A Study On Semantics, Types, and Languages For Databases and Object-Oriented Programming

MS-CIS-89-60
LOGIC & COMPUTATION 15

Atsushi Ohori

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104

October 1989

ACKNOWLEDGMENTS:
This research was supported in part by grants NSF IRI86-10617, ARO DAA6-29-84-K-0061, ONR N00014-88-K-0634 and OKI Electric Industry Co., Japan

A STUDY OF SEMANTICS , TYPES AND LANGUAGES FOR DATABASES AND OBJECT-ORIENTED PROGRAMMING

ATSUSHI OHORI

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy.

1989

[illegible]

Peter Buneman

Co-supervisor of Dissertation

Val Breazu-Tannen

Val Breazu-Tannen

Co-supervisor of Dissertation

Jean H Gallier

Jean Gallier

Graduate Group Chairperson

ABSTRACT

A STUDY OF SEMANTICS, TYPES AND LANGUAGES FOR DATABASES AND OBJECT-ORIENTED PROGRAMMING

ATSUSHI OHORI

CO-SUPERVISORS : PETER BUNEMAN AND VAL BREAZU-TANNEN

The purpose of this thesis is to investigate a type system for databases and object-oriented programming and to design a statically typed programming language for these applications. Such a language should ideally have a static type system that supports:

- polymorphism and static type inference,
- rich data structures and operations to represent various data models for databases including the relational model and more recent complex object models,
- central features of object-oriented programming including user definable class hierarchies, multiple inheritance, and data abstraction,
- the notion of extents and object-identities for object-oriented databases.

Without a proper formalism, it is not obvious that the construction of such a type system is possible. This thesis attempts to construct one such formalism and proposes a programming language that uniformly integrate all of the above features.

The specific contributions of this thesis include:

- A simple semantics for ML polymorphism and axiomatization of the equational theory of ML.
- A uniform generalization of the relational model to arbitrary complex database objects that are constructed by labeled records, labeled variants, finite sets and recursive definition.
- A framework for semantics of types for complex database objects.
- The notion of *conditional typing schemes* that extends Milner's typing scheme for ML to a wide range of complex structures and operations.
- A formulation of the notion of classes and inheritance in an ML style static type system.

- The notion of views that enable us to represent object-oriented databases in an ML style static type system.
- A proposal of a polymorphic programming language, Machiavelli, for databases and object-oriented programming with static type inference.

In addition to the above technical contributions, a prototype implementation of Machiavelli that embodies most of the features presented in this thesis has been done in Standard ML of New Jersey.

ACKNOWLEDGEMENTS

I am indebted to my co-adviser Peter Buneman. He originally introduced me the area of types and semantics for database programming languages and recommended me to undertake the doctoral study. Without his enthusiastic encouragement, this dissertation would have never existed. Special thanks are also to my co-adviser Val Breazu-Tannen for his guiding me the are of syntax and semantics of types for programming languages and for his encouragements. Moreover, their constructive criticism and our continuous discussion directly contribute to the work.

I am grateful to my external commitee members Tomasz Imielinski and John Mitchell for very useful discussion at several opportune moments and many helpful comments. I am as grateful to the other members of my thesis committee, Jean Gallier, Aravind Joshi, and Dale Miller for their interest in my work and their invaluable comments.

Many thanks are also to Luca Cardelli, Carl Gunter, Richard Hull, David MacQueen, Mitchell Wand and Aaron Watters for very helpful discussions and suggestions. Moreover Mitchell Wand made several constructive criticism on the draft paper on which a part of chapter 5 is based.

I am grateful to OKI Electric Industry Co., Japan. They granted my leave of absence for the study at University of Pennsylvania and provided financial support. Among them, special thanks are to Kenji Tateoka, Eiji Nohara, and Toshiyuki Matsumoto who gave me assistance and support at critical moments.

Finally my special gratitude goes to my wife Hiroko. Without her, the endevour of graduate study would have been arid and trying experience.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivations and Purposes

The term "impedance mismatch" has been coined [75, 13] to describe the phenomenon that the structures and operations available in a programming language do not usually match those needed for database systems. This problem is painfully familiar to anyone who has used a high-level programming language to communicate with a database. This mismatch is particularly unfortunate when database programming cannot share the benefits of recent developments in the theory of types in programming languages. Among them most important ones are ***polymorphism*** [94, 78] and ***static type inference*** [78, 34], which should have had apparent practical advantages for many database applications. A similar situation exists in the field of object-oriented programming. In the development of object-oriented languages [44, 33], various practical ideas such as *classes* and ***multiple inheritance*** have been proposed and implemented. The advantages of these features seem orthogonal to those of conventional notion of types and the integration of them into a type system of a programming language is highly desirable. However, a proper formalism that enables us to integrate these features in a type system with polymorphism and static type inference has not yet been well established.

The motivation of this thesis is to attempt to solve these mismatch problems and to develop a programming language that unifies databases and object-oriented programming in a modern type system. Such a language should provide a programming environment where the programmer can enjoy both the capability of database management and the desirable features of object-oriented programming with all the benefits of a modern type system. Recent studies in the area of object-oriented databases [13, 17] and "semantic" data models [58] suggest that such integration is highly desirable. Such a language should be also suitable for many other applications whose main interest is manipulation of highly structured data such as knowledge representation [18] and natural language processing [97]. It is therefore hoped that the integration should also contribute to solve the "high-level" impedance mismatches between database systems and other applications.

One desirable feature of a programming language for those data intensive application is *static type-checking.* The main objective of static type-checking is to detect inconsistency in applications of operations to data *before* program execution. This eliminates certain programming errors at early stages of programming development. This should be particularly important for data intensive applications. Data structures such as schemes in database systems and class definitions in object-oriented programming are large and complex structures. Much of programming errors in these applications would show up as *type errors* were those data structures a part of the type structure of the program. Therefore a type system in which such errors can be anticipated by a static analysis of the program is, I believe, a prerequisite for a good programming language for data intensive applications. As argued in [106], static type-checking may also contributes to efficient execution of programs by eliminating run-time type-checking.

Until recently, statically typed languages were justly criticized for being too cumbersome. First, they did not allow certain types of generic code; and second the type declarations, while admittedly useful as documentation, were often tedious and obvious. With the inventions of polymorphism [94, 78] and static type inference [78, 34], however, I believe that a static type system can overcome these disadvantages. The ML family of languages – including Standard ML [48] and Miranda [107] – are successful examples. In those languages, the programmer is not required to specify types of programs. The type system *infers* a most general polymorphic type for any type correct programs. By this mechanism, those languages achieve much of the flexibility of dynamically typed languages like Lisp without sacrificing the advantages of the static type-checking. Moreover, I believe that an ML style type system can achieve a proper integration of databases and object-oriented systems in a statically typed programming language if it is extended with the following features:

1. records and variants to represent various data structures,

2. the structures and operations rich enough to represent various data models for databases including the relational model [29], nested relations [36, 62, 95] and complex object models [59, 14, 4],

3. user definable classes, data abstraction and multiple inheritance for object-oriented programming,

4. object-identities and extents for object-oriented data models [13, 17].

The last two years have seen considerable research into the integration of records and variants in an ML style type system to support an aspect of object-oriented programming [111, 105, 85, 63, 38, 93] – including a contribution of a part of this study ([85]) – which shows that the integration of the first is now possible. However, there seems no existing approach that integrates the other three features in an ML style type system. For 2, the problem is seen by simply noting that no existing polymorphic type system can represent even the relational model – perhaps the simplest form of a data model for databases. As pointed out in [10], no existing type system can type-check the polymorphic *natural join* operation. For 3, there have been some efforts [7] and suggestions [63] toward the integration of user definable classes, data abstraction and inheritance in a static type system. However, to my knowledge, there is no formal system that integrates these features in an ML style type inference system. Moreover, there appears to be no static type system of any kind that successfully deals with 4. The purpose of this thesis is to develop an ML style type system that uniformly integrates all the above features and to propose a programming language having that type system.

It is also desirable for the language to have a clean mathematics semantics. Such a semantics should provide a better understanding of the interaction of various features of the language and should be useful for further extensions of the language. This thesis also attempts to construct a semantic framework for the polymorphic core of the proposed language.

The following example illustrates the flavor of the language proposed in this thesis. Consider a function which takes a set of records (i.e. a relation) with Name and Salary information and returns the set of all Name values that correspond to Salary values over 100K. For example, applied to the relation (set of records)

$$\begin{array}{l}\{[Name = "Joe", Salary = 23456],\\ \;[Name = "Fred", Salary = 123456],\\ \;[Name = "Helen", Salary = 132000]\}\end{array}$$

this function should yield the set $\{"Fred", "Helen"\}$. Such a function is written in the language as follows:

fun *wealthy*(S) =**select** $x.Name$

where $x \in S$
with $x.Salary > 100000$;

The **select**...**where**...**with** form is simple syntactic sugar for more basic program structure. Although no data types are mentioned in the code, the type system infers the following type information

$$wealthy : \{[(s_1)Salary : int, Name : s_2]\} \rightarrow \{s_2\}$$

which means that *wealthy* is a function that takes a homogeneous set of records, each of type $[(s_1)Salary : int, Name : s_2]$ and returns a homogeneous set of values of type s_2, where s_2 is a type variable ranging over all types on which equality is defined and $[(s_1)Salary : int, Name : s_2]$ is a type variable ranging over arbitrary record types that contains $Salary : int$ and $Name : s_2$ fields. Consequently, the type system will allow *wealthy* to be applied, for example, to relations of type

$$\{[Name : string, Age : int, Salary : int]\}$$

and also to relations of type

$$\{[Name : [First : string, Last : string], Weight : int, Salary : int]\}.$$

Moreover, the type system statically checks the type correctness of each application and computes the result type by instantiating the type variables.

1.2 Organization of Thesis

One of the purposes of this thesis is to design a typed programming language that is rich enough to represent various data structures used in a wide rage of data intensive applications. For such a language, recursively defined types and objects are essential. In order to treat recursive structures uniformly, we use *regular trees* as a mathematical tool to represent them. In order to make this thesis self-contained, I have gathered in chapter 2 standard results about regular trees as well as some standard mathematical notations and definitions. Everything there is standard knowledge. My recommendation to the reader is to go over this chapter quickly to familialize himself notations and then to use this chapter as references when needed.

The rest of the thesis consists of the following five investigation to achieve the purpose stated in the previous section:

1. an analysis of ML and a construction of a semantic framework for ML polymorphism,
2. a type system for complex database objects and its semantics,
3. an extension of ML type inference method to records, variants and complex database objects,
4. a formulation of classes for object-oriented programming,
5. a method to represent object-oriented databases in an ML style type system.

All of these five investigations lead to the proposal of Machiavelli – a polymorphic language for databases and object-oriented programming. The first of them gives a framework to understand ML's syntactic properties and to define a denotational semantics. This will serve as a starting point of the development of Machiavelli. The second investigation formulates the structures of domains for databases and constructs a type system for complex database objects. Then, by combining the above two, the third investigation develops the polymorphic core of Machiavelli as an extension of ML. The rest of the two extend the core language to represent object-oriented systems and object-oriented databases. I think that this thesis is best read by reading from cover to cover going through all of the above five topics in that order. However, in order to reconcile diverse interests of the readers, I organize each of these five investigations as a relatively self-contained chapter (chapter 3 to chapter 7). For this reason, I do not make a independent chapter for backgrounds of this entire thesis but include an introductory section in each chapter giving enough background for each topic. In particular the analysis of ML in chapter 3 and the construction of a framework for database domains in chapter 4 can be read independently. The subsequent developments are of course based on the above two investigations. However, they can be read using the above two chapters as references. I will try to provide appropriate reference keys to major results in chapter 3 and 4. The readers whose main interest is type systems of programming languages and their denotational semantics would start with chapter 3 and then go to chapter 5 and 6. The readers whose main interest is data models and database programming may start with chapter 4 continue on chapter 5 and then go to chapter 7. The readers whose main interest is object-oriented programming and programming language design may go directly to chapter 5 and continue on chapter 6 and 7. In the rest of this section, I outline these five investigations.

In chapter 3, I begin our investigation with ML type system. After analyzing the existing approachs to ML's type inference system and its semantics, an alternative proof system for ML type inference is given. This proof system only requires simple types but is shown to be equivalent (in a very natural sense) to the proof system given by Damas and Milner [34]. A semantic framework for ML polymorphism is then proposed and an axiomatization of the equational theory of ML terms is given. The proposed framework requires no more semantic material than what is needed for modeling the simple type discipline, yet it provides a better account for ML's implicit type system. The axiomatization of the equational theory corresponds exactly to the proposed semantics. The analogs of the completeness theorems that Friedman proved [37] for the simply typed lambda calculus are proved with respect to the proposed semantics. The framework is then extended to languages with various type constructors, constants and recursive types (via regular trees). At the end of the chapter, it is also shown that certain full abstraction result for typed languages can be transferred to the corresponding ML like languages.

Chapter 4 investigates type systems for databases and their semantics and proposes a concrete type system for complex database objects. It starts the investigation with the analysis of the relational model. The relational model is characterized independently of the underlying tuple structure. By generalizing this abstract analysis, the notions of *database type systems* and *database domains* are defined as characterizations of the structures of type systems for databases and their semantics domains. Based on these abstract characterizations, a concrete database type system for complex database objects and its database domain are constructed. The proposed type system allows arbitrary complex structures constructed by labeled records, labeled variants, finite sets and recursive definition. Moreover, it is a proper generalization of the relational model to those complex structures. In addition to standard operations for records, variants and sets, *join* and *projection* are available as polymorphically typed computable functions on those complex structures (even on recursively defined structures).

By combining the results of the previous two chapters, chapter 5 defines a programming language as an extension of ML. This language is the polymorphic core of Machiavelli – the language I propose in this thesis. In order to develop a type inference algorithm for the core language, a new paradigm for type inference – the notion of *conditional typing schemes* – is proposed. By allowing typing schemes to include conditions on substitutions of type variables, it extends Milner's method uniformly to a wide range of structures and operations. Using this mechanism, an algorithm to compute a principal conditional typing scheme is developed. At the end of the chapter, extended examples of database programming are given.

Chapter 6 extends the core language with parameterized class declarations, which allows the programmer to build a hierarchy of classes connected by multiple inheritance declarations. This extension achieves a proper integration of multiple inheritance in object-oriented programming and ML style abstract data types with type parameterization. A formal system to type-check class definitions and to infer typings of raw terms containing method names are defined. It is then shown that the extended language is sound with respect to the type system of the core language by showing the property that if a typing is provable in the extended type system then the typing obtained by unfolding all class names and method names by their implementations is provable in the type system of the core language. The type inference algorithm for the core language defined in the previous chapter is then extended to include classes. It is shown that it still computes a principal conditional typing scheme for any typable raw terms.

Chapter 7 presents a method to represents object-oriented databases. Major properties of objects in object-oriented databases can be captured by the combinations of references and variants. I then introduce the notion of *views* (sets of structures with "identities") and show that various operations in object-oriented databases are naturally represented by the combination of viewing functions and the operation join and projection generalized in chapter 4.

Chapter 8 concludes the thesis and discuss some possible topics for further investigation.

Appendix contains the abstract syntax of Machiavelli (including classes definitions).

Chapter 2

Mathematical Preliminaries

2.1 Basic Notions and Notations

The *domain* and the *range* of a function f are denoted by $dom(f)$ and $ran(f)$ respectively. We write $f : A \longrightarrow B$ for a function f such that $dom(f) = A$ and $ran(f) = B$. If f is a function and $A \subseteq dom(f)$ then $f(A)$ is the set $\{f(a) | a \in A\}$. The *restriction* of a function f to a set $A \subseteq dom(f)$, denoted by $f{\restriction}^{A}$, is the function f' such that $dom(f') = A$ and $f'(a) = f(a)$ for all $a \in A$. We write $\{x_1 := v_1, \ldots, x_n := v_n\}$ for the function f such that $dom(f) = \{x_1, \ldots, x_n\}$ and $f(x_i) = v_i$ $(1 \leq i \leq n)$ (assuming that all x_i are distinct). For a function $f : A \longrightarrow B$, $f\{x := v\}$ is the function f' such that $dom(f') = A \cup \{x\}$, $f(x) = v$ and $f'(a) = f(a)$ for all a such that $a \neq x$. If f, g are functions such that $ran(g) \subseteq dom(f)$ then $f \circ g$ is the *composition* of f and g defined as $dom(f \circ g) = dom(g)$ and $f \circ g(x) = f(g(x))$ for all $x \in dom(g)$.

For a set A, $\boldsymbol{P}(A)$ is the set of all finite subset of A. If A is a finite set then we denote by $|A|$ the cardinality of A. Let A, B be a sets. The *product* of A and B, denoted by $A \times B$, is the set $\{(x, y) | x \in A, y \in B\}$. If A is the singleton set $\{x\}$ then we write $x \times B$ for $\{x\} \times B$. Similarly for $B \times x$.

A *relation* on a set A is a subset of $A \times A$. If r is a relation on A and $x, y \in A$, we usually write $x\ r\ y$ to denote $(x, y) \in r$. Let r be a relation on a set A, x, y, z be any elements in A. r is *reflexive* iff $x\ r\ x$. r is *symmetric* iff $x\ r\ y$ implies $y\ r\ x$. r is *antisymmetric* iff $x\ r\ y$ and $y\ r\ x$ implies $x = y$. r is *transitive* iff $x\ r\ y$ and $y\ r\ z$ implies $x\ r\ z$.

A *preorder* on a set A is a reflexive transitive relation on A. A *partial order* (or *oredering*) is an antisymmetric preorder. A partial order r on A is a *linear order* iff for any $x, y \in A$ either $x\ r\ y$ or $y\ r\ x$.

Definition 2.1 (Preordered Sets and Partially Ordered Sets) *A preordered set $(P, \preceq)$ is a set P endowed with a preorder $\preceq$ on P. A partially ordered set (or poset) $(P, \sqsubseteq)$ is a set P endowed with a partial order $\sqsubseteq$ on P.*

When the preorder (partial order) $\preceq$ ($\sqsubseteq$) is understood, we sometimes write P for $(P, \preceq)$ ($(P, \sqsubseteq)$) and say a preordered set (a poset) P.

Let $(P, \preceq)$ be a preordered set and $A \subseteq P$. An element $p \in P$ is an *upper bound* of A iff $a \preceq p$ for all $a \in A$. An element p is a *least upper bound* of A iff for any element of $x \in P$, $p \preceq x$ iff x is an upper bound of A. An element $p \in P$ is a *lower bound* of A iff $p \preceq a$ for all $a \in A$. An element p is a *greatest lower bound* of A iff for any element of $x \in P$, $x \preceq p$ iff x is a lower bound of A. The following property is an immediate consequence of the definition:

Proposition 2.1 *In a poset, least upper bounds and greatest lower bounds are unique.* ▮

Let $(P, \sqsubseteq)$ be a poset and $A \subseteq P$. We write $\sqcup A$ for the least upper bound of A and $\sqcap A$ for the greatest lower bound of A (if they exists). For $x, y \in P$, we write $x \sqcup y$ and $x \sqcap y$ respectively for $\sqcup\{x, y\}$ and $\sqcap\{x, y\}$.

Definition 2.2 (Pairwise Bounded Join Property) *A preordered set $(P, \preceq)$ has the pairwise bounded join property iff for any $p_1, p_2 \in P$ if $\{p_1, p_2\}$ has an upper bound then it has a least upper bound.*

Let $(P, \preceq)$ be a preordered set. For an element $p \in P$, we denote by $[p]$ the *equivalence class* containing p defined as $[p] = \{x | x \leq p, p \leq x\}$.

Definition 2.3 (Quotient Poset) *The quotient poset $[(P, \preceq)]$ of a preordered set $(P, \leq)$ is the poset $(P/\equiv, \preceq/\equiv)$ where $P/\equiv\ = \{[p] | p \in P\}$ and $[p]\ \preceq/\equiv [q]$ iff $p \preceq q$.*

It is easily verified that the relation $\preceq/\equiv$ in the above definition is well defined in the sense that it does not depend on the representatives of equivalence classes. For $[(P, \leq)]$, the following results is an immediately consequence of the definition:

Lemma 2.1 *If $(P, \preceq)$ has the pairwise bounded join property then so does $[(P, \preceq)]$.* ▮

2.2 Labeled Trees

We shall use *labeled trees* to represent types and data structures. In particular, a class of trees called *regular tree* play a central role. The main reference on this subject is a work by Courcelle [32]. In this section I have gathered from [32] definitions and standard results (with some adaptation of their presentations) that are useful in the subsequent investigation. I include proofs only for those results whose presentations differ from those found in [32].

2.2.1 Definitions and Basic Properties

Let X be a set of symbols. A *string over* A is a finite sequence of symbols in A. We denote by X^* the set of all string over X. The *empty* string is denoted by ϵ. We identify an element $x \in X$ and the string consisting of the symbol x (the string of length one). The *concatenation* of a string a and a string b is denoted by $a \cdot b$. A string a is a *prefix* of a string b if there is another string c such that $b = a \cdot c$. A prefix a of a string b is *proper* if $a \neq b$. The length of a string a is denoted by $|a|$. (Note that the notation $|x|$ is overloaded.) For a natural number n and $x \in X$, we write x^n for the string of n x's. For convenience, we define $x^0 = \epsilon$. If $A \subseteq X^*$ and $a \in X^*$, then we denote by $a \cdot A$ the set $\{a \cdot b | b \in A\}$ and by A/a the set $\{b | a \cdot b \in A\}$.

Throughout this thesis, we assume that there is a given countably infinite set $\mathcal{L}$ of symbols (ranged over by $l, l_1, \ldots$), called *labels*, equipped with a linear order $\ll$. For a technical convenience we assume that $\mathcal{L}$ is closed under products, i.e. there is an injective function $prodcode : (\mathcal{L} \times \mathcal{L}) \rightarrow \mathcal{L}$. We use the injection $prodcode$ implicitly and treat $\mathcal{L}$ as if it satisfies $(\mathcal{L} \times \mathcal{L}) \subset \mathcal{L}$. In particular, $(\mathcal{L} \times \mathcal{L})^* \subseteq \mathcal{L}^*$. On $(\mathcal{L} \times \mathcal{L})^*$ we define the mappings $first^*$, $second^*$ inductively as follows:

$$\begin{aligned} first^*(\epsilon) &= \epsilon \\ first^*(a \cdot (l_1, l_2)) &= first^*(a) \cdot l_1 \\ second^*(\epsilon) &= \epsilon \\ second^*(a \cdot (l_1, l_2)) &= second^*(a) \cdot l_2 \end{aligned}$$

On $\{(a, b) | a \in \mathcal{L}^*, b \in \mathcal{L}^*, |a| = |b|\}$, we define $pair^*$ as follows:

$$\begin{aligned} pair^*(\epsilon, \epsilon) &= \epsilon \\ pair^*(a \cdot l_1, b \cdot l_2) &= pair^*(a, b) \cdot (l_1, l_2) \end{aligned}$$

For $a \in (\mathcal{L} \times \mathcal{L})^*$, the following equation always holds:

$$pair^*(first^*(a), second^*(a)) = a$$

$$\begin{tikzcd}
 & f \arrow[dl, "l_2"'] \arrow[dr, "l_1"] & & \\
h & & k \arrow[dr, "l_2"] & \\
 & & & f
\end{tikzcd}$$

Figure 2.1: An example of a finite labeled tree

The following definition of labeled trees is due to Aït-Kaci[6]. Let $\boldsymbol{F}$ be a (not necessarily finite) set of symbols.

Definition 2.4 (Labeled Trees) *A labeled $\boldsymbol{F}$-tree is a function $\alpha : A \longrightarrow \boldsymbol{F}$ such that*

1. *A is a prefix-closed subset of $\mathcal{L}^*$, i.e. for any $a, b \in \mathcal{L}^*$, if $a \cdot b \in A$ then $a \in A$, and*

2. *A is finitely branching, i.e. if $a \in A$ then the set $\{l | a \cdot l \in A\}$ is finite.*

If $dom(\alpha)$ is finite then α is *finite* otherwise it is *infinite*. The set of all labeled $\boldsymbol{F}$-trees and the set of all finite labeled $\boldsymbol{F}$-trees are denoted respectively by $\boldsymbol{T}^{\infty}(\boldsymbol{F})$ and $\boldsymbol{T}(\boldsymbol{F})$.

As an example, let α be the function such that $dom(\alpha) = \{\epsilon, l_1, l_2, l_1 \cdot l_2\}$ and

$$\begin{aligned}
\alpha(\epsilon) &= f \\
\alpha(l_1) &= k \\
\alpha(l_2) &= h \\
\alpha(l_1 \cdot l_2) &= f
\end{aligned}$$

Then α is the tree shown in figure 2.1. As an example of infinite labeled trees, let $list^{int}$ be the following function:

$$\begin{aligned}
dom(list^{int}) = \; & \{(r \cdot tl)^n | n \geq 0\} \cup \\
& \{(r \cdot tl)^n \cdot r | n \geq 0\} \cup \\
& \{(r \cdot tl)^n \cdot l | n \geq 0\} \cup \\
& \{(r \cdot tl)^n \cdot r \cdot hd | n \geq 0\}
\end{aligned}$$

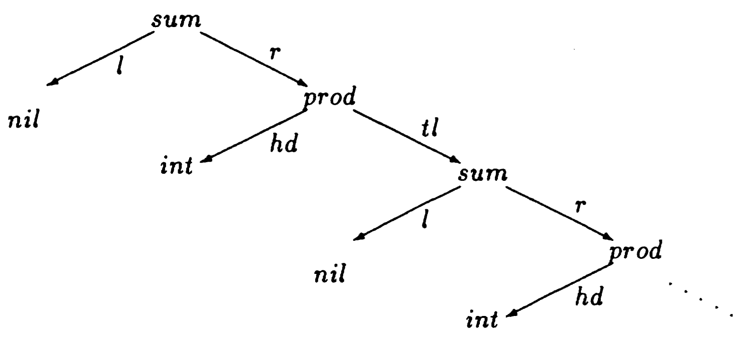

Figure 2.2: An example of an infinite labeled tree

and

$$\begin{aligned}
list^{int}((r \cdot tl)^n) &= sum \\
list^{int}((r \cdot tl)^n \cdot r) &= prod \\
list^{int}((r \cdot tl)^n \cdot l) &= nil \\
list^{int}((r \cdot tl)^n \cdot r \cdot hd) &= int
\end{aligned}$$

Then $list^{int}$ is the tree depicted in figure 2.2. This tree can be regarded as a represetation of the type of integer lists.

If $\alpha \in \boldsymbol{T}^{\infty}(\boldsymbol{F})$ and $a \in dom(\alpha)$ then α/a is the tree α' such that $dom(\alpha') = dom(\alpha)/a$, and for all $b \in dom(\alpha')$, $\alpha'(b) = \alpha(a \cdot b)$.

