' \sigma'$ matches $T_1' \tau'$. Since $T_1' t_i = t_i$ for all the t_i , and $T_1' \mathcal{X}_{NULL} = \langle \phi, \mathcal{X}_{NULL} \rangle$, it follows that T_1' is a matching substitution for C . Furthermore, by the definition of T_1' and \mathcal{Q}_C , it follows that T_1 is a matching substitution for C .

Since T_2 is c -simple, $T_1; T_2$ is also a matching substitution for C . Then, clearly, so is $(T_1; T_2) \upharpoonright (\mathcal{Q} \cup \mathcal{Q}_C)$, and thus so is $T \upharpoonright (\mathcal{Q} \cup \mathcal{Q}_C)$. By the definition of \mathcal{Q}_C , T is a matching substitution for C , proving the lemma. \blacksquare

In order to show how to “factor” substitutions, we state precisely what it means for all the type and row variables occurring in the range of the most general unifier to be distinct. Since we do not want these type and row variables to conflict with the set \mathcal{Q} in our notion of most general matching substitutions, we incorporate \mathcal{Q} into our definition.

Definition 4.3.2 *A substitution S chooses variables freely on a set \mathcal{Q} of type variables and row variables if no type variable or row variable in \mathcal{Q} occurs in $\text{range}(S)$ and*

- *For each $t \in \mathcal{Q}$, no type variable or row variable appears twice in St .*
- *For each $\mathcal{X} \in \mathcal{Q}$, no type variable or row variable appears twice in $S\mathcal{X}$.*
- *For any distinct $s, t \in \mathcal{Q}$, no type variable or row variable in Ss appears in St .*
- *For any distinct $\mathcal{X}, \mathcal{Y} \in \mathcal{Q}$, no type variable or row variable in $S\mathcal{X}$ appears in $S\mathcal{Y}$.*
- *For any $t, \mathcal{X} \in \mathcal{Q}$, no type variable or row variable in St appears in $S\mathcal{X}$.*

In the proof of correctness for the algorithm MATCH, we will need to use the property that choosing variables freely is preserved under composition of appropriately defined substitutions.

Lemma 4.3.3 *Suppose that a substitution S chooses variables freely on a set \mathcal{Q} of type variables and row variables, and that a substitution T chooses variables freely on the set of all type variables and row variables occurring in all St or $S\mathcal{X}$ with $t \in \mathcal{Q}$, $\mathcal{X} \in \mathcal{Q}$. If $S;T$ is defined, then $S;T$ chooses variables freely on \mathcal{Q} .*

The proof is straightforward and we omit it.

As alluded to previously, we will need to be able to “factor” a substitution S into a substitution S_1 that chooses variables freely on a set \mathcal{Q} of type and row variables composed with a simple substitution S_2 . To prove this property, we need to develop a linear ordering on the “components” of a type. First of all, we consider a record type $\langle h, \mathcal{Z} \rangle$ to be written out as $h(l_1), h(l_2), \dots, h(l_n), \mathcal{Z}$, where $\text{dom}(h) = \bigcup_{1 \leq i \leq n}$ and the l_i are in lexicographic order. (The other types are written out in the expected manner.) Assuming this order, we can unambiguously refer to the m -th type variable occurring in a type, or similarly, the m -th row variable occurring in a type. Assuming that each occurrence of a type variable, row variable, base type, and NULL determines a “position” in a type, we can also unambiguously refer to the k -th position in a type.

Lemma 4.3.4 *Let \mathcal{Q}^t be a finite set of type variables and \mathcal{Q}^r be a finite set of row variables, and let $\mathcal{Q} = \mathcal{Q}^t \cup \mathcal{Q}^r$. For any substitution S , there are substitutions S_1 and S_2 such that S_1 and S_2 are computable, S_1 chooses variables freely on \mathcal{Q} , S_2 is simple, and $S =_{\mathcal{Q}} S_1; S_2$.*

Proof. The proof is analogous to that of [Mit84]. To define S_1 and S_2 , let t_1, t_2, \dots be an enumeration of \mathcal{Q}^t and let $\mathcal{X}_1, \mathcal{X}_2, \dots$ be an enumeration of \mathcal{Q}^r , and let us partition the complement of \mathcal{Q}^t into disjoint sets $\mathcal{U}_1, \mathcal{U}_2, \dots$ and partition the complement of \mathcal{Q}^r into disjoint sets $\mathcal{W}_1, \mathcal{W}_2, \dots$. It will be convenient to choose some enumeration of each \mathcal{U}_i , say $\mathcal{U}_i = \{t_{i,1}, t_{i,2}, \dots\}$ and some enumeration of each \mathcal{W}_j , say $\mathcal{W}_j = \{\mathcal{X}_{j,1}, \mathcal{X}_{j,2}, \dots\}$.

For each $t_i \in \mathcal{Q}^t$, let $S_1 t_i$ be the type expression derived by replacing the m -th type variable occurrence in St_i with the m -th type variable $t_{2i,m}$ from \mathcal{U}_{2i} and by replacing the

n -th row variable occurrence in St_i with the n -th row variable $\mathcal{X}_{2i,n}$ from \mathcal{W}_{2i} . For each $\mathcal{X}_j \in \mathcal{Q}^r$, let $S_1\mathcal{X}_j$ be the type expression derived by replacing the m -th type variable occurrence in $S\mathcal{X}_j$ with the m -th type variable $t_{2j+1,m}$ from \mathcal{U}_{2j+1} and by replacing the n -th row variable occurrence in $S\mathcal{X}_j$ with the n -th row variable $\mathcal{X}_{2j+1,n}$ from \mathcal{W}_{2j+1} . Let S_2 map $t_{2i,m}$ back to the m -th type variable in St_i , $t_{2j+1,m}$ back to the m -th type variable in $S\mathcal{X}_j$, $\mathcal{X}_{2i,n}$ back to the n -th type variable in St_i , and $\mathcal{X}_{2j+1,n}$ back to the n -th type variable in $S\mathcal{X}_j$. Clearly, S_2 is simple, and thus $S_1;S_2$ is defined. It is easily verified that S_1 chooses variables freely on \mathcal{Q} and that $S =_{\mathcal{Q}} S_1;S_2$. Moreover, all type expressions are of finite length, the complements of \mathcal{Q}^t and \mathcal{Q}^r do not need to be fully partitioned or fully enumerated. Thus, S_1 and S_2 are computable by the procedure outlined above. ■

To show that the substitution that our MATCH algorithm computes is the most general matching substitution, we will need to use the fact that any substitution S_1 with the properties outlined above is “most general” in the following precise sense.

Lemma 4.3.5 *Let \mathcal{Q} be a finite set of type variables and row variables. Furthermore, let S be any substitution, and let S_1 and S_2 be substitutions such that S_1 chooses variables freely on \mathcal{Q} , S_2 is simple, and $S =_{\mathcal{Q}} S_1;S_2$. If $S =_{\mathcal{Q}} T_1;T_2$ for some simple substitution T_2 , then there exists a substitution W such that $S_1;W$ is defined and $T_1 =_{\mathcal{Q}} S_1;W$.*

Proof. Since S_2 and T_2 are simple, it follows that, for any $t \in \mathcal{Q}$, the type expressions St , S_1t , and T_1t must match. Similarly, for any $\mathcal{X} \in \mathcal{Q}$, the type expressions $S\mathcal{X}$, $S_1\mathcal{X}$, and $T_1\mathcal{X}$ must match.

For any $t \in \mathcal{Q}$, consider the k -th position of S_1t and the k -th position T_1t . By the definition of “simple”, it follows that either both positions contain type variables, both positions contain row variables, or both positions contain the *same* ground type. In the first case, since S_1 chooses variables freely on \mathcal{Q} , there is a well-defined substitution W mapping the type variable occurring in the k -th position of S_1t to the type variable occurring in the k -th position of T_1t . Similarly, in the second case, since S_1 chooses variables freely on \mathcal{Q} , there is a well-defined substitution W mapping the row variable occurring in the k -th position of S_1t to (ϕ, \mathcal{Y}) , where \mathcal{Y} is the row variable occurring in the k -th position of T_1t . (An analogous statement holds for any $\mathcal{X} \in \mathcal{Q}$.)

Clearly, W is simple, and thus $S_1;W$ is defined. It is to verify that $T_1 =_{\mathcal{Q}} S_1;W$, proving the lemma. ■

Having developed the necessary technical machinery, we now prove the main lemma of this section; namely, that there exists an algorithm MATCH that, given a set C of possibly non-matching subtype assertions and a set \mathcal{Q} of type and row variables, computes a most general substitution for C with respect to \mathcal{Q} . We also prove that the domain of $\text{MATCH}(C, \mathcal{Q})$ is subset of $\mathcal{Q} \cup \mathcal{Q}_C$, where \mathcal{Q}_C is the set of type variables and row variables that occur in C , since, for technical reasons, we will need to use this fact in our proof of completeness of GE in Section 4.6.4.

Lemma 4.3.6 *Let C be a set of subtype assertions $\sigma \subseteq \tau$, where σ and τ may not necessarily match, and let \mathcal{Q} be a finite set of type variables and row variables. Let \mathcal{Q}_C be the set of type variables and row variables that occur in C . There is an algorithm MATCH such that if there is a matching substitution for C , $\text{MATCH}(C, \mathcal{Q})$ produces a most general matching substitution with respect to $\mathcal{Q} \cup \mathcal{Q}_C$. Otherwise, $\text{MATCH}(C, \mathcal{Q})$ fails.*

Furthermore, if $\text{MATCH}(C, \mathcal{Q})$ succeeds, then $\text{dom}(\text{MATCH}(C, \mathcal{Q})) \subseteq (\mathcal{Q} \cup \mathcal{Q}_C)$.

Proof. The algorithm is as follows:

$\text{MATCH}(C, \mathcal{Q}) =$

let \mathcal{Q}_C be the set of type variables and row variables that occur in C
 $t_{int}, t_{real}, t_{bool}, t_{string}$, and \mathcal{X}_{NULL} be “fresh” with respect to $\mathcal{Q} \cup \mathcal{Q}_C$
 C' be the set C with
 all occurrences of **int** replaced with t_{int} ,
 all occurrences of **real** replaced with t_{real} ,
 all occurrences of **bool** replaced with t_{bool} ,
 all occurrences of **string** replaced with t_{string} , and
 all occurrences of **NULL** replaced with \mathcal{X}_{NULL}
 $E = \{\sigma' = \tau' \mid \sigma' \subseteq \tau' \in C'\}$
 $\mathcal{Q}_{C'} = \mathcal{Q}_C \cup \{t_i \mid i \in \{\mathbf{int}, \mathbf{real}, \mathbf{bool}, \mathbf{string}\}\} \cup \{\mathcal{X}_{NULL}\}$
 $S = \text{UNIFY}(E, \mathcal{Q} \cup \mathcal{Q}_{C'})$
 if, for any t_i such that $i \in \{\mathbf{int}, \mathbf{real}, \mathbf{bool}, \mathbf{string}\}$,
 $S t_i = \sigma \rightarrow \tau$ for any σ, τ or
 $S t_i = \langle h, \mathcal{Z} \rangle$ for any h, \mathcal{Z} or
 $S \mathcal{X}_{NULL} = \langle h, \mathcal{Z} \rangle$ for any $h \neq \phi$
 then *fail*
 else let S_1 and S_2 be the substitutions such that
 $S =_{\mathcal{Q} \cup \mathcal{Q}_{C'}} S_1; S_2$
 and S_1 chooses variables freely on $\mathcal{Q} \cup \mathcal{Q}_{C'}$
 and S_2 is simple
 $S_1' = S_1 \upharpoonright (\mathcal{Q} \cup \mathcal{Q}_C)$
 return S_1'

We first prove that if $\text{MATCH}(C, \mathcal{Q})$ succeeds, then there is a matching substitution for C . More specifically, we show that $\text{MATCH}(C, \mathcal{Q})$ produces a matching substitution for C . Suppose that $\text{MATCH}(C, \mathcal{Q})$ succeeds. By Lemma 4.2.1, S is the most general unifier for E with respect to $\mathcal{Q} \cup \mathcal{Q}_{C'}$. Furthermore, S maps all the t_i to types that are not function types or record types and maps \mathcal{X}_{NULL} to a record with ϕ as its first component. Since E does not contain $NULL$ or any base types and S is a most general unifier for E (w.r.t to $\mathcal{Q} \cup \mathcal{Q}_{C'}$), we have that for all $\sigma' = \tau' \in E$, both $S\sigma'$ and $S\tau'$ do not contain $NULL$ or any base types.