Definition 2.5 *The set of subtrees of a tree α, denoted by **Subtrees**(α), is the set $\{\alpha/a | a \in dom(\alpha)\}$.*

For any element $f \in \boldsymbol{F}$, we also denote by f the one node tree such that $dom(f) = \{\epsilon\}$ and $f(\epsilon) = f$. Let $\alpha_1, \ldots, \alpha_n \in \boldsymbol{T}^{\infty}(\boldsymbol{F})$, $l_1, \ldots, l_n \in \mathcal{L}$ and $f \in \boldsymbol{F}$. We write $f(l_1 = \alpha_1, \ldots, l_n = \alpha_n)$ to denote the tree α such that $dom(\alpha) = \{\epsilon\} \cup (l_1 \cdot dom(\alpha_1)) \cup \cdots \cup (l_n \cdot dom(\alpha_n))$, $\alpha(\epsilon) = f$, $\alpha(l_i \cdot a) = \alpha_i(a)$ for all $a \in dom(\alpha_i)$ $(1 \leq i \leq n)$. For example, an equation that holds for the tree $list^{int}$ in the above example can be written using this notation as follows:

$$list^{int} = sum(l = nil, r = prod(hd = int, tl = list^{int})).$$

A labeled tree can be regarded as a notational variant of a tree defined in [32] based on a *tree domain* [45]. Let N be the set of natural numbers and N_+ be the set of positive ones. A tree domain A is a subset of N_+^* such that (1) for any $a, b \in N_+^*$, if $a \cdot b \in A$ then $a \in A$, (2) for any $a \in N_+^*$ and $n \in N_+$, if $a \cdot n \in A$ then for any $1 \leq i \leq n$, $a \cdot i \in A$. A *ranked alphabet* is a set of symbols F associated with a mapping $r : F \longrightarrow N$ called a *ranking function*.

Definition 2.6 *Let F be a ranked alphabet with the ranking function r. An F-tree α is a function $\alpha : A \longrightarrow F$ such that A is a tree domain and for any $a \in A$ if $r(\alpha(a)) = n$ then $a \cdot i \in A$ iff $1 \leq i \leq n$.*

The set of all F-trees is denoted by $\boldsymbol{t}^\infty(F)$. For a set $\boldsymbol{F}$ of function symbols, $F_{\boldsymbol{F}}$ is the ranked alphabet $\boldsymbol{F} \times \boldsymbol{P}(\mathcal{L})$ with the ranking function r defined as $r((f, L)) = |L|$. The set of labeled trees $\boldsymbol{T}^\infty(\boldsymbol{F})$ has one-to-one correspondence to the set of trees $\boldsymbol{t}^\infty(F_{\boldsymbol{F}})$. Let $\alpha \in \boldsymbol{T}^\infty(\boldsymbol{F})$ with $dom(\alpha) = A$. Define a function $\theta_\alpha : A \longrightarrow N_+^*$ by induction on the length $|a|$ of $a \in A$ as follows:

$$\begin{aligned} \theta_\alpha(\epsilon) &= \epsilon, \\ \theta_\alpha(a \cdot l) &= \theta(a) \cdot i \text{ where } i \text{ is the natural number such that } l \text{ is the } i\text{-th smallest label} \\ &\qquad \text{under} \ll \text{in } \{l | a \cdot l \in A\}. \end{aligned}$$

Also define a function $\eta_\alpha : A \longrightarrow F_{\boldsymbol{F}}$ as follows:

$$\eta_\alpha(a) = (\alpha(a), \{l | a \cdot l \in A\}).$$

Now define a function $\phi : \boldsymbol{T}^\infty(\boldsymbol{F}) \longrightarrow \boldsymbol{t}^\infty(F_{\boldsymbol{F}})$ as $dom(\phi(\alpha)) = \theta_\alpha(A)$ and $\phi(\alpha)(a) = \eta_\alpha(\theta_\alpha^{-1}(a))$.

Proposition 2.2 *The function $\phi : \boldsymbol{T}^\infty(\boldsymbol{F}) \longrightarrow \boldsymbol{t}^\infty(F_{\boldsymbol{F}})$ is a bijection.*

Proof We first show that θ_α is injective and therefore ϕ is well defined. By simple induction, $|\theta_\alpha(a)| = |a|$. Suppose $\theta_\alpha(a) = \theta_\alpha(b)$. We need to show that $a = b$. The proof is by induction on the length of $\theta_\alpha(a)$. Suppose $\theta_\alpha(a) = \epsilon$. Then $|a| = |b| = |\epsilon|$ and therefore $a = b = \epsilon$. Suppose $\theta_\alpha(a) = c \cdot i$. $|a| = |b| = |c| + 1$. Let $a = a' \cdot l_a$ and $b = b' \cdot l_b$. Then since $\theta_\alpha(a) = \theta_\alpha(b)$ and by the definition of θ, $\theta_\alpha(a') \cdot i = \theta_\alpha(b') \cdot i$. This implies $\theta_\alpha(a') = \theta_\alpha(b')$. Then by the induction hypothesis, $a' = b'$. By the definition of θ_α, $l_a = l_b$ and therefore $a = b$.

By the definition of θ_α it is clear that $\theta_\alpha(A)$ is a tree domain. For the arity restriction, suppose $r((\phi(\alpha))(a)) = n$. Then $(\phi(\alpha))(a) = (f, \{l_1, \ldots, l_n\})$ such that $\{l_1, \ldots, l_n\} = \{l | (\theta_\alpha^{-1}(a)) \cdot l \in dom(\alpha)\}$. Then by the definition of θ_α, $a \cdot i \in dom(\phi(\alpha))$ iff $i \leq n$. Therefore $\phi(\alpha) \in \boldsymbol{t}^\infty(F_{\boldsymbol{F}})$.

For $\beta \in \boldsymbol{t}^\infty(F_{\boldsymbol{F}})$, define a function μ_β on $dom(\beta)$ by induction on the length of strings as follows:

$$\begin{aligned} \mu_\beta(\epsilon) &= \epsilon, \\ \mu_\beta(a \cdot i) &= \mu_\beta(a) \cdot l \text{ where } l \text{ is the } i\text{-th smallest label under} \ll \text{in } L \\ &\qquad \text{such that } \beta(a) = (f, L). \end{aligned}$$

and a function ν_β as $\nu_\beta(a) = f$ iff $\beta(a) = (f, L)$ for some L. Now define a function ψ on $t^\infty(F_{\boldsymbol{F}})$ such that $dom(\psi(\beta)) = \mu_\beta(dom(\beta))$ and $\psi(\beta)(a) = \nu_\beta(\mu_\beta^{-1}(a))$. Similar to the proof that θ_α is injective, it is shown that μ_β is injective. By the definition of μ_β it is also clear that $\mu_\beta(dom(\beta))$ is prefix closed. Therefore ψ is well defined and $\psi(\beta) \in \boldsymbol{T}^\infty(\boldsymbol{F})$.

Let $\alpha : A \rightarrow \boldsymbol{F}$ be any element in $\boldsymbol{T}^\infty(\boldsymbol{F})$. We show $\psi(\phi(\alpha)) = \alpha$ by showing the properties that (1) for any a, $a \in dom(\alpha)$ iff $a \in dom(\psi(\phi(\alpha)))$ and that (2) for any $a \in A$, $\alpha(a) = (\psi(\phi(\alpha)))(a)$. Since $dom(\psi(\phi(\alpha))) = \mu_{\phi(\alpha)}(\theta_\alpha(dom(\alpha)))$, (1) is shown by showing that for all $a \in dom(\alpha)$, $\mu_{\phi(\alpha)}(\theta_\alpha(a)) = a$. Proof is by induction on the length of a. Basis is trivila. Let $a \cdot l \in dom(\alpha)$ and i be the natural number such that l is the i^{th} smallest label in $\{l | a \cdot l \in dom(\alpha)\}$. Then we have:

$$\begin{aligned}
&\mu_{\phi(\alpha)}(\theta_\alpha(a \cdot l)) \\
&= \mu_{\phi(\alpha)}(\theta_\alpha(a) \cdot i) \\
&= (\mu_{\phi(\alpha)}(\theta_\alpha(a))) \cdot l' \text{ where } l' \text{ is the } i^{th} \text{ smallest label in } L \text{ such that} \\
&\quad \phi(\alpha)(\theta_\alpha(a)) = (f, L) \text{ for some } f.
\end{aligned}$$

But by the definition of ϕ, $\phi(\alpha)(\theta_\alpha(a)) = (\alpha(a), \{l | a \cdot l \in dom(\alpha)\})$. Therefore $l = l'$ and hence by the induction hypothesis, $\mu_{\phi(\alpha)}(\theta_\alpha(a \cdot l)) = a \cdot l$. For (2), suppose $\psi(\phi(\alpha))(a) = f$. By the definition of ψ, $\phi(\alpha)(\mu_{\phi(\alpha)}^{-1}(a)) = (f, L)$ for some L. By the definition of ϕ, $\alpha(\theta_\alpha^{-1}(\mu_{\phi(\alpha)}^{-1}(a))) = f$. But we have shown that $\mu_{\phi(\alpha)}(\theta_\alpha(a)) = a$. This implies that $\theta_\alpha^{-1}(\mu_{\phi(\alpha)}^{-1}(a)) = a$. Therefore $\alpha(a) = f$.

The property $\phi(\psi(\alpha)) = \alpha$ is shown by similar reasoning. Therefore $\phi^{-1} = \psi$. ∎

Because of this connection, we can regard labeled trees as trees and vice versa. In particular, all properties on trees shown in [32] can be applied to labeled trees. In what follows, we will use the term trees for labeled trees.

Lemma 2.2 *Let α, α' be trees of the forms $\alpha = f(l_1 = \alpha_1, \ldots, l_n = \alpha_n)$, $\alpha' = f(l'_1 = \alpha'_1, \ldots, l'_n = \alpha'_n)$. They are equal iff $f = g$, $\{l_1, \ldots, l_n\} = \{l'_1, \ldots, l'_n\}$ and $\alpha/l_i = \alpha'/l_i$ for all $1 \leq i \leq n$.* ∎

The following two lemmas hold only for finite trees:

Lemma 2.3 (Definition by Structural Induction) *There exists one and only one mapping ξ : $\boldsymbol{T}(\boldsymbol{F}) \rightarrow A$ such that*

1. *$\xi(f) = base(f)$ for all $f \in \boldsymbol{F}$,*
2. *$\xi(f(l_1 = \alpha_1, \ldots, l_n = \alpha_n)) = step(f, (l_1, \xi(\alpha_1)), \ldots, (l_n, \xi(\alpha_n)))$*

where base and step are given mappings of the following types:

$$\begin{array}{rcl} base & : & \boldsymbol{F} \rightarrow A, \\ step & : & \bigcup_{\{l_1,\ldots,l_n\}\in \boldsymbol{P}(\mathcal{L})} \boldsymbol{F} \times (l_1 \times A) \times \cdots \times (l_n \times A) \rightarrow A. \blacksquare \end{array}$$

Lemma 2.4 (Proof by Structural Induction) *In order to prove a property P on the set $\boldsymbol{T}(\boldsymbol{F})$, it suffices to prove that:*

1. *for all $f \in \boldsymbol{F}$, $P(f)$,*

2. *for all $f \in \boldsymbol{F}$, $\{l_1, \ldots, l_n\} \in \boldsymbol{P}(\mathcal{L})$, and $\alpha_1, \ldots, \alpha_n \in \boldsymbol{T}(\boldsymbol{F})$, if $P(\alpha_1), \ldots, P(\alpha_n)$ then $P(f(l_1 = \alpha_1, \ldots, l_n = \alpha_n))$.* $\blacksquare$

2.2.2 Substitutions and Unifications on Trees

We first introduce trees containing *variables*. Let V be a set of *variables* disjoint from $\boldsymbol{F}$. We denote by $\boldsymbol{T}^{\infty}(\boldsymbol{F}, V)$ the set of trees generated by the set of function symbols $\boldsymbol{F} \cup V$ such that there are no outgoing edge from variable nodes, i.e. $\alpha \in \boldsymbol{T}^{\infty}(\boldsymbol{F}, V)$ iff $\alpha \in \boldsymbol{T}^{\infty}(\boldsymbol{F} \cup V)$ and if $\alpha(a) \in V$ then there is no $l \in \mathcal{L}$ such that $a \cdot l \in dom(\alpha)$. $\boldsymbol{T}(\boldsymbol{F}, V)$ is the set of finite trees in $\boldsymbol{T}^{\infty}(\boldsymbol{F}, V)$.

Definition 2.7 (Substitutions) *A first-order substitution (or simply substitution) θ is a function from V to $\boldsymbol{T}^{\infty}(\boldsymbol{F}, V)$ such that $\theta(v) \neq v$ for only finitely many $v \in V$. Let $\alpha \in \boldsymbol{T}^{\infty}(\boldsymbol{F}, V)$ be any tree. The result of simultaneous substitution of $\theta(v)$ for $v \in V$ in α, denoted by $\theta^*(\alpha)$, is the tree α' defined as:*

$$dom(\alpha') = dom(\alpha) \cup \bigcup \{a \cdot dom(\theta(v)) | a \in dom(\alpha), \alpha(a) = v\}$$

and

$$\begin{array}{rcl} \alpha'(a) & = & \alpha(a) \text{ if } a \in dom(\alpha) \text{ and } \alpha(a) \notin V, \\ \alpha'(a) & = & \theta(\alpha(b))(c) \text{ if } a = b \cdot c, \alpha(b) \in V. \end{array}$$

Since α and $\theta(v)$ (for any v) are trees, $\theta^*(\alpha)$ is a well defined tree.

When restricted to finite trees, the above definiton of application of substitution is equivalent to the following inductive definition:

$$\begin{array}{rcl} \theta^*(v) & = & \theta(v) \text{ if } v \in V, \\ \theta^*(f(l_1 = \alpha_1, \ldots, l_n = \alpha_n)) & = & f(l_1 = \theta^*(\alpha_1), \ldots, l_n = \theta^*(\alpha_n)) \\ & & \text{for all } f \in \boldsymbol{F}, \{l_1, \ldots, l_n\} \in \boldsymbol{P}(\mathcal{L}) \end{array}$$

which is also characterized by the unique homomorphic extension of θ to $\boldsymbol{T}(\boldsymbol{F}, V)$. The function θ^* on $\boldsymbol{T}^\infty(\boldsymbol{F}, V)$ can be also defined by a unique extension of the above inductive definition [32]. Since various syntactic structures such as pairs and sequences can be regarded as trees, we apply θ^* directly to those syntactic structures containing trees.

If θ is a substitution, then we denote by *dom*(θ) the set $\{v | v \in V, \theta(v) \neq v\}$. Let V be a set of variables. A *restriction* of a substitution θ to V, denoted by $\theta\restriction^V$, is the substitution θ' defined as follows:

$$\theta'(v) = \begin{cases} \theta(v) & \text{if } v \in V \\ v & \text{otherwise.} \end{cases}$$

Note that notations *dom*(θ) and $\theta\restriction^V$ are overloaded with the corresponding notions of functions. Distinction of them should be clear from our usage of meta notations and the context. If *dom*$(\theta) = \{l_1, \ldots, l_n\}$ and $\theta(v_i) = \alpha_i$ $(1 \leq i \leq n)$ then we shall use the notation $[v_1 := \alpha_1, \ldots, v_n := \alpha_n]$ and $\alpha[v_1 := \alpha_1, \ldots, v_n := \alpha_n]$ for θ and $\theta^*(\alpha)$ respectively. If θ, η are substitutions then their *composition* is the substitution defined as $\theta^* \circ \eta$. In what follows, we will identify the mapping θ^* with θ. In particular, we write $\theta \circ \eta$ for the composition of θ and η

A substitution θ is *ground for* a tree α if $\theta(\alpha) \in \boldsymbol{T}^\infty(\boldsymbol{F})$. A tree α' is a *substitution instance* (or simply *instance*) of α if there is some substitution θ such that $\alpha' = \theta(\alpha)$. If θ is gound for α then α' is a *ground instance* of α.

Definition 2.8 (Unifier) *A substitution θ is a unifier of trees $\alpha, \beta \in \boldsymbol{T}^\infty(\boldsymbol{F}, V)$ if $\theta(\alpha) = \theta(\beta)$. A unifier θ is more general than θ' if there is another substitution η such that $\theta' = \eta \circ \theta$.*

Substitutions induce the following preorder on trees containing variables:

Definition 2.9 *Let $\alpha, \beta \in \boldsymbol{T}^\infty(\boldsymbol{F}, V)$. α is more general than β, denoted by $\beta \precsim \alpha$, iff there is some substitution θ such that $\beta = \theta(\alpha)$.*

2.2.3 Regular Trees

An important class of trees in $\boldsymbol{T}^\infty(\boldsymbol{F})$ is the set of *regular trees*. Since "'all properties' of regular trees are decidable"[32], they provide rich yet computationally feasible data structures for databases and other information systems.

Definition 2.10 (Regular Trees) *A tree $\alpha \in \boldsymbol{T}^\infty(\boldsymbol{F})$ is regular iff the set* **Subtrees**(α) *is finite.*

The set of all regular trees in $\boldsymbol{T}^\infty(\boldsymbol{F})$ is denoted by $\boldsymbol{R}(\boldsymbol{F})$.

On regular trees, the following properties hold.

Proposition 2.3 *1.* $\boldsymbol{T}(\boldsymbol{F}) \subset \boldsymbol{R}(\boldsymbol{F}) \subset \boldsymbol{T}^{\infty}(\boldsymbol{F})$.

2. Any subtree of a regular tree is regular.

3. The set of symbols occurring in a regular tree is finite.

4. $\boldsymbol{R}(\boldsymbol{F})$ *is closed under substitution. We mean by this that* $\theta(\alpha)$ *is regular if* $\theta(v)$ *is regular for all* $v \in V$. ▌

Intuitive way of understanding the definition is that regular trees are trees that contain only finite amount of information. This intuition corresponds to the property that a regular tree has a finite representation. There are several equivalent representations of regular trees. Following [6], we use Moore machines to represent them.

Definition 2.11 (Moore Machine) *A Moore machine is a 5-tuple* $(Q, s, \boldsymbol{F}, \delta, o)$, *where* Q *is a finite set of states,* s *is a distinguished element in* Q *called the start state,* $\boldsymbol{F}$ *is the set of output symbols,* δ *is a partial function from* $Q \times \mathcal{L}$ *to* Q *called the state transition function such that for any* $q \in Q$, $\{l \in \mathcal{L} | \delta(q,l)$ *is defined*$\}$ *is finite and* o *is the output function from* Q *to* $\boldsymbol{F}$.

In the above definition, the input alphabet is implicitly assumed to be the fixed set $\mathcal{L}$ of labels. Because of the restriction on δ, a Moore machine under the above definition behaves like a Moore machine under a standard definition (such as in [52]) where the input alphabet $\mathcal{L}$ is finite and δ is defined as a total function on $Q \times \mathcal{L}$.

As is done in standard finite state automata [52], we extend δ to the partial function δ^* on $Q \times \mathcal{L}^*$ as follows:

$$\begin{aligned}
\delta^*(q, \epsilon) &= q, \\
\delta^*(q, l) &= q' \text{ for all } l \in \mathcal{L} \text{ such that } \delta(q,l) = q', \\
\delta^*(q, a \cdot l) &= q'' \text{ for all } a \in \mathcal{L}^*, l \in \mathcal{L} \text{ such that } \delta^*(q,a) = q', \delta(q', l) = q''.
\end{aligned}$$

A state $q \in Q$ is *reachable* if there is some $a \in \mathcal{L}^*$ such that $\delta^*(s,a) = q$. a is called a *path* from s to q. Each state $q \in Q$ in a Moore machine $M = (Q, s, \boldsymbol{F}, \delta, o)$ represents a function form a subset of $\mathcal{L}^*$ to $\boldsymbol{F}$. Define $M(q)$ as the function such that $dom(M(q)) = \{a \in \mathcal{L}^* | \delta^*(q,a) = q' \text{ for some } q' \in Q\}$ and $M(q)(a) = o(\delta^*(q,a))$ for all $a \in dom(M)$.

The following theorem establishes the relationship between Moore machines and regular trees, which corresponds to the equivalence between regular trees and *regular systems* shown in [32].

Theorem 2.1 *For any Moore machine $M = (Q, s, \boldsymbol{F}, \delta, o)$, $M(q) \in \boldsymbol{R}(\boldsymbol{F})$ for any $q \in Q$. Conversely, for any regular tree $\alpha \in \boldsymbol{R}(\boldsymbol{F})$ there is a Moore machine $M = (Q, s, \boldsymbol{F}, \delta, o)$ such that $\alpha = M(s)$.*

Proof Let $M = (Q, s, \boldsymbol{F}, \delta, o)$ be a Moore machine. For any $a, b \in \mathcal{L}^*$, and $q \in Q$, if $\delta^*(q, a \cdot b) = q'$ for some q', then by the definition of δ^*, $\delta^*(q, a) = q''$ for some q''. Therefore $dom(M(q))$ is prefix closed. By the restriction of δ, $\{l | a \cdot l \in dom(M(q))\}$ is finite. Therefore $M(q) \in \boldsymbol{T}^{\infty}(\boldsymbol{F})$. Since $M(q)/a = M(\delta^*(q, a))$ for all $a \in dom(M(q))$, $|\boldsymbol{Subtrees}(M(q))| \leq |Q|$ and hence finite. This establishes $M(q) \in \boldsymbol{R}(\boldsymbol{F})$.

Let $\alpha_0 \in \boldsymbol{R}(\boldsymbol{F})$ be any regular tree with the set of subtrees $\boldsymbol{Subtrees}(\alpha_0) = \{\alpha_0, \ldots, \alpha_n\}$. Define the Moore machine $M_{\alpha_0} = (\{q_0, \ldots, q_n\}, q_0, \boldsymbol{F}, \delta, o)$ as follows:

1. δ is the function such that $\delta(p_i, l)$ is defined and equal to q_j iff $l \in dom(\alpha_i)$ and $\alpha_i / l = \alpha_j$,
2. o is defined as $o(q_i) = \alpha_i(\epsilon)$.

Then for any $a \in \mathcal{L}^*$, it is shwon by simple induction on $|a|$ that $a \in dom(M_{\alpha_0}(q_0))$ iff $a \in dom(\alpha_0)$ and $M_{\alpha_0}(q)(a) = f$ iff $\alpha_0(a) = f$. Therefore $M_{\alpha_0}(q) = \alpha$. ▌

We say that a regular tree α is represented by a Moore machine M if $M(s) = \alpha$.

The following construction on Moore machines will be often useful to determine various relations on regular trees via Moore machines.

Definition 2.12 (Product Machine) *Let $\simeq$ be an equivalence relation on $\mathcal{L}$. Given two Moore machines $M_1 = (Q_1, s_1, \boldsymbol{F}_1, \delta_1, o_1)$ and $M_2 = (Q_2, s_2 \boldsymbol{F}_2, \delta_2, o_2)$, a product machine of M_1 and M_2 modulo $\simeq$, denoted by $(M_1 \times M_2)/\simeq$, is the Moore machine $(Q, s, \boldsymbol{F}, \delta, o)$ such that:*

1. *$Q = (Q_1 \cup \{\$\}) \times (Q_2 \cup \{\$\})$ where $\$$ is a new distinguished symbol that does not appear in M_1 or M_2,*
2. *$s = (s_1, s_2)$,*
3. *$\boldsymbol{F} = (\boldsymbol{F}_1 \cup \{\$\}) \times (\boldsymbol{F}_2 \cup \{\$\})$*
4. *$\delta((x, y), l)$ is defined and equal to (x', y') iff one of the following holds:*
   - *(a) $l = (l_1, l_2)$, $l_1 \neq \$, l_2 \neq \$$, $l_1 \simeq l_2$, $x \in Q_1, y \in Q_2$, and $\delta_1(x, l_1) = x', \delta_2(y, l_2) = y'$,*
   - *(b) $l = (l_1, l_2)$, $l_1 \neq \$, l_2 = \$$, $x \in Q_1$, $\delta_1(x, l_1) = x'$, $y' = \$$ and either $y = \$$ or there is no l_2' such that $l_1 \simeq l_2'$ and $\delta_2(y, l_2')$ is defined,*

(c) $l = (l_1, l_2)$, $l_1 = \$, l_2 \neq \$$, $y \in Q_2$, $\delta_2(y, l_2) = y'$, $x' = \$$ and either $x = \$$ or there is ~~no~~ l'_1 such that $l'_1 \simeq l_2$ and $\delta_1(x, l'_1)$ is defined,

5. *$o((x_1, x_2)) = (O_1, O_2)$ such that $O_i = o_i(x_i)$ if $x_i \in Q_i$ otherwise $O_i = \$$ ($i \in \{1,2\}$).*

If $\simeq$ is the identity relation $=$ on $\mathcal{L}$ then we write $M_1 \times M_2$ for $(M_1 \times M_2)/{=}$. The construction of a product machine is clearly effective.

For a given equivalence relation $\simeq$, $\overset{*}{\simeq}$ is the equivalence relation on $\mathcal{L}^*$ defined as follows:

$$\begin{aligned} \epsilon &\overset{*}{\simeq} \epsilon \\ a \cdot l_1 &\overset{*}{\simeq} b \cdot l_2 \text{ if } a \overset{*}{\simeq} b \text{ and } l_1 \simeq l_2. \end{aligned}$$

For product machines, the following are immediate consequences of the definition:

Lemma 2.5 *Let $M_1 = (Q_1, s_1, \boldsymbol{F}_1, \delta_1, o_1)$, $M_2 = (Q_2, s_2, \boldsymbol{F}_2, \delta_2, o_2)$ and $(Q, s, \boldsymbol{F}, \delta, o) = (M_1 \times M_2)/{\simeq}$.*

1. *If $\delta^*(s, a) = (q_1, q_2), q_1 \in Q_1, q_2 \in Q_2$ then $first^*(a) \overset{*}{\simeq} second^*(a)$ and $\delta_1^*(s_1, first^*(a)) = q_1$, $\delta_2^*(s_2, second^*(a)) = q_2$. Conversely, if there are a, b such that $a \overset{*}{\simeq} b$, $\delta_1^*(s_1, a) = q_1$ and $\delta_2^*(s_2, b) = q_2$ then $\delta^*(s, pair^*(a, b)) = (q_1, q_2)$.*

2. *If $\delta^*(s, a) = (q, x), q \in Q_1$ then $\delta_1^*(s_1, first^*(a)) = q$ and $first^*(o((q, x))) = o_1(q)$. If $\delta^*(s, a) = (x, q), q \in Q_2$ then $\delta_2^*(s_2, second^*(a)) = q$ and $second^*(o(x, q)) = o_2(q)$.*

3. *If $\delta_1^*(s_1, a) = q$ then there is some b such that $first^*(b) = a$, $\delta^*(s, b) = (q, x)$ and $o_1(q) = first^*(o((q, x)))$. If $\delta_2^*(s_2, a) = q$ then there is some b such that $second^*(b) = a$ and $\delta^*(s, b) = (x, q)$ and $o_2(q) = second^*(o((q, x)))$.* ▮

On Moore machines defined on the same set of output symbols $\boldsymbol{F}$, we define an equivalence relation $\approx$ as $M_1 \approx M_2$ iff $M_1(s_1) = M_2(s_2)$ where s_1, s_2 are the start states of M_1, M_2 respectively.

Theorem 2.2 *The relation $M_1 \approx M_2$ is decidable.*

Proof The idea of the following proof is due to Hopcroft and Karp [53], who defined an algorithm to decide whether two finite state automata accept a same regular set or not.

Let $M_1 = (Q_1, s_1, \boldsymbol{F}_1, \delta_1, o_1)$, $M_2 = (Q_2, s_2, \boldsymbol{F}_2, \delta_2, o_2)$. We will show that $M_1 \approx M_2$ iff for any reachable state q in $M_1 \times M_2$, q is of the form (q_1, q_2) such that $q_1 \in Q_1, q_2 \in Q_2$ and $o(q) = (f, f)$ for some $f \in \boldsymbol{F}$. Suppose $M_1 \approx M_2$. $dom(M_1(s_1)) = dom(M_2(s_2))$ and for any $a \in dom(M_1(s_1))$,

$o_1(\delta_1^*(s_1,a)) = o_2(\delta_2^*(s_2,a))$. Therefore, for any $a \in \mathcal{L}^*$, there is some reachable $q_1 \in Q_1$ such that $\delta_1^*(s_1,a) = q_1$ and $o_1(q_1) = f$ for some $f \in \boldsymbol{F}$ iff there is some reachable $q_2 \in Q_2$ such that $\delta_2^*(s_2,a) = q_2$ and $o_2(q_2) = f$. Then by lemma 2.5, for any reachable state q in $M_1 \times M_2$, q is of the form (q_1,q_2) and $o(q) = (f,f)$ for some $f \in \boldsymbol{F}$.

Conversely, suppose $M_1 \times M_2$ satisfies the condition. Then $\{pair^*(a,a)|a \in dom(M_1(s_1))\} = \{pair^*(a,a)|a \in dom(M_2(s_2))\}$ and for any $a \in dom(M_1(s_1))$, $o(\delta^*((s_1,s_2),pair^*(a,a)) = (o_1(\delta_1^*(s_1,a)), o_2(\delta_2^*(s_2,a))) = (f,f)$ for some $f \in \boldsymbol{F}$. This implies $M_1(s_1) = M_2(s_2)$. ▌

By the relationship between Moore machines and regular trees (theorem 2.1) the above theorem implies:

Corollary 2.1 *Equality on regular trees is decidable.*

This property was first shown by Ginali [41]. Alternative proofs can be found in [32].

Another important property on regular trees is the decidability of the unification problem shown by Huet [55]:

Theorem 2.3 (Huet) *There is an algorithm $\mathcal{U}$ which, given a pair of regular trees $\alpha, \beta \in \boldsymbol{R}(\boldsymbol{F}, V)$, returns either a substitution or failure such that if it returns a substitution then it is a most general unifier of α and β otherwise they have no unifier.* ▌

Finally we define term representations of Moore machines (and therefore regular trees). Let e denote terms given by the following syntax:

$$e ::= v \mid f \mid f(l = e, \ldots, l = e) \mid (rec\ v.\, e)$$

where v stands for auxiliary variables, f stands for a given set $\boldsymbol{F}$ of output symbols, and l stands for the set $\mathcal{L}$ of input symbols. A variable occurrence v is *bound occurrence* if it is in $(rec\ v \ldots)$ otherwise it is *free*. A term e is *proper* if it does not contain free variables and if $e \equiv (rec\ v.\, e')$ then e' is a term of the form $f(l_1 = e_1, \ldots, l_n = e_n)$. We denote the set of proper terms generated by $\boldsymbol{F}$ by the following syntax:

$$e ::= f \mid f(l = e, \ldots, l = e) \mid (rec\ v.\, e(v)).$$

A proper term e denotes the Moore machine $M_e = (Q, q_0, \boldsymbol{F}, \delta, o)$ defined as follows:

1. $Q = \{q_f|$ for each occurrence f in $e\}$,

2. $q_0 = q_f$ where f is the outmost occurrence of an output symbol in e,

3. $o(q_f) = f$, and

4. δ is the following function:

$$\begin{aligned}
\delta(q_f, l) &= q_g \text{ if } f, g \text{ are occurrences in a subterm of the forms } f(\ldots, l = g, \ldots), \\
&\quad f(\ldots, l = g(\ldots), \ldots), \text{ or } f(\ldots, l = (rec\ v.\, g \ldots), \ldots), \\
\delta(q_f, l) &= q_g \text{ if } f \text{ is the occurrence in a subterm of the form } f(\ldots, l = v, \ldots) \text{ and} \\
&\quad g \text{ is the occurrence in its innermost surrounding subterm of the form} \\
&\quad (rec\ v.\, g(\ldots)).
\end{aligned}$$

Conversely, for any Moore machine $M = (Q, s, \boldsymbol{F}, \delta, o)$ there is a term e_M that represents M. Define a partial ordering $\leq$ on reachable states in Q as follows: $q \leq q'$ iff the shortest path from s to q is a prefix of the shortest path form s to q'. We define two mappings $\mathcal{R}_1, \mathcal{R}_2$ respectively on $Q \times Q$ and Q as follows:

$$\mathcal{R}_1(p, q) = \begin{cases} q & \text{if } q \leq p \\ \mathcal{R}_2(q) & \text{otherwise} \end{cases}$$

and

$$\begin{aligned}
\mathcal{R}_2(p) &= (rec\ p.\, o(p)(l_1 = \mathcal{R}_1(p, \delta(p, l_1)), \ldots, l_n = \mathcal{R}_1(p, \delta(p, l_n)))) \\
&\quad \text{where } \{l_1, \ldots, l_n\} = \{l | \delta(p, l) \text{ is defined}\}.
\end{aligned}$$

Since $\leq$ is a well founded partial ordering, the above definition is well defined. Then the term $\mathcal{R}_2(s)$ represents M.

As an example, let $M = (\{q_1, q_2, q_3\}, q_1, \{f, g, h, \ldots\}, \delta, o)$ such that

$$\begin{aligned}
\delta(q_1, l_1) &= q_2, \\
\delta(q_1, l_2) &= q_3, \\
\delta(q_2, l_3) &= q_1,
\end{aligned}$$

and

$$\begin{aligned}
o(q_1) &= f, \\
o(q_2) &= g, \\
o(q_3) &= h.
\end{aligned}$$

Then $\mathcal{R}_2(M) = (rec\, q_1.f(l_1 = (rec\, q_2.g(l_3 = q_1)), l_2 = (rec\, q_3.h)))$. Note that a term representation of a Moore machine is not unique. The above machine (modulo renaming of state names) also has the following simpler term representation: $(rec\ q.\, f(l_1 = g(l_3 = q), l_2 = h))$.

Chapter 3

Analysis of ML : its Syntax and Semantics

This chapter analyzes the syntactic properties of the polymorphic core of the programming language ML and proposes a framework for denotational semantics for ML polymorphism. These analyses provide bases for the subsequent development of a type system and a language for databases and object-oriented programming. Most of the results of this chapter were presented in [84].

3.1 Introduction

ML is a strongly typed programming language sharing with other typed languages the property that the type correctness of a program is completely checked by static analysis of the program – usually done at compile time. Among other strongly typed languages, one feature that distinguishes ML is its *implicit* type system. Unlike explicitly-typed languages such as Algol [109], Pascal [64] and Ada [60], ML does not require type specifications of bound variables (formal parameters). The type of a program is automatically *inferred* by ML type system. Through this type inference mechanism, ML achieves much of the convenience of dynamically typed languages without sacrificing the desired feature of complete static type-checking. As an example, consider the following definition of the factorial function in ML:

```
fun fact n = if n = 1 then 1
             else n * (fact (n - 1));
```

Besides the notational differences, the above definition has the identical structure to the following definition in Lisp:

```
(defun fact (n)
       (cond ((equal n 1) 1)
             (t (mul n (fact (sub n 1))))))
```

In particular, both of them have no mention of types. However, in ML, the compiler statically *infers* the type `int -> int` of `fact`. By this mechanism, the type correctness of ML programs is completely checked at compile time. This contrast with Lisp (and any other dynamically typed languages), where type errors such as the one in `(defun foo ...  (fact '("a" "b" "c"))..)` are not caught until something goes wrong at run-time, often with a disastrous consequence.

Another important feature of ML is that it supports *polymorphism* in a static type system. This is achieved by inferring a *most general* (or *principal*) *type-scheme* of any type correct program. A principal type-scheme of a program represents the set of all possible types of the program, capturing the polymorphic nature of the untyped program code. By this mechanism, ML also achieves much of the flexibility of dynamically typed languages in a static type system. For example, from the following definition of identity function:

```
fun id x = x;
```

ML type system infers the following type-scheme:

```
'a -> 'a
```

where `'a` is a *type variable* representing arbitrary types. As a consequence, `id` can be used as an identity function of any type of the form $\tau \rightarrow \tau$. The type correctness of each application of `id` is statically checked. Moreover, the result type of the application is also statically determined. For example, `id("a")` yields an expression of type `string` and `id(3)` yields an expression of type `int`.

There are two major existing approaches to denotational semantics for ML polymorphism; the one by Milner [78] (extended by MacQueen, Plotkin and Sethi [72]) based on an untyped language and the other by Mitchell and Harper [79] based on an explicitly-typed language using Damas and Milner's type inference system [34]. As I shall suggest in this chapter, however, neither of them properly explains the behavior of ML programs. Because of the implicit type system, ML behaves differently from both untyped languages and explicitly-typed languages. In order to understand ML, we need to develop a framework for denotational semantics and equational theories that give precise account for ML's implicit type system. The goal of this chapter is to propose such a framework, which will provide a basis to extend safely its type system to include various structures and operations for databases and object-oriented programming. In the rest of this section, we review the two existing approaches in subsection 3.1.1, 3.1.2 and outline our approach in subsection 3.1.3.

3.1.1 Milner's original semantics

In [78], Milner proposed a semantic framework for ML based on a semantics of an *untyped* language. He defined the following two classes of types:

$$\begin{aligned} \tau &::= b \mid \tau \rightarrow \tau, \\ \rho &::= t \mid b \mid \rho \rightarrow \rho \end{aligned}$$

where b stands for base types and t stands for type variables. Here we call them *types* and *type-schemes* respectively. Type-schemes containing type variables represent all their substitution instances and correspond to polymorphic types. He defined the preorder of *generalness* on type-schemes as the preorder on trees induced by substitution (definition 2.9), i.e. ρ is more general than ρ' iff ρ' is an substitution instance of ρ. He then gave the algorithm $\mathcal{W}$ that infers most general type-schemes for the following raw terms:

$$e ::= x \mid (\lambda x.\, e) \mid (e\ e) \mid \textbf{if } e \textbf{ then } e \textbf{ else } e \mid \textbf{fix } x\ e \mid \textbf{let } x = e \textbf{ in } e \textbf{ end}.$$

He interpreted an ML typing $e\ :\ \rho$ as the semantion assertion $[\![e]\!] \in [\![\rho]\!]$, i.e. the denotation of e is an element of the denotation of ρ and showed that the type inference algorithm $\mathcal{W}$ is sound under this interpretation. The denotation of a raw term is defined as an element of a domain satisfying the following domain equation:

$$V = B_1 + \cdots + B_n + [V \rightarrow V] + \{wrong\}$$

where $B_1, \ldots, B_n$ are domains corresponding to base types and *wrong* represents run-time error. The denotation of a type is defined as a subset of V not containing *wrong*. The denotation of a type-scheme is defined as the intersection of the denotations of all its instance types. This semantics was extended to recursive types by MacQueen, Plotkin and Sethi [72]. (See also [51, 30] for related studies.)

This semantics explains the polymorphic nature of ML programs and verifies that ML typing discipline prevents all run-time type errors. However, this semantics does not completely fit the operational behavior of ML programs. As an example, consider the following two raw terms e_1 and

e_2 with their principal type-schemes:

$$e_1 \equiv \lambda x \lambda y.\, y \;:\; t_1 \to t_2 \to t_2,$$
$$e_2 \equiv \lambda x \lambda y.\, (\lambda z \lambda w.\, w)(xy)y \;:\; (t_3 \to t_4) \to t_3 \to t_3$$

where parentheses are omitted, assuming left association of applications. Under the call-by-name version of Milner's semantics, which is also the semantics defined by MacQueen, Plotkin and Sethi, the above two raw terms have the same meaning. Indeed, if we were to ignore their type-schemes and regard them as terms in the untyped lambda calculus, then they would be β-convertible to each other and would be regarded as equal terms. However, ML is apparently a typed language, and as terms of ML, these two behave quite differently. For example, under any evaluation strategy, the term $((e_1\ 1)\ 2)$ is evaluated to 2 but $((e_2\ 1)\ 2)$ is not even a legal term and ML compiler reports a type error. This is one of the most noticeable difference between meanings of terms and should be distinguished by any semantics. From this example, we can also see that the equality on ML programs is different from the equality on terms in the untyped lambda calculus.

Moreover, this semantics requires a model of the set of*all* untyped lambda terms, many of which do not have typing and therefore do not correspond to ML programs.

3.1.2 Damas-Milner type inference system and Mitchell-Harper's analysis

Damas and Milner presented a proof system for typing judgements of ML [34]. They redefined the set of types of ML as the following two classes:

$$\rho ::= t \mid b \mid \rho \to \rho,$$
$$\pi ::= \rho \mid \forall t.\, \pi.$$

ρ is Milner's type-scheme. We call π a *generic type-scheme. Free* type variables and *bound* type variables are defined as in the second-order lambda calculus (or the polymorphic lambda calculus) [94, 42]. We write $\pi[\rho_1/t_1, \ldots, \rho_n/t_n]$ for the generic type-scheme obtained from π by simultaneously substituting each free occurrence t_i by ρ_i.

Definition 3.1 *A generic type-scheme* $\forall t_1 \cdots t_n.\, \rho$ *is a generic instance of* $\forall t'_1 \cdots t'_m.\, \rho'$ *if each* t_j *is not free in* $\forall t'_1 \cdots t'_m.\, \rho'$ *and* $\rho = \rho'[t'_1 := \rho_1, \ldots, t'_m := \rho_m]$ *for some type-schemes* $\rho_1, \ldots, \rho_m$. *A type* π *is more general than* π', *denoted by* $\pi' \precsim_{\mathrm{DM}} \pi$, *if* π' *is a generic instance of* π.

Note that the relation $\pi' \precsim_{\mathrm{DM}} \pi$ is decidable. A Damas-Milner *type assignment scheme* Γ is a function from a finite subset of variables to generic type-schemes.

Definition 3.2 (Damas-Milner Type Inference System) *A Damas-Milner typing scheme is a formula of the form* $\Gamma \rhd e : \pi$ *that is derivable in the following proof system:*

(VAR) $\qquad \Gamma \rhd x : \pi \quad \textit{if } \Gamma(x) = \pi$

(GEN) $$\frac{\Gamma \rhd e : \pi}{\Gamma \rhd e : \forall t.\, \pi} \quad \textit{if } t \textit{ not free in } \Gamma$$

(INST) $$\frac{\Gamma \rhd e : \pi}{\Gamma \rhd e : \pi'} \quad \textit{if } \pi' \preceq_{\mathrm{DM}} \pi$$

(ABS) $$\frac{\Gamma\{x := \rho_1\} \rhd e : \rho_2}{\Gamma \rhd \lambda x.\, e : \rho_1 \rightarrow \rho_2}$$

(APP) $$\frac{\Gamma \rhd e_1 : \rho_1 \rightarrow \rho_2 \qquad \Gamma \rhd e_2 : \rho_1}{\Gamma \rhd (e_1\ e_2) : \rho_2}$$

(LET) $$\frac{\Gamma \rhd e_1 : \pi \qquad \Gamma\{x := \pi\} \rhd e_2 : \rho}{\Gamma \rhd \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} : \rho}$$

We write $\boldsymbol{DM} \vdash \Gamma \rhd e : \pi$ if $\Gamma \rhd e : \pi$ is derivable in the proof system. In this formalism, ML terms are typing schemes. We call them Damas-Milner terms.

Based on this derivation system, Mitchell and Harper proposed another framework to explain implicit type system of ML [79]. In what follows, we shall only discuss their analysis of the core of ML. However, it should be mentioned that their approach also provides an elegant treatment of Standard ML's modules [47].

They defined an explicitly-typed language, called Core-XML. The set of types of Core-XML is the same as those in Damas-Milner system. The set of *pre-terms* of Core-XML is given by the following abstract syntax:

$$M ::= x \mid (M\ M) \mid (\lambda x : \rho.\, M) \mid (M\ \rho) \mid (\lambda t.\, M) \mid \mathbf{let}\ x : \pi = M\ \mathbf{in}\ M\ \mathbf{end}$$

where $(M\ \rho)$ is a type application and $(\lambda t.\, M)$ is a type abstraction.

Definition 3.3 (Terms of Core-XML) *Core-XML terms are formulae of the form* $\Gamma \rhd M : \pi$ *that are derivable in the following proof system:*

(VAR) $\qquad \Gamma \rhd x : \pi \quad \textit{if } \Gamma(x) = \pi$

$$(\text{TABS})\quad \frac{\Gamma \rhd M : \pi}{\Gamma \rhd (\lambda t.\, M) : \forall t.\, \pi} \quad \textit{if } t \textit{ not free in } \Gamma$$

$$(\text{TAPP})\quad \frac{\Gamma \rhd M : \forall t.\, \pi}{\Gamma \rhd (M\ \rho) : \pi[\rho/t]}$$

$$(\text{ABS})\quad \frac{\Gamma\{x := \rho_1\} \rhd M : \rho_2}{\Gamma \rhd (\lambda x : \rho_1.\, M) : \rho_1 \to \rho_2}$$

$$(\text{APP})\quad \frac{\Gamma \rhd M_1 : \rho_1 \to \rho_2 \qquad \Gamma \rhd M_2 : \rho_1}{\Gamma \rhd (M_1\ M_2) : \rho_2}$$

$$(\text{LET})\quad \frac{\Gamma \rhd M_1 : \pi \qquad \Gamma\{x := \pi\} \rhd M_2 : \rho}{\Gamma \rhd \textbf{let } x : \pi = M_1 \textbf{ in } M_2 \textbf{ end} : \rho}$$

We write $\boldsymbol{MH} \vdash \Gamma \rhd M : \pi$ if $\Gamma \rhd M : \pi$ is derivable from the above typing rules.

Define the *type erasure* of a pre-term M, denoted by $erase(M)$, as follows:

$$\begin{array}{rcl}
erase(x) & = & x \\
erase((M_1\ M_2)) & = & (erase(M_1)\ erase(M_2)) \\
erase((\lambda x : \rho.\, M)) & = & (\lambda x.\, erase(M)) \\
erase((\lambda t.\, M)) & = & erase(M) \\
erase((M\ \rho)) & = & erase(M) \\
erase(\textbf{let } x : \pi = M_1 \textbf{ in } M_2 \textbf{ end}) & = & \textbf{let } x = erase(M_1) \textbf{ in } erase(M_2) \textbf{ end}
\end{array}$$

They showed the following relationships between Core-XML and Damas-Milner system.

Theorem 3.1 (Mitchell-Harper) *If* $\boldsymbol{MH} \vdash \Gamma \rhd M : \pi$ *then* $\boldsymbol{DM} \vdash \Gamma \rhd erase(M) : \pi$. *If* $\boldsymbol{DM} \vdash \Gamma \rhd e : \pi$ *then there exists a Core-XML pre-term* M *such that* $erase(M) \equiv e$ *and* $\boldsymbol{MH} \vdash \Gamma \rhd M : \pi$. *Moreover,* M *can be computed effectively from a proof of* $\Gamma \rhd e : \pi$. ∎

Based on this relationship, they concluded that Core-XML and Damas-Milner system are "equivalent" and regarded ML as a "convenient shorthand" for Core-XML.

If we could indeed regard ML terms as syntactic shorthands for Core-XML terms then equational theory and model theory could be those of Core-XML. Core-XML is a restricted form of the second-order lambda calculus whose equational theory and model theory are well investigated (see [20] and references therein). However, the above result does not establish any syntactic mapping from Damas-Milner terms to Core-XML terms. It only established a correspondence between Core-XML terms and *derivations* of Damas-Milner terms, which can be infinitely many for a single Damas-Milner term. This means that there are, in general, infinitely many distinct Core-XML terms that correspond to a given Damas-Milner term. For example, consider the Damas-Milner term:

$$\emptyset \rhd (\lambda x \lambda y.\, y)(\lambda x.\, x) \;:\; t \rightarrow t.$$

Any Core-XML term of the form

$$(\lambda x : \rho \rightarrow \rho \lambda y : t.\, y)(\lambda x : \rho.\, x)$$

for any type-scheme ρ corresponds to the above typing.

One way to overcome this difficulty is to *choose* a particular Core-XML term among possibly infinitely many choices. Such a choice seems possible if we assume a particular type inference algorithm, but we would like to avoid such an assumption as a part of a formal characterization of ML. Another possibility would be to consider a Damas-Milner term as an equivalence class of Core-XML terms. One plausible equivalence relation is the convertibility (or equality) relation. If a Damas-Milner term corresponds to a (subset of) convertibility class of Core-XML terms then any model of Core-XML in which the convertibility relation is valid yields a semantics of Damas-Milner terms. Unfortunately, however, a Damas-Milner term in general do not correspond to a (subset of) convertibility class of Core-XML terms. As a counter example, consider the following Damas-Milner term:

$$\{x : \forall t.\, t \rightarrow int, y : \forall t.\, t \rightarrow t\} \rhd (x\ y) \;:\; int.$$

The following two Core-XML terms both correspond to derivations of the above term:

$$\{x : \forall t.\, t \rightarrow int, y : \forall t.\, t \rightarrow t\} \rhd ((x\ (bool \rightarrow bool))\ (y\ bool)) \;:\; int,$$

$$\{x : \forall t.\, t \rightarrow int, y : \forall t.\, t \rightarrow t\} \rhd ((x\ (int \rightarrow int))\ (y\ int)) \;:\; int.$$

But these two terms are both in normal form and are therefore not convertible.

We also think that Damas-Milner system and the corresponding explicitly-typed language Core-XML are too strong to explain ML's type system. As argued by Milner in [78], it is ML's unique feature and advantage that ML supports polymorphism without introducing explicit type abstraction and type application. Note that this account of ML only used non-generic type-schemes. As such a language, ML can be better understood without using generic type-schemes, whose semantics requires the construction of very large spaces.

3.1.3 A simple framework for ML polymorphism

From the above analyses, it appears that ML is different from both untyped languages and explicitly typed languages. In order to understand ML properly we will develop a framework for semantics that accounts for ML's implicit type system. Such a semantics should be useful to reason about various properties of ML programs including equality on programs and operational semantics. A strategy was already suggested in Mitchell-Harper approach. We can use an explicitly typed language as an "intermediate language" to define a semantics of ML. In this chapter, we use the simply typed lambda calculus. Usage of the simply typed lambda calculus to explain ML polymorphism was suggested in Wand's analysis [110], where ML terms are regarded as shorthands for terms in the simply typed lambda calculus. Wand's approach, however, shares the same difficulty as in Mitchell-Harper's analysis. It only gives meanings to derivations. Moreover, it does not deal with polymorphic terms, i.e. those terms whose type-schemes contain type variables.

We first define an inference system and semantics of ML *typings* (typing schemes that do not contain type variables) and then generalize them to ML terms (i.e. typing schemes). Parallel to the relationship between Damas-Milner system and Core-XML, derivations of typings in our system correspond to terms of the simply typed lambda calculus. Here is the crucial point in the development of our semantic approach: we show that if two typed terms correspond to derivations of a same ML typing then they are β-convertible (theorem 3.7). This guarantees that any semantics of the simply typed lambda calculus, in which the rule (β) is sound, indeed yields a semantics of ML typings. We regard a general ML term as a representation of a *set* of typings. The denotation of an ML term is then defined as the set of denotations of the typings indexed by the set of types represented by its type-scheme. For example, we regard the denotation $[\![\emptyset \rhd \lambda x.\, x : t \rightarrow t]\!]$ as the set $\{(\tau \rightarrow \tau, [\![\lambda x : \tau.\, x]\!]) | \tau \in Type\}$.

Equational theories are defined not on raw terms but on typing schemes. Two typing schemes are equal iff their type-schemes are equal and raw terms are convertible to each other. This definition correctly models the behavior of ML programs. Type-schemes determine the compile-time behavior of programs and raw terms determine their run-time behavior. We then prove the soundness and completeness of equational theories. This confirms that our notion of semantics precisely captures and justifies the informal intuition behind the behavior of ML programs.

Our semantic framework can be extended to languages with constants, type constructors and recursive types (via infinite regular trees). Our semantic framework can also be related to certain operational semantics. We show that if a semantics of the typed lambda calculus is fully abstract with respect to an operational semantics then the corresponding semantics of ML is also fully abstract with respect to an operational semantics that satisfies certain reasonable properties in connection with the operational semantics of the typed lambda calculus. This results enables us to transfer various existing results for full abstraction of typed languages to ML-like languages. A limitation to this program is due to the fact that our interpretation needs the soundness of the rule (β). Such models, of course, while good for "call-by-name" evaluation, are not computationally adequate for the usual "call-by-value" evaluation of ML programs. Thus, our full abstraction result seems helpful only for ML-like languages with "lazy" evaluation strategy such as Miranda [107], Lazy ML [11] and Haskel [54].

3.2 The Language Core-ML

We first present our framework for the set of pure raw terms, the same set analyzed in [34, 79]. We call the pure language Core-ML. Later in section 3.5 we extend our frameworks to a language allowing constants, arbitrary set of type constructors and recursive types (via regular trees).

3.2.1 Raw terms, types and type-schemes

We assume that we are given a countably infinite set of variables Var (ranged over by x).

Definition 3.4 (Raw Terms of Core-ML) *The set of raw terms of Core-ML (ranged over by e) is defined by the following abstract syntax:*

$$e ::= x \mid (e\ e) \mid \lambda x.\, e \mid \mathbf{let}\ x = e\ \mathbf{in}\ e\ \mathbf{end}$$

The notion of bound variables and free variables in a term are defined as in the lambda calculus [49, 15] with the additional rule that x in $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end}$ is a bound variable. We write $FV(e)$ for the set of free variables in e. We write $e[e_1/x_1, \ldots, e_n/x_n]$ for the raw term obtained from e by simultaneously replacing free occurrences of $x_1, \ldots, x_n$ by $e_1, \ldots, e_n$ with necessary bound variable renaming.