Let $S_1'' = S_1 \upharpoonright (\mathcal{Q} \cup \mathcal{Q}_{C'})$. Since S_2 is simple, and by the properties of S mentioned above, it follows that for all the t_i , there exists a type variable s_i such that $S_1'' t_i = s_i$. Also, by the same reasoning, $S_1'' \mathcal{X}_{NULL} = \langle \phi, \mathcal{X}' \rangle$, for some \mathcal{X}' . Let

$$S_3 = [s_{int}/t_{int}, s_{real}/t_{real}, s_{bool}/t_{bool}, s_{string}/t_{string}, \langle \phi, \mathcal{X}' \rangle / \mathcal{X}_{NULL}]$$

Since S_1 chooses variables freely on $\mathcal{Q} \cup \mathcal{Q}_C$, we have that

$$S_1'; S_3 =_{\mathcal{Q} \cup \mathcal{Q}_{C'}} S_1''$$

Thus, since E contains only type and row variables from \mathcal{Q}_C , we have that $S_1'; S_3; S_2$ unifies E . Noting that $S_3; S_2$ is simple, we can use Lemma 4.3.1 to prove that S is a matching substitution for C . Then, clearly, S_1' must be a matching substitution for C .

To prove the other direction, suppose that there is a matching substitution T for C . By Lemma 4.3.1, there exist substitutions T_1, T_2 , and T_3 such that $T =_{\mathcal{Q} \cup \mathcal{Q}_C} T_1; T_2$, T_2 is c -simple, T_3 is simple, and such that $T_1'; T_3$ unifies E , where $T_1' = T_1 \upharpoonright (\mathcal{Q} \cup \mathcal{Q}_C)$. By

Lemma 4.2.1, we can conclude that $S = \text{UNIFY}(E, \mathcal{Q} \cup \mathcal{Q}_{C'})$ succeeds, and that there exists a substitution U such that $S;U$ is defined and such that $S;U =_{\mathcal{Q} \cup \mathcal{Q}_{C'}} T_1';T_3$. Furthermore, by Lemma 4.3.1, $T_1';T_3$ maps all the t_i to types that are not function types or record types and maps \mathcal{X}_{NULL} to a record with ϕ as its first component. Thus, this property must also hold true for S . By Lemma 4.3.4, it follows that S_1 and S_2 exist and are computable; thus, $\text{MATCH}(C, \mathcal{Q})$ succeeds. By the proof above, S_1' is a matching substitution for C .

It remains to show that there is a substitution W such that $S_1';W$ is defined and $T =_{\mathcal{Q} \cup \mathcal{Q}_C} S_1';W$. We first note that $S =_{\mathcal{Q} \cup \mathcal{Q}_C} S_1';S_2$. We let $S' = S \upharpoonright (\mathcal{Q} \cup \mathcal{Q}_C)$, and let $U' = U \upharpoonright (\mathcal{Q} \cup \mathcal{Q}_C \cup \text{range}(S'))$. Since $\mathcal{Q}_C \subseteq \mathcal{Q}_{C'}$, we have that $S_1';S_2;U'$ is defined and that

$$S;U' =_{\mathcal{Q} \cup \mathcal{Q}_C} S_1';S_2;U' =_{\mathcal{Q} \cup \mathcal{Q}_C} T_1';T_3$$

By Lemma 4.3.4, we can “factor” $S_2;U'$ into $U_1;U_2$ such that

$$S_2;U' =_{\mathcal{Q} \cup \mathcal{Q}_C \cup \text{range}(S_1')} U_1;U_2$$

and such that U_1 chooses variables freely on $\mathcal{Q} \cup \mathcal{Q}_C \cup \text{range}(S_1')$ and U_2 is simple. Since $S_1';S_2$ is defined and $S_2;U' =_{\text{range}(S_1')} U_1;U_2$, it follows that $S_1';U_1$ is defined as well. By Lemma 4.3.3, we have that $S_1';U_1$ chooses variables freely on $\mathcal{Q} \cup \mathcal{Q}_C$. Moreover,

$$S_1';S_2;U' =_{\mathcal{Q} \cup \mathcal{Q}_C} S_1';U_1;U_2$$

By Lemma 4.3.5, there exists a simple substitution V such that

$$S_1';U_1;V =_{\mathcal{Q} \cup \mathcal{Q}_C} T_1'$$

Thus, $S_1';U_1;V =_{\mathcal{Q} \cup \mathcal{Q}_C} T_1'$. Putting it all together, we have that

$$S_1';U_1;V;T_2 =_{\mathcal{Q} \cup \mathcal{Q}_C} T_1;T_2 =_{\mathcal{Q} \cup \mathcal{Q}_C} T$$

Letting $W = U_1;V;T_2$ proves this direction of the lemma.

To finish the proof of the lemma, we note that if $\text{MATCH}(C, \mathcal{Q})$ succeeds, then, clearly, $\text{dom}(S_1') \subseteq (\mathcal{Q} \cup \mathcal{Q}_C)$. ■

4.4 The Type Inference Algorithm

This section presents the type inference algorithm, GE, for ML^+ . GE is written in an applicative, pattern-matching style, and is defined by the mutually recursive clauses given in Table 4.1.

The algorithms UNIFY and MATCH play a crucial role in GE, while a subsidiary role is played by algorithms for applying substitutions, composing substitutions, subtracting two type environments, computing the union of two type environments, and pasting two functions mapping labels to types. All of these subsidiary algorithms are assumed to *fail* when the desired result is not well-defined. For example, while we write $S\sigma$ for the application of a substitution S to type expression σ , we assume in our algorithm GE that if the result is undefined (not a well-formed expression), then the entire algorithm will terminate in error. This notational convention makes the pseudo-code in Table 4.1 more readable.

$\text{GE}(b) = \emptyset \mid \{c \subseteq t\}, \emptyset \supset b: t$, where c is the type of b

$\text{GE}(q) =$

let $\mathcal{Q} =$ the type and row variables in τ_q , where τ_q is the built-in type for q
 $T = \text{MATCH}(\{\tau_q \subseteq t\}, \mathcal{Q} \cup \{t\})$, where t is fresh with respect to \mathcal{Q}
in $\emptyset \mid T \bullet \{\tau_q \subseteq t\}, \emptyset \supset q: Tt$

$\text{GE}(x) = \emptyset \mid \{s \subseteq t\}, \{x: s\} \supset x: t$

$\text{GE}(\langle \phi, \text{EMPTY} \rangle) = \emptyset \mid \{\langle \phi, \text{NULL} \rangle \subseteq \langle \phi, \mathcal{X} \rangle\}, \emptyset \supset \langle \phi, \text{EMPTY} \rangle: \langle \phi, \mathcal{X} \rangle$

$\text{GE}(\langle \phi, u \rangle) = \emptyset \mid \{\langle \phi, \mathcal{X} \rangle \subseteq \langle \phi, \mathcal{Y} \rangle\}, \{u: \langle \phi, \mathcal{X} \rangle\} \supset \langle \phi, u \rangle: \langle \phi, \mathcal{Y} \rangle$

$\text{GE}(\langle f, E \rangle) =$ (where f is non-empty)

let $R_l \mid C_l, A_l \supset f(l): \sigma_l = \text{GE}(f(l))$, for all $l \in \text{dom}(f)$
with the type and row variables in $\text{GE}(f(l))$ renamed to be distinct from those in $\text{GE}(f(l'))$
for all $l \neq l'$ where $l, l' \in \text{dom}(f)$
 $R_E \mid C_E, A_E \supset \langle \phi, E \rangle: \langle h_E, \mathcal{Z} \rangle = \text{GE}(\langle \phi, E \rangle)$
with the type and row variables in $\text{GE}(\langle \phi, E \rangle)$ renamed to be distinct
from those in $\text{GE}(f(l))$, for all $l \in \text{dom}(f)$
 $\mathcal{Q}_l =$ the type and row variables in $R_l \mid C_l, A_l \supset f(l): \sigma_l$, for all $l \in \text{dom}(f)$
 $\mathcal{Q}_E =$ the type and row variables in $R_E \mid C_E, A_E \supset \langle \phi, E \rangle: \langle h_E, \mathcal{Z} \rangle$
 $T = \text{UNIFY}(\{\alpha = \beta \mid w: \alpha \in A_l \text{ and } w: \beta \in A_{l'} \text{ for } l \neq l', l, l' \in \text{dom}(f)$
or $w: \alpha \in A_l \text{ and } w: \beta \in A_E \text{ for } l \in \text{dom}(f)\}, \mathcal{Q}_E \cup \bigcup_{l \in \text{dom}(f)} \mathcal{Q}_l)$
 $S = T; \text{MATCH}(TC_E \cup \bigcup_{l \in \text{dom}(f)} TC_l, \text{range}(T) \cup \mathcal{Q}_E \cup \bigcup_{l \in \text{dom}(f)} \mathcal{Q}_l)$
in $SR_E \cup \bigcup_{l \in \text{dom}(f)} SR_l \mid S \bullet C_E \cup \bigcup_{l \in \text{dom}(f)} S \bullet C_l, SA_E \cup \bigcup_{l \in \text{dom}(f)} SA_l \supset \langle f, E \rangle: S \langle h + h_E, \mathcal{Z} \rangle$
where $\text{dom}(h) = \text{dom}(f)$ and $\forall l \in \text{dom}(h). h(l) = \sigma_l$

$\text{GE}(MN) =$

let $R_1 \mid C_1, A_1 \supset M: \sigma = \text{GE}(M)$
 $R_2 \mid C_2, A_2 \supset N: \tau = \text{GE}(N)$
with type and row variables renamed to be distinct from those in $\text{GE}(M)$
 $\mathcal{Q}_1 =$ the type and row variables in $R_1 \mid C_1, A_1 \supset M: \sigma$
 $\mathcal{Q}_2 =$ the type and row variables in $R_2 \mid C_2, A_2 \supset N: \tau$
 $T = \text{UNIFY}(\{\alpha = \beta \mid w: \alpha \in A_1 \text{ and } w: \beta \in A_2\} \cup \{\sigma = \tau \rightarrow t\}, \mathcal{Q}_1 \cup \mathcal{Q}_2)$
where t is fresh with respect to $\mathcal{Q}_1 \cup \mathcal{Q}_2$
 $S = T; \text{MATCH}(TC_1 \cup TC_2, \text{range}(T) \cup \mathcal{Q}_1 \cup \mathcal{Q}_2 \cup \{t\})$
in $SR_1 \cup SR_2 \cup \{S\tau\} \mid S \bullet C_1 \cup S \bullet C_2, SA_1 \cup SA_2 \supset MN: St$

$\text{GE}(\text{fn } P \Rightarrow M) =$

let $R_p \mid C_p, A_p \supset P: \sigma = \text{GE}(P)$
 $R_e \mid C_e, A_e \supset M: \tau = \text{GE}(M)$
with type and row variables renamed to be distinct from those in $\text{GE}(P)$
 $\mathcal{Q}_p =$ the type and row variables in $R_p \mid C_p, A_p \supset P: \sigma$
 $\mathcal{Q}_e =$ the type and row variables in $R_e \mid C_e, A_e \supset M: \tau$
 $T = \text{UNIFY}(\{\alpha = \beta \mid w: \alpha \in A_p \text{ and } w: \beta \in A_e\}, \mathcal{Q}_p \cup \mathcal{Q}_e)$
 $S = T; \text{MATCH}(TC_p \cup TC_e, \text{range}(T) \cup \mathcal{Q}_p \cup \mathcal{Q}_e)$
in $SR_p \cup SR_e \mid (S \bullet C_p)^{op} \cup S \bullet C_e, (SA_e - SA_p) \supset \text{fn } P \Rightarrow M: S(\sigma \rightarrow \tau)$

4.5 Soundness

In this section, we prove Theorem 4.5.3, the first of the two main theorems of the thesis, which shows that GE is *sound* with respect to the R -typing system and generates the most general restricted typing. More precisely, it shows that for any expression M , if $GE(M)$ yields a restricted typing statement, then every instance of that restricted typing is derivable in the R -typing system.

We will prove Theorem 4.5.3 in the following manner. We will first prove Lemma 4.5.2, which shows that, for any expression M , if $GE(M)$ succeeds and yields a restricted typing, then that restricted typing is provable in the R -typing system. Then, by Theorem 3.6.7, it follows easily that every instance of that restricted typing is provable.

In order to prove Theorem 4.5.2, we will need to use the following fact about the action of GE on patterns: namely, that it infers a type environment that contains mappings for exactly the free variables of the pattern.

Lemma 4.5.1 *If $GE(P) = R \mid C, A \supset P: \sigma$, then $vars(A) = vars(P)$.*

We omit the proof, as it follows by a simple induction on the structure of terms.

Using some of the lemmas developed in Chapter 3 as well as the above lemma, we now prove the desired lemma about the algorithm. In our proof, we extend our notion of unification to type environments and say informally that “ S unifies A_1 and A_2 ” if, for all w such that $w: \alpha \in A_1$ and $w: \beta \in A_2$ for some α and β , $S\alpha$ is syntactically like $S\beta$.

Lemma 4.5.2 *If $GE(M)$ succeeds and yields $R \mid C, A \supset M: \sigma$, then $R \mid C, A \supset M: \sigma$ is a provable restricted typing statement.*

Proof. The proof is analogous to [Mit84]. We proceed by induction on the structure of terms.