In order to show various properties of raw terms by induction, we define their complexity measure. We first define *let degree* that is to measure nesting of let expressions. A *let dgree assignmen* L is a function form a subset of variables to natural numbers. Define *let dgree of e under L*, denoted by $ld(L, e)$, by induction on the structure of e as follows:

$$ld(L, x) = \begin{cases} 0 & \text{if } x \notin dom(L) \\ L(x) & \text{if } x \in dom(L) \end{cases}$$

$$
\begin{aligned}
ld(L,(e_1\ e_2)) &= ld(L,e_1)+ld(L,e_2)\\
ld(L,\lambda x.\,e) &= ld(L\lceil^{dom(L)\setminus\{x\}},e)\\
ld(L,\textbf{let } x=e_1 \textbf{ in } e_2 \textbf{ end}) &= 1+ld(L\{x:=ld(L,e_1)\},e_2)
\end{aligned}
$$

For this mesurement, we have the following substitution property:

Lemma 3.1 *For any raw terms e_1, e_2,*

$$ld(L\{x:=ld(L,e_1)\},e_2)=ld(L,e_2[e_1/x]).$$

Proof The proof is by induction on the structure of e_2. We only show the case for $e_2 \equiv \textbf{let } x = e_2^1 \textbf{ in } e_2^2 \textbf{ end}$. Other cases can be easily shown by using the induction hypothesis and the definition of $ld(L,e)$. Suppose $e_2 \equiv \textbf{let } y = e_2^1 \textbf{ in } e_2^2 \textbf{ end}$:

1. Subcase $x = y$:

$$
\begin{aligned}
&ld(L\{x:=ld(L,e_1)\},\textbf{let } x=e_2^1 \textbf{ in } e_2^2 \textbf{ end})\\
&= 1+ld(L\{x:=ld(L,e_1)\}\{x:=ld(L\{x:=ld(L,e_1)\},e_2^1)\},e_2^2)\\
&\qquad \text{(by the definition of } ld(L,e))\\
&= 1+ld(L\{x:=ld(L\{x:=ld(L,e_1)\},e_2^1)\},e_2^2)\\
&= 1+ld(L\{x:=ld(L,e_2^1[e_1/x])\},e_2^2)\\
&\qquad \text{(by the induction hypothesis)}
\end{aligned}
$$

But since $(\textbf{let } x = e_2^1 \textbf{ in } e_2^2 \textbf{ end})[e_1/x] \equiv \textbf{let } x = e_2^1[e_1/x] \textbf{ in } e_2^2 \textbf{ end}$, by the definition of $ld(L,e)$,

$$ld(L,(\textbf{let } x=e_2^1 \textbf{ in } e_2^2 \textbf{ end})[e_1/x]) = 1+ld(L\{x:=ld(L,e_2^1[e_1/x])\},e_2^2).$$

Therefore $ld(L\{x:=ld(L,e_1)\},e_2)=ld(L,e_2[e_1/x])$.

2. Subcase $x \neq y$:

$$
\begin{aligned}
&ld(L\{x:=ld(L,e_1)\},\textbf{let } y=e_2^1 \textbf{ in } e_2^2 \textbf{ end})\\
&= 1+ld(L\{x:=ld(L,e_1)\}\{y:=ld(L\{x:=ld(L,e_1)\},e_2^1)\},e_2^2)\\
&\qquad \text{(by the definition of } ld(L,e))\\
&= 1+ld(L\{y:=ld(L\{x:=ld(L,e_1)\},e_2^1)\}\{x:=ld(L,e_1)\},e_2^2)\\
&\qquad \text{(since } x\neq y)\\
&= 1+ld(L\{y:=ld(L\{x:=ld(L,e_1)\},e_2^1)\},e_2^2[e_1/x])
\end{aligned}
$$

(by the induction hypothesis)

$$= \quad 1 + ld(L\{y := ld(L, e_2^1[e_1/x])\}, e_2^2[e_1/x])$$

(by the induction hypothesis)

But since (**let** $y = e_2^1$ **in** e_2^2 **end**)$[e_1/x]$ = **let** $y = e_2^1[e_1/x]$ **in** $e_2^2[e_1/x]$ **end**, by the definition of $ld(L, e)$,

$$ld(L, (\textbf{let } y = e_2^1 \textbf{ in } e_2^2 \textbf{ end})[e_1/x]) = 1 + ld(L\{y := ld(L, e_2^1[e_1/x])\}, e_2^2[e_1/x]).$$

Therefore $ld(L\{x := ld(L, e_1)\}, e_2) = ld(L, e_2[e_1/x])$.

▮

Definition 3.5 (Complexity Measure of Raw Terms) *The complexity of a raw term e is the lexicographical pairing of $ld(\emptyset, e)$ and the size of e.*

By lemma 3.1, we immediately have the following property of the complexity of raw terms:

Proposition 3.1 *1. For any raw term e_1, e_2, the complexity of* **let** $x = e_1$ **in** e_2 **end** *is strictly greater than that of $e_2[e_1/x]$.*

2. For any raw term e, the complexity of e is strictly greater than that of any proper subterm of e. ▮

The intended meaning of **let** $x = e_1$ **in** e_2 **end** is to bind x to e_1 in e_2 and to denote operationally the expression $e_2[e_1/x]$. For a raw term e, the *let expansion* of e, *letexpd*(e), is the raw term without **let**-expression obtained from e by repeatedly replacing the outmost subterm of the form **let** $x = e_1$ **in** e_2 **end** by $e_2[e_1/x]$.

Proposition 3.2 *For any raw term e, letexpd(e) exists.*

Proof Sinse each expansion step strictly reduces the complexity of the raw term, the expansion process terminates, yielding a unique raw term. ▮

We assume that we are given a set $\mathcal{B}$ of base types (ranged over by b) and a countably infinite set $Tvar$ of type variables (ranged over by t).

Definition 3.6 (Types and Type-schemes of Core-ML) *The set of types, Type ranged over by τ, is given by the following abstract syntax:*

$$\tau ::= b \mid \tau \rightarrow \tau.$$

The set of type-schemes, *Tscheme ranged over by* ρ, *is given by the following abstract syntax:* —

$$\rho ::= t \mid b \mid \rho \to \rho.$$

Since *Tscheme* can be regarded as a set of trees, the notion of substitutions, instances e.t.c. we defined in section 2.2.2 apply to *Tscheme*.

3.2.2 Typings, typing schemes and terms of Core-ML

A *type assignment* $\mathcal{A}$ is a function from a finite subset of *Var* to *Type*.

Definition 3.7 (Core-ML Typings) *A typing is a formula of the form* $\mathcal{A} \rhd e : \tau$ *that is derivable in the following proof system:*

(VAR) $\qquad \mathcal{A} \rhd x : \tau \quad$ *if* $\mathcal{A}(x) = \tau$

(APP) $$\frac{\mathcal{A} \rhd e_1 : \tau_1 \to \tau_2 \qquad \mathcal{A} \rhd e_2 : \tau_1}{\mathcal{A} \rhd (e_1\ e_2) : \tau_2}$$

(ABS) $$\frac{\mathcal{A}\{x := \tau_1\} \rhd e : \tau_2}{\mathcal{A} \rhd \lambda x.\, e : \tau_1 \to \tau_2}$$

(LET) $$\frac{\mathcal{A} \rhd e_1[e_2/x] : \tau \qquad \mathcal{A} \rhd e_2 : \tau'}{\mathcal{A} \rhd \textbf{let } x = e_2 \textbf{ in } e_1 \textbf{ end} : \tau}$$

In the rule (LET), τ' may be any type. We write $\boldsymbol{ML} \vdash \mathcal{A} \rhd e : \tau$ if $\mathcal{A} \rhd e : \tau$ is derivable in the above proof system. A *derivation* Δ of $\mathcal{A} \rhd e : \tau$ is a proof tree for $\mathcal{A} \rhd e : \tau$ in the above proof system.

For this proof system we have the following properties:

Lemma 3.2 *If* $\boldsymbol{ML} \vdash \mathcal{A} \rhd e : \tau$ *then* $dom(\mathcal{A}) \supseteq FV(e)$.

Proof By induction on the complexity of e. ▌

Lemma 3.3 *If* $\boldsymbol{ML} \vdash \mathcal{A} \rhd e : \tau$ *then* $\boldsymbol{ML} \vdash \mathcal{A}' \rhd e : \tau$ *for any* $\mathcal{A}'$ *such that* $\mathcal{A} \subseteq \mathcal{A}'$ *(as graphs).*

Proof By induction on the complexity of e. ▌

A *type assignment scheme* Σ is a function from a finite subset of *Var* to *Tscheme*.

Definition 3.8 (Typing schemes) *A typing scheme is a formula of the form* $\Sigma \rhd e : \rho$ *such that for any substitution* θ *ground for* Σ *and* ρ, $\boldsymbol{ML} \vdash \theta(\Sigma) \rhd e : \theta(\rho)$.

In other words, a typing scheme is a formula whose ground instances are typings. We write $\boldsymbol{ML} \vdash \Sigma \rhd e : \rho$ if $\Sigma \rhd e : \rho$ is a typing scheme.

In [78, 34], an ordering is defined only on type-schemes (generic type-schemes) and a type inference algorithm is defined with respect to a given type assignment. Here, we follow [79] and generalize the ordering on type-schemes ($\precsim$, definition 2.9) to typing schemes and characterize the type inference problem based on the ordering.

Definition 3.9 (Preoreder on Typing schemes) *A typing scheme* $\Sigma_1 \rhd e : \rho_1$ *is more general than a typing scheme* $\Sigma_2 \rhd e : \rho_2$, *denoted by* $\Sigma_2 \rhd e : \rho_2 \precsim_{\text{ML}} \Sigma_1 \rhd e : \rho_1$, *if* $(\Sigma_2\lceil^{dom(\Sigma_1)}, \rho_2) \precsim (\Sigma_1, \rho_1)$.

We also use $\precsim_{\text{ML}}$ as a relation on the set of pairs of a type assignment scheme and a type-scheme and write $(\Sigma_2, \rho_2) \precsim_{\text{ML}} (\Sigma_1, \rho_1)$ if if $(\Sigma_2\lceil^{dom(\Sigma_1)}, \rho_2) \precsim (\Sigma_1, \rho_1)$. Note that more general also means less entries in a type assignment scheme. A typing scheme $\Sigma \rhd e : \rho$ is *most general* (or *principal*) if $\Sigma' \rhd e : \rho' \precsim_{\text{ML}} \Sigma \rhd e : \rho$ for any typing scheme $\Sigma' \rhd e : \rho'$. We then have:

Proposition 3.3 *If* $\Sigma \rhd e : \rho$ *is a principal typing scheme then* $\{\mathcal{A} \rhd e : \tau | \mathcal{A} \rhd e : \tau \precsim_{\text{ML}} \Sigma \rhd e : \rho\} = \{\mathcal{A} \rhd e : \tau | \boldsymbol{ML} \vdash \mathcal{A} \rhd e : \tau\}$.

Proof The inclusion

$$\{\mathcal{A} \rhd e : \tau | \mathcal{A} \rhd e : \tau \precsim_{\text{ML}} \Sigma \rhd e : \rho\} \subseteq \{\mathcal{A} \rhd e : \tau | \boldsymbol{ML} \vdash \mathcal{A} \rhd e : \tau\}$$

is by definition of typing schemes and by lemma 3.3. The inverse inclusion is an immediate consequence of the facts that typings are also typing schemes and $\Sigma \rhd e : \rho$ is principal. ▮

This means that a principal typing scheme represents the set of all provable typings. In what follows, we regard typing schemes as representatives of equivalence classes under the preorder $\precsim_{\text{ML}}$. This equivalence relation corresponds to the relation induced by renaming of type variables (without "collapsing" distinct variables).

Definition 3.10 (Terms of Core-ML) *Terms of Core-ML are (not necessarily principal) typing schemes.*

According to this definition, it is not clear whether it is decidable or not that a given string of symbols is a term or not. The answer is positive as a consequence of the decidability of type-checking problem below.

Non principal typing schemes correspond to programs with (partial) type specifications which are supported in ML and can be easily added to our definition. A term containing type variables corresponds to a polymorphic program in Core-ML. A raw term e in a term $\Sigma \rhd e : \rho$ represents the computational contents of the term and determines its run-time behavior. The pair (Σ, ρ) represents the typing contexts in which e is meaningful and determines the compile-time behavior of the term.

3.2.3 Type Inference Problem

Under our characterization, the problem of type-checking in Core-ML is stated as follows:

> given a type assignment scheme Σ, a raw term e and a type-scheme ρ, determine whether $\boldsymbol{ML} \vdash \Sigma \rhd e : \rho$ or not.

The type inference problem is stated as follows:

> given a raw term e, determine the set $\{(\Sigma, \rho) | \boldsymbol{ML} \vdash \Sigma \rhd e : \rho\}$.

The following theorem, which is essentially due to Hindley [50], solves both of the problems:

Theorem 3.2 *There is an algorithm $\mathcal{PTS}$ which, given any raw term e, yields either failure or (Σ, ρ) such that if $\mathcal{PTS}(e) = (\Sigma, \rho)$ then $\Sigma \rhd e : \rho$ is a principal typing scheme otherwise e has no typing.*

Proof In the following proof, we assume a linear order $<$ on Var and treat (Σ, ρ) as a tree $((x_1 : \rho_1), \dots, (x_n, \rho_n), \rho)$ where $\{x_1, \dots, x_n\} = dom(\Sigma), \Sigma(x_i) = \rho_i$ and $x_1 < \cdots < x_n$. Algorithm $\mathcal{PTS}$ is defined by cases. In the following description of the algorithm, if unification $\mathcal{U}(\dots)$ fails then the algorithm returns *failure*.

$$\mathcal{PTS}(e) = (\Sigma, \rho) \text{ where}$$

(1) Case $e \equiv x$:

$\Sigma = \{x := t\}$ (t fresh)

$\rho = t$

```
(2) Case $e \equiv (e_1\ e_2)$:
    let
        $(\Sigma_1, \rho_1) = \mathcal{PTS}(e_1)$
        $(\Sigma_2, \rho_2) = \mathcal{PTS}(e_2)$
        $\Sigma_1' = \Sigma_1\{x_1^1 := t_1^1, \ldots, x_n^1 := t_n^1\}$ where
            $\{x_1^1, \ldots, x_n^1\} = dom(\Sigma_2) \setminus dom(\Sigma_1)$, $(t_1^1, \ldots, t_n^1$ fresh)
        $\Sigma_2' = \Sigma_2\{x_1^2 := t_1^2, \ldots, x_m^2 := t_m^2\}$ where
            $\{x_1^2, \ldots, x_m^2\} = dom(\Sigma_1) \setminus dom(\Sigma_2)$, $(t_1^2, \ldots, t_m^2$ fresh)
        $\theta = \mathcal{U}((\Sigma_1', \rho_1), (\Sigma_2', \rho_2 \rightarrow t))$ ($t$ fresh)
    in
        $\Sigma = \theta(\Sigma_1')$,
        $\rho = \theta(t)$.

(3) Case $\equiv \lambda x.\, e_1$:
    let
        $(\Sigma_1, \rho_1) = \mathcal{PTS}(e_1)$
    in
        if $x \in dom(\Sigma_1)$ then
            $\Sigma = \Sigma_1 \restriction^{dom(\Sigma_1) \setminus \{x\}}$,
            $\rho = \Sigma_1(x) \rightarrow \rho_1$
        else
            $\Sigma = \Sigma_1$,
            $\rho = t \rightarrow \rho_1$ ($t$ fresh).

(4) Case $e \equiv$ **let** $x = e_1$ **in** $e_2$ **end**:
    let
        $(\Sigma_1, \rho_1) = \mathcal{PTS}(e_1)$
        $(\Sigma_2, \rho_2) = \mathcal{PTS}(e_2[e_1/x])$
        $\Sigma_1' = \Sigma_1\{x_1^1 := t_1^1, \ldots, x_n^1 := t_n^1\}$ where
            $\{x_1^1, \ldots, x_n^1\} = dom(\Sigma_2) \setminus dom(\Sigma_1)$, $(t_1^1, \ldots, t_n^1$ fresh)
        $\Sigma_2' = \Sigma_2\{x_1^2 := t_1^2, \ldots, x_m^2 := t_m^2\}$ where
            $\{x_1^2, \ldots, x_m^2\} = dom(\Sigma_1) \setminus dom(\Sigma_2)$, $(t_1^2, \ldots, t_m^2$ fresh)
        $\theta = \mathcal{U}(\Sigma_1', \Sigma_2')$
```

in

$\Sigma = \theta(\Sigma_2'),$
$\rho = \theta(\rho_2).$

Since in each case $\mathcal{PTS}$ is called on raw terms with strictly smaller complexity, the algorithm always terminates.

In order to show the desired property of the algorithm $\mathcal{PTS}$ we use the following results:

Lemma 3.4 *If $\mathcal{PTS}(e) = (\Sigma, \rho)$ then $FV(e) = dom(\Sigma)$.*

Proof By induction on the complexity of e. The basis is trivial. The induction step is by cases in terms of the structure of e. Cases other than $\lambda x.\, e'$ are immediate consequences of the induction hypothesis. Suppose $e \equiv \lambda x.\, e'$. By the induction hypothesis, $FV(e') = dom(\Sigma_1)$. If $x \in FV(e')$ then $dom(\Sigma) = dom(\Sigma_1) \setminus \{x\} = FV(e') \setminus \{x\} = FV(\lambda x.\, e')$ otherwise $dom(\Sigma) = dom(\Sigma_1) = FV(e') = FV(\lambda x.\, e')$. ▮

We also use the following property of the unification algorithm $\mathcal{U}$, which follows directly from theorem 2.3:

Proposition 3.4 *Let a, b be terms that do no share common variables. For any term c, $c \precsim a, c \precsim b$ iff $\mathcal{U}(a,b) = \theta$ and $c \precsim \theta(a)$ for some θ.* ▮

Using these properties, we show the necessary property of the algorithm by showing the property of principal typing scheme: $\boldsymbol{ML} \vdash \mathcal{A} \rhd e : \tau$ iff $\mathcal{A} \rhd e : \tau \precsim_{\mathrm{ML}} \Sigma \rhd e : \rho$ (proposition 3.3). Proof is by cases in terms of the structure of e. The type assignment schemes (Σ,Σ_1 etc), type-schemes (ρ, ρ_1 etc) and type variables (t, t_1 etc) in the following proof refer to those in the corresponding cases of the description of the algorithm.

1. Case $e \equiv x$: $\boldsymbol{ML} \vdash \mathcal{A} \rhd x : \tau$ iff $x \in dom(\mathcal{A})$ and $\mathcal{A}(x) = \tau$. Then by the definition of $\precsim_{\mathrm{ML}}$, $\boldsymbol{ML} \vdash \mathcal{A} \rhd x : \tau$ iff $(\mathcal{A}, \tau) \precsim_{\mathrm{ML}} (\{x := t\}, t)$.

2. Case $e \equiv (e_1\ e_2)$: By the typing rules, $\boldsymbol{ML} \vdash \mathcal{A} \rhd (e_1\ e_2) : \tau$ iff

   (A) there is some τ_1 such that $\boldsymbol{ML} \vdash \mathcal{A} \rhd e_1 : \tau_1 \rightarrow \tau$ and $\boldsymbol{ML} \vdash \mathcal{A} \rhd e_2 : \tau_1$.

   By the induction hypothesis, (A) iff

   (B) there is some τ_1 such that $(\mathcal{A}, \tau_1 \rightarrow \tau) \precsim_{\mathrm{ML}} (\Sigma_1, \rho_1)$, $(\mathcal{A}, \tau_1) \precsim_{\mathrm{ML}} (\Sigma_2, \rho_2)$.

By lemma 3.2 and lemma 3.4, $dom(\mathcal{A}) \supseteq FV(e_1) \cup FV(e_2) = dom(\Sigma_1) \cup dom(\Sigma_2)$. Then by the constructions of Σ_1', Σ_2', (B) iff

(C) there is some τ_1 such that $(\mathcal{A}, \tau_1 \rightarrow \tau) \precsim_{\text{ML}} (\Sigma_1', \rho_1)$ and $(\mathcal{A}, \tau_1) \precsim_{\text{ML}} (\Sigma_2', \rho_2)$.

Since t introduced in the algorithm is fresh, (C) iff

(D) there is some τ_1 such that $(\mathcal{A}, \tau_1 \rightarrow \tau) \precsim_{\text{ML}} (\Sigma_1', \rho_1)$ and $(\mathcal{A}, \tau_1 \rightarrow \tau) \precsim_{\text{ML}} (\Sigma_2', \rho_2 \rightarrow t)$.

By definition of $\mathcal{PTS}$, (Σ_1', ρ_1) and $(\Sigma_2', \rho_2 \rightarrow t)$ do not share type variables. Therefore by proposition 3.4, (D) iff there is some τ_1 such that $(\mathcal{A}, \tau_1 \rightarrow \tau) \precsim_{\text{ML}} (\theta(\Sigma_2'), \theta(\rho_2 \rightarrow t))$ where $\mathcal{U}((\Sigma_1', \rho_1), (\Sigma_2', \rho_2 \rightarrow t)) = \theta$. Therefore $\boldsymbol{ML} \vdash \mathcal{A} \triangleright (e_1\ e_2) : \tau$ iff $(\mathcal{A}, \tau) \precsim_{\text{ML}} (\Sigma, \rho)$.

3. Case $e \equiv \lambda x.\, e_1$: By the typing rules, $\boldsymbol{ML} \vdash \mathcal{A} \triangleright \lambda x.\, e_1 : \tau$ iff there are some τ_1, τ_2 such that $\tau = \tau_1 \rightarrow \tau_2$ and $\boldsymbol{ML} \vdash \mathcal{A}\{x := \tau_1\} \triangleright e_1 : \tau_2$. By the induction hypothesis, $\boldsymbol{ML} \vdash \mathcal{A}\{x := \tau_1\} \triangleright e_1 : \tau_2$ iff $(\mathcal{A}\{x := \tau_1\}, \tau_2) \precsim_{\text{ML}} (\Sigma_1, \rho_1)$. Suppose $x \in dom(\Sigma_1)$. Then $(\mathcal{A}\{x := \tau_1\}, \tau_2) \precsim_{\text{ML}} (\Sigma_1, \rho_1)$ iff $(\mathcal{A}, \tau_1 \rightarrow \tau_2) \precsim_{\text{ML}} (\Sigma_1\lceil^{dom(\Sigma_1)\setminus\{x\}}, \Sigma_1(x) \rightarrow \rho_1)$. Suppose $x \notin dom(\Sigma_1)$. Then since t is fresh $(\mathcal{A}\{x := \tau_1\}, \tau_2) \precsim_{\text{ML}} (\Sigma_1, \rho_1)$ iff $(\mathcal{A}, \tau_1 \rightarrow \tau_2) \precsim_{\text{ML}} (\Sigma_1, t \rightarrow \rho_1)$. Therefore $\boldsymbol{ML} \vdash \mathcal{A} \triangleright \lambda x.\, e_1 : \tau$ iff $(\mathcal{A}, \tau) \precsim_{\text{ML}} (\Sigma, \rho)$.

4. Case $e \equiv$ **let** $x = e_1$ **in** e_2 **end**. By the typing rules, $\boldsymbol{ML} \vdash \mathcal{A} \triangleright$ **let** $x = e_1$ **in** e_2 **end** $: \tau$ iff

   (A) there is some τ_1 such that $\boldsymbol{ML} \vdash \mathcal{A} \triangleright e_1 : \tau_1$ and $\boldsymbol{ML} \vdash \mathcal{A} \triangleright e_2[e_1/x] : \tau$.

   By the induction hypothesis, (A) iff there is some τ_1 such that $(\mathcal{A}, \tau_1) \precsim_{\text{ML}} (\Sigma_1, \rho_1)$, $(\mathcal{A}, \tau) \precsim_{\text{ML}} (\Sigma_2, \rho_2)$. Similar to the case for $e \equiv (e_1\ e_2)$, $(\mathcal{A}, \tau_1) \precsim_{\text{ML}} (\Sigma_1, \rho_1)$, and $(\mathcal{A}, \tau) \precsim_{\text{ML}} (\Sigma_2, \rho_2)$ iff $(\mathcal{A}, \tau_1) \precsim_{\text{ML}} (\theta(\Sigma_1'), \theta(\rho_1))$, $(\mathcal{A}, \tau) \precsim_{\text{ML}} (\theta(\Sigma_2'), \theta(\rho_2))$ where $\mathcal{U}(\Sigma_1', \Sigma_2') = \theta$. Therefore $\boldsymbol{ML} \vdash \mathcal{A} \triangleright$ **let** $x = e_1$ **in** e_2 **end** $: \tau$ iff $(\mathcal{A}, \tau) \precsim_{\text{ML}} (\Sigma, \rho)$.

(End of the proof of theorem 3.2) ∎

The decidability of the type-checking problem follows from the decidability of the relation $\Sigma_1 \triangleright e : \rho_1 \precsim_{\text{ML}} \Sigma_2 \triangleright e : \rho_2$. The set $\{(\Sigma, \rho) | \boldsymbol{ML} \vdash \Sigma \triangleright e : \rho\}$ is determined by the principal typing scheme using proposition 3.3.

3.2.4 Relation to Damas-Milner System

The typing derivation system for Core-ML is significantly simpler than that of Damas-Milner system and has a particularly simpler proof for the existence of a complete type inference algorithm as demonstrated above. Nevertheless, for closed terms, they are essentially equivalent in the sense of the following two theorems (theorem 3.3 and 3.4). For any generic type-scheme $\pi = \forall t_1 \dots t_n . \rho_0$, define the type scheme ρ_π as $\rho_\pi = \rho_0[t_1 := t'_1, \dots, t_n := t'_n]$ where $t'_1, \dots, t'_n$ are fresh type variables.

Theorem 3.3 *For a closed raw term e, if* $\boldsymbol{DM} \vdash \emptyset \rhd e : \pi$ *then* $\boldsymbol{ML} \vdash \emptyset \rhd e : \rho_\pi$.

Proof The proof uses the following lemmas:

Lemma 3.5 *For any* e, Γ, π, $\boldsymbol{DM} \vdash \Gamma \rhd e : \pi$ *iff* $\boldsymbol{DM} \vdash \Gamma\lceil^{FV(e)} \rhd e : \pi$.

Proof By induction on the height of a derivation of $\Gamma \rhd e : \pi$. ▌

Lemma 3.6 *For any* e, Γ, π, $\boldsymbol{DM} \vdash \Gamma \rhd e : \pi$ *iff* $\boldsymbol{DM} \vdash \Gamma \rhd e : \rho_\pi$.

Proof Suppose $\boldsymbol{DM} \vdash \Gamma \rhd e : \pi$. Then by the rule (INST), $\boldsymbol{DM} \vdash \Gamma \rhd e : \rho_\pi$. Suppose $\boldsymbol{DM} \vdash \Gamma \rhd e : \rho_\pi$. Since $t'_1, \dots, t'_n$ are fresh, by repeated applications of the rule (GEN), $\boldsymbol{DM} \vdash \Gamma \rhd e : \forall t'_1 \dots t'_n . \rho_\pi$. But since $\pi \precsim_{\mathrm{DM}} \forall t'_1 \dots t'_n . \rho_\pi$, by the rule (INST), $\boldsymbol{DM} \vdash \Gamma \rhd e : \pi$. ▌

Lemma 3.7 *If* $\boldsymbol{DM} \vdash \Gamma \rhd e : \pi$ *then it has a derivation such that all applications of the rule* (INST) *are immediately preceded by an instance of the axiom scheme* (VAR).