If M is an integer b , then $GE(M) = \emptyset \mid \{\mathbf{int} \subseteq t\}, \emptyset \supset b: t$. By axiom (*int*), we have that $\vdash \emptyset \mid \emptyset, \emptyset \supset b: \mathbf{int}$. Lemma 3.6.2 yields $\vdash \emptyset \mid \{\mathbf{int} \subseteq t\}, \emptyset \supset b: \mathbf{int}$, and a use of (*coerce*) then yields $\vdash \emptyset \mid \{\mathbf{int} \subseteq t\}, \emptyset \supset b: t$, as desired. The proof is analogous if b is a real, a boolean, or a string.

If M is a variable x , then $GE(M) = \emptyset \mid \{s \subseteq t\}, \{x: s\} \supset x: t$. By axiom (*var*), we have that $\vdash \emptyset \mid \{x: s\}, \emptyset \supset x: s$. Lemma 3.6.2 yields $\vdash \emptyset \mid \{s \subseteq t\}, \{x: s\} \supset x: s$, and a use of (*coerce*) then yields $\vdash \emptyset \mid \{s \subseteq t\}, \{x: s\} \supset x: t$, as desired.

If M is of the form $\langle \phi, \mathbf{EMPTY} \rangle$, then $GE(M) = \emptyset \mid \{\langle \phi, \mathbf{NULL} \rangle \subseteq \langle \phi, \mathcal{X} \rangle\}, \emptyset \supset \langle \phi, \mathbf{EMPTY} \rangle: \langle \phi, \mathcal{X} \rangle$, for some \mathcal{X} . We have that $\vdash \emptyset \mid \emptyset, \emptyset \supset \langle \phi, \mathbf{EMPTY} \rangle: \langle \phi, \mathbf{NULL} \rangle$ by axiom (*rec1*). We have that $\vdash \emptyset \mid \{\langle \phi, \mathbf{NULL} \rangle \subseteq \langle \phi, \mathcal{X} \rangle\}, \emptyset \supset \langle \phi, \mathbf{EMPTY} \rangle: \langle \phi, \mathbf{NULL} \rangle$ by Lemma 3.6.2, and we get $\vdash \emptyset \mid \{\langle \phi, \mathbf{NULL} \rangle \subseteq \langle \phi, \mathcal{X} \rangle\}, \emptyset \supset \langle \phi, \mathbf{EMPTY} \rangle: \langle \phi, \mathcal{X} \rangle$ by a use of (*coerce*), as desired.

If M is of the form $\langle \phi, u \rangle$, then $GE(M) = \emptyset \mid \{\langle \phi, \mathcal{X} \rangle \subseteq \langle \phi, \mathcal{Y} \rangle\}, \emptyset \supset \langle \phi, u \rangle: \langle \phi, \mathcal{X} \rangle$. By axiom (*rec2*), we have that $\vdash \emptyset \mid \emptyset, \{\langle \phi, u \rangle: \langle \phi, \mathcal{X} \rangle\}, \emptyset \supset \langle \phi, u \rangle: \langle \phi, \mathcal{X} \rangle$. Lemma 3.6.2 yields $\vdash \emptyset \mid \{\langle \phi, \mathcal{X} \rangle \subseteq \langle \phi, \mathcal{Y} \rangle\}, \emptyset \supset \langle \phi, u \rangle: \langle \phi, \mathcal{X} \rangle$, and a use of (*coerce*) then yields $\vdash \emptyset \mid \{\langle \phi, \mathcal{X} \rangle \subseteq \langle \phi, \mathcal{Y} \rangle\}, \emptyset \supset \langle \phi, u \rangle: \langle \phi, \mathcal{Y} \rangle$, as desired.

If M is a built-in constant q , then $GE(M) = \emptyset \mid T \bullet \{\tau_q \subseteq t\}, \emptyset \supset q: Tt$, for some t . By the properties of MATCH, T is a matching substitution for $\{\tau_q \subseteq \tau\}$. Thus, by axiom (*const*), we have that $\vdash \emptyset \mid \emptyset, \emptyset \supset q: T\tau_q$. Lemma 3.6.2 yields $\vdash \emptyset \mid T \bullet \{\tau_q \subseteq t\}, \emptyset \supset q: T\tau_q$. By Lemma 3.5.5, $T \bullet \{\tau_q \subseteq t\} \vdash T\tau_q \subseteq Tt$. Thus, a use of (*coerce*) yields $\vdash \emptyset \mid T \bullet \{\tau_q \subseteq t\}, \emptyset \supset q: Tt$, as desired.

If M is of the form $\langle f, E \rangle$, where f is non-empty, then we can assume inductively that $\vdash R_E | C_E, A_E \supset \langle \phi, E \rangle : \langle h_E, \mathcal{Z} \rangle$ and that

$$\forall l \in \text{dom}(f). \vdash R_l | C_l, A_l \supset f(l) : h(l),$$

where $\text{dom}(h) = \text{dom}(f)$, and $\forall l \in \text{dom}(h). h(l) = \sigma_l$. Since, by assumption, GE succeeds, we can assume that $h + h_E$ is defined, and that S respects C_E and all the C_l , S is defined on R_E and all the R_l , on A_E and all the A_l , and on $\langle h + h_E, \mathcal{Z} \rangle$. Therefore, S is also defined on $\langle h_E, \mathcal{Z} \rangle$.

Thus, by Lemma 3.6.7, we have that $\vdash SR_E | S \bullet C_E, SA_E \supset \langle \phi, E \rangle : S \langle h_E, \mathcal{Z} \rangle$ and that

$$\forall l \in \text{dom}(f). \vdash SR_l | S \bullet C_l, SA_l \supset f(l) : S(h(l))$$

Since, by assumption, $GE(M)$ succeeds, we can assume that $SA_E \cup \bigcup_{l \in \text{dom}(f)} SA_l$ is a well-formed set. Thus, by Lemma 3.6.3, Lemma 3.6.2 and Lemma 3.6.1, we can infer that

$$\vdash SR_E \cup \bigcup_{l \in \text{dom}(f)} SR_l | S \bullet C_E \cup \bigcup_{l \in \text{dom}(f)} S \bullet C_l, SA_E \cup \bigcup_{l \in \text{dom}(f)} SA_l \supset \langle \phi, E \rangle : S \langle h_E, \mathcal{Z} \rangle$$

and that

$$\forall l \in \text{dom}(f). \vdash SR_E \cup \bigcup_{l \in \text{dom}(f)} SR_l | S \bullet C_E \cup \bigcup_{l \in \text{dom}(f)} S \bullet C_l, SA_E \cup \bigcup_{l \in \text{dom}(f)} SA_l \supset f(l) : S(h(l))$$

Since, by assumption, $S \langle h + h_E, \mathcal{Z} \rangle$ is defined, a use of $(rec\beta)$ yields

$$\vdash SR_E \cup \bigcup_{l \in \text{dom}(f)} SR_l | S \bullet C_E \cup \bigcup_{l \in \text{dom}(f)} S \bullet C_l, SA_E \cup \bigcup_{l \in \text{dom}(f)} SA_l \supset \langle f, E \rangle : S \langle h + h_E, \mathcal{Z} \rangle,$$

as desired.

If M is of the form $M'N'$, then we can assume inductively that $\vdash R_1 | C_1, A_1 \supset M' : \sigma$, and that $\vdash R_2 | C_2, A_2 \supset N' : \tau$. Since, by assumption, $GE(M)$ succeeds, we can assume that S respects C_1 and C_2 and that S is defined on R_1, R_2, A_1, A_2, τ , and t . We first need to show that S is defined on σ .

Assume for the sake of contradiction that S is not defined on σ . Since, by assumption, $GE(M)$ succeeds, T unifies $\sigma = \tau \rightarrow t$. Thus, it must be that $\text{MATCH}(TC_1 \cup TC_2)$ is not defined on $T\sigma$, which implies that $\text{MATCH}(TC_1 \cup TC_2)$ is not defined on $T(\tau \rightarrow t)$. But then, S is not defined on $\tau \rightarrow t$, which leads to a contradiction. Thus, we can conclude that S is defined on σ and unifies $\sigma = \tau \rightarrow t$.

By Lemma 3.6.7, we have that $\vdash SR_1 | S \bullet C_1, SA_1 \supset M' : S\sigma$, and that $\vdash SR_2 | S \bullet C_2, SA_2 \supset N' : S\tau$. Since, by assumption, $GE(M)$ succeeds, we can assume that $SA_1 \cup SA_2$ is a well-defined set. Therefore, by Lemma 3.6.3, Lemma 3.6.2, and Lemma 3.6.1, we have that

$$\vdash SR_1 \cup SR_2 \cup \{S\tau\} | S \bullet C_1 \cup S \bullet C_2, SA_1 \cup SA_2 \supset M'N' : S\tau \rightarrow St$$

and that

$$\vdash SR_1 \cup SR_2 \cup \{S\tau\} | S \bullet C_1 \cup S \bullet C_2, SA_1 \cup SA_2, \supset M'N' : S\tau$$

A use of rule (app) then yields

$$\vdash SR_1 \cup SR_2 \cup \{S\tau\} | S \bullet C_1 \cup S \bullet C_2, SA_1 \cup SA_2, \supset M'N' : St,$$

as desired.

If M is of the form $\mathbf{fn} P \Rightarrow M'$, then we can assume inductively that $\vdash R_p | C_p, A_p \supset P : \sigma$, and that $\vdash R_e | C_e, A_e \supset M' : \tau$. Since, by assumption, $GE(M)$ succeeds, we can assume that S respects C_p and C_e and is defined on $R_p, R_e, A_p, A_e, \sigma$ and τ . Thus, by Lemma 3.6.7, we have that $\vdash SR_p | S \bullet C_p, SA_p \supset P : S\sigma$, and that $\vdash SR_e | S \bullet C_e, SA_e \supset M' : S\tau$. Since, by assumption, S unifies A_p and A_e , we have that

$$(SA_e)[SA_p] = SA_p \cup SA_e$$

and is a well-defined set. Thus, by Lemma 3.6.3, Lemma 3.6.2, and Lemma 3.6.1, we can conclude that

$$\vdash SR_p \cup SR_e | S \bullet C_p \cup (S \bullet C_e)^{op}, SA_p \supset P : S\sigma$$

and that

$$\vdash SR_p \cup SR_e | (S \bullet C_p)^{op} \cup S \bullet C_e, (SA_e)[SA_p] \supset M' : S\tau.$$

By Lemma 4.5.1, we can infer that $\mathit{vars}(A_p) = \mathit{vars}(P)$. Since $\mathit{vars}(SA_p) = \mathit{vars}(A_p)$, a use of rule (*abs*) yields

$$\vdash SR_p \cup SR_e | (S \bullet C_p)^{op} \cup S \bullet C_e, SA_e \supset \mathbf{fn} P \Rightarrow M' : S(\sigma \rightarrow \tau).$$

Since $\mathit{vars}(\mathbf{fn} P \Rightarrow M') = \mathit{vars}(M') - \mathit{vars}(P)$, we have by Lemma 3.6.3 that $\mathit{vars}(SA_e) \supseteq \mathit{vars}(M') - \mathit{vars}(P)$. Thus, since $\mathit{vars}(SA_e - SA_p) = \mathit{vars}(SA_e) - \mathit{vars}(P) \supseteq \mathit{vars}(M') - \mathit{vars}(P)$, we have by Lemma 3.6.3 that

$$\vdash SR_p \cup SR_e | (S \bullet C_p)^{op} \cup S \bullet C_e, SA_e - SA_p \supset \mathbf{fn} P \Rightarrow M' : S(\sigma \rightarrow \tau),$$

proving the theorem. ■

We now formally state the main theorem concerning the soundness of GE with respect to the R -typing system.

Lemma 4.5.3 *If $GE(M)$ succeeds and yields $R | C, A \supset M : \sigma$, then every instance of $R | C, A \supset M : \sigma$ is provable.*

The proof follows easily by Lemma 4.5.2 and Lemma 3.6.7.

4.6 Completeness

This section proves Theorem 4.6.4, the second main theorem of this thesis, which shows that GE is *complete* with respect to the R -typing system and generates the most general restricted typing. More precisely, it shows that for any expression M , if a restricted typing for M is derivable in the R -typing system, then GE yields a restricted typing statement of which the provable restricted typing for M is an instance.

In our proof of Theorem 4.6.4, we will need to use an induction on the structure of terms in ML^+ ; that is, for any typable expression M , we will show inductively that the antecedents in the derivation of a provable typing for M are instances of the typing yielded by GE on the corresponding subexpressions of M . However, due to the rule (*coerce*) there may be many possible derivations for a provable typing, and thus, our inductive reasoning becomes more complicated. In order to simplify this reasoning, we show that, as in [Mit84], every typing derivation may be put in a certain “normal form”, in which the rule (*coerce*) is used only after the axioms in the typing system. Reasoning about the normal form derivation of provable typings greatly simplifies our proof of Theorem 4.6.4.

Lemma 4.6.1 *Suppose $R|C, A \supset M:\sigma$ is provable. Then there is a derivation with precisely the same cut formulas in which rule (*coerce*) is used only immediately after the typing axioms (*int*), (*real*), (*bool*), (*string*), (*var*), (*rec1*), (*rec2*), and (*const*).*

Proof. The proof is similar to that of [Mit84]. We think of the proof of a restricted typing statement as a tree, where each leaf is an instance of the axiom (*int*), (*real*), (*bool*), (*string*), (*var*), (*rec1*), (*rec2*), or (*const*). We think of each node as labeled by both the restricted typing statement proved at that node and the final rule used in that proof. Given a proof of a restricted typing statement, we define the *degree* of the proof to be the number of pairs of internal tree nodes (α, β) such that there is a path from a leaf *through* α to β and node β is labeled with (*coerce*). Intuitively, the degree of the proof gives a measure of how far the occurrences of (*coerce*) are from the leaves. We note that a proof has degree zero iff the rule (*coerce*) is used only immediately after the axioms and show by induction on the degree of a proof that every provable statement has a proof of degree zero.

If the final two rules used in the proof are both (*coerce*), then $\vdash R|C, A \supset M:\sigma$ must have followed by the antecedents $\vdash R|C, A \supset M:\gamma$ and $C \vdash \gamma \subseteq \sigma$ for some γ . Also, $\vdash R|C, A \supset M:\gamma$ must have followed by the antecedents $\vdash R|C, A \supset M:\tau$ and $C \vdash \tau \subseteq \gamma$ for some τ . By (*trans*), we can infer that

$$C \vdash \tau \subseteq \sigma.$$

Therefore, we can derive $\vdash R|C, A \supset M:\sigma$ directly from $\vdash R|C, A \supset M:\tau$ by (*coerce*), thus reducing the degree of the proof.

If the final two rules used in the proof are (*app*) followed by (*coerce*), then M is of the form $M = M'N'$ and $\vdash R|C, A \supset M:\sigma$ must have followed by the antecedents $\vdash R|C, A \supset M'N':\gamma$ and $C \vdash \gamma \subseteq \sigma$ for some γ . Also, for some τ , where $R = R' \cup \{\tau\}$, $\vdash R|C, A \supset M'N':\gamma$ must have followed by the antecedents $\vdash R'|C, A \supset M':\tau \rightarrow \gamma$ and $\vdash R'|C, A \supset N':\tau$. By (*arrow*), we can infer that

$$C \vdash \tau \rightarrow \gamma \subseteq \tau \rightarrow \sigma.$$

Thus, we can derive $\vdash R'|C, A \supset M':\tau \rightarrow \sigma$ directly from $\vdash R'|C, A \supset M':\tau \rightarrow \gamma$ by (*coerce*), and then proceed to derive $\vdash R|C, A \supset M'N':\sigma$ by (*app*), thus reducing the degree of the proof.

If the final two rules used in a proof are (*abs*) followed by (*coerce*), then M is of the form $M = \mathbf{fn} P \Rightarrow M'$ and σ is of the form $\sigma = \sigma_1 \rightarrow \sigma_2$. Thus, $\vdash R|C, A \supset \mathbf{fn} P \Rightarrow M':\sigma_1 \rightarrow \sigma_2$ must have followed by the antecedents $\vdash R|C, A \supset \mathbf{fn} P \Rightarrow M':\gamma_1 \rightarrow \gamma_2$ and $C \vdash \gamma_1 \rightarrow \gamma_2 \subseteq \sigma_1 \rightarrow \sigma_2$ for some γ_1 and γ_2 . Also, $\vdash R|C, A \supset \mathbf{fn} P \Rightarrow M':\gamma_1 \rightarrow \gamma_2$ must have followed by the antecedents $\vdash R|C^{op}, A' \supset P:\gamma_1$ and $\vdash R|C, A[A'] \supset M':\gamma_2$ for some A' such that $\mathit{vars}(A') = \mathit{vars}(P)$. By Lemma 3.5.2,

$$C \vdash \sigma_1 \subseteq \gamma_1 \text{ and } C \vdash \gamma_2 \subseteq \sigma_2;$$

thus, $C^{op} \vdash \gamma_1 \subseteq \sigma_1$. Therefore, we can derive $\vdash R|C^{op}, A' \supset P:\sigma_1$ directly from $\vdash R|C^{op}, A' \supset P:\gamma_1$ by (*coerce*), and we can derive $\vdash R|C, A[A'] \supset M':\sigma_2$ directly from $\vdash R|C, A[A'] \supset M':\gamma_2$ by (*coerce*). We can then proceed to derive $\vdash R|C^{op}, A \supset \mathbf{fn} P \Rightarrow M':\sigma_1 \rightarrow \sigma_2$ by (*abs*), thus reducing the degree of the proof by 1.

If the final two rules used in a proof are (*rec3*) followed by (*coerce*), then M is of the form $M = \langle f, E \rangle$, and σ is of the form $\sigma = \langle h, \mathcal{Z} \rangle$, where $\mathit{dom}(f) \neq \phi$ and $\mathit{dom}(h) \neq \phi$. We can thus assume that, for some $\langle h', \mathcal{Z}' \rangle$ where $\mathit{dom}(h') = \mathit{dom}(h)$, $\vdash R|C, A \supset M:\sigma$

must have followed by the antecedents $\vdash R \mid C, A \supset M: \langle h', \mathcal{Z}' \rangle$ and $C \vdash \langle h', \mathcal{Z}' \rangle \subseteq \langle h, \mathcal{Z} \rangle$. Also, we can assume that $\vdash R \mid C, A \supset M: \langle h', \mathcal{Z}' \rangle$ must have followed by the antecedents $\forall l \in \text{dom}(f). \vdash R \mid C, A \supset f(l): h_1'(l)$ and $\vdash R \mid C, A \supset \langle \phi, E \rangle: \langle h_2', \mathcal{Z}' \rangle$ for some h_1' and h_2' , where $h' = h_1' + h_2'$. By Lemma 3.5.3,

$$C \vdash \langle \phi, \mathcal{Z}' \rangle \subseteq \langle \phi, \mathcal{Z} \rangle,$$

and

$$\forall l \in \text{dom}(h). C \vdash h(l) \subseteq h'(l).$$

Since $\text{dom}(h) = \text{dom}(h')$ and $\text{dom}(h_1') \cap \text{dom}(h_2') = \emptyset$, we can divide h into $h = h_1 + h_2$, where $\text{dom}(h_1) = \text{dom}(h_1')$ and $\text{dom}(h_2) = \text{dom}(h_2')$. Since $h_2 \subseteq h$, we can infer by (*record*) that

$$C \vdash \langle h_2', \mathcal{Z}' \rangle \subseteq \langle h_2, \mathcal{Z} \rangle.$$

We can then derive $\vdash R \mid C, A \supset \langle \phi, E \rangle: \langle h_2, \mathcal{Z} \rangle$ directly from $\vdash R \mid C, A \supset \langle \phi, E \rangle: \langle h_2', \mathcal{Z}' \rangle$ by (*coerce*). Moreover, for all $l \in \text{dom}(f)$, we can derive $\vdash R \mid C, A \supset f(l): h_1(l)$ directly from $\vdash R \mid C, A \supset f(l): h_1'(l)$ by (*coerce*), since $h_1 \subseteq h$. We can then derive $\vdash R \mid C, A \supset M: \langle h, \mathcal{Z} \rangle$ by (*rec3*), reducing the degree of the proof by 1, and proving the lemma. ■

In our proof of Theorem 4.6.4, we will need the following property about the action of GE on record expressions which contain no explicit labels: namely, that such records are always inferred to be of a record type whose first component is the empty function.

Lemma 4.6.2 *If $GE(\langle \phi, E \rangle) = R \mid C, A \supset \langle \phi, E \rangle: \langle h, \mathcal{Z} \rangle$, then $\text{dom}(h) = \emptyset$.*

The proof proceeds by simple case analysis.

We will also need the following property of substitutions, which shows that, in a precise technical sense, substitution composition merges naturally with the operation \bullet .

Lemma 4.6.3 *Suppose that \mathcal{Q} is a set of type variables and row variables and that C is a set of possibly non-matching subtype assertions that contains only type variables and row variables from \mathcal{Q} . Furthermore, suppose that S, T and W are substitutions such that $T; S$ is defined and $W =_{\mathcal{Q}} T; S$. If W and T are matching substitutions for C , then*

1. S respects $(T \bullet C)$
2. $S \bullet (T \bullet C) \vdash W \bullet C$
3. $W \bullet C \vdash S \bullet (T \bullet C)$

Proof. To prove (1), we note that, clearly, S is a matching substitution for TC . It is easy to show by induction on the structure of terms that S must respect $T \bullet C$.

To prove (2), we note that, by Lemma 3.5.5, $T \bullet C \vdash TC$. By Lemma 3.6.5, we have that

$$\forall \sigma' \subseteq \tau' \in TC. S \bullet (T \bullet C) \vdash \text{ATOMIC}(S\sigma' \subseteq S\tau')$$

Thus, it follows that $\forall \sigma \subseteq \tau \in C. S \bullet (T \bullet C) \vdash \text{ATOMIC}(S(T\sigma) \subseteq S(T\tau))$. Therefore, $S \bullet (T \bullet C) \vdash (T; S) \bullet C$. Since C contains only type variables and row variables from \mathcal{Q} , we have that $W \bullet C = (T; S) \bullet C$, as desired.

To prove (3), we note that, clearly, $(T; S)$ respects C . By Lemma 3.5.5, it follows that $\forall \sigma \subseteq \tau \in C. (T; S) \bullet \{\sigma \subseteq \tau\} \vdash (T; S)\{\sigma \subseteq \tau\}$.

By induction on the structure of terms and by Lemma 3.5.2 and Lemma 3.5.3, it is easy to show that

$$\forall \sigma \subseteq \tau \in C. (T; S) \bullet \{\sigma \subseteq \tau\} \vdash \bigcup_{\sigma' \subseteq \tau' \in \text{ATOMIC}(T\sigma \subseteq T\tau)} \{S\sigma' \subseteq S\tau'\}$$

By Lemma 3.5.5,

$$\forall \sigma \subseteq \tau \in C. (T; S) \bullet \{\sigma \subseteq \tau\} \vdash \bigcup_{\sigma' \subseteq \tau' \in \text{ATOMIC}(T\sigma \subseteq T\tau)} \text{ATOMIC}(S\sigma' \subseteq S\tau')$$

Thus, we have that $(T; S) \bullet C \vdash S \bullet (T \bullet C)$. Since C contains only type variables and row variables from \mathcal{Q} , $W \bullet C = (T; S) \bullet C$, proving the lemma. \blacksquare

Using the above lemmas as well as some of the lemmas developed in Chapter 3, we now prove our main theorem about the completeness of GE with respect to the R -typing system.

Theorem 4.6.4 *If $\vdash R \mid C, A \supset M : \gamma$, then $GE(M)$ succeeds and produces a restricted typing statement with $R \mid C, A \supset M : \gamma$ as an instance.*

Proof. The proof is similar to [Mit84], except that we show that the substitution which provides the instance is defined on $GE(M)$. We proceed by induction on the structure of terms.

Suppose that M is an integer b , and that $\vdash R \mid C, A \supset b : \gamma$. By Lemma 4.6.1, we can assume without loss of generality that the proof consists of a use of (int) followed by a use of $(coerce)$. Thus, $C \vdash \mathbf{int} \subseteq \gamma$. By Lemma 3.5.1, we can conclude that \mathbf{int} matches γ .

It is easy to see that $GE(b)$ always succeeds. It remains to show that $\vdash R \mid C, A \supset b : \gamma$ is an instance of $GE(b)$ by some substitution S . Let $S = [\gamma/t]$. Clearly, S respects $\{\mathbf{int} \subseteq t\}$. Thus, by Lemma 3.5.5, we can conclude that $C \vdash S \bullet \{\mathbf{int} \subseteq t\}$, giving the proof of this case. The proof is analogous if b is a real, a boolean, or a string.

Suppose that M is a variable x , and that $\vdash R \mid C, A \supset x : \gamma$. By Lemma 4.6.1, we can assume without loss of generality that the proof consists of a use of (var) followed by a use of $(coerce)$. By Lemma 3.6.3, x must occur in A . Let τ be the type expression with $x : \tau \in A$, and note that $C \vdash \tau \subseteq \gamma$. By Lemma 3.5.1, γ and τ must match.

It is easy to see that $GE(x)$ always succeeds. It remains to show that $\vdash R \mid C, A \supset x : \gamma$ is an instance of $GE(x)$ by some substitution S . Let $S = [\gamma/t, \tau/s]$. Clearly S respects $\{s \subseteq t\}$; thus, by Lemma 3.5.5, we can conclude that $C \vdash S \bullet \{s \subseteq t\}$. Moreover, since $x : \tau \in A$, it follows that $A \supseteq S\{x : s\}$, completing the proof of this case.