Proof We first show the following property on typing derivations:

> if a derivation Δ of $\Gamma \rhd e : \rho$ contains a sub-derivation Δ' of $\Gamma \rhd e : \rho'$ then ρ is a substitution instance of ρ'.

By the typing rules, only rules applied between the root of Δ' and the root of Δ are either (GEN) or (INST). The property is then shown by a simple induction on the number of the rules applied between the two roots.

The lemma is proved by induction on the structure of e using the above property.

1. Case of $e \equiv x$: Any derivation of $\Gamma \rhd x : \pi$ must have the node $\Gamma \rhd x : \Gamma(x)$ at its leaf and can only contain applications of the rules (GEN) and (INST). It is then shown by induction that Γ must satisfies either $\pi \precsim_{\mathrm{DM}} \Gamma(x)$ or $\pi = \Gamma(x)$. If $\Gamma(x) = \pi$ then the one node typing derivation tree

$$\frac{}{\Gamma \rhd x : \pi} \quad (\mathrm{VAR})$$

satisfies the condition. Otherwise $\pi \preceq_{DM} \Gamma(x)$ then the following derivation tree satisfies the condition:

$$\cfrac{\cfrac{}{\Gamma \rhd x : \Gamma(x)} \quad (\text{VAR})}{\Gamma \rhd x : \pi} \quad (\text{INST})$$

2. Case of $e \equiv (e_1\ e_2)$: By the typing rules, lemma 3.6 and the property shown in the beginning of the proof, there are derivations Δ_1, Δ_2 respectively of $\Gamma \rhd e_1 : \rho_1 \rightarrow \rho_\pi$ and $\Gamma \rhd e_2 : \rho_1$ for some ρ_1. Then by the induction hypothesis, there are derivations Δ_1', Δ_2' of $\Gamma \rhd e_1 : \rho_1 \rightarrow \rho_\pi$ and $\Gamma \rhd e_2 : \rho_1$ satisfying the condition. Then we have the following derivation of $\Gamma \rhd (e_1\ e_2) : \rho_\pi$ satisfies the condition:

$$\frac{\Delta_1' \qquad \Delta_2'}{\Gamma \rhd (e_1\ e_2) : \rho_\pi} \quad (\text{APP})$$

Since the type variables introduced in ρ_π are fresh, by repeated applications of the rule (GEN), we have a derivation of $\Gamma \rhd (e_1\ e_2) : \pi$ satisfying the condition.

3. Case of $e \equiv \lambda x.e$: By the typing rules and the property shown in the beginning of the proof, there is a derivation Δ of $\Gamma\{x := \rho_1\} \rhd e : \rho_2$ such that $\rho_\pi = \rho_1 \rightarrow \rho_2$. By the induction hypothesis, there is also a derivation Δ' of $\Gamma\{x := \rho_1\} \rhd e : \rho_2$ satisfying the condition. Then we have the following derivation of $\Gamma \rhd \lambda x.\, e : \rho_1 \rightarrow \rho_2$ satisfies the condition:

$$\frac{\Delta'}{\Gamma \rhd \lambda x.\, e : \rho_1 \rightarrow \rho_2} \quad (\text{ABS})$$

Similar to the case for $e \equiv (e_1\ e_2)$, there is a derivation of $\Gamma \rhd \lambda x.\, e : \pi$ satisfying the condition.

4. Case of $e \equiv$ **let** $x = e_1$ **in** e_2 **end**: By the typing rules and the property shown in the beginning of the proof, $\boldsymbol{DM} \vdash \Gamma\{x := \pi'\} \rhd e_2 : \rho_\pi$ and $\boldsymbol{DM} \vdash \Gamma \rhd e_1 : \pi'$ for some π'. By the induction hypothesis, there are derivations Δ_1, Δ_2 respectively of $\Gamma\{x := \pi'\} \rhd e_2 : \rho$ and $\Gamma \rhd e_1 : \pi'$ satisfying the condition. Then we have the following derivation of $\Gamma \rhd$ **let** $x = e_1$ **in** e_2 **end** $: \rho_\pi$ satisfies the condition:

$$\frac{\Delta_1 \qquad \Delta_2}{\Gamma \rhd \textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end} : \rho_\pi} \quad (\text{LET})$$

Similar to the case for $e \equiv (e_1\ e_2)$, there is a derivation of $\Gamma \rhd$ **let** $x = e_1$ **in** e_2 **end** $: \pi$ satisfying the condition.

■

Lemma 3.8 *There is some π_1 such that $\boldsymbol{DM} \vdash \Gamma \rhd e_1 : \pi_1$ and $\boldsymbol{DM} \vdash \Gamma\{x := \pi_1\} \rhd e_2 : \pi$ iff there is some π_2 such that $\boldsymbol{DM} \vdash \Gamma \rhd e_1 : \pi_2$ and $\boldsymbol{DM} \vdash \Gamma \rhd e_2[e_1/x] : \pi$.*

Proof In order to prove this lemma, we need the follopwing theorem proved by Damas and Milner [34]:

Proposition 3.5 (Damas-Milner) *Let Γ be any type assignment scheme. If e has a typing under Γ then there is a generic type scheme π such that for any typing scheme $\boldsymbol{DM} \vdash \Gamma \rhd e : \pi'$, $\pi' \precsim_{\mathsf{DM}} \pi$.* ■

This is a direct corollaries of a stronger theorem shown in [34]. If e has a typing scheme under Γ, then we call a generic type scheme satisfying the above property as *principal typing scheme under* Γ. Using this proposition, we proved the lemma by showing the following stronger property:

> Let π_1 be a principal type scheme of e_1 under Γ. $\boldsymbol{DM} \vdash \Gamma\{x := \pi_1\} \rhd e_2 : \pi$ iff $\boldsymbol{DM} \vdash \Gamma \rhd e_2[e_1/x] : \pi$ iff

Since it is easily verified that the provability of typing schemes is preserved by renaming of bound variables, we assume without any loss of generality that x is distinct from all bound variables in e. By lemma 3.5 we can also assume that $x \notin dom(\Gamma)$.

Let Δ be a derivation for $\boldsymbol{DM} \vdash \Gamma\{x := \pi_1\} \rhd e_2 : \pi$. Let Δ' be a tree obtained form Δ by replacing each occurrence of x in Δ by e_1 and deleting all the entry $x := \pi_1$. Only typing rules in $\boldsymbol{DM} \vdash$ that depend on the structure of terms are the rules (VAR) and (ABS). All other typing rules depend only on types of subterms. Therefore the applications of the typing rules in Δ' other than (ABS) and (VAR) remain valid inference steps. Since x is distinct from any bound variables and the rule (ABS) does not depend on the structure of the body e of the lambda term $\lambda y.\, e$, the applications of the rule (ABS) in Δ' are also valid inference steps. Δ' is therefore a valid derivation tree *except for the subtrees of the form*:

$$\frac{}{\Gamma \rhd e_2[e_1/x] : \pi_1} \quad (\mathsf{VAR})$$

Now let Δ'' be the tree obtained from Δ' by replacing all subtrees of the above form by a derivation for

$$\Gamma \rhd e_1 : \pi_1$$

Since π_1 is a principal typing scheme under Γ such a tree always exists. Then Δ'' is a valid derivation tree for $\boldsymbol{DM} \vdash \Gamma \rhd e_2[e_1/x] : \pi$. ▮▮

We now prove the theorem. Suppose $\boldsymbol{DM} \vdash \Gamma \rhd e : \pi$. By lemma 3.5 and lemma 3.6, $\boldsymbol{DM} \vdash \emptyset \rhd e : \rho_\pi$. We need to show $\boldsymbol{ML} \vdash \emptyset \rhd e : \rho_\pi$. Proof is by induction on $ld(\emptyset, e)$.

Basis: By lemma 3.7 and the fact that e does not contain **let**-expression, $\emptyset \rhd e : \rho_\pi$ has a derivation Δ such that it does not contain applications of (GEN) or (INST). This means that any ground instance of Δ is a derivation in $\boldsymbol{ML} \vdash$. Therefore $\boldsymbol{ML} \vdash \emptyset \rhd e : \tau$ for any ground instance $(\mathcal{A}, \tau)$ of (Σ, ρ_π). Hence $\boldsymbol{ML} \vdash \emptyset \rhd e : \rho_\pi$.

Induction Step: Proof is by cases in terms of the structure of e. Cases other than that of $e \equiv$ **let** $x = e_1$ **in** e_2 **end** are immediate consequences of the corresponding induction hypothesis. Suppose $\boldsymbol{DM} \vdash \emptyset \rhd$ **let** $x = e_1$ **in** e_2 **end** $: \rho_\pi$. Then by typing rules and the property shown in the proof of lemma 3.7, $\boldsymbol{DM} \vdash \{x := \pi_1\} \rhd e_2 : \rho_\pi$ and $\boldsymbol{DM} \vdash \emptyset \rhd e_1 : \pi_1$ for some π_1. By lemma 3.8 and 3.6, this is equivalent to $\boldsymbol{DM} \vdash \emptyset \rhd e_2[e_1/x] : \rho_\pi$ and $\boldsymbol{DM} \vdash \emptyset \rhd e_1 : \rho'$ for some ρ'. Since $ld(\emptyset, e_2[e_1/x])$ and $ld(\emptyset, e_1)$ are strictly less than $ld(\emptyset,$ **let** $x = e_1$ **in** e_2 **end**$)$, by the induction hypothesis, $\boldsymbol{ML} \vdash \emptyset \rhd e_2[e_1/x] : \rho_\pi$ and $\boldsymbol{ML} \vdash \emptyset \rhd e_1 : \rho'$ for some ρ'. Then by the rule (LET) in $\boldsymbol{ML} \vdash$, $\boldsymbol{ML} \vdash \emptyset \rhd$ **let** $x = e_1$ **in** e_2 **end** $: \tau$ for any instance τ of ρ_π. This implies $\boldsymbol{ML} \vdash \emptyset \rhd$ **let** $x = e_1$ **in** e_2 **end** $: \rho_\pi$ ▮

Theorem 3.4 *For a closed raw term e, if $\boldsymbol{ML} \vdash \emptyset \rhd e : \rho$ then $\boldsymbol{DM} \vdash \emptyset \rhd e : \rho$.*

Proof By induction on $ld(\emptyset, e)$ using lemma 3.8. If $ld(\emptyset, e) = 0$ then any derivation in $\boldsymbol{ML} \vdash$ is also a derivation in $\boldsymbol{DM} \vdash$. Induction step is by cases in terms of the structure of e. Cases other than **let** $x = e_1$ **in** e_2 **end** are immediate consequences of the induction hypothesis. Suppose $\boldsymbol{ML} \vdash \emptyset \rhd$ **let** $x = e_1$ **in** e_2 **end** $: \rho$. By typing rules in $\boldsymbol{ML} \vdash$, $\boldsymbol{ML} \vdash \emptyset \rhd e_2[e_1/x] : \rho$ and $\boldsymbol{ML} \vdash \emptyset \rhd e_1 : \rho'$ for some ρ'. By the induction hypothesis, $\boldsymbol{DM} \vdash \emptyset \rhd e_2[e_1/x] : \rho$ and $\boldsymbol{DM} \vdash \emptyset \rhd e_1 : \rho'$. Then by lemma 3.8, there is some π such that $\boldsymbol{DM} \vdash \{x := \pi\} \rhd e_2 : \rho$ and $\boldsymbol{DM} \vdash \emptyset \rhd e_1 : \pi$. Then by the rule (LET) in $\boldsymbol{DM} \vdash$, $\boldsymbol{DM} \vdash \emptyset \rhd$ **let** $x = e_1$ **in** e_2 **end** $: \rho$. ▮

As we have demonstrated through theorem 3.2, 3.3, and 3.4, ML's syntactic properties are understood without using generic type-schemes. This corresponds to our semantics which only requires the semantic space of the simply typed lambda calculus. However, our typing derivation system suggests a potentially inefficient type inference algorithm. The algorithm $\mathcal{PTS}$ we defined in theorem 3.2 is indeed potentially inefficient compared to algorithm $\mathcal{W}$ defined in [78]. $\mathcal{PTS}$ infers a typing scheme of **let** $x = e_1$ **in** e_2 **end** by inferring a typing scheme of $e_2[e_1/x]$. This may involves repeated inferences of a typing scheme of e_1 because of multiple occurrences of x in e_2, which is clearly redundant. The extra typing rules for generic type-schemes in Damas-Milner system and the corresponding control structures of the algorithm $\mathcal{W}$ can be regarded as a mechanism to eliminate the redundancy and could be considered as implementation aspects of ML type inference.

3.2.5 Equational theories of Core-ML

An equation of an equational theory of Core-ML is a formula of the form $\Sigma \rhd e_1 = e_2 : \rho$.

Definition 3.11 (ML-theory) *An ML-theory consists of a given set of equations E_{ML} satisfying the properties:*

$$\Sigma \rhd e_1 = e_2 : \rho \in E \textit{ iff for any ground instance } (\mathcal{A}, \tau) \textit{ of } (\Sigma, \rho),\ \mathcal{A} \rhd e_1 = e_2 : \tau \in E$$

and the following set of rules: the axiom schemes $(\alpha), (\beta), (\eta)$, the inference rule scheme (ξ) obtained from respective rule schemes in the untyped lambda calculu [15] by tagging *Σ and ρ, the set of rule schemes for usual equational reasoning (i.e. reflexivity, symmetry, transitivity and congruence), the following axiom scheme:*

$$(let) \qquad \Sigma \rhd (\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end}) = (e_2[e_1/x]) : \rho,$$

and the following inference rule scheme:

$$(thinning) \quad \frac{\Sigma \rhd e_1 = e_2 : \rho}{\Sigma' \rhd e_1 = e_2 : \rho} \quad \textit{if } \Sigma \subseteq \Sigma' \textit{ (as graphs).}$$

We call a set of equations E_{ML} satisfying the above property as a set of *ML-equations*. We write $E_{\mathrm{ML}} \vdash_{\mathrm{ML}} \Sigma \rhd e_1 = e_2 : \rho$ if $\Sigma \rhd e_1 = e_2 : \rho$ is derivable from the axioms and E_{ML} using the inference rules. A set of ML-equations E_{ML} determines the ML-theory $Th_{\mathrm{ML}}(E_{\mathrm{ML}})$. We sometimes regard $Th_{\mathrm{ML}}(E_{\mathrm{ML}})$ as the set of all equations that are provable by the theory.

ML-theory is intended to model euality among terms of Core-ML. For this purpose, we are usually interested in only those euqations that correspond to pairs of Core-ML terms.

Definition 3.12 (Well typed ML-equation) *An equation $\Sigma \rhd e_1 = e_2 : \rho$ is well typed if $\boldsymbol{ML} \vdash \Sigma \rhd e_1 : \rho$ and $\boldsymbol{ML} \vdash \Sigma \rhd e_2 : \rho$.*

We also say that a set of ML equations is well typed if all its elements are well typed. It is easily checked that if E_{ML} is well typed and axioms are restricted to well type equations then $Th_{\mathrm{ML}}(E_{\mathrm{ML}})$

is also well typed. In what follows, we restrict a set of ML-equations E_{ML} and axioms to be well typed ones. For example, $\Sigma \rhd ((\lambda x.\, e_1)\ e_2) = e_1[e_2/x] \ :\ \rho$ is an instance of the axiom shceme (β) only if $\boldsymbol{ML} \vdash \Sigma \rhd ((\lambda x.\, e_1)\ e_2) \ :\ \rho$ and $\boldsymbol{ML} \vdash \Sigma \rhd e_1[e_2/x] \ :\ \rho$.

The theory $Th_{\mathrm{ML}}(\emptyset)$ corresponds to the equality on Core-ML terms. We write $\Sigma \rhd e_1 =_{\mathrm{ML}} e_2 \ :\ \rho$ for $\emptyset \vdash_{\mathrm{ML}} \Sigma \rhd e_1 = e_2 \ :\ \rho$.

If we exclude the rule of symmetry from the set of rules, then we have the notion of *reductions*. We write $E_{\mathrm{ML}} \vdash_{\mathrm{ML}} \Sigma \rhd e_1 \twoheadrightarrow e_2 \ :\ \rho$ if $\Sigma \rhd e_1 \ :\ \rho$ is reducible to $\Sigma \rhd e_2 \ :\ \rho$ using E_{ML} and the set of rules. In particular, the empty set determines the $\beta\eta$-reducibility, for which we write $\mathcal{A} \rhd e_1 \twoheadrightarrow_{\mathrm{ML}} e_2 \ :\ \tau$.

3.3 Semantics of Core-ML

In this section, we first define the explicitly-typed language $T\Lambda$ that corresponds to derivations of Core-ML typings. We then define the semantics of Core-ML relative to a semantics of $T\Lambda$.

3.3.1 Explicitly-typed language $T\Lambda$ and its semantics

The set of types of $T\Lambda$ is exactly the set *Type* of types of Core-ML. The set of pre-terms is given by the following abstract syntax:

$$M ::= x \mid (M\ M) \mid \lambda x : \tau.\, M.$$

Definition 3.13 (Terms of $T\Lambda$) *The set of terms of $T\Lambda$ is the set of formulae of the form $\mathcal{A} \rhd M \ :\ \tau$ that are derivable in the following proof system:*

(VAR) $\qquad \mathcal{A} \rhd x \ :\ \tau \qquad$ *if* $\Gamma(x) = \tau$

(ABS) $$\frac{\mathcal{A}\{x := \tau_1\} \rhd M \ :\ \tau_2}{\mathcal{A} \rhd (\lambda x : \tau_1.\, M) \ :\ \tau_1 \rightarrow \tau_2}$$

(APP) $$\frac{\mathcal{A} \rhd M_1 \ :\ \tau_1 \rightarrow \tau_2 \qquad \mathcal{A} \rhd M_2 \ :\ \tau_1}{\mathcal{A} \rhd (M_1\ M_2) \ :\ \tau_2}$$

Note that in this system, a formula has at most one derivation. We write $\boldsymbol{T\Lambda} \vdash \mathcal{A} \rhd M \ :\ \tau$ if $\mathcal{A} \rhd M \ :\ \tau$ is derivable from the above typing rules. $T\Lambda$ is clearly a representation of the simply typed lambda calculus [49], whose equational theory and model theory are well understood.

An equation of an equational theory of $T\Lambda$ is a formula of the form $\mathcal{A} \rhd M_1 = M_2 : \tau$.

Definition 3.14 ($T\Lambda$-theory) *A $T\Lambda$-theory consists of a given set $E_{T\Lambda}$ of equations and the following set of rules: the axiom schemes $(\alpha), (\beta), (\eta)$ and the inference rule scheme (ξ) of the simply typed lambda calculus [49], the set of rule schemes for usual equational reasoning and the following inference rule scheme:*

$$(thinning) \quad \frac{\mathcal{A} \rhd M_1 = M_2 : \tau}{\mathcal{A}' \rhd M_1 = M_2 : \tau} \; \text{if } \mathcal{A} \subseteq \mathcal{A}' \text{ (as graphs)}$$

We write $E_{T\Lambda} \vdash_{T\Lambda} \mathcal{A} \rhd M_1 = M_2 : \tau$ if $\mathcal{A} \rhd M_1 = M_2 : \tau$ is derivable from the axioms and $E_{T\Lambda}$ using the inference rules. A set of $T\Lambda$ equations $E_{T\Lambda}$ determines the $T\Lambda$-theory $Th_{T\Lambda}(E_{T\Lambda})$. We sometimes regard $Th_{T\Lambda}(E_{T\Lambda})$ as the set of all equations that are provable by the theory.

Parallel to ML-theory, we deifine well typed $T\Lambda$ equations.

Definition 3.15 (Well typed $T\Lambda$ equation) *An equation $\Sigma \rhd M_1 = M_2 : \rho$ is well typed if $\boldsymbol{T\Lambda} \vdash \Sigma \rhd M_1 : \rho$ and $\boldsymbol{T\Lambda} \vdash \Sigma \rhd M_2 : \rho$.*

We also say that a set of $T\Lambda$ equations is well typed if all its elements are well typed. It is easily checked that if $E_{T\Lambda}$ is well typed and axioms are restricted to well type equations then $Th_{T\Lambda}(E_{T\Lambda})$ is also well typed. In what follows, we restrict a set of $T\Lambda$ equations $E_{T\Lambda}$ and axioms to be well typed ones.

The following notations and notions are defined parallel to those in Core-ML: $\mathcal{A} \rhd M_1 =_{T\Lambda} M_2 : \tau$, the notion of reductions, $E_{T\Lambda} \vdash_{T\Lambda} \mathcal{A} \rhd M_1 \twoheadrightarrow M_2 : \tau$, and $\mathcal{A} \rhd M_1 \longrightarrow_{T\Lambda} M_2 : \tau$.

Following Friedman, [37] we define a *model* of $T\Lambda$ as follows:

Definition 3.16 (Frames and Extensional Frames) *A frame is a pair $(\mathcal{F}, \bullet)$ where $\mathcal{F}$ is a set $\{F_\tau | \tau \in Type\}$ such that each F_τ is non-empty and $\bullet$ is a family of binary operations $\bullet_{\tau_1,\tau_2} : F_{\tau_1 \to \tau_2} \times F_{\tau_1} \to F_{\tau_2}$. A frame is extensional if*

$$\forall \tau_1, \tau_2 \in Type, \forall f, g \in F_{\tau_1 \to \tau_2}.(\forall d \in F_{\tau_1}.f \bullet d = g \bullet d) \Longrightarrow f = g$$

We usually write $\mathcal{F}$ for $(\mathcal{F}, \bullet)$. Given a frame $\mathcal{F}$, a map $\phi : F_{\tau_1} \to F_{\tau_2}$ is *representable* if there is some $f \in F_{\tau_1 \to \tau_2}$ such that $\forall d \in F_{\tau_1}.\phi(d) = f \bullet d$ (f is a *representative* of ϕ). In an extensional frame, representatives are unique. For a frame $\mathcal{F}$ and a type assignment $\mathcal{A}$, an $\mathcal{F}\mathcal{A}$-environment ε is a mapping from $dom(\mathcal{A})$ to $\bigcup \mathcal{F}$ such that $\varepsilon(x) \in F_{\mathcal{A}(x)}$. We write $Env^{\mathcal{F}}(\mathcal{A})$ for the set of all $\mathcal{F}\mathcal{A}$-environments.

Definition 3.17 (Models of $T\Lambda$) *An extensional frame $\mathcal{F}$ is a model of $T\Lambda$ if there is a semantic mapping $[\![]\!]$ on terms of $T\Lambda$ satisfying the following equations: for any $\varepsilon \in Env^{\mathcal{M}}(\mathcal{A})$,*

$$\begin{aligned}
[\![\mathcal{A} \rhd x : \tau]\!]\varepsilon &= \varepsilon(x) \\
[\![\mathcal{A} \rhd \lambda x : \tau_1 .\, M : \tau_1 \rightarrow \tau_2]\!]\varepsilon &= \textit{the representative of } \phi \textit{ such that} \\
&\quad (\forall d \in F_{\tau_1})(\phi(d) = [\![\mathcal{A}\{x := \tau_1\} \rhd M : \tau_2]\!]\varepsilon\{x := d\}) \\
[\![\mathcal{A} \rhd (M\ N) : \tau]\!]\varepsilon &= [\![\mathcal{A} \rhd M : \tau_1 \rightarrow \tau]\!]\varepsilon \bullet [\![\mathcal{A} \rhd N : \tau_1]\!]\varepsilon
\end{aligned}$$

Note that for a given extensional frame, such a semantic mapping does not necessarily exist, but if one exists then it is unique. If $\mathcal{M}$ is a model, then we write $\mathcal{M}[\![]\!]$ for the unique semantic mapping.

An equation $\mathcal{A} \rhd M = N : \tau$ is *valid* in a model $\mathcal{M}$, write $\mathcal{M} \models_{T\Lambda} \mathcal{A} \rhd M = N : \tau$, if $\mathcal{M}[\![\mathcal{A} \rhd M : \tau]\!] = \mathcal{M}[\![\mathcal{A} \rhd M : \tau]\!]$. Let $\boldsymbol{Valid}^{T\Lambda}(\mathcal{M})$ be the set of all $T\Lambda$ equations that are valid in $\mathcal{M}$. Write $\mathcal{M} \models_{T\Lambda} F$ for $F \subseteq \boldsymbol{Valid}^{T\Lambda}(\mathcal{M})$. For $T\Lambda$ we have the following soundness and completeness of equational theories [37]:

Theorem 3.5 (Friedman) *For any model $\mathcal{M}$ and any $T\Lambda$-theory $Th_{T\Lambda}(E_{T\Lambda})$, if $\mathcal{M} \models_{T\Lambda} E_{T\Lambda}$ then $Th_{T\Lambda}(E_{T\Lambda}) \subseteq \boldsymbol{Valid}^{T\Lambda}(\mathcal{M})$. For any $T\Lambda$-theory T, there exists a model $\mathcal{T}$ such that $\boldsymbol{Valid}^{T\Lambda}(\mathcal{T}) = T$.* ▮

3.3.2 Relationship between $T\Lambda$ and Core-ML

Parallel to the relationship between Damas-Milner system and Core-XML, derivations of Core-ML typings correspond to terms of $T\Lambda$. Define a mapping *typedterm* on derivations of Core-ML typings as follows:

(1) If Δ is the one node derivation tree

$$\frac{}{\mathcal{A} \rhd x : \tau} \quad \text{(VAR)}$$

then $typedterm(\Delta) = x$.

(2) If Δ is a tree of the form

$$\frac{\Delta_1}{\mathcal{A} \rhd \lambda x .\, e_1 : \tau_1 \rightarrow \tau_2} \quad \text{(ABS)}$$

then $typedterm(\Delta) = \lambda x : \tau_1 .\, typedterm(\Delta_1)$.

(3) If Δ is a tree of the form

$$\frac{\Delta_1 \qquad \Delta_2}{\mathcal{A} \rhd (e_1\ e_2) : \tau} \quad (\text{APP})$$

then $typedterm(\Delta) = (typedterm(\Delta_1)\ typedterm(\Delta_2))$.

(5) If Δ is a tree of the form

$$\frac{\Delta_1 \qquad \Delta_2}{\mathcal{A} \rhd \textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end} : \tau} \quad (\text{LET})$$

then $typedterm(\Delta) = typedterm(\Delta_1)$.