Suppose that M is of the form $\langle \phi, \text{EMPTY} \rangle$ and that $\vdash R \mid C, A \supset M : \gamma$. By Lemma 4.6.1, we can assume without loss of generality that the proof consists of a use of $(rec1)$ followed by a use of $(coerce)$. Thus, $C \vdash \langle \phi, \text{NULL} \rangle \subseteq \gamma$. By Lemma 3.5.1, we can conclude that $\langle \phi, \text{NULL} \rangle$ matches γ . Thus, $\gamma = \langle \phi, \mathcal{Z} \rangle$ for some \mathcal{Z} .

It is easy to see that $GE(\langle \phi, \text{EMPTY} \rangle)$ always succeeds. It remains to show that $R \mid C, A \supset \langle \phi, \text{EMPTY} \rangle : \gamma$ is an instance of $GE(\langle \phi, \text{EMPTY} \rangle)$ by some substitution S . Let

$$S = [\langle \phi, \mathcal{Z} \rangle / \mathcal{X}]$$

Since $\gamma = S\langle \phi, \mathcal{X} \rangle$, it follows that S respects $\{\langle \phi, \text{NULL} \rangle \subseteq \langle \phi, \mathcal{X} \rangle\}$. Thus, by Lemma 3.5.5, we have that $C \vdash S \bullet \{\langle \phi, \text{NULL} \rangle \subseteq \langle \phi, \mathcal{X} \rangle\}$, completing the proof of this case.

Suppose that M is of the form $\langle \phi, u \rangle$ and that $\vdash R \mid C, A \supset M : \gamma$. By Lemma 4.6.1, we can assume without loss of generality that the proof consists of a use of $(rec2)$ followed by

a use of (*coerce*). By Lemma 3.6.3, u must occur in A . Let $\langle h, \mathcal{Z} \rangle$ be the type expression with $u: \langle h, \mathcal{Z} \rangle \in A$, and note that $C \vdash \langle h, \mathcal{Z} \rangle \subseteq \gamma$. By Lemma 3.5.1, γ and $\langle h, \mathcal{Z} \rangle$ must match. Thus, $\gamma = \langle h', \mathcal{Z}' \rangle$ for some h' and \mathcal{Z}' such that $\text{dom}(h) = \text{dom}(h')$ and $\forall l \in \text{dom}(h). h(l)$ matches $h'(l)$.

It is easy to see that $GE(\langle \phi, u \rangle)$ always succeeds. It remains to show that $R \mid C, A \supset \langle \phi, u \rangle: \gamma$ is an instance of $GE(\langle \phi, u \rangle)$ by some substitution S . Let

$$S = [\langle h', \mathcal{Z}' \rangle / \mathcal{Y}, \langle h, \mathcal{Z} \rangle / \mathcal{X}]$$

Clearly, S respects $\{\langle \phi, \mathcal{X} \rangle \subseteq \langle \phi, \mathcal{Y} \rangle\}$; thus, by Lemma 3.5.5, we can conclude that $C \vdash S \bullet \{\langle \phi, \mathcal{X} \rangle \subseteq \langle \phi, \mathcal{Y} \rangle\}$. Moreover, since $u: \langle h, \mathcal{Z} \rangle \in A$, it follows that $A \supseteq S\{u: \langle \phi, \mathcal{X} \rangle\}$, completing the proof of this case.

Suppose that M is a built-in constant q and that $\vdash R \mid C, A \supset q: \gamma$. By Lemma 4.6.1, we can assume without loss of generality that the proof consists of a use of (*const*) followed by a use of (*coerce*). Thus, we can assume that, for some substitution V that is defined on τ_q , (where τ_q is the built-in type for q), $C \vdash V\tau_q \subseteq \gamma$. By Lemma 3.5.1, $V\tau_q$ and γ must match.

Let $\mathcal{Q}' = \mathcal{Q} \cup \{t\}$. Also, let W be the substitution with $\text{dom}(W) = \mathcal{Q}'$ and such that, for all type variables and row variables in \mathcal{Q} , W behaves identically to V ; furthermore, $Wt = \gamma$. Clearly, W is defined on τ_q and $W\tau_q = V\tau_q$; therefore, W is a matching substitution for $\{\tau_q \subseteq t\}$. By Lemma 4.3.6, $\text{MATCH}(\{\tau_q \subseteq t\}, \mathcal{Q}')$ succeeds, and $W =_{\mathcal{Q}'} T; U$ for some substitution U where $T; U$ is defined. Also by Lemma 4.3.6, $\text{dom}(T) \subseteq \mathcal{Q}'$; thus, $T \upharpoonright \mathcal{Q}' = T$. Therefore, since $\text{dom}(W) = \mathcal{Q}'$,

$$W = T; U', \text{ where } U' = U \upharpoonright (\text{range}(T) \cup \mathcal{Q}').$$

It remains to show that $R \mid C, A \supset q: \gamma$ is an instance of $GE(q)$ by some substitution S . Let $S = U'$. Then, by Lemma 4.6.3, S respects $T \bullet \{\tau_q \subseteq t\}$, and $W \bullet \{\tau_q \subseteq t\} \vdash S \bullet (T \bullet \{\tau_q \subseteq t\})$. Since $W \bullet \{\tau_q \subseteq t\} = \text{ATOMIC}(V\tau_q \subseteq \gamma)$, we have by Lemma 3.5.5 that $C \vdash W \bullet \{\tau_q \subseteq t\}$, completing the proof of this case.

Suppose that M is of the form $M'N'$ and that $\vdash R \mid C, A \supset M: \gamma$. By Lemma 4.6.1, we can assume without loss of generality that the proof ends in a use of (*app*). Thus, for some γ' ,

$$\vdash R' \mid C, A \supset M': \gamma' \rightarrow \gamma$$

and

$$\vdash R' \mid C, A \supset N': \gamma',$$

where $R' = R \cup \{\gamma'\}$. We can assume inductively that $R' \mid C, A \supset M': \gamma' \rightarrow \gamma$ is an instance of $GE(M') = R_1 \mid C_1, A_1 \supset M': \sigma$ and that $R' \mid C, A \supset N': \gamma'$ is an instance of $GE(N') = R_2 \mid C_2, A_2 \supset N': \tau$. Thus, there exist substitutions V_1 and V_2 such that V_1 respects C_1 and is defined on R_1, A_1 , and σ , V_2 respects C_2 and is defined on R_2, A_2 , and τ , and such that

$$R' \supseteq V_1 R_1, C \vdash V_1 \bullet C_1, A \supseteq V_1 A_1, \gamma' \rightarrow \gamma = V_1 \sigma$$

and

$$R' \supseteq V_2 R_2, C \vdash V_2 \bullet C_2, A \supseteq V_2 A_2, \gamma' = V_2 \tau.$$

We note that no type or row variable in \mathcal{Q}_1 appears in \mathcal{Q}_2 , and that t is not contained in \mathcal{Q}_1 or \mathcal{Q}_2 . Thus, there exists a well-defined substitution V whose domain is $\{t\} \cup \mathcal{Q}_1 \cup \mathcal{Q}_2$ and such that V maps any type variable s in its domain to $V_1 s$ if $s \in \mathcal{Q}_1$ and to $V_2 s$ if

$s \in \mathcal{Q}_2$. (The analogous definition holds for row variables in the domain of V .) Also, V maps t to γ .

It is easy to see that V respects C_1 and C_2 and is defined on $R_1, R_2, A_1, A_2, \sigma$ and τ , and that

$$R' \supseteq VR_1, C \vdash V \bullet C_1, A \supseteq VA_1, \gamma' \rightarrow \gamma = V\sigma$$

and

$$R' \supseteq VR_2, C \vdash V \bullet C_2, A \supseteq VA_2, \gamma' = V\tau.$$

Therefore, V must unify $\{\alpha = \beta \mid w: \alpha \in A_1 \text{ and } w: \beta \in A_2\}$. Also, since $Vt = \gamma$, we have that $V\sigma = V\tau \rightarrow Vt = V(\tau \rightarrow t)$; thus, V unifies $\sigma = \tau \rightarrow t$. Let $\mathcal{Q} = \mathcal{Q}_1 \cup \mathcal{Q}_2 \cup \{t\}$. By Lemma 4.2.1, it follows that $T = \text{UNIFY}(\{\alpha = \beta \mid w: \alpha \in A_1 \text{ and } w: \beta \in A_2\} \cup \{\sigma = \tau \rightarrow t\}, \mathcal{Q})$ succeeds, and that $V =_{\mathcal{Q}} T; U$ for some substitution U such that $T; U$ is defined. Equivalently, $V =_{\mathcal{Q}} T; U'$, where $U' = U \upharpoonright (\text{range}(T) \cup \mathcal{Q})$.

Since only type and row variables from \mathcal{Q} occur in C_1 and C_2 , clearly U' respects $(TC_1 \cup TC_2)$. Therefore, by Lemma 4.3.6, we can assume that $\text{MATCH}(TC_1 \cup TC_2, \text{range}(T) \cup \mathcal{Q})$ succeeds. Furthermore, letting $T' = \text{MATCH}(TC_1 \cup TC_2, \text{range}(T) \cup \mathcal{Q})$, we have that $U' =_{\text{range}(T) \cup \mathcal{Q}} T'; W$ for some substitution W where $T'; W$ is defined. Since $\text{dom}(U') \subseteq (\text{range}(T) \cup \mathcal{Q})$, and since, by Lemma 4.3.6, $\text{dom}(T') \subseteq (\text{range}(T) \cup \mathcal{Q})$, we have that

$$U' = (T'; W) \upharpoonright (\text{range}(T) \cup \mathcal{Q}) = T'; W \upharpoonright (\text{range}(T') \cup \text{range}(T) \cup \mathcal{Q}).$$

Thus, since $T; U'$ is defined, so is $T; (T'; W \upharpoonright (\text{range}(T') \cup \text{range}(T) \cup \mathcal{Q}))$, and thus so is $S = T; T'$. Therefore, $V =_{\mathcal{Q}} T; T'; W =_{\mathcal{Q}} S; W$.

We recall that all the type variables and row variables occurring in any of $C_1, C_2, R_1, R_2, A_1, A_2, \sigma, \tau$, and t , are contained in \mathcal{Q} . Thus, S respects C_1 and C_2 and is defined on $R_1, R_2, A_1, A_2, \sigma$ and τ . Since T unifies A_1 and A_2 , so does S . Thus, $SA_1 \cup SA_2$ is a well-formed set. Furthermore, since $V =_{\mathcal{Q}} S; W$, it is easy to see that W is defined on $SR_1, SR_2, SA_1, SA_2, S\sigma$ and $S\tau$. By Lemma 4.6.3, we have that W respects $S \bullet C_1$ and $S \bullet C_2$ and that $C \vdash W \bullet (S \bullet C_1 \cup S \bullet C_2)$.

Clearly, $W(S\sigma) = \gamma$, $A \supseteq W(SA_1 \cup SA_2)$, and $R' \supseteq W(SR_1 \cup SR_2)$. Since $V\tau = \gamma'$, we have that $R = R' \cup W(S\tau)$. Thus, $R \supseteq W(SR_1 \cup SR_2 \cup S\tau)$. Therefore, $R \mid C, A \supset M'N': \gamma$ is an instance of $GE(M'N')$ by substitution W .

Suppose that M is of the form $\text{fn } P \Rightarrow M'$ and $\vdash R \mid C, A \supset M: \gamma$. By Lemma 4.6.1, we can assume without loss of generality that the proof ends in a use of (abs) . Thus, for some A', γ_1 , and γ_2 , where $\text{vars}(A') = \text{vars}(P)$ and $\gamma = \gamma_1 \rightarrow \gamma_2$,

$$\vdash R \mid C^{op}, A' \supset P: \gamma_1$$

and

$$\vdash R \mid C, A[A'] \supset M': \gamma_2.$$

We can thus assume inductively that $R \mid C^{op}, A' \supset P: \gamma_1$ is an instance of $GE(P) = R_p \mid C_p, A_p \supset P: \sigma$ and that $R \mid C, A[A'] \supset M': \gamma_2$ is an instance of $GE(M) = R_e \mid C_e, A_e \supset M': \tau$. Thus, there exist substitutions V_p and V_e such that V_p respects C_p and is defined on R_p, A_p , and σ , V_e respects C_e and is defined on R_e, A_e , and τ , and such that

$$R \supseteq V_p R_p, C^{op} \vdash V_p \bullet C_p, A' \supseteq V_p A_p, \gamma_1 = V_p \sigma$$

and

$$R \supseteq V_e R_e, C \vdash V_e \bullet C_e, A[A'] \supseteq V_e A_e, \gamma_2 = V_e \tau.$$

Since no type or row variables in \mathcal{Q}_p appear in \mathcal{Q}_e , we can define the substitution V whose domain is $\mathcal{Q}_p \cup \mathcal{Q}_e$ and such that V maps any type variable t in its domain to $V_p t$ if $t \in \mathcal{Q}_p$ and to $V_e t$ if $t \in \mathcal{Q}_e$. The action of V on row variables in its domain is analogous. It is easy to see that V respects C_p and C_e , is defined on $R_p, R_e, A_p, A_e, \sigma$ and τ , and that

$$R \supseteq V R_p, C^{op} \vdash V \bullet C_p, A' \supseteq V A_p, \gamma_1 = V \sigma$$

and

$$R \supseteq V R_e, C \vdash V \bullet C_e, A[A'] \supseteq V A_e, \gamma_2 = V \tau.$$

By Lemma 4.5.1, $\text{vars}(A_p) = \text{vars}(P)$. Thus, $A' = V A_p$, and thus, $A[V A_p] \supseteq V A_e$. This implies that if $w: V \alpha \in V A_e$ and $w: V \beta \in V A_p$, then $V \alpha = V \beta$. Let $\mathcal{Q} = \mathcal{Q}_p \cup \mathcal{Q}_e$. By Lemma 4.2.1, it follows that $\text{UNIFY}(\{\alpha = \beta \mid w: \alpha \in A_p \text{ and } w: \beta \in A_e\}, \mathcal{Q})$ succeeds, and that $V =_{\mathcal{Q}} T; U$ for some U such that $T; U$ is defined. Equivalently, $V =_{\mathcal{Q}} T; U'$, where $U' = U \uparrow (\text{range}(T) \cup \mathcal{Q})$.

Since only type and row variables from \mathcal{Q} occur in C_p and C_e , clearly U respects $(TC_p \cup TC_e)$. Thus, by Lemma 4.3.6, we can assume that that $\text{MATCH}(TC_p \cup TC_e, \text{range}(T) \cup \mathcal{Q})$ succeeds. Furthermore, letting $T' = \text{MATCH}(TC_p \cup TC_e, \text{range}(T) \cup \mathcal{Q})$, we have that $U' =_{\text{range}(T) \cup \mathcal{Q}} T'; W$ for some substitution W where $T'; W$ is defined. Since $\text{dom}(U') \subseteq (\text{range}(T) \cup \mathcal{Q})$, and since, by Lemma 4.3.6, $\text{dom}(T') \subseteq (\text{range}(T) \cup \mathcal{Q})$, we have that

$$U' = (T'; W) \uparrow (\text{range}(T) \cup \mathcal{Q}) = T'; W \uparrow (\text{range}(T') \cup \text{range}(T) \cup \mathcal{Q}).$$

Thus, since $T; U'$ is defined, so is $T; (T'; W \uparrow (\text{range}(T') \cup \text{range}(T) \cup \mathcal{Q}))$, and thus so is $S = T; T'$. Therefore, $V =_{\mathcal{Q}} T; T'; W =_{\mathcal{Q}} S; W$.

We recall that all the type variables and row variables occurring in any of $C_p, C_e, R_p, R_e, A_p, A_e, \sigma$, and τ , are contained in \mathcal{Q} . Clearly, S respects C_p and C_e and is defined on $R_p, R_e, A_p, A_e, \sigma$ and τ . Since T unifies A_p and A_e , so does S . Thus, $S A_e - S A_p$ is a well-defined set. Furthermore, since $V = S; W$, it is easy to see that W is defined on $S R_p, S R_e, S A_p, S A_e, S \sigma$ and $S \tau$. By Lemma 4.6.3, W respects $S \bullet C_p$ and $S \bullet C_e$ and $C \vdash W \bullet (S \bullet C_p^{op} \cup S \bullet C_e)$. It is easy to see that

$$W(S \sigma \rightarrow S \tau) = \gamma_1 \rightarrow \gamma_2, \text{ and } R \supseteq W(S R_p \cup S R_e).$$

Since $A' = W(S A_p)$, we know that $A[A'] = A_1 \cup W(S A_p)$, where $A = A_1 \cup \{w: \alpha \mid w: \alpha \in A \text{ and } w: \beta \in W(S A_p)\}$. (Note that these are “disjoint” unions, since these sets are well-formed.) If $w: \alpha \in W(S A_e - S A_p)$, then $w \notin \text{vars}(W(S A_p))$. Thus, since $A[A'] \supseteq W(S A_e) \supseteq W(S A_e - S A_p)$, $w: \alpha \in A_1$. Thus, $w: \alpha \in A$, and therefore $A \supseteq W(S A_e - S A_p)$, as desired.

Thus, $R \mid C, A \supset \text{fn } P \Rightarrow M': \gamma$ is an instance of $GE(\text{fn } P \Rightarrow M')$ by substitution W .

Suppose that M is of the form $\langle f, E \rangle$, where f is non-empty, and $\vdash R \mid C, A \supset M: \gamma$. By Lemma 4.6.1, we can assume without loss of generality that the proof ends in a use of $(\text{rec}3)$. Therefore, we can assume that for some h_1, h_2 , and \mathcal{Z}' where $h_1 + h_2$ is well-defined, $\text{dom}(f) = \text{dom}(h_1)$, and $\gamma = \langle h_1 + h_2, \mathcal{Z}' \rangle$,

$$\vdash R \mid C, A \supset \langle \phi, E \rangle: \langle h_2, \mathcal{Z}' \rangle$$

and

$$\forall l \in \text{dom}(f). \vdash R \mid C, A \supset f(l): h_1(l).$$

We can thus assume inductively that $R \mid C, A \supset \langle \phi, E \rangle: \langle h_2, \mathcal{Z}' \rangle$ is an instance of $GE(\langle \phi, E \rangle) = R_E \mid C_E, A_E \supset \langle \phi, E \rangle: \langle h_E, \mathcal{Z} \rangle$ and that for all $l \in \text{dom}(f)$, $R \mid C, A \supset f(l): h_1(l)$ is an instance of $GE(f(l)) = R_l \mid C_l, A_l \supset f(l): h(l)$.

Thus, there exists a substitution V_E such that V_E respects C_E and is defined on R_E, A_E , and $\langle h_E, \mathcal{Z} \rangle$ and such that

$$R \supseteq V_E R_E, C \vdash V_E \bullet C_E, A \supseteq V_E A_E, \langle h_2, \mathcal{Z}' \rangle = V_E \langle h_E, \mathcal{Z} \rangle.$$

Also, for all $l \in \text{dom}(f)$, there exists a substitution V_l such that V_l respects C_l and is defined on R_l, A_l , and $h(l)$ and such that

$$R \supseteq V_l R_l, C \vdash V_l \bullet C_l, A \supseteq V_l A_l, h_1(l) = V_l(h(l)).$$

Since the type and row variables that occur in \mathcal{Q}_E and in the \mathcal{Q}_l are all distinct there is a well-defined substitution V whose domain is $\mathcal{Q}_E \cup \bigcup_{l \in \text{dom}(f)} \mathcal{Q}_l$, and such that for all type variables t in its domain, $Vt = V_E t$ if $t \in \mathcal{Q}_E$, and, for any $l \in \text{dom}(f)$, $Vt = V_l t$ if $t \in \mathcal{Q}_l$. An analogous definition holds for the action of V on row variables in its domain.

It is easy to see that V respects C_E and is defined on R_E, A_E and $\langle h_E, \mathcal{Z} \rangle$ and that, for all $l \in \text{dom}(f)$, V respects C_l and is defined on R_l, A_l , and $h(l)$. Moreover, we have that

$$R \supseteq V R_E, C \vdash V \bullet C_E, A \supseteq V A_E, \langle h_2, \mathcal{Z}' \rangle = V \langle h_E, \mathcal{Z} \rangle.$$

and for all $l \in \text{dom}(f)$,

$$R \supseteq V R_l, C \vdash V \bullet C_l, A \supseteq V A_l, h_1(l) = V(h(l)).$$

Let $\mathcal{Q} = \mathcal{Q}_E \cup \bigcup_{l \in \text{dom}(f)} \mathcal{Q}_l$. Since V must unify A_E and all of the A_l , we can conclude by Lemma 4.2.1 that $T = \text{UNIFY}(\{\alpha = \beta \mid w: \alpha \in A_l \text{ and } w: \beta \in A_{l'}, \text{ for } l \neq l', l, l' \in \text{dom}(f), \text{ or } w: \alpha \in A_l \text{ and } w: \beta \in A_E, \text{ for } l \in \text{dom}(f)\}, \mathcal{Q})$ succeeds. Furthermore, $V =_{\mathcal{Q}} T; U$ for some U such that $T; U$ is defined. Equivalently, $V =_{\mathcal{Q}} T; U'$, where $U' = U \upharpoonright (\text{range}(T) \cup \mathcal{Q})$.

Since only type and row variables from \mathcal{Q} occur in C_E and all the C_l , clearly U must respect TC_E and all of the TC_l . Thus, by Lemma 4.3.6, we can assume that that $\text{MATCH}(TC_E \cup \bigcup_{l \in \text{dom}(f)} TC_l, \text{range}(T) \cup \mathcal{Q})$ succeeds. Furthermore, letting

$$T' = \text{MATCH}(TC_E \cup \bigcup_{l \in \text{dom}(f)} TC_l, \text{range}(T) \cup \mathcal{Q}),$$

we have that $U' =_{\text{range}(T) \cup \mathcal{Q}} T'; W$ for some substitution W where $T'; W$ is defined. Since $\text{dom}(U') \subseteq (\text{range}(T) \cup \mathcal{Q})$, and since, by Lemma 4.3.6, $\text{dom}(T') \subseteq (\text{range}(T) \cup \mathcal{Q})$, we have that

$$U' = (T'; W) \upharpoonright (\text{range}(T) \cup \mathcal{Q}) = T'; W \upharpoonright (\text{range}(T') \cup \text{range}(T) \cup \mathcal{Q}).$$

Thus, since $T; U'$ is defined, so is $T; (T'; W \upharpoonright (\text{range}(T') \cup \text{range}(T) \cup \mathcal{Q}))$, and thus so is $S = T; T'$. Therefore, $V =_{\mathcal{Q}} T; T'; W =_{\mathcal{Q}} S; W$.

We recall that all the type variables and row variables occurring in $C_E, A_E, R_E, \langle h_E, \mathcal{Z} \rangle$, any of the C_l, R_l, A_l, σ_l , are contained in \mathcal{Q} . Thus, S respects C_E and all the C_l , and is defined on R_E and all the R_l , on A_E and all the A_l , and on $\langle h_E, \mathcal{Z} \rangle$ and all the $h(l)$. Since T unifies A_E and all the A_l , so does S ; thus, $SA_E \cup \bigcup_{l \in \text{dom}(f)} SA_l$ is well-formed.

It is easy to see that W is defined on SR_E and all the SR_l , on SA_E and all the SA_l , and on $S\langle h_E, \mathcal{Z} \rangle$ and all the $S\langle h(l) \rangle$. By Lemma 4.6.3, W respects $S \bullet C_E$ and all the $S \bullet C_l$ and

$$C \vdash W \bullet (S \bullet C_E \cup \bigcup_{l \in \text{dom}(f)} S \bullet C_l).$$

Clearly,

$$R \supseteq W(SR_E \cup \bigcup_{l \in \text{dom}(f)} SR_l), \quad A \supseteq W(SA_E \cup \bigcup_{l \in \text{dom}(f)} SA_l).$$

All that remains to be shown is that $h + h_E$ is well-defined, S is defined on $\langle h + h_E, \mathcal{Z} \rangle$, W is defined on $S\langle h + h_E, \mathcal{Z} \rangle$, and that $\langle h_1 + h_2, \mathcal{Z}' \rangle = W(S\langle h + h_E, \mathcal{Z} \rangle)$.

By Lemma 4.6.2, $h_E = \phi$. Thus, $\langle h + h_E, \mathcal{Z} \rangle = \langle h, \mathcal{Z} \rangle$, and is well-formed. Then, $V\langle h + h_E, \mathcal{Z} \rangle = \langle h; V + \text{left}(V\langle \phi, \mathcal{Z} \rangle), \text{right}(V\langle \phi, \mathcal{Z} \rangle) \rangle = \langle h_1 + h_2, \mathcal{Z} \rangle$. Since, by assumption, $h_1 + h_2$ is well-defined, so is $V\langle h + h_E, \mathcal{Z} \rangle$. Then, so is $S\langle h + h_E, \mathcal{Z} \rangle$ and $W(S\langle h + h_E, \mathcal{Z} \rangle)$. Furthermore, $W(S\langle h + h_E, \mathcal{Z} \rangle) = \langle h_1 + h_2, \mathcal{Z} \rangle$.

Thus, $R | C, A \supset \langle f, E \rangle : \gamma$ is an instance of $GE(\langle f, E \rangle)$ by substitution W , proving the theorem. \blacksquare

4.7 Relating GE to the Original Typing System

In this chapter, we have shown that the type inference algorithm GE is sound and complete with respect to the R -typing system. However, the reader may wonder how GE relates to the *original* typing system defined in Chapter 3.