The *type erasure* of a pre-term M, denoted by $erase(M)$, is the raw term defined as follows:

$$\begin{aligned} erase(x) &= x \\ erase((M_1\ M_2)) &= (erase(M_1)\ erase(M_2)) \\ erase((\lambda x : \rho.\, M)) &= (\lambda x.\, erase(M)) \end{aligned}$$

The following theorem corresponds to theorem 3.1:

Theorem 3.6 *If* $\boldsymbol{TA} \vdash \mathcal{A} \rhd M : \tau$ *then there is a derivation* Δ *of* $\mathcal{A} \rhd erase(M) : \tau$ *in Core-ML such that* $typedterm(\Delta) \equiv M$. *If* Δ *is a typing derivation of* $\mathcal{A} \rhd e : \tau$ *then* $letexpd(e) \equiv erase(typedterm(\Delta))$ *and* $\boldsymbol{TA} \vdash \mathcal{A} \rhd typedterm(\Delta) : \tau$.

Proof The first statement is easily proved by induction on the structure M.

The second statement is shown by induction on the hight of Δ. We only show the case for *let* expression. Suppose Δ is a typing derivation of $\mathcal{A} \rhd \textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end} : \tau$ then Δ must be of the form:

$$\frac{\Delta_1 \qquad \Delta_2}{\mathcal{A} \rhd \textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end} : \tau} \quad (\text{LET})$$

By the definition of *typedterm*, $typedterm(\Delta) = typedterm(\Delta_1)$. By the typing rules, Δ_1 is a derivation of the typing $\mathcal{A} \rhd e_2[e_1/x] : \tau$. By the induction hypothesis, $letexpd(e_2[e_1/x]) \equiv erase(typedterm(\Delta))$ and $\boldsymbol{TA} \vdash \mathcal{A} \rhd typedterm(\Delta) : \tau$. But by the definition of *letexpd*, $letexpd(e_2[e_1/x]) = letexpd(\textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end})$. ∎

Unlike the relationship between Core-XML and Damas-Milner system, we also have the following desired property:

Theorem 3.7 *If Δ_1, Δ_2 are typing derivations of a same typing $\mathcal{A} \rhd e : \tau$ then the following equation holds:*

$$\mathcal{A} \rhd \mathit{typedterm}(\Delta_1) =_{T\Lambda} \mathit{typedterm}(\Delta_2) : \tau.$$

Proof The proof uses the following lemmas:

Lemma 3.9 *Let $\mathcal{A} \rhd M : \tau$ and $\mathcal{A} \rhd e : \tau$ be respectively $T\Lambda$ term and Core-ML term such that $erase(M) \equiv e$. If $\mathcal{A} \rhd M \twoheadrightarrow_{T\Lambda} M' : \tau$ then there is e' such that $erase(M') \equiv e'$ and $\mathcal{A} \rhd e \twoheadrightarrow_{\mathrm{ML}} e' : \tau$. Conversely, if $\mathcal{A} \rhd e \twoheadrightarrow_{\mathrm{ML}} e' : \tau$ then there is M' such that $erase(M') \equiv e'$ and $\mathcal{A} \rhd M \twoheadrightarrow_{T\Lambda} M' : \tau$.*

Proof This is proved by observing the following facts: (1) there is a one-one correspondence between the set of $\beta\eta$-redexes in M and the set of $\beta\eta$-redexes in e, (2) if $erase((\lambda x : \tau. M_1)\ M_2) \equiv ((\lambda x.e_1)\ e_2)$ then $erase(M_1[x := M_2]) \equiv e_1[x := e_2]$, and (3) if $erase(\lambda x : \tau. M x) \equiv (\lambda x.ex)$ then $erase(M) \equiv e$. ∎

Note that this result, combined with the property of the reduction rule (*let*) and the connection between $T\Lambda$ terms and typing derivations of ML implies that if $T\Lambda$ has the strong normalization property then so does Core-ML, which was suggested in [49, remark 15.32]. Technical difficulty of treating bound variables mentioned in [49, remark 15.32] was overcome by our presentation of $T\Lambda$.

Lemma 3.10 *If two terms $\mathcal{A} \rhd M_1 : \tau$ and $\mathcal{A} \rhd M_2 : \tau$ are in β-normal form and $erase(M_1) \equiv erase(M_2)$ then $M_1 \equiv M_2$.*

Proof The proof is by induction on the structure of M_1. The basis is trivial. The induction step is by cases.

1. Case of $M_1 \equiv \lambda x : \tau_1. M_1'$: By the typing rules, $\boldsymbol{T\Lambda} \vdash \mathcal{A}\{x := \tau_1\} \rhd M_1' : \tau_2$ for some τ_2 and $\tau = \tau_1 \to \tau_2$. Since $erase(M_1) \equiv erase(M_2)$, M_2 must be of the form $\lambda x : \tau_1'. M_2'$ such that $erase(M_1') \equiv erase(M_2')$. By the typing rules, $\boldsymbol{T\Lambda} \vdash \mathcal{A}\{x := \tau_1'\} \rhd M_2 : \tau_2'$ for some τ_2' and $\tau = \tau_1' \to \tau_2'$. Therefore $\tau_1 = \tau_1', \tau_2 = \tau_2'$. By definition, $\mathcal{A}\{x := \tau_1\} \rhd M_1' : \tau_2$ and $\mathcal{A}\{x := \tau_1\} \rhd M_2' : \tau_2$ must be also in β-normal form. Then by the induction hypothesis, $M_1' \equiv M_2'$. This implies $M_1 \equiv M_2$.

2. Case of $M_1 \equiv (\cdots(x\ M_1^1)\cdots M_1^n)$: By the typing rules, $\boldsymbol{T\Lambda} \vdash \mathcal{A} \rhd M_1^i : \tau_1^i$ for some τ_1^i, $1 \le i \le n$. It is shown by simple induction that $\mathcal{A}(x) = \tau_1^1 \to \tau_1^2 \to \cdots \to \tau_1^n \to \tau$. Since $erase(M_1) \equiv erase(M_2)$, M_2 must be of the form $(\cdots(x\ M_2^1)\cdots M_2^n)$ and $erase(M_1^i) \equiv erase(M_2^i), 1 \le i \le n$. Then similarly we have $\boldsymbol{T\Lambda} \vdash \mathcal{A} \rhd M_2^i : \tau_2^i$ for some τ_2^i, $1 \le i \le n$ and $\mathcal{A}(x) = \tau_2^1 \to \tau_2^2 \to \cdots \to \tau_2^n \to \tau$. This implies $\tau_1^i = \tau_2^i, 1 \le i \le n$. Then by the induction hypothesis, $M_1^i \equiv M_2^i, 1 \le i \le n$. Hence we have $M_1 \equiv M_2$.

Since M_1 is in β-normal form, we have exhausted all cases. ▮

We now prove the theorem. Let $M_1 \equiv typedterm(\Delta_1), M_2 \equiv typedterm(\Delta_2)$. Also let $\mathcal{A} \rhd M_1' : \tau$, $\mathcal{A} \rhd M_2' : \tau$ be normal form terms such that $\mathcal{A} \rhd M_1 \twoheadrightarrow_{T\Lambda} M_1' : \tau$ and $\mathcal{A} \rhd M_2 \twoheadrightarrow_{T\Lambda} M_2' : \tau$ (such M_1', M_2' always exist). By lemma 3.9, there are normal form terms $\mathcal{A} \rhd e_1 : \tau$ and $\mathcal{A} \rhd e_2 : \tau$ such that $erase(M_1') \equiv e_1, erase(M_2') \equiv e_2$ and $\mathcal{A} \rhd e \twoheadrightarrow_{\text{ML}} e_1 : \tau$ and $\mathcal{A} \rhd e \twoheadrightarrow_{\text{ML}} e_2 : \tau$. By the uniqueness of normal form, $e_1 \equiv e_2$. Thus $erase(M_1') \equiv erase(M_2')$. Then by lemma 3.10, $M_1' \equiv M_2'$. (End of the proof of theorem 3.7) ▮

3.3.3 Semantics of Core-ML

We define the semantics of Core-ML relative to a model of $T\Lambda$. We first define the semantics of Core-ML typings and then "lift" them to general Core-ML terms.

Definition 3.18 (Semantics of Core-ML Typings) *The semantics of Core-ML typings relative to a model $\mathcal{M}$ of $T\Lambda$ is defined as*

$$\mathcal{M}[\![\mathcal{A} \rhd e : \tau]\!]^{\text{ML}}\varepsilon = \mathcal{M}[\![\mathcal{A} \rhd typedterm(\Delta) : \tau]\!]\varepsilon$$

for some derivation Δ for $\mathcal{A} \rhd e : \tau$.

By theorem 3.7 and the soundness of $T\Lambda$ theories (theorem 3.5), this definition does not depend on the choice of Δ.

For a given type assignment scheme Σ, the set of *admissible type assignments under* Σ denoted by $TA(\Sigma)$ is the set $\{\mathcal{A} | dom(\Sigma) \subseteq dom(\mathcal{A}), \exists\theta. \mathcal{A}\restriction^{dom(\Sigma)} = \theta(\Sigma)\}$. Under a given type assignment $\mathcal{A}$, the set $TP(\mathcal{A}, \Sigma \rhd e : \rho)$ of the types associated with a term $\Sigma \rhd e : \rho$ is the set $\{\tau | \exists\theta. (\mathcal{A}\restriction^{dom(\Sigma)}, \tau) = \theta(\Sigma, \rho)\}$. For a model $\mathcal{M} = (\{F_\tau | \tau \in Type\}, \bullet)$ and a set of types S, we write $\Pi\tau \in S. F_\tau$ for the direct product (i.e. the space of functions f such that $dom(f) = S, f(\tau) \in F_\tau$).

Definition 3.19 (Semantics of Core-ML Terms) *The semantics $\mathcal{M}[\![\Sigma \rhd e : \rho]\!]^{\text{ML}}$ of a Core-ML term $\Sigma \rhd e : \rho$ relative to a model $\mathcal{M}$ is the function which takes a type assignment $\mathcal{A} \in TA(\Sigma)$*

and an environment $\varepsilon \in Env^{\mathcal{M}}(\mathcal{A})$ and returns an element in $\Pi\tau \in TP(\mathcal{A}, \Sigma \rhd e : \rho)$. F_τ, defined as follows:

$$\mathcal{M}[\![\Sigma \rhd e : \rho]\!]^{\text{ML}}\mathcal{A}\varepsilon = \{(\tau, \mathcal{M}[\![\mathcal{A} \rhd e : \tau]\!]^{\text{ML}}\varepsilon) | \tau \in TP(\mathcal{A}, \Sigma \rhd e : \rho)\}$$

For example,

$$\mathcal{M}[\![\emptyset \rhd \lambda x.\, x : t \to t]\!]^{\text{ML}}\mathcal{A}\varepsilon = \{(\tau \to \tau, \mathcal{M}[\![\mathcal{A} \rhd \lambda x : \tau.\, x : \tau \to \tau]\!]\varepsilon) | \tau \in Type\}$$

Now if each element of $F_{\tau_1 \to \tau_2}$ is a function from F_{τ_1} to F_{τ_2} then by the extensionality property of $\mathcal{M}$, we have

$$\mathcal{M}[\![\emptyset \rhd \lambda x.\, x : t \to t]\!]^{\text{ML}}\mathcal{A}\varepsilon = \{(\tau \to \tau, id_{F_\tau}) | \tau \in Type\}$$

where id_X is the identity function on X.

3.4 Soundness and Completeness of Core-ML Theories

Let $\mathcal{M}$ be a given model of $T\Lambda$. $\mathcal{M}$ also determines the semantics of ML. We say that an equation $\mathcal{A} \rhd e_1 = e_2 : \rho$ is *valid* in $\mathcal{M}$, write $\mathcal{M} \models_{\text{ML}} \Sigma \rhd e_1 = e_2 : \rho$, iff $\mathcal{M}[\![\Sigma \rhd e_1 : \rho]\!]^{\text{ML}} = \mathcal{M}[\![\Sigma \rhd e_2 : \rho]\!]^{\text{ML}}$ (as mappings). Let $\boldsymbol{Valid}^{\text{ML}}(\mathcal{M})$ be the set of all equations in Core-ML that are valid in $\mathcal{M}$. Write $\mathcal{M} \models_{\text{ML}} F$ for $F \subseteq \boldsymbol{Valid}^{\text{ML}}(\mathcal{M})$.

Theorem 3.8 (Soundness of Core-ML Theories) *Let E_{ML} be any set of ML-equations and $\mathcal{M}$ be any model. If $\mathcal{M} \models_{\text{ML}} E_{\text{ML}}$, then $Th_{\text{ML}}(E_{\text{ML}}) \subseteq \boldsymbol{Valid}^{\text{ML}}(\mathcal{M})$.*

Proof Define mappings Φ, Ψ between sets of ML-equations and sets of $T\Lambda$-equations as:

$$\begin{aligned}
\Phi(E_{\text{ML}}) &= \{\mathcal{A} \rhd M = N : \tau | \exists(\mathcal{A} \rhd e_1 = e_2 : \tau) \in E_{\text{ML}} \text{ such that} \\
&\qquad erase(M) \equiv letexpd(e_1), erase(N) \equiv letexpd(e_2)\} \\
\Psi(E_{T\Lambda}) &= \{\Sigma \rhd e_1 = e_2 : \rho | \forall(\mathcal{A}, \tau) \text{ if } (\mathcal{A}, \tau) \precsim (\Sigma, \rho) \text{ then } \exists(\mathcal{A} \rhd M_1 = M_2 : \tau) \in E_{T\Lambda} \\
&\qquad \text{such that } erase(M_1) \equiv letexpd(e_1), erase(M_2) \equiv letexpd(e_2)\}
\end{aligned}$$

The proof uses the following lemmas.

Lemma 3.11 *For any set of ML-equations E_{ML}, $\Psi(Th_{T\Lambda}(\Phi(E_{\text{ML}}))) = Th_{\text{ML}}(E_{\text{ML}})$.*

Proof By our assumptions on E_{ML} and the properties of the rules of ML-theories, $\Sigma \rhd e_1 = e_2 : \rho \in Th_{\text{ML}}(E_{\text{ML}})$ iff for all ground instance $(\mathcal{A}, \tau)$ of (Σ, ρ), $\mathcal{A} \rhd e_1 = e_2 : \tau \in Th_{\text{ML}}(E_{\text{ML}})$.

By definition of Ψ, $\Psi(Th_{T\Lambda}(\Phi(E_{\mathrm{ML}})))$ also has this property. It is therefore enough to show that $\mathcal{A} \rhd e_1 = e_2 : \tau \in Th_{\mathrm{ML}}(E_{\mathrm{ML}})$ iff $\mathcal{A} \rhd e_1 = e_2 : \tau \in \Psi(Th_{T\Lambda}(\Phi(E_{\mathrm{ML}})))$, which is proved by the relationship between sets of rules of $T\Lambda$ and those of Core-ML and the definition of Ψ. ▮

Lemma 3.12 *For any model $\mathcal{M}$, $\boldsymbol{Valid}^{\mathrm{ML}}(\mathcal{M}) = \Psi(\boldsymbol{Valid}^{T\Lambda}(\mathcal{M}))$.*

Proof Suppose $\Sigma \rhd e_1 = e_2 : \rho \in \boldsymbol{Valid}^{\mathrm{ML}}(\mathcal{M})$. For any ground instance $(\mathcal{A}, \tau)$ of (Σ, ρ), $\mathcal{M}[\![\mathcal{A} \rhd e_1 : \tau]\!]^{\mathrm{ML}} = \mathcal{M}[\![\mathcal{A} \rhd e_2 : \tau]\!]^{\mathrm{ML}}$. Let Δ_1, Δ_2 be derivations of $\mathcal{A} \rhd e_1 : \tau$ and $\mathcal{A} \rhd e_2 : \tau$ respectively. Then $erase(typedterm(\Delta_1)) \equiv letexpd(e_1)$, $erase(typedterm(\Delta_2)) \equiv letexpd(e_2)$, and $\mathcal{M}[\![\mathcal{A} \rhd typedterm(\Delta_1) : \tau]\!] = \mathcal{M}[\![\mathcal{A} \rhd typedterm(\Delta_2) : \tau]\!]$. Therefore by definition $\Sigma \rhd e_1 = e_2 : \rho \in \Psi(\boldsymbol{Valid}^{T\Lambda}(\mathcal{M}))$. Conversely, suppose $\Sigma \rhd e_1 = e_2 : \rho \in \Psi(\boldsymbol{Valid}^{T\Lambda}(\mathcal{M}))$. Let $(\mathcal{A}, \tau)$ be any instance of (Σ, ρ). By the definition of Ψ, there are M_1, M_2 such that, $\mathcal{A} \rhd M_1 = M_2 : \tau \in \boldsymbol{Valid}^{T\Lambda}(\mathcal{M})$, $erase(M_1) \equiv letexpd(e_1), erase(M_2) \equiv letexpd(e_2)$. Let Δ_1, Δ_2 be derivations of $\mathcal{A} \rhd e_1 : \tau$ and $\mathcal{A} \rhd e_2 : \tau$ respectively. Then it is shown by using lemmas 3.9 and 3.10 that $\mathcal{A} \rhd typedterm(\Delta_1) =_{T\Lambda} M_1 : \tau$ and $\mathcal{A} \rhd typedterm(\Delta_2) =_{T\Lambda} M_2 : \tau$. Then by theorem 3.5, $\mathcal{M}[\![\mathcal{A} \rhd e_1 : \tau]\!]^{\mathrm{ML}} = \mathcal{M}[\![\mathcal{A} \rhd e_2 : \tau]\!]^{\mathrm{ML}}$. Since $(\mathcal{A}, \tau)$ is arbitrary instance of (Σ, ρ), we have $\Sigma \rhd e_1 = e_2 : \rho \in \boldsymbol{Valid}^{\mathrm{ML}}(\mathcal{M})$. ▮

We now conclude the proof of the theorem. Suppose $\mathcal{M} \models_{\mathrm{ML}} E_{\mathrm{ML}}$. By the definitions of Ψ and $\mathcal{M}[\![]\!]$, $\mathcal{M} \models_{T\Lambda} \Phi(E_{\mathrm{ML}})$. By theorem 3.5, $Th_{T\Lambda}(\Phi(E_{\mathrm{ML}})) \subseteq \boldsymbol{Valid}^{T\Lambda}(\mathcal{M})$. Since Ψ is monotone with respect to $\subseteq$, by lemma 3.11 and 3.12, $Th_{\mathrm{ML}}(E_{\mathrm{ML}}) \subseteq \boldsymbol{Valid}^{\mathrm{ML}}(\mathcal{M})$. (End of the proof of theorem 3.8) ▮

Theorem 3.9 (Relative Completeness of Core-ML Theories) *For any set of ML-equations E_{ML} and any model $\mathcal{M}$, if $\boldsymbol{Valid}^{T\Lambda}(\mathcal{M}) = Th_{T\Lambda}(\Phi(E_{\mathrm{ML}}))$ then $\boldsymbol{Valid}^{\mathrm{ML}}(\mathcal{M}) = Th_{\mathrm{ML}}(E_{\mathrm{ML}})$*

Proof By lemma 3.11 and 3.12. ▮

Then by theorem 3.5, we have:

Corollary 3.1 (Completeness of Core-ML Theories) *For any ML-theory G, there exists a model $\mathcal{G}$ such that $\boldsymbol{Valid}^{\mathrm{ML}}(\mathcal{G}) = G$.* ▮

As a special case of theorem 3.9, for any model $\mathcal{M}$, we have $\boldsymbol{Valid}^{\mathrm{ML}}(\mathcal{M}) = Th_{\mathrm{ML}}(\emptyset)$ if $\boldsymbol{Valid}^{T\Lambda}(\mathcal{M}) = Th_{T\Lambda}(\emptyset)$. Now let $\mathcal{S}$ be a *full type structure*, that is, let F_b be a countably infinite set, $F_{\tau_1 \to \tau_2}$ be the set of all functions from F_{τ_1} to F_{τ_2} and $\bullet$ is the function application. Friedman showed that [37] $\boldsymbol{Valid}^{T\Lambda}(\mathcal{S}) = Th_{T\Lambda}(\emptyset)$. Then we have:

Corollary 3.2 $Valid^{\text{ML}}(\mathcal{S}) = Th_{\text{ML}}(\emptyset)$. ∎

This means that $=_{\text{ML}}$ is sound and complete for the full type structure generated by countably infinite base sets. Since $=_{\text{ML}}$ is decidable (see remark on lemma 3.9), this implies that the set of all true ML equations in the full type structure is decidable.

3.5 Extensions of Core-ML

As a programming language, Core-ML should be extended to support recursion and various data types including recursive types. This is done by adding constants and extending the set of types and type-schemes as (possibly infinite) trees generated by various type constructor symbols. We call the extended language ML.

We assume that we are given a set of type constructor symbols *Tycon* (always containing the function type constructor : $\rightarrow$). As observed in [31, 110], an appropriate class of infinite trees to support recursive types is the set of regular trees.

Definition 3.20 (Types and Type-schemes of ML) *The set Type of types of ML is the set $\boldsymbol{R}(Tycon)$ of regular trees. The set Tscheme of type-schemes is the set $\boldsymbol{R}(Tycon, V)$ of regular trees.*

As an example of a recursive types, the following infinite type-scheme in a term representation we have defined in section 2.2 represents a polymorphic list type:

$$(rec\ v.\ nil + (t \times v))$$

where $+$ and $\times$ are binary type constructors representing sum and product and *nil* is a trivial type which has only one element representing the empty list. The following recursive type corresponds to the type of the set of all pure lambda terms:

$$(rec\ v.\ v \rightarrow v).$$

We also extend the language with a set of constant symbols. In order to preserve ML's implicit type system, we assume that we are given a set *Const* of pairs of a constant symbol and a type. We write $c : \tau$ for an element of *Const*.

Definition 3.21 (Raw Terms of ML) *The set of raw terms of ML is given by the following syntax:*

$$e ::= x \mid c \mid \lambda x.\, e \mid (e\ e) \mid \textbf{let } x = e \textbf{ in } e \textbf{ end}$$

where c stands for the set of constant symbols that appear in Const. —

For example, products can be introduced by assuming the following set of constants

$$\begin{aligned} pair &: \tau_1 \to \tau_2 \to (\tau_1 \times \tau_2) \text{ for each } \tau_1, \tau_2, \\ first &: (\tau_1 \times \tau_2) \to \tau_1 \text{ for each } \tau_1, \tau_2, \\ second &: (\tau_1 \times \tau_2) \to \tau_2 \text{ for each } \tau_1, \tau_2. \end{aligned}$$

Definition 3.22 (Typings of ML) *The typing derivation system for ML is the one obtained from that of Core-ML by adding the following axiom:*

(CONST) $\mathcal{A} \rhd c : \tau$ *if* $c : \tau \in Const$

On the types of constants, we need the following assumption to preserve the existence of principal typing scheme and decidability of type inference problem:

Assumption 3.1 *For each constant symbol c appears in Const, there is a type-scheme ρ such that the set of all ground instances of ρ coincides with the set $\{\tau | c : \tau \in Const\}$.*

We write $c : \rho$ for such a type-scheme ρ. This condition is satisfied by many standard data structures. For example, the sets of types of *pair*, *first*, *second* are represented by the following type-schemes:

$$\begin{aligned} pair &: t_1 \to t_2 \to (t_1 \times t_2), \\ first &: (t_1 \times t_2) \to t_1, \\ second &: (t_1 \times t_2) \to t_2. \end{aligned}$$

As we will see in chapter 5, however, there are data structures and operations essential to databases and object-oriented programming that do not satisfy the assumption.

Under this assumption, the type inference problem of ML is still decidable.

Theorem 3.10 *There is an algorithm $\mathcal{PTS}^+$ which, given any raw term e satisfying assumption 3.1, yields either failure or (Σ, ρ) such that if $\mathcal{PTS}^+(e) = (\Sigma, \rho)$ then $\Sigma \rhd e : \rho$ is a principal typing scheme otherwise e has no typing.*

Proof Algorithm $\mathcal{PTS}^+$ is obtained from algorithm $\mathcal{PTS}$ defined in the proof of theorem 3.2 by adding the case:

(5) Case $e \equiv c$:

$\Sigma = \emptyset$,

ρ is the type-scheme such that $c : \rho$

Clearly this addition does not change the termination property of the algorithm.

The proof that $\mathcal{PTS}^+$ has the desired property is obtained by adding the case for $e \equiv c$ to the inductive proof of lemma 3.4 and the inductive proof of the property that $\boldsymbol{ML} \vdash \mathcal{A} \rhd e : \tau$ iff $\mathcal{A} \rhd e : \tau \precsim_{\mathrm{ML}} \Sigma \rhd e : \rho$ in the proof of theoremg 3.2. Both of them are immediate consequences of the definitions.

All other parts of the proof remain valid without any change. In particular, the extension of the set of types to regular trees does not change the proof since the proof of theorem 3.2 uses an unification on regular trees. ∎

With those extensions, ML uniformly support both recursive types and recursion. For recursion, the special term constructor **fix** $x.e$ (which is built in **fun** declaration in Standard ML) is no longer necessary. As observed in [72], if recursive types are allowed then fixed point combinators are typable. For example, the following well known fixed point combinators:

$$Y_{curry} = \lambda f.\, (\lambda x.\, f(xx))(\lambda x.\, f(xx)),$$

and

$$Y_{turing} = (\lambda z \lambda x.\, x(zzx))(\lambda z \lambda x.\, x(zzx))$$

have the following principal typing schemes:

$$\boldsymbol{ML} \vdash \emptyset \rhd Y_{curry} : (t \to t) \to t,$$

and

$$\boldsymbol{ML} \vdash \emptyset \rhd Y_{turing} : (t \to t) \to t.$$

Moreover, the algorithm $\mathcal{PTS}$ can infer these principal typing schemes. Later in section 5.6 we will consider another fixed point combinator given by Plotkin [90] that represents recursive function definition under the call-by-value evaluation strategy.

The extended language also infers recursive types for recursively defined functions. This eliminates the mandatory requirement of recursive type declarations in Standard ML. As an example suppose we have the following primitives for sum types:

$$isr \;:\; (t_1 + t_2) \to bool,$$

```
- fun self x = x x;
> val self :  (rec v. v → t)

- fun loop x = self self x;
> val loop :  t1 → t2

- fun isnil l = if isl l then true else false;
> val isnil :  (t1 + t2) → bool

- fun car l = if isnil l then loop Null else first (outr l);
> val car :  (t1 + (t2 × t3)) → t2

- fun cdr l = if isnil l then loop Null else second (outr l);
> val cdr :  (t1 + (t2 × t3)) → t3

- fun length l = if isnil l then 0 else 1 + (length (cdr l));
> val length :  (rec v. (t1 + (t2 × v)) → int
```

Figure 3.1: Examples of Type Inference with Recursive Types

$$\begin{aligned}
isl &: (t_1 + t_2) \rightarrow bool,\\
inl &: t_1 \rightarrow (t_1 + t_2),\\
inr &: t_1 \rightarrow (t_2 + t_1),\\
outl &: (t_1 + t_2) \rightarrow t_1,\\
outr &: (t_1 + t_2) \rightarrow t_2
\end{aligned}$$

together with the primitive constants *pair,first* and *second* for products we have defined earlier. Figure 3.1 shows type inference for recursive types by simulating an interactive session using Standard ML conventions.