If we say that an *unrestricted typing* $C, A \supset M : \sigma$ is an instance of $R' | C', A' \supset M : \sigma'$ by substitution S whenever S is defined on R' and $C, A \supset M : \sigma$ is an instance of $C', A' \supset M : \sigma'$ by S , we have the following corollaries for soundness and completeness.

Corollary 4.7.1 *Suppose that $GE(M)$ succeeds and produces a restricted typing statement $R | C, A \supset M : \sigma$. If $C', A' \supset M : \sigma'$ is an instance of $R | C, A \supset M : \sigma$, then $\vdash C', A' \supset M : \sigma'$.*

The proof follows easily by Theorem 4.5.2 and Lemma 3.6.7.

Corollary 4.7.2 *If $\vdash C, A \supset M : \sigma$, then $GE(M)$ succeeds and produces a restricted typing statement with $C, A \supset M : \sigma$ as an instance.*

The corollary is proved by noting that if $C, A \supset M : \sigma$ is provable, then there is a proof using some set of cut formulas.

Therefore, given an expression M , GE computes a most general *restricted* typing statement for M such that $C, A \supset M : \sigma$ is provable in the original typing system iff it is an instance of this most general restricted typing statement.

Appendix A

Unification: Algorithm and Proof of Correctness

This appendix is devoted to proving Lemma 4.2, which states that an algorithm UNIFY exists that, given a set E of equations between types and a co-infinite set \mathcal{Q} of type variables and row variables, computes a most general unifier for E of equations with respect to \mathcal{Q} . In this appendix, we present the algorithm UNIFY, which uses an algorithm UNIFIER that we also present here. In order to prove that UNIFY is correct, we use some properties about most general unifiers for equations between certain types. We prove these properties in Lemmas A.1, A.2, and A.3, and then use these lemmas to prove Lemma 4.2.

Our algorithm UNIFIER is quite similar to the unification algorithm of [Wan87]; however, it fixes a bug in Wand's algorithm for the case of unifying two extended records. As we will discuss in the proof of Lemma 4.2, Wand's termination proof is incorrect, since the halting measure does not necessarily decrease after every iteration of the algorithm. We fix this bug by using some of the properties of unification developed in Lemma A.3 in defining our algorithm UNIFIER, and we prove that UNIFIER does terminate.

The algorithm UNIFIER is defined as follows:

$$\text{UNIFIER}(\phi, \mathcal{Q}) = \phi$$

$$\begin{aligned} \text{UNIFIER}(E' \cup \{c_1 = c_2\}, \mathcal{Q}) = \\ \text{if } c_1 \neq c_2 \text{ then } \textit{fail} \\ \text{else } \text{UNIFIER}(E', \mathcal{Q}) \end{aligned}$$

$$\begin{aligned} \text{UNIFIER}(E' \cup \{\sigma_1 \rightarrow \sigma_2 = \tau_1 \rightarrow \tau_2\}, \mathcal{Q}) = \\ \text{UNIFIER}(E' \cup \{\sigma_1 = \tau_1, \sigma_2 = \tau_2\}, \mathcal{Q}) \end{aligned}$$

$$\begin{aligned} \text{UNIFIER}(E' \cup \{t = \tau\}, \mathcal{Q}) = \\ \text{if } t = \tau \text{ then } \text{UNIFIER}(E', \mathcal{Q} \cup \{t\}) \\ \text{else if } t \text{ occurs in } \tau \text{ then } \textit{fail} \\ \text{else let } \mathcal{Q}' = \mathcal{Q} \cup \{t\} \cup \{\text{the type and row variables in } \tau\} \\ \text{in } [\tau/t]; \text{UNIFIER}([\tau/t]E', \mathcal{Q}') \end{aligned}$$

$$\begin{aligned} \text{UNIFIER}(E' \cup \{\langle h, \text{NULL} \rangle = \langle h', \text{NULL} \rangle\}, \mathcal{Q}) = \\ \text{if } \text{dom}(h) \neq \text{dom}(h') \text{ then } \textit{fail} \\ \text{else } \text{UNIFIER}(E' \cup \bigcup_{l \in \text{dom}(h)} \{h(l) = h'(l)\}, \mathcal{Q}) \end{aligned}$$

$$\begin{aligned}
& \text{UNIFIER}(E' \cup \{\langle h, \mathcal{X} \rangle = \langle h', \mathcal{X}' \rangle\}, \mathcal{Q}) = \\
& \quad \text{if } \text{dom}(h) \neq \text{dom}(h') \text{ then } \text{fail} \\
& \quad \text{else UNIFIER}(E' \cup \bigcup_{l \in \text{dom}(h)} \{h(l) = h'(l)\}, \mathcal{Q} \cup \{\mathcal{X}'\}) \\
\\
& \text{UNIFIER}(E' \cup \{\langle h, \mathcal{X} \rangle = \langle h', \text{NULL} \rangle\}, \mathcal{Q}) = \\
& \quad \text{if } \text{dom}(h) \not\subseteq \text{dom}(h') \text{ then } \text{fail} \\
& \quad \text{else let } h_1 = h' \upharpoonright (\text{dom}(h') - \text{dom}(h)) \\
& \quad \quad \text{in if } \mathcal{X} \text{ occurs in } \langle h_1, \text{NULL} \rangle \text{ then } \text{fail} \\
& \quad \quad \text{else let } \mathcal{Q}' = \mathcal{Q} \cup \{\mathcal{X}'\} \cup \{\text{the type and row variables in the range of } h_1\} \\
& \quad \quad \quad \text{in } [\langle h_1, \text{NULL} \rangle / \mathcal{X}]; \text{UNIFIER}([\langle h_1, \text{NULL} \rangle / \mathcal{X}](E' \cup \bigcup_{l \in \text{dom}(h)} \{h(l) = h'(l)\}), \mathcal{Q}') \\
\\
& \text{UNIFIER}(E' \cup \{\langle h, \mathcal{X} \rangle = \langle h', \mathcal{X}' \rangle\}, \mathcal{Q}) = \quad (\text{where } \mathcal{X} \neq \mathcal{X}') \\
& \quad \text{let } h_1 = h' \upharpoonright (\text{dom}(h') - \text{dom}(h)) \\
& \quad \quad h_2 = h \upharpoonright (\text{dom}(h) - \text{dom}(h')) \\
& \quad \quad \mathcal{Q}_E = \text{the type and row variables in } E' \cup \{\langle h, \mathcal{X} \rangle = \langle h', \mathcal{X}' \rangle\} \\
& \quad \quad \mathcal{Y} \text{ be a fresh row variable with respect to } \mathcal{Q} \cup \mathcal{Q}_E \\
& \quad \quad \text{in if } \mathcal{X} \text{ occurs in } \langle h_1, \mathcal{Y} \rangle \text{ or } \mathcal{X}' \text{ occurs in } \langle h_2, \mathcal{Y} \rangle \\
& \quad \quad \quad \text{OR} \\
& \quad \quad \quad \text{if } \mathcal{X} \text{ occurs in } \langle h_2, \mathcal{Y} \rangle \text{ and } \mathcal{X}' \text{ occurs in } \langle h_1, \mathcal{Y} \rangle \\
& \quad \quad \quad \text{then } \text{fail} \\
& \quad \quad \quad \text{else let } V = [\langle h_1, \mathcal{Y} \rangle / \mathcal{X}, \langle h_2, \mathcal{Y} \rangle / \mathcal{X}']; [\langle h_1, \mathcal{Y} \rangle / \mathcal{X}, \langle h_2, \mathcal{Y} \rangle / \mathcal{X}'] \\
& \quad \quad \quad \quad \mathcal{Q}' = \mathcal{Q} \cup \{\mathcal{X}, \mathcal{X}'\} \cup \{\text{the type and row variables in the range of } h_1 \text{ or } h_2\} \\
& \quad \quad \quad \quad \text{in } V; \text{UNIFIER}(V(E' \cup \bigcup_{l \in \text{dom}(h) \cap \text{dom}(h')} \{h(l) = h'(l)\}), \mathcal{Q}') \\
\\
& \text{UNIFIER}(E' \cup \{\sigma = \tau\}, \mathcal{Q}) = \\
& \quad \text{if } \sigma = \tau \text{ is an equation between any other types} \\
& \quad \quad \text{then } \text{fail}
\end{aligned}$$

Using this algorithm UNIFIER, we define our algorithm UNIFY, which simply checks that the substitution returned by UNIFIER is defined on all the types in the original set of equations. We define UNIFY as follows:

$$\begin{aligned}
& \text{UNIFY}(E, \mathcal{Q}) = \\
& \quad \text{let } S = \text{UNIFIER}(E, \mathcal{Q}) \\
& \quad \text{if, for all } \sigma = \tau \in E, S \text{ is defined on } \sigma \text{ and } \tau \\
& \quad \quad \text{then return } S \\
& \quad \quad \text{else } \text{fail}
\end{aligned}$$

Before proceeding to prove the correctness of UNIFY, we first develop some properties of unification that we will need in that proof. The first such property concerns unifying a type variable with a type.

Lemma A.1 *A substitution T unifies $E' \cup \{t = \tau\}$ iff $t = \tau$ and T unifies E' OR*

1. $t \neq \tau$ and

2. t does not occur in τ and
3. there exists a substitution T' such that $[\tau/t]; T'$ is defined, and $T = [\tau/t]; T'$, and T' unifies $[\tau/t]E'$.

We omit the proof, as it is quite similar to [Rob65].

We also need the following property about unifying an extended record with a fixed record.

Lemma A.2 *A substitution T unifies $E' \cup \{\langle h, \mathcal{X} \rangle = \langle h', NULL \rangle\}$ iff*

1. $dom(h) \subseteq dom(h')$ and
2. \mathcal{X} does not occur in $\langle h_1, NULL \rangle$, where $h_1 = h' \upharpoonright (dom(h') - dom(h))$ and
3. $[\langle h_1, NULL \rangle / \mathcal{X}](E' \cup \bigcup_{l \in dom(h)} \{h(l) = h'(l)\})$ is defined and
4. there exists a substitution T' such that $[\langle h_1, NULL \rangle / \mathcal{X}]; T'$ is defined and $T = [\langle h_1, NULL \rangle / \mathcal{X}]; T'$ and T' unifies $[\langle h_1, NULL \rangle / \mathcal{X}](E' \cup \bigcup_{l \in dom(h)} \{h(l) = h'(l)\})$

Again, we omit the proof, as it uses the same sort of reasoning as the proof of the Lemma A.1.

The final property that we will need concerns unifying two extended records.

Lemma A.3 *Suppose that $E = E' \cup \{\langle h, \mathcal{X} \rangle = \langle h', \mathcal{X}' \rangle\}$. Let \mathcal{Q} be a co-infinite set of type variables and row variables, and let \mathcal{Q}_E be the set of type variables and row variables occurring in E . Also, let $V = [\langle h_1, \mathcal{Y} \rangle / \mathcal{X}, \langle h_2, \mathcal{Y} \rangle / \mathcal{X}']; [\langle h_1, \mathcal{Y} \rangle / \mathcal{X}, \langle h_2, \mathcal{Y} \rangle / \mathcal{X}']$, for some row variable \mathcal{Y} that is fresh with respect to $\mathcal{Q} \cup \mathcal{Q}_E$. A substitution T unifies E iff*

1. \mathcal{X} does not occur in $\langle h_1, \mathcal{Y} \rangle$, where $h_1 = h' \upharpoonright (dom(h') - dom(h))$ and
2. \mathcal{X}' does not occur in $\langle h_2, \mathcal{Y} \rangle$, where $h_2 = h \upharpoonright (dom(h) - dom(h'))$ and
3. either \mathcal{X} does not occur in $\langle h_2, \mathcal{Y} \rangle$ or \mathcal{X}' does not occur in $\langle h_1, \mathcal{Y} \rangle$ and
4. V is defined and
5. $V(E' \cup \bigcup_{l \in dom(h) \cap dom(h')} \{h(l) = h'(l)\})$ is defined and
6. there exists a substitution T' such that $V; T'$ is defined and $T = \mathcal{Q} \cup \mathcal{Q}_E V; T'$ and T' unifies $V(E' \cup \bigcup_{l \in dom(h) \cap dom(h')} \{h(l) = h'(l)\})$ and
7. T is defined on $\langle h, \mathcal{X} \rangle$ and $\langle h', \mathcal{X}' \rangle$.

Again, we omit the proof here.

Using the lemmas developed above, we now give the proof of Lemma 4.2.

Lemma 4.2 *Let E be a set of equations of the form $\sigma = \tau$ and let \mathcal{Q} be a co-infinite set of type variables and row variables. Let \mathcal{Q}_E be the set of type variables and row variables that occur in E . There exists an algorithm UNIFY such that whenever there is a unifying substitution for E , UNIFY(E, \mathcal{Q}) produces a most general unifying substitution with respect to $\mathcal{Q} \cup \mathcal{Q}_E$. Otherwise, UNIFY(E, \mathcal{Q}) fails.*

Proof. In order to prove the lemma, it is sufficient to show the following three properties of UNIFIER:

1. UNIFIER(E, \mathcal{Q}) halts.
2. If there is a substitution T that unifies E , then UNIFIER(E, \mathcal{Q}) succeeds and produces a substitution S such that S unifies E . Furthermore, there exists a substitution U such that $S;U$ is defined and $T = \mathcal{Q} \cup \mathcal{Q}_E S;U$.
3. If there is no unifying substitution for E , then UNIFIER(E, \mathcal{Q}) either *fails* or produces a substitution S such that for some $\sigma = \tau \in E$, S is not defined on at least one of σ and τ .

In order to prove (1), we define the *degree* of a set E to be the pair $\langle m, n \rangle$, where m is the number of *distinct* type variables and row variables that occur in E , and n is the number of occurrences of $\rightarrow, \langle \rangle$ plus the total number of (possibly non-distinct) type variables, row variables, ground types, and *NULL* that occur in E . We say that $\langle m, n \rangle$ is smaller than $\langle m', n' \rangle$ if either $m < m'$, or $m = m'$ and $n < n'$. We will show that each clause of the algorithm reduces the degree of the set E that is passed around. We note here that a set E has degree $\langle 0, 0 \rangle$ iff E is empty.

The proof proceeds by induction on the structure of $\sigma = \tau$, where $E = E' \cup \{\sigma = \tau\}$, and $\langle m, n \rangle$ is the degree of E .

Suppose that $\sigma = \tau$ is of the form $c_1 = c_2$. To prove (1), we note that, clearly, the degree of E' is $\langle m, n - 2 \rangle$. To prove (2) and (3), we note that the degree of E' is smaller than the degree of E , and that $\mathcal{Q}_E = \mathcal{Q}_{E'}$. The proof of (2) and (3) then follows easily from the inductive hypothesis.

Suppose that $\sigma = \tau$ is of the form $\sigma_1 \rightarrow \sigma_2 = \tau_1 \rightarrow \tau_2$. To prove (1), we observe that, clearly, the degree of E' is $\langle m, n - 2 \rangle$. To prove (2) and (3), we first note that a substitution T unifies E iff T unifies $E' \cup \{\sigma_1 = \tau_1, \sigma_2 = \tau_2\}$. We also note that the degree of $E' \cup \{\sigma_1 = \tau_1, \sigma_2 = \tau_2\}$ is smaller than the degree of E , and that both of these sets contain the same type variables and row variables. The proof of (2) and (3) again follows easily from the inductive hypothesis.

Suppose that $\sigma = \tau$ is of the form $\langle h, NULL \rangle = \langle h', NULL \rangle$. To prove (1), we observe that, clearly, the degree of E' is $\langle m, n - 4 \rangle$. To prove (2) and (3), we first note that a substitution T unifies E iff $\text{dom}(h) = \text{dom}(h')$ and T unifies $(E' \cup \bigcup_{l \in \text{dom}(h)} \{h(l) = h'(l)\})$. We also note that the degree of $(E' \cup \bigcup_{l \in \text{dom}(h)} \{h(l) = h'(l)\})$ is smaller than the degree of E , and that both of these sets contain the same type variables and row variables. The proof of (2) and (3) again follows easily from the inductive hypothesis.

Suppose that $\sigma = \tau$ is of the form $\langle h, \mathcal{X} \rangle = \langle h', \mathcal{X} \rangle$. To prove (1), we note that, clearly, the degree of E' is $\langle m, n - 4 \rangle$. To prove (2) and (3), we first note that a substitution T unifies E iff $\text{dom}(h) = \text{dom}(h')$ and T unifies $(E' \cup \bigcup_{l \in \text{dom}(h)} \{h(l) = h'(l)\})$ and T is defined on $\langle h, \mathcal{X} \rangle$ and $\langle h', \mathcal{X} \rangle$. We also note that the degree of $(E' \cup \bigcup_{l \in \text{dom}(h)} \{h(l) = h'(l)\})$ is smaller than the degree of E , and that $\mathcal{Q}_E = \mathcal{Q}_{(E' \cup \bigcup_{l \in \text{dom}(h)} \{h(l) = h'(l)\})} \cup \{\mathcal{X}\}$. The proof of (2) and (3) then follows easily from the inductive hypothesis.

Suppose that $\sigma = \tau$ is of the form $t = \tau$. To prove (1), we first note that if UNIFIER succeeds then either $t = \tau$ or t does not occur in τ . In the first case, the degree of E' is clearly $\langle m, n - 2 \rangle$; in the second case, the first component of the degree of $[\tau/t]E'$ is at most $m - 1$. To prove (2) and (3), we use Lemma A.1 and again proceed by case analysis. In the case that $t = \tau$, we note that the degree of E' is less than the degree of E , and that $\mathcal{Q}_E = \mathcal{Q}_{E'} \cup \{t\}$. In the case that $t \neq \tau$, we note that the degree of $[\tau/t]E'$ is less than the degree of E , and that $\mathcal{Q}_E = \mathcal{Q}_{[\tau/t]E'} \cup \{t\} \cup \{\text{the type and row variables in } \tau\}$. Using

these properties and Lemma A.1, the proofs of (2) and (3) for both cases follow from the inductive hypothesis.

Suppose that $\sigma = \tau$ is of the form $\langle h, \mathcal{X} \rangle = \langle h', NULL \rangle$. To prove (1), we observe that if UNIFIER succeeds, then the first component of the degree of $[\langle h_1, NULL \rangle / \mathcal{X}]E'$ is at most $m - 1$. To prove (2) and (3), we use Lemma A.2. We first note that the degree of $[\langle h_1, NULL \rangle / \mathcal{X}]E'$ is less than the degree of E , and that $\mathcal{Q}_E = \mathcal{Q}_{[\langle h_1, NULL \rangle / \mathcal{X}]E'} \cup \{\mathcal{X}\} \cup \{\text{the type and row variables in the range of } h_1\}$. To prove (2), we also note that, if there is a substitution that unifies E , then, since $\langle h_1, NULL \rangle$ contains only type and row variables from \mathcal{Q}_E , Lemma A.2 implies that $[\langle h_1, NULL \rangle / \mathcal{X}; \text{UNIFIER}([\langle h_1, NULL \rangle / \mathcal{X}](E' \cup \bigcup_{l \in \text{dom}(h)} \{h(l) = h'(l)\}), \mathcal{Q}')] is defined. Using these properties and Lemma A.2, the proofs of (2) and (3) follow from the inductive hypothesis.$

Suppose that $\sigma = \tau$ is of the form $\langle h, \mathcal{X} \rangle = \langle h', \mathcal{X}' \rangle$. To prove (1), we observe that if UNIFIER succeeds, then the first component of the degree of $V(E')$ is at most $m - 1$. To prove (2) and (3), we use Lemma A.3. We first note that the degree of $V(E')$ is less than the degree of E , and that $\mathcal{Q}_E = \mathcal{Q}_{[\langle h_1, NULL \rangle / \mathcal{X}]E'} \cup \{\text{the type and row variables in the range of } h_1 \text{ or } h_2\} \cup \{\mathcal{X}, \mathcal{X}'\}$. To prove (2), we also note that, if there is a substitution T that unifies E , then, since $\langle h_1, \mathcal{Y} \rangle$ and $\langle h_2, \mathcal{Y} \rangle$ contain only type and row variables from \mathcal{Q}_E , Lemma A.3 implies that $V; \text{UNIFIER}([\langle h_1, NULL \rangle / \mathcal{X}](E' \cup \bigcup_{l \in \text{dom}(h) \cap \text{dom}(h')} \{h(l) = h'(l)\}), \mathcal{Q}')$ is defined. By the inductive hypothesis, we can deduce that this substitution is equal to T with respect to $\mathcal{Q} \cup \mathcal{Q}_E$, and thus this substitution is defined on $\langle h, \mathcal{X} \rangle$ and $\langle h', \mathcal{X}' \rangle$. Using these properties and Lemma A.3, the proofs of (2) and (3) follow from the inductive hypothesis, proving the lemma.

As mentioned at the beginning of the appendix, our algorithm UNIFIER corrects a bug in the unification algorithm of [Wan87]. Wand's algorithm uses the substitution

$$[\langle h_1, \mathcal{Y} \rangle / \mathcal{X}, \langle h_2, \mathcal{Y} / \mathcal{X}' \rangle]$$

instead of the substitution

$$[\langle h_1, \mathcal{Y} \rangle / \mathcal{X}, \langle h_2, \mathcal{Y} / \mathcal{X}' \rangle]; [\langle h_1, \mathcal{Y} \rangle / \mathcal{X}, \langle h_2, \mathcal{Y} / \mathcal{X}' \rangle]$$

which we call V . However, applying the simpler substitution used by Wand to the set $(E' \cup \bigcup_{l \in \text{dom}(h) \cap \text{dom}(h')} \{h(l) = h'(l)\})$ does not necessarily decrease the halting measure, since \mathcal{X}' may occur in h_1 or \mathcal{X} may occur in h_2 . We correct this difficulty in Wand's algorithm through our use of the substitution V . ■

Bibliography

- [BCGS89] V. Breazu-Tannen, T. Coquand, C. Gunter, and A. Scedrov. Inheritance and explicit coercion. In *4-th IEEE Symp. Logic in Computer Science*, pages 112-133, June 1989.
- [BL88] K. Bruce and G. Longo. A modest model of records, inheritance and bounded quantification. In *3-rd IEEE Symp. Logic in Computer Science*, pages 38-50, June 1988.
- [Car84] L. Cardelli. A semantics of multiple inheritance. In *Proc. Int. Symp. on Semantics of Data Types, Sophia-Antipolis (France), Springer LNCS 173*, pages 51-68, June 1984.
- [CCHMO89] P. Canning, W. Cook, W. Hill, J. Mitchell, and W. Olthoff. F-bounded polymorphism for object-oriented programming. In *Proc. 4-th Int. Conference of Functional Programming and Architecture*, pages (to appear), September 1989.
- [CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471-522, 1985.
- [Coo89] W. Cook. A denotational definition of inheritance. Ph.D. Thesis, Brown University, March 1989.
- [GR83] A. Goldberg and D. Robson. *Smalltalk-80: The language and its implementation*. Addison Wesley, 1983.
- [FM88] Y. Fuh and P. Mishra. Polymorphic subtype inference: closing the theory-practice gap. Manuscript, 1988.
- [JM88] L. Jategaonkar and J.C. Mitchell. ML with Extended Pattern Matching and Subtypes (prelimary version). In *Proceedings of ACM Conference on LISP and Functional Programming*, pages 198-211, July 1988.
- [Kam88] S. Kamin. Inheritance in smalltalk-80: a denotational definition. In *Proc. 15-th ACM Symp. on Principles of Programming Languages*, pages 80-87, January 1988.
- [Mil78] R. Milner. A theory of type polymorphism in programming. *JCSS*, 17, 1978. pages 348-375.
- [Mil85] R. Milner. The standard ml core language. *Polymorphism*, 2(2), 1985. 28 pages. An earlier version appeared in Proc. 1984 ACM Symp. on Lisp and Functional Programming.

- [Mit84] J.C. Mitchell. Coercion and type inference (summary). In *Proc. 11-th ACM Symp. on Principles of Programming Languages*, pages 175–185, January 1984.
- [OB88] A. Ohori and P. Buneman. Type inference in a database programming language. In *Proceedings of ACM Conference on LISP and Functional Programming*, pages 174–183, July 1988.
- [Red88] U. Reddy. Objects as closures: abstract semantics of object-oriented languages. In *Proceedings of ACM Conference on LISP and Functional Programming*, pages 289–297, July 1988.
- [Rem89] D. Remy. Typechecking records and variants in a natural extension of ML. In *Proc. 16-th ACM Symp. on Principles of Programming Languages*, pages 77–88, January 1989.
- [Rey80] J.C. Reynolds. *Using Category Theory to Design Implicit Conversions and Generic Operators*, pages 211–2580. Springer-Verlag Lecture Notes in Computer Science, Vol. 94, 1980.
- [Rob65] J.A. Robinson. A machine oriented logic based on the resolution principle. *JACM*, 12(1):23–41, 1965.
- [Sta88] R. Stansifer. Type inference with subtypes. In *Proc. 15-th ACM Symp. on Principles of Programming Languages*, pages 88–97, January 1988.
- [Str86] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.
- [Wan87] M. Wand. Complete type inference for simple objects. In *Proc. 2-nd IEEE Symp. on Logic in Computer Science*, pages 37–44, June 1987.
- [Wand88] M. Wand. Corrigendum: Complete type inference for simple objects. In *Proc. 3-rd IEEE Symp. on Logic in Computer Science*, page 132, June 1988.
- [Wand89] M. Wand. Type inference for record concatenation and multiple inheritance. In *Proc. 4-th IEEE Symp. on Logic in Computer Science*, pages 92–97, June 1989.