In order to define a semantics of the extended language, we need to extend the simply typed lambda calculus $T\Lambda$ and its semantics. We call the extended language $T\Lambda^+$.

Definition 3.23 (Syntax of $T\Lambda^+$) *The set of pre-terms of $T\Lambda^+$ is the set of pre-terms of $T\Lambda$ extended with the set of typed constants $\{c^\tau | c : \tau \in Const\}$. The proof system for typings of $T\Lambda^+$ is the one obtained from that of $T\Lambda$ by adding the following rule:*

(CONST) $\quad \mathcal{A} \rhd c^\tau : \tau.$

The notion of models is also extended with constants.

Definition 3.24 (Models of $T\Lambda^+$) *An extended frame is a frame with a function C on constants* *such that $C(c^\tau) \in F_\tau$. An extensional extended frame is a model of $T\Lambda^+$ if there is a semantic mapping $[\![]\!]$ on terms of $T\Lambda^+$ satisfying the conditions of the model of $T\Lambda$ (definition 3.17) and the following equation:*

$$[\![\mathcal{A} \rhd c^\tau : \tau]\!]\varepsilon = C(c^\tau).$$

Breazu-Tannen and Meyer extended [19] Friedman's soundness and completeness of equational theories to languages with constants and a set of types satisfying arbitrary constraints. Since the set of types of $T\Lambda^+$ satisfies their definition of *type algebra*, the soundness and completeness of equational theories (theorem 3.5) still holds for $T\Lambda^+$.

The relationship between $T\Lambda^+$ terms and derivations of ML typings is essentially unchanged and theorem 3.6 still holds (by adding the case for constants). However, theorem 3.7 no longer holds for $T\Lambda^+$. There are non convertible $T\Lambda^+$ terms that correspond to a same ML typing. For example, consider the ML typing:

$$\emptyset \rhd (second((pair\ \lambda x.\,x)1)) : int.$$

The following two $T\Lambda^+$ terms both correspond to derivations of the above typing:

$$\emptyset \rhd (second^{((int\to int)\times int)\to int}\ ((pair^{(int\to int)\to int\to((int\to int)\times int)}\ \lambda x : int.\,x)1)) : int,$$

$$\emptyset \rhd (second^{((bool\to bool)\times int)\to int}\ ((pair^{(bool\to bool)\to int\to((bool\to bool)\times int)}\ \lambda x : bool.\,x)1)) : int.$$

But they are not convertible to each other (in $=_{T\Lambda}$). An obvious implication of this fact is that we cannot interpret constants arbitrarily. In order to define a semantics of ML terms via a semantics of $T\Lambda^+$, we need the following restriction:

Definition 3.25 *A model $\mathcal{M}$ of $T\Lambda^+$ is abstract if $erase(M) \equiv erase(N)$ implies $\mathcal{M}[\![\mathcal{A} \rhd M : \tau]\!] = \mathcal{M}[\![\mathcal{A} \rhd N : \tau]\!]$.*

Abstract models are models in which the following equations are valid:

$$(erasure) \qquad \mathcal{A} \rhd M \ = \ N : \tau \qquad \text{if } erase(M) = erase(N).$$

By the completeness theorem for equational theories, $T\Lambda^+$ always has an abstract model. We further think that the class of abstract models covers a wide range of standard models of languages with standard set of constants. For example, ordinary interpretation of *pair* and *second* certainly satisfy the above condition and suggests an abstract model. Any abstract model of $T\Lambda^+$ yields a semantics of ML. The definition of a semantics of ML relative to an abstract model of $T\Lambda^+$ is the same as before (definition 3.18). The well definedness of the definition follow directly from the property of abstract model instead of theorem 3.7. The soundness and completeness of equational theories of ML (theorem 3.8 and 3.9) hold with respect to the class of abstract models. Proofs are the same as before except that we use the condition of abstract models in place of theorem 3.7. The condition of abstract models can be regarded as a necessary condition for ***fully abstract models*** we will exploit in the next section.

3.6 Full Abstraction of ML

One desired property of a denotational semantics of a programming language is ***full abstraction*** [77, 92, 81, 76], which roughly says that the denotational semantics coincides with the operational semantics. In this section, we will show that if a model of $T\Lambda^+$ is fully abstract for an operational semantics of $T\Lambda^+$ then it is also fully abstract for the corresponding operational semantics of ML.

Following [92, 76], we define an operational semantics as a partial function on closed terms of base types. Let $\mathcal{E}^{T\Lambda}, \mathcal{E}^{\mathrm{ML}}$ be respectively the evaluation functions of $T\Lambda^+$ and ML determining their operational semantics. We write $\mathcal{E}(X) \Downarrow y$ to means that $\mathcal{E}(X)$ is defined and equal to y. On the operational semantics of $T\Lambda^+$ we assume that it depend only on structure of terms. Formally, we assume $\mathcal{E}^{T\Lambda}$ to satisfy the following property:

Assumption 3.2 *For two terms $\emptyset \rhd M : b$ and $\emptyset \rhd N : b$ if $erase(M) \equiv erase(N)$ then $\mathcal{E}^{T\Lambda}(\emptyset \rhd M : b) \Downarrow \emptyset \rhd c^b : b$ iff $\mathcal{E}^{T\Lambda}(\emptyset \rhd N : b) \Downarrow \emptyset \rhd c^b : b$.*

We believe that this condition is satisfied by most operational semantics of explicitly-typed programming languages. On the operational semantics of $\mathcal{E}^{\mathrm{ML}}$ we assume the following property on evaluation of **let**-expressions:

Assumption 3.3 *$\mathcal{E}^{\mathrm{ML}}(\emptyset \rhd e : b) \Downarrow \emptyset \rhd c : b$ iff $\mathcal{E}^{\mathrm{ML}}(\emptyset \rhd letexpd(e) : b) \Downarrow \emptyset \rhd c : b$.*

This condition correspond to the equality axiom (*let*). Note that the rule (*let*) corresponds to the rule (β) and does not agree with the call-by-value evaluation strategy. Finally we assume the following relationship between the operational semantics of $T\Lambda^+$ and that of ML:

Assumption 3.4 *For terms $\emptyset \rhd M : b$ of $T\Lambda^+$ and $\emptyset \rhd e : b$ of ML, if $erase(M) \equiv e$ then $\mathcal{E}^{T\Lambda}(\emptyset \rhd M : b) \Downarrow \emptyset \rhd c^b : b$ iff $\mathcal{E}^{\mathrm{ML}}(\emptyset \rhd e : b) \Downarrow \emptyset \rhd c : b$.*

We believe that in most cases it is routine to construct $\mathcal{E}^{\mathrm{ML}}$ from given $\mathcal{E}^{T\Lambda}$ that satisfies the condition and vice versa.

A context $C[\,]$ in $T\Lambda^+$ is a $T\Lambda^+$ pre-term with one "hole" in it. We omit a formal definition. A context $C[\,]$ is a *closing b-context for* $\mathcal{A} \rhd M : \tau$ if there is a derivation of $\boldsymbol{T\Lambda} \vdash \emptyset \rhd C[M] : b$ such that its sub-derivation of (the occurrence in $C[M]$ of) M is a derivation of $\mathcal{A} \rhd M : \tau$.

Definition 3.26 (Operational Equivalence in $T\Lambda$) *Two $T\Lambda^+$ terms $\mathcal{A} \rhd M : \tau$ and $\mathcal{A} \rhd N : \tau$ are operationally equivalent, denoted by $\mathcal{A} \rhd M \overset{T\Lambda}{\approx} N : \tau$, iff for any closing b-context $C[\,]$ for these two terms, $\mathcal{E}^{T\Lambda}(\emptyset \rhd C[M] : b) \Downarrow \emptyset \rhd c^b : b$ iff $\mathcal{E}^{T\Lambda}(\emptyset \rhd C[N] : b) \Downarrow \emptyset \rhd c^b : b$.*

In ML, under our assumption on **let**-expressions, it is enough to consider raw terms and contexts that do not contain **let**-expression. Therefore we define a context $c[\,]$ in ML as a context of the untyped lambda calculus. A context $c[\,]$ is a *closing b-context for* $\Sigma \rhd e : \rho$ if there is a derivation of $\emptyset \rhd c[e] : b$ such that its subderivation of e is a derivation of an instance of $\Sigma \rhd e : \rho$.

Definition 3.27 (Operational Equivalence in ML) *Two ML terms $\Sigma \rhd e_1 : \rho$ and $\Sigma \rhd e_2 : \rho$ are operationally equivalent, denoted by $\Sigma \rhd e_1 \overset{\mathrm{ML}}{\approx} e_2 : \rho$, iff for any closing b-context $c[\,]$ for these two terms, $\mathcal{E}^{\mathrm{ML}}(\emptyset \rhd c[e_1] : b) \Downarrow \emptyset \rhd c : b$ iff $\mathcal{E}^{\mathrm{ML}}(\emptyset \rhd c[e_2] : b) \Downarrow \emptyset \rhd c : b$.*

Definition 3.28 (Full Abstraction) *A model $\mathcal{M}$ is fully abstract for $\mathcal{E}^{T\Lambda}$ if $\mathcal{M} \models_{T\Lambda} \mathcal{A} \rhd M = N : \tau$ iff $\mathcal{A} \rhd M \overset{T\Lambda}{\approx} N : \tau$. A mode $\mathcal{M}$ is fully abstract for $\mathcal{E}^{\mathrm{ML}}$ if $\mathcal{M} \models_{\mathrm{ML}} \Sigma \rhd e_1 = e_2 : \rho$ iff $\Sigma \rhd e_1 \overset{\mathrm{ML}}{\approx} e_2 : \rho$.*

Note that a model $\mathcal{M}$ is fully abstract for $\mathcal{E}^{T\Lambda}$ then it is an abstract model (definition 3.25). This means that any fully abstract model for $\mathcal{E}^{T\Lambda}$ yields a semantics of ML. Moreover, we have:

Theorem 3.11 *If a model $\mathcal{M}$ is fully abstract for $\mathcal{E}^{T\Lambda}$ then $\mathcal{M}$ is also fully abstract for $\mathcal{E}^{\mathrm{ML}}$.*

Proof Let $\mathcal{M}$ be any fully abstract model for $\mathcal{E}^{T\Lambda}$. By our assumption on $\mathcal{E}^{\mathrm{ML}}$ and the definition of $\mathcal{M}[\![\,]\!]^{\mathrm{ML}}$, it is sufficient to show the condition of full abstraction for $\mathcal{E}^{\mathrm{ML}}$ for terms that do not contain **let**-construct. Suppose $\Sigma \rhd e_1 \overset{\mathrm{ML}}{\approx} e_2 : \rho$, where e_1, e_2 do not contain **let**-construct. By the definition of $\overset{\mathrm{ML}}{\approx}$, $\mathcal{A} \rhd e_1 \overset{\mathrm{ML}}{\approx} e_2 : \tau$ for any ground instance $(\mathcal{A}, \tau)$ of (Σ, ρ). Let $\mathcal{A} \rhd M : \tau, \mathcal{A} \rhd N : \tau$ be $T\Lambda^+$ terms that correspond to derivations of $\mathcal{A} \rhd e_1 : \tau$ and $\mathcal{A} \rhd e_2 : \tau$ respectively. Let $C[]$ be any closing b-context for $\mathcal{A} \rhd M : \tau$ and $\mathcal{A} \rhd N : \tau$. Let $c[]$ be the context obtained by erasing all type sppecification. Then $erae(C[M]) \equiv c[e_1]$, $erase(N) = c[e_2]$ and $c[]$ is a closing b contex for $\mathcal{A} \rhd e_1 : \tau$ and $\mathcal{A} \rhd e_2 : \tau$. By assumption 3.4, $\mathcal{E}^{T\Lambda}(\mathcal{A} \rhd C[M] : b) \Downarrow \emptyset \rhd c^b : b$ iff $\mathcal{E}^{\mathrm{ML}}(\mathcal{A} \rhd c[e_1] : b) \Downarrow \emptyset \rhd c : b$ and $\mathcal{E}^{T\Lambda}(\mathcal{A} \rhd C[N] : b) \Downarrow \emptyset \rhd c^b : b$ iff $\mathcal{E}^{\mathrm{ML}}(\mathcal{A} \rhd c[e_2] : b) \Downarrow \emptyset \rhd c : b$. Since $\mathcal{A} \rhd e_1 \overset{\mathrm{ML}}{\approx} e_2 : \tau, \mathcal{A} \rhd M \overset{T\Lambda}{\approx} N : \tau$. By the full abstraction of $\mathcal{M}$ for $\mathcal{E}^{T\Lambda}$ and by the definition of $\mathcal{M}[\![]\!]^{\text{ML}}$, $\mathcal{M} \models_{\text{ML}} \Sigma \rhd e_1 = e_2 : \rho$. Conversely, suppose $\mathcal{M} \models_{\text{ML}} \Sigma \rhd e_1 = e_2 : \rho$, where e_1, e_2 do not contain **let**-construct. Let $c[\,]$ be any closing b-context for $\Sigma \rhd e_1 : \rho$ and $\Sigma \rhd e_1 : \rho$. Let Δ_1 be a derivation of $\emptyset \rhd c[e_1] : b$ such that it contains a subderivation Δ_2 of $\mathcal{A} \rhd e_1 : \tau$ where $(\mathcal{A}, \tau)$ is an instance of (Σ, ρ). Since $c[\,]$ is a closing b-context for $\Sigma \rhd e_1 : \rho$, such Δ_1 always exists. Since $\Sigma \rhd e_2 : \rho$ is a term, there is a derivation Δ_3 of $\mathcal{A} \rhd e_2 : \tau$. Then by typing rules and the definition of contexts, the derivation Δ_4 obtained from Δ_1 by replacing the subtree Δ_2 by Δ_3 is a derivation of $\emptyset \rhd c[e_2] : b$. Let $M_1 \equiv \mathit{typedterm}(\Delta_1)$, $M_2 \equiv \mathit{typedterm}(\Delta_2)$, $M_3 \equiv \mathit{typedterm}(\Delta_3)$, $M_4 \equiv \mathit{typedterm}(\Delta_4)$. Then $\mathit{erase}(M_2) \equiv e_1$, $\mathit{erase}(M_3) \equiv e_2$. Clearly M_1, M_4 respectively contain M_2, M_3 as subterms. Moreover, the $T\Lambda^+$ contexts obtained from M_1, M_4 by replacing respectively M_2, M_3 with the 'hole' are identical. Call this context $C[\,]$. Then $\mathit{erase}(C[M_2]) \equiv c[e_1]$ and $\mathit{erase}(C[M_3]) \equiv c[e_2]$. Since $\mathcal{M} \models_{\text{ML}} \Sigma \rhd e_1 = e_2 : \rho$, $\mathcal{M} \models_{T\Lambda} \mathcal{A} \rhd M_2 = M_3 : \tau$. By the full abstraction of $\mathcal{M}$ for $\mathcal{E}^{T\Lambda}$, $\mathcal{E}^{T\Lambda}(\emptyset \rhd C[M_2] : \tau) \Downarrow \emptyset \rhd c^b : b$ iff $\mathcal{E}^{T\Lambda}(\emptyset \rhd C[M_3] : \tau) \Downarrow \emptyset \rhd c^b : b$. By assumption 3.4, $\mathcal{E}^{\text{ML}}(\emptyset \rhd c[e_1] : b) \Downarrow \emptyset \rhd c : b$ iff $\mathcal{E}^{\text{ML}}(\emptyset \rhd c[e_2] : b) \Downarrow \emptyset \rhd c : b$. ∎

The importance of this result is that we can immediately apply results already developed for explicitly typed languages to implicitly typed language with ML polymorphism. As an example, Plotkin constructed [92] a fully abstract model of his language PCF with *parallel conditionals.* It is not hard to define the "ML version" of PCF (with parallel conditionals) by deleting type specifications of bound variables and adding let expressions. Its operational semantics can be also defined in such a way that it satisfies our assumptions. We then immediately have a fully abstract model for the ML-version of PCF.

Chapter 4

Database Domains

This chapter constructs a theory of database domains and proposes a type system for complex database objects. They will not only provide a unform framework for various data models for databases including the relational model, nested relations, and complex object models but they also enable us to integrate those data models into an ML style polymorphic type system we have investigated in the previous chapter. This integration will be carried out in the next chapter. Most of the results in this chapter were presented in [83].

4.1 Introduction

There have been a number of attempts to develop data models to represent complex database objects beyond the first-normal-form relational model. Examples include nested relations [36, 2, 89, 96] and complex object models [59, 14, 4]. (See also [57] for a survey.) As we have argued in section 1.1, however, these complex data structures and associated database operations have not been well integrated in a type system of a programming language, creating the problem of "impedance mismatch". I believe that the major source of this mismatch problem is poor understanding of the properties of types for databases and the structures of domains for database objects. Traditionally, the theory of types of programming languages has been focussed on function types and domains of functions. Neither the properties of database type systems nor their relationship to type systems of programming languages have been well investigated.

The goal of this chapter is to construct a theory of database type systems that will serve as a "bridge" between complex data models and type systems of programming languages and to propose a concrete database type system that is rich enough to represent a wide range of complex database objects. Later in chapter 5, by integrating the type system we develop in this chapter and ML type system we have analyzed in the previous chapter, we will develop a strongly typed polymorphic programming language for databases and other data intensive applications. In the rest of this section, we overview the data structures and operations needed for databases and outline our strategy.

4.1.1 Data Structures for Database Objects

As suggested by Cardelli [24], one way to represent complex objects in a programming language is to use labeled records and labeled disjoint unions (or labeled variants) found in many programming languages such as Pascal, Standard ML [47], Amber [23] and Galileo [7]. The following is an example of a labeled record expression:

$$[Name = [Firstname = "Joe", Lastname = "Doe"], Dept = "Sales", Office = 278].$$

Types for expressions can be easily defined. For example, the above record is given the following type:

$$[Name : [Firstname : string, Lastname : string], Dept : string, Office : int].$$

Assuming computable equality on each atomic type, equality on expressions that do not contain functions is computable and it is not hard to introduce (finite) set expressions on those complex expressions. These three data constructors – record, variant and set – are basic building blocks for complex object models. In database literature, they are respectively called *aggregation*, *generalization* and *grouping*. Tuples in the relational model [29] are represented by records that only contain atomic values and relations are simply sets of those records. Data structures in various forms of non-first-normal-form relations [36, 2, 89, 96] are represented by the combination of record and set constructors. Data structures in complex object models [4, 59, 14] correspond to unrestricted nested structures generated by the above three constructors. When combined with recursive definiton (or cyclic data constructor), unrestricted nesting of these three constructors seem rich enough to represent virtually all complex data models.

It is not hard to integrate these data structures into a type system of a programming language. Many languages allow unrestricted nesting of records and variants. Some languages such as Miranda [107] also allow recursively defined types and expressions. As we have mentioned, finite sets can also be introduced in those type systems. Moreover, recent studies on type inference [111, 85, 63, 93] – including a contribution of a part of this study ([85]) which will be presented in the next chapter

– show that these data structures can be integrated in a polymorphic type system with static type inference. Therefore, as far as data structures are concerned, type systems of several programming languages seem to have sufficient expressive power to represent databases.

4.1.2 Operations on Complex Objects

In a programming language, in addition to the operations that construct these structures, the following standard operations are available (or can be easily added):

- field selection from a record,
- field modification (update) of a record,
- cases analysis for a variant,
- standard set theoretic operations and a primitive for mapping a function over a set.

It is therefore tempting to represent a database of complex objects as a set of complex expressions which is manipulated by functions defined using the above primitive operations.

An obvious problem of this approach is that, in practice, both expressions and sets become very large and contain a great deal of redundancy. This problem is elegantly solved in the relational model by the introduction of the two database operations – (*natural*) *join* and *projection*. Instead of representing a database as one large set (relation) of complex tuples, we can first project it onto various small relations and then represent a database as a collections of those small relations. Larger relations are obtained by joining these small relations when needed. In order to integrate complex database objects in a programming language, it is therefore essential to generalize join and projection so that they work uniformly on complex expressions and to introduce them in a programming language. I further believe that properly generalized join and projection together with standard operations on complex expressions form a sufficiently rich set of operations for complex database objects. It will be also shown in chapter 7 that join and projection play essential roles in manipulating object-oriented databases.

There have been some arguments on expressive power of sets of operations on data structures for databases. A well known example is that the relational algebra cannot compute the transitive closure of a given relation [5]. However, the lack of such computational power does not imply incompleteness of the set of operations on database objects. It simply suggests the desideratum that data models should be integrated into a standard computational paradigm such as function abstraction and recursion, which are readily available in programming languages. For example, it is unnatural to try to compute a transitive closure by a set of primitive operations for records and sets. It is even more unnatural to require such computational power for those primitive operations. The computation of a transitive closure naturally suggests iteration or recursion and is of course computable if the relational algebra is integrated in a programming language where recursion or iteration are available.

4.1.3 A Strategy to Generalize Join and Projection

There are several efforts to generalize join and projection beyond the first normal form relations [96, 88, 36, 62]. However, their definitions still depend on the underlying tuple structures using some forms of *unnesting* or *flattening* operations. For example, in [96] the notion of *partition normal form relations* was introduced, which is essentially those that can be transformed into first normal form relations by unnesting. By imposing further restrictions on partition normal form relations they extended join to non first normal form relations. However, the justifications for these ad hoc restrictions are not clear besides the fact that the relational algebra including join can be extended to those restricted non first normal form relations. Here we would like to extend join and projection uniformly to arbitrary complex database objects including recursively defined ones in such a way that they can be integrated in an ML style type system as polymorphically typed computable functions. It should be worth noting that join and projection in the relational model are polymorphic operations in the sense that are defined uniformly over relations of various types.

Join and projection in the relational model are based on the underlying operations that compute a join of tuples and a projection of a tuple. By regarding tuples as *partial descriptions* of real-world entities, we can characterize these operations as special cases of very general operations on partial descriptions; the one that *combines* two consistent descriptions and the one that *throws away* part of a given description. For example, if we consider the following non-flat tuples

$$t_1 = [Name = [Fn = "Joe"]]$$

and

$$t_2 = [Name = [Ln = "Doe"]]$$

as partial descriptions, then the combination of the two should be

$$t = [Name = [Fn = "Joe", Ln = "Doe"]].$$

Conversely, the tuple t_1 is considered as the result of the projection of the partial description t on the structure specified by the type

$$[Name : string, [Fn : string]].$$

Operations that combine partial information also arise in other areas of applications. Examples include the meet operation on Aït-Kaci's ψ-terms [6] and the "unification" operation on *feature structures* representing linguistic information (see [102] for a survey).

Based on this general intuition, in this chapter, we propose a framework for type systems for database objects and their denotational semantics. We then construct a concrete database type system and its semantic domain. The type system contains arbitrarily complex expressions definable by labeled records, labeled variants, finite sets and recursive definition. On its semantic domain, join and projection are defined as polymorphically typed computable functions. Furthermore, we carry out these constructions in a completely effective way. In our framework, we require types and objects to be finitely representable and the properties needed to define database operations to be decidable. This means that, once we have constructed a type system and its semantic domain based on our framework, it not only provides an uniform and elegant explanation of the properties of the type system and the structures of domain of complex database objects, but it also provides representations and algorithms to integrate them into a programming language.

We start with our investigation by analyzing the relational model. This analysis will also serve as an introduction to the subsequent abstract characterizations of database type systems and their semantic domains. Based on the analysis of the relational model, in section 4.3, we characterize the structures of type systems in which polymorphic join and polymorphic projection are definable and propose a framework for their denotational semantic. In section 4.4, we define a concrete type system for complex database objects and construct its semantic domain. A part of the construction of the semantic domain (subsection 4.4.4) is based on the idea we have developed in [22] that a certain ordering on powerdomains can be used to generalize the relational join uniformly to complex objects and the idea due to Aït-Kaci [6] that a rich yet computationally feasible domain of values is nicely represented by *labeled regular trees*. I have also noticed that Rounds' recent work [97] achieves a result similar to one presented in subsection 4.4.4.

4.2 Analysis of the Relational Model

We first give a standard definition of the relational model. Since our purpose is to extract the essence of the type structure of the model, we define the model as a typed data structure. We also integrate *null values* in the model. The importance of null values has been widely recognized and

$$\{\![[Name = \text{"}Joe\ Doe\text{"}, Age = 21, Salary = 21000] \\ [Name = \text{"}John\ Smith\text{"}, Age = null_{int}, Salary = 34000]\ ]\!\} \\ : \{\![[Name : string, Age : int, Salary : int]]\!\}$$

Name:string	Age:int	Salary:int
"Joe Doe"	21	21000
"John Smith"	$null_{int}$	34000

Figure 4.1: A Simple Relation and its Representation as a Table

several approaches have been proposed [16, 100, 68, 112]. Among them, we adopt the approach that null values represent *non-informative* values [112]. This approach fits well in our paradigm that database objects are partial descriptions and plays a crucial role in our theory of semantic domains of database type systems, which will be developed in the next section.

We continue to assume that $\mathcal{L}$ and $\mathcal{B}$ are respectively a given set of labels and a given set of base types. We also assume that we are given a set $\{D_b | b \in \mathcal{B}\}$ of pairwise disjoint sets of atomic values. It should be noted that D_b does not necessarily coincide with the set F_b for the type b in a model of a programming language we have defined in section 3.3. It is required that $D_b \subseteq F_b$ (or there is some injective mapping from D_b to F_b) but we do not assume the inverse inclusion. For example, when recursive functions are definable, F_d should contain a value that corresponds to expressions that diverge, which is not an element of D_b. For each base type b, we introduce a special symbol $null_b$ for the null value of the type b. We say that c has the type b if $c \in D_b$ or $c = null_b$.

Definition 4.1 (Tuples and Relations) *A tuple type T has the following syntax $[l_1 : b_1, \ldots, l_n : b_n]$ where $l_1, \ldots, l_n$ are pairwise distinct elements of $\mathcal{L}$ and $b_1 \ldots, b_n \in \mathcal{B}$. A tuple t of the tuple type $[l_1 : b_1, \ldots, l_n : b_n]$ is a term of the form $[l_1 = c_1, \ldots, l_n = c_n]$ such that c_i has the type b_i $(1 \leq i \leq n)$. A relation type (or relation scheme in the database literature) R is a term of the form $\{\![T]\!\}$ for some tuple type T. A relation instance r of the relation type $\{\![T]\!\}$ is a term of the form $\{\![t_1, \ldots, t_n]\!\}$ such that each t_i $(1 \leq i \leq n)$ is a tuple of the type T.*

Regarding a tuple t as a function from a finite subset $L \subset \mathcal{L}$ to $\bigcup_{b \in \mathcal{B}} D_b \cup \{null_b | b \in \mathcal{B}\}$, we write $dom(t)$ for the set of labels in t and $t(l)$ for the value corresponding to the label l. Similar notations are used for tuple types. Figure 4.1 shows a simple example of relation instance and its standard representation as a table.

Relation instances are terms representing sets, for which the following equations hold:

$$\{\![t_1, \ldots, t_n]\!\} = \{\![t_{i_1}, \ldots, t_{i_n}]\!\} \text{ if } i_1, \ldots, i_n \text{ is a permutation of } 1, \ldots, n$$

and

$$\{\!\{t_1, t_2, t_3, \ldots\}\!\} = \{\!\{t_2, t_3, \ldots\}\!\} \text{ if } t_1 = t_2.$$

We consider relation instances as equivalence classes of the above equality. Under this equality, relation instances behave exactly like sets of tuples, on which ordinary set-theoretic operations are defined. Based on this fact, we treat relation instances as sets of tuples and apply ordinary set-theoretic notions directly to them. Readers might think that this strictly syntactic treatment only introduces (trivial but annoying) complication to the model that were simpler and more intuitive if we treated them just as sets. This had been true if we were only interested in sets of finite tuples such as flat relations in the relational model. However, it is no longer possible to maintain such intuitive treatment when we allow infinite structures through recursive definition. Our syntactic treatment provides a uniform way to deal with complex structures involving recursively defined data.

Among the operations in the relational algebra, we only define join and projection. As we have argued, these two operations make the model a successful data model for databases. They also distinguish the model from standard type systems of programming languages. As we will see in section 5.7, other operations are definable using standard operations on records and sets.

Two tuple types T_1, T_2 are *consistent* if for all $l \in dom(T_1) \cap dom(T_2)$, $T_1(l) = T_2(l)$. Let T_1, T_2 be two consistent tuple types. Define $jointype(T_1, T_2)$ as the type T such that $dom(T) = dom(T_1) \cup dom(T_2)$ and $T(l) = T_1(l)$ if $l \in dom(T_1)$ otherwise $T(l) = T_2(l)$. The two tuples t_1, t_2 are *consistent* if for all $l \in dom(t_1) \cap dom(t_2)$ one of the following hold: (1) $t_1(l) = t_2(l)$, (2) $t_1(l) = null_b$ and $t_2(l) \in D_b$ or (3) $t_1(l) \in D_b$ and $t_2(l) = null_b$. Two relation types $\{\!\{T_1\}\!\}, \{\!\{T_2\}\!\}$ are *consistent* if T_1, T_2 are consistent. For two consistent relation types $\{\!\{T_1\}\!\}, \{\!\{T_2\}\!\}$, define $jointype(\{\!\{T_1\}\!\}, \{\!\{T_2\}\!\})$ as the relation type $\{\!\{jointype(T_1, T_2)\}\!\}$.

Definition 4.2 (Relational Join) *If t_1, t_2 are consistent tuples then the join of t_1, t_2, denoted by $join(t_1, t_2)$, is the following tuple t:*

$$\begin{aligned} dom(t) &= dom(t_1) \cup dom(t_2), \\ t(l) &= \begin{cases} t_1(l) & \textit{if } l \in dom(t_1) \textit{ and either } l \notin dom(t_2) \textit{ or } t_2(l) = null_b \\ t_2(l) & \textit{otherwise} \end{cases} \end{aligned}$$

If r_1, r_2 are relation instances having consistent relation types then the (natural) join of r_1, r_2, denoted by $join(r_1, r_2)$, is the relation instance $\{\!\{join(t_1, t_2) | t_1 \in r_1, t_2 \in r_2, t_1, t_2$ are consistent$\}\!\}$.

For $join(t_1, t_2)$ and $join(r_1, r_2)$ the following properties hold:

Proposition 4.1 *Let t_1, t_2 be tuples of the type T_1, T_2 respectively. If $join(t_1, t_2)$ is defined then it has the type $jointype(T_1, T_2)$.*

Let r_1, r_2 be relation instances of the type R_1, R_2 respectively. If $join(r_1, r_2)$ is defined then it has the type $jointype(R_1, R_2)$.

Proof By definition, $dom(t) = dom(jointype(T_1, T_2))$. Let l be any label in $dom(t)$. Then either $join(t_1, t_2)(l) = t_1(l)$ with the type $T(l)$ or $join(t_1, t_2)(l) = t_2(l)$ with the type $T_2(l)$. But by definition, $T_1(l) = jointype(T_1, T_2)(l)$ for all $l \in dom(T_1)$. Similarly for T_2. Thus t has the type $jointype(T_1, T_2)$.

R_1, R_2 must respectively be of the forms $\{\!\{T_1\}\!\}, \{\!\{T_2\}\!\}$. Let t be any tuple in $join(r_1, r_2)$. By definition, there are some $t_1 \in r_1$ and $t_2 \in r_2$ such that $t = join(t_1, t_2)$. By the previous result, t has the type $jointype(T_1, T_2)$. Then by definition, r has the type $\{\!\{jointype(T_1, T_2)\}\!\}$, which is equal to $jointype(R_1, R_2)$. ▌

Definition 4.3 (Relational Projection) *If t is a tuple of the form $[l_1 = c_1, \ldots, l_n = c_n, \ldots]$ such that each c_i has the type b_i $(i \leq n)$ then the projection of t onto the type $T = [l_1 : b_1, \ldots, l_n : b_n]$, denoted by $project^T(t)$, is the tuple $[l_1 = c_1, \ldots, l_n = c_n]$.*

If r is the relation instance such that for all $t \in r$, $project^T(t)$ is defined then the projection of r onto the type $\{\!\{T\}\!\}$, denoted by $project^{\{\!\{T\}\!\}}(r)$, is the relation instance $\{\!\{project^T(t) | t \in r\}\!\}$.

For $project^T(t)$ and $project^{\{\!\{T\}\!\}}(r)$, the following properties are immediate consequence of the definitions:

Proposition 4.2 *If $project^T(t)$ is defined then it has the type T. If $project^{\{\!\{T'\}\!\}}(r)$ is defined then it has the type $\{\!\{T\}\!\}$.* ▌

When restricted to tuples without null values, the above definitions are straightforward translations of the corresponding definitions in the relational model found for example in [108, 35, 73]. The operation *join* is extended to relations containing null values. Figure 4.2 shows an example of a join of relations containing null values.

Note that the definition of join reflects the intended semantics of null values. Projection is specified by a type not just a set of labels. This will allow us to generalize the relational projection to complex structures.

Remark: The combination of non-informative null values and join operation may sometimes

Name	Age	Salary
"Joe Doe"	21	21000
"John Smith"	$null_{int}$	34000

r_1

Name	Age	Office
"Joe Doe"	$null_{int}$	103
"John Smith"	$null_{int}$	278
"Mary Jones"	41	556

r_2

Name	Age	Salary	Office
"Joe Doe"	21	21000	103
"John Smith"	$null_{int}$	34000	278

$join(r_1, r_2)$

Figure 4.2: Join of Relations Containing Null Values

yield counter-intuitive results.[1] For example, the join of the two relations

Course	Instructor
"Math110"	*"K. Jones"*
$null_{string}$	*"S. Brown"*

Course	Student
"CIS310"	*"Joe Doe"*
$null_{string}$	*"John Smith"*

is the relation

Course	Instructor	Student
"Math110"	*"K. Jones"*	*"John Smith"*
"CIS310"	*"S. Brown"*	*"Joe Doe"*
$null_{string}$	*"S. Brown"*	*"John Smith"*

which suggests that instructor *"S. Brown"* is related to students *"Joe Doe"*, *"John Smith"*, although no such relationship is implied by the two original relations. Based on this observation, it has been argued [61] that join (as well as other relational operations) could not be extended "semantically correctly" to this form of null values. This is, however, not the problem of interpretation of null values but the problem of join operation. Without null values, join still yields same kind of counter-intuitive results. For example, the join of

Name	Age
"Joe Doe"	21
"John Smith"	21

Age	Salary
21	21000
21	34000

is the relation

[1] This was pointed out to me by Tomasz Imielinski

Name	Age	Salary
"*Joe Doe*"	21	21000
"*Joe Doe*"	21	34000
"*John Smith*"	21	21000
"*John Smith*"	21	34000

which suggests that "*Joe Doe*" is related to both the salaries of 21000 and 34000. Intuitively, the situation is stated that natural join as defined in the relational model does not necessarily "preserve semantics of relations". In order to investigate the problem, we need to construct a model of our "real world" and to define semantics of relations with respect to the model. This is out of the scope of this thesis and I refer the interested readers to [82] where this problem was fully investigated in the context of flat relations with null values and it was shown that join can be given a satisfactory semantics which is completely compatible to non informative null values. This study can be extended to complex database objects we are investigating in this chapter.

Returning to our problem of extending join and projection, the above definitions apparently depend on the underlying structure of flat tuples. Here, we would like to characterize join and projection independently of the underlying data structures so that we can generalize them uniformly to a wide range of complex data structures and introduce them into a type system of a programming language. Our guiding intuition is the idea we have exploited in [22] that database objects are *partial descriptions* of real-world entities and are *ordered* in terms of their "goodness of descriptions". The idea of partial description was originally suggested by Lipski [69]. The corresponding order structure was studied by Biskup [16] and Zaniolo [112] in connection to null values and is closely related to the orderings on ψ-terms [6] and directed graphs [97].

For generality and simplicity, we treat tuples and relations uniformly. We call both tuple types and relation types *flat description types* (ranged over by σ) and tuples and relation instances *flat descriptions* (ranged over by d). For each flat description type σ, we write D_σ for the set of descriptions of the type σ. A flat description type represents a structure of descriptions. Such structures are naturally *ordered* to represent the intuition that one *contains* the other. For example, if $\sigma_1 = [Name : string, Age : int]$ and $\sigma_2 = [Name : string, Age : int, Office : int]$, then the structure represented by σ_2 contains the structure represented by σ_1. This intuitive idea is formalized by the following ordering:

Definition 4.4 (Ordering on Flat Description Types) *The information ordering $\leqslant$ on flat description types is the smallest relation containing:*

$$[l_1 : b_1, \ldots, l_n : b_n] \quad \leqslant \quad [l_1 : b_1, \ldots, l_n : b_n, \ldots],$$

$$\{\!\{\sigma_1\}\!\} \;\leqslant\; \{\!\{\sigma_2\}\!\} \text{ if } \sigma_1 \leqslant \sigma_2.$$

This relation is clearly a partial order. Moreover, since it is based on the inclusion of fields of records, this ordering has the following properties:

1. $\leqslant$ on the set of description types has the pairwise bounded join property, and
2. the ordering relation $\leqslant$ is decidable and least upper bounds (if they exist) are effectively computable.

The importance of this ordering is that it provides the following characterization of the types of the relational join and the relational projection:

Theorem 4.1 (Types of Relational Join and Projection) *Let d_1, d_2 be flat descriptions of the types σ_1, σ_2 respectively.*

1. If $join(d_1, d_2)$ is defined then $\sigma_1 \sqcup \sigma_2$ exists and $join(d_1, d_2)$ has the type $\sigma_1 \sqcup \sigma_2$.

2. If $project^{\sigma}(d_1)$ is defined then $\sigma \leqslant \sigma_1$ and $project^{\sigma}(d_1)$ has the type σ.

Proof The property of join follows from proposition 4.1 and the fact that $jointype(\sigma_1, \sigma_2)$ is defined iff $\sigma_1 \sqcup \sigma_2$ is defined and, when they are defined then their values are equal. (In what follows, we usually write $F = G$ to mean that F is defined iff G is defined and when they are defined then their values are equal). The property of *project* follows from proposition 4.2 and its definition. ∎

We can then give the following type schemes (polymorphic types) to join and projection:

$$\begin{array}{rcll} join & : & (\sigma_1 \times \sigma_2) \rightarrow \sigma_1 \sqcup \sigma_2 & \text{for all } \sigma_1, \sigma_2 \text{ such that } \sigma_1 \sqcup \sigma_2 \text{ exists}, \\ project^{\sigma_1} & : & \sigma_2 \rightarrow \sigma_1 & \text{for all } \sigma_1, \sigma_2 \text{ such that } \sigma_1 \leqslant \sigma_2. \end{array}$$

Since the ordering relation is decidable and least upper bounds are effectively computable, these types allow us to type-check expressions containing joins and projections.

We next characterize these operations themselves using orderings on descriptions. As observed in [16, 112], the introduction of null values induces the following ordering on tuples:

$$[l_1 = x_1, \ldots, l_n = x_n] \sqsubseteq [l_1 = y_1, \ldots, l_n = y_n] \text{ iff either } x_i = null_b \text{ or } x_i = y_i \; (1 \le i \le n).$$

This ordering is interpreted as the ordering of "goodness of descriptions". The following is an example of this ordering:

$$[Name = \text{"}Joe\ Doe\text{"}, Age = null_{int}] \sqsubseteq [Name = \text{"}Joe\ Doe\text{"}, Age = 21].$$

It is clear that for any tuple type T this ordering is a partial order on D_T with the pairwise bounded join property. Join on tuples of a same type is characterized as the least upper bound operation under this ordering, which formalizes our intuition that join is an operation that combines partial descriptions:

Proposition 4.3 (Join of Flat Tuples) *If $t_1, t_2 \in D_T$ then $join(t_1, t_2) = t_1 \sqcup t_2$.*

Proof By definitions. ▌

For a relation type R, an appropriate ordering on D_R to characterize join on D_R turns out to be the ordering known as *Smyth powerdomain ordering* [104]. To define the ordering, we first define the preorder $\preceq$:

$$\{\!\{t_1, \dots, t_n\}\!\} \preceq \{\!\{t'_1, \dots, t'_m\}\!\} \text{ if } \forall t'_j \in \{t'_1, \dots, t'_m\} \exists t_i \in \{t_1, \dots, t_n\}. \, t_i \sqsubseteq t'_j.$$

The relation $\preceq$ is not antisymmetric. However, we can take the quotient poset (definition 2.3) induced by the preorder:

Proposition 4.4 *For any relation type R, $[(D_R, \preceq)]$ is a poset with the pairwise bounded join property.*

Proof $\preceq$ is clearly transitive and reflexive and therefore $(D_R, \preceq)$ is a preordered set. Let r_1 and r_2 be any elements in D_R under $\preceq$. Let $r = \{\!\{join(t_1, t_2) | t_1 \in r_1, t_2 \in r_2, t_1, t_2 \text{ are consistent}\}\!\}$. Since $t_1 \sqcup t_2 = join(t_1, t_2)$, as a special case of the result shown in [104], r is a least upper bound of r_1 and r_2. Then the proposition follows from lemma 2.1. ▌

We regard a relation instance as a representative of the corresponding equivalence class induced by the above preorder and write $d_1 \sqcup d_2$ for the least upper bound of the corresponding equivalence classes. We also write $(D_R, \sqsubseteq)$ for $[(D_R, \preceq)]$. Readers are referred to [22, 82] for the intuition and relevance of this ordering in various aspects of databases. For the purpose of formalizing the relational model, this ordering provides the following characterization of join on relations we have shown in [22]:

Proposition 4.5 (Join of Flat Relations) *If $r_1, r_2 \in D_R$ then $join(r_1, r_2) = r_1 \sqcup r_2 = r$.* ▌

In order to characterize projections and joins of descriptions of different types, we interpret the partially ordered space of flat description types by a space of domains connected by *coercions*.

Definition 4.5 (Coercions between Relational Domains) *The set of up-coercions is the set of mappings* $\{\phi_{\sigma_1 \to \sigma_2} | \sigma_1 \leqslant \sigma_2\}$ *defined as*

1. *if* $\sigma_1 = [l_1 : b_1, \ldots, l_n : b_n]$, $\sigma_2 = [l_1 : b_1, \ldots, l_n : b_n, l_{n+1} : b_{n+1}, \ldots, l_{n+m} : b_{n+m}]$ $(n, m \geq 0)$ *then*

$$\phi_{\sigma_1 \to \sigma_2}([l_1 = c_1, \ldots, l_n = c_n]) = \\ [l_1 = c_1, \ldots, l_n = c_n, l_{n+1} = null_{b_{n+1}}, \ldots, l_{n+m} = null_{b_{n+m}}],$$

2. *if* $\sigma_1 = \{\!\{\sigma_1'\}\!\}$, $\sigma_2 = \{\!\{\sigma_2'\}\!\}$ *and* $\sigma_1' \leqslant \sigma_2'$ *then*

$$\phi_{\sigma_1 \to \sigma_2}(r) = \{\!\{\phi_{\sigma_1' \to \sigma_2'}(t) | t \in r\}\!\}.$$

The set of down-coercions is the set of mappings $\{\psi_{\sigma_1 \to \sigma_2} | \sigma_2 \leqslant \sigma_1\}$ *defined as*

1. *if* $\sigma_1 = [l_1 : b_1, \ldots, l_n : b_n, \ldots]$ *and* $\sigma_2 = [l_1 : b_1, \ldots, l_n : b_n]$ $(n \geq 0)$ *then*

$$\psi_{\sigma_1 \to \sigma_2}([l_1 = c_1, \ldots, l_n = c_n, \ldots]) = [l_1 = c_1, \ldots, l_n = c_n],$$

2. *if* $\sigma_1 = \{\!\{\sigma_1'\}\!\}$, $\sigma_2 = \{\!\{\sigma_2'\}\!\}$ *and* $\sigma_2' \leqslant \sigma_1'$ *then*

$$\psi_{\sigma_1 \to \sigma_2}(r) = \{\!\{\psi_{\sigma_1' \to \sigma_2'}(t) | t \in r\}\!\}.$$

Intuitively, an up-coercion coerces a description to a description of larger structure by "padding" extra part of structure with null values. A down-coercion on the other hand coerces a description to a description of a smaller structure by "throwing away" part of its structure. For example, if

$$\begin{aligned} \sigma_1 &= [Name : string, Age : int], \\ \sigma_2 &= [Name : string, Office : int], \\ \sigma_3 &= [Name : string, Age : int, Office : int], \\ t_1 &= [Name = \text{"}Joe\text{"}, Age = 21], \\ t_2 &= [Name = \text{"}Joe\text{"}, Office = 278], \\ t_3 &= [Name = \text{"}Joe\text{"}, Age = 21, Office = 278] \end{aligned}$$

then

$$\begin{aligned} \phi_{\sigma_1 \to \sigma_3}(t_1) &= [Name = \text{"}Joe\text{"}, Age = 21, Office = null_{int}], \\ \phi_{\sigma_2 \to \sigma_3}(t_2) &= [Name = \text{"}Joe\text{"}, Age = null_{int}, Office = 278], \\ \psi_{\sigma_3 \to \sigma_1}(t_3) &= t_1, \\ \psi_{\sigma_3 \to \sigma_2}(t_3) &= t_2. \end{aligned}$$

We then have the following equations:

$$\begin{aligned} join(t_1, t_2) &= \phi_{\sigma_1 \to \sigma_3}(t_1) \sqcup_{D_{\sigma_3}} \phi_{\sigma_2 \to \sigma_3}(t_2), \\ project_{\sigma_1}(t_3) &= \psi_{\sigma_3 \to \sigma_1}(t_3), \\ project_{\sigma_2}(t_3) &= \psi_{\sigma_3 \to \sigma_2}(t_3). \end{aligned}$$

This example suggests that computing a join of descriptions of types σ_1, σ_2 corresponds to coercing them to descriptions of the type $\sigma_1 \sqcup \sigma_2$ followed by computing their least upper bound. The projections correspond to down-coercions. Indeed we have:

Theorem 4.2 (Relational Join and Projection) *Let d_1 and d_2 be any flat descriptions of types σ_1, σ_2 respectively and σ be any type such that $\sigma \leqslant \sigma_1$.*

$$\begin{aligned} join(d_1, d_2) &= \phi_{\sigma_1 \to (\sigma_1 \sqcup \sigma_2)}(d_1) \sqcup_{D_{\sigma_1 \sqcup \sigma_2}} \phi_{\sigma_2 \to (\sigma_1 \sqcup \sigma_2)}(d_2), \\ project^{\sigma}(d_1) &= \psi_{\sigma_1 \to \sigma}(d_1). \end{aligned}$$

Proof By the definitions of ϕ and *join*, $join(d_1, d_2) = join(\phi_{\sigma_1 \to (\sigma_1 \sqcup \sigma_2)}(d_1), \phi_{\sigma_2 \to (\sigma_1 \sqcup \sigma_2)}(d_2))$. Then the property of *join* follows from propositions 4.3 and 4.5. The property of projection is by definitions. ▮

The semantic space of the relational model is therefore characterized by the set

$$\{(D_\sigma, \sqsubseteq) | \sigma \text{ is a flat description type}\}$$

connected by the set of pairs of up- and down-coercions

$$\{(\phi_{\sigma_1 \to \sigma_2}, \psi_{\sigma_2 \to \sigma_1}) | \sigma_1 \leqslant \sigma_2\}$$

associated with the set of join operations $\{join_{(\sigma_1 \times \sigma_2) \to (\sigma_1 \sqcup \sigma_2)} | \sigma_1 \sqcup \sigma_2 \text{ exists}\}$ defined as

$$join_{(\sigma_1 \times \sigma_2) \to (\sigma_1 \sqcup \sigma_2)}(d_1, d_2) = \phi_{\sigma_1 \to (\sigma_1 \sqcup \sigma_2)}(d_1) \sqcup_{D_{\sigma_1 \sqcup \sigma_2}} \phi_{\sigma_2 \to (\sigma_1 \sqcup \sigma_2)}(d_2)$$

and the set of projection operations $\{project^{\sigma_1}_{\sigma_2 \to \sigma_1} | \sigma_1 \leqslant \sigma_2\}$ defined as

$$project^{\sigma_1}_{\sigma_2 \to \sigma_1}(d) = \psi_{\sigma_2 \to \sigma_1}(d).$$

The importance of this characterization is that it applies to any set of domains on which we can define information orderings and appropriate sets of coercions. Based on this analysis, in the next section, we formally define the structures of type systems for databases and their semantic domains.

4.3 Database Domains

As a generalization of the set of flat description types in the relational model, we define the notion of *database type systems*:

Definition 4.6 (Database Type Systems) *A database type system is a poset of types such that*

1. *it has the pairwise bounded join property, and*
2. *the ordering relation is decidable and least upper bounds (if they exist) are effectively computable.*

We call each type in a database type system a *description type*. A description type represents a structure of descriptions and the ordering on types represents the containment ordering of the structures they represent. The pairwise bounded join condition is necessary for the types of joins to be well defined. The decidability conditions is necessary for effective type-checking.

Each description type should denote a domain of descriptions. As a generalization of domains of flat descriptions in the relational model, we define the notion of *description domains*:

Definition 4.7 (Description Domains) *A description domain is a poset $(D, \sqsubseteq)$ satisfying:*

1. *D has the bottom element $null_D$, i.e. for any $d \in D, null_D \sqsubseteq d$,*
2. *D has the pairwise bounded join property,*
3. *the ordering relation $\sqsubseteq$ is decidable and least upper bounds (if they exist) are effectively computable.*

Condition 1 allows us to represent a non-informative value which is essential for partial descriptions. Condition 2 states that if we have two consistent descriptions then the combination of the two is also representable as a description. This is necessary for join to be well defined. Condition 3 is needed for effective computation of joins and other operations.

It should be noted that description domains are models of types of database objects and not models of general types in programming languages such as function types. In particular, they should not be confused with *Scott domains* [101] which is used to give semantics to untyped lambda calculus and programming languages with recursively defined functions [98]. Both notions share similar order structures and are based on a similar intuition that values are ordered in terms of "goodness of approximation". However, the properties of the two orderings are fundamentally different. The ordering on a description domain is just a computable predicate, which is introduced to generalize join and projection as computable polymorphic functions on complex database objects. On the other hand *Scott ordering* can be regarded as a predicate on the computability itself and in principle not computable. As an example of the difference, the bottom element in a description domain is simply an atomic value and does not corresponds to non terminating computation (or "divergent value") denoting the bottom element in a Scott domain. We also do not assume the *directed completeness*. As we will see in the next section, recursive (cyclic) types and objects are restricted to those that have a finite representation and are modeled by regular values not by limit points of ascending chains of the ordering.

By abstracting underlying tuple structures in the definition of up-coercions and down-coercions between relational domains, we interpret an ordering on description types as a relation induced by a special class of mappings between description domains. A function $f : D_1 \to D_2$ between description domains D_1, D_2 is *monotone* iff for any $x, y \in D_1$, $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$.

Definition 4.8 (Embeddings and Projections) *A monotone function $\phi : D_1 \to D_2$ is an embedding if there exists a function $\psi : D_2 \to D_1$ such that (1) for any $x \in D_2$, $\phi(\psi(x)) \sqsubseteq x$ and (2) for any $x \in D_1$, $\psi(\phi(x)) = x$. The function ψ is called a projection.*

A pair of embedding and projection is a special case of *Galois connections* (or *adjunctions*), for which the following result is well known [40]:

Lemma 4.1 *Given an embedding $\phi : D_1 \to D_2$, the corresponding projection is uniquely determined by ϕ.* ▌

If ϕ is an embedding, we sometimes denote by ϕ^R the corresponding projection.

If a pair of description domains (D_1, D_2) has an embedding-projection pair $(\phi : D_1 \to D_2, \psi : D_2 \to D_1)$ then D_2 contains an isomorphic copy $D_1' = \phi(D_1)$ of D_1 and for any element d in D_2 there is a unique maximal element $d' \in D_1'$ such that $d' \sqsubseteq d$. We regard this property as the semantics of the ordering of description types. ϕ maps an element $d \in D_1$ to the least element $d' \in D_2$ such that d' contains all information in d. ψ maps an element $d \in D_2$ to a unique maximal element $d' \in D_1$ that contains only information in d and is regarded as a database projection from D_2 to D_1. The set of up-coercions we have defined on relational domains are indeed a set of embeddings between relational domains. The corresponding set of projections is exactly the set of down-coercions.

Our characterization of the ordering on types can be regarded as a refinement of one of the characterizations of subtypes proposed by Bruce and Wegner [21], where the notion of subtypes is characterized in three ways; one of them being that the larger set contains an isomorphic copy of the smaller. It is also related to the notion of *information capacity* of data structures studied in [56] where an ordering on various data structures was defined by using mappings between sets of objects.

Finally we define a semantic space of a database type system as a space of description domains partially ordered by a set of embedding-projection pairs.

Definition 4.9 (Database Domains) *A database domain is a pair (Dom, Emb) of a set of description domains Dom and a set of embeddings Emb among Dom satisfying the following conditions:*

1. *For any two domains $D_1, D_2 \in Dom$, there is at most one $\phi \in Emb$ such that $\phi : D_1 \rightarrow D_2$. We write $\phi_{D_1 \rightarrow D_2}$ for an embedding from D_1 to D_2.*
2. *For any domain $D \in Dom$, $\phi_{D \rightarrow D} \in Emb$.*
3. *Emb is closed under composition.*
4. *For any two domains $D_1, D_2 \in Dom$, if there is some $D \in Dom$ such that $\phi_{D_1 \rightarrow D} \in Emb$ and $\phi_{D_2 \rightarrow D} \in Emb$ then there is a unique $D' \in Dom$ depending only on D_1, D_2 such that $\phi_{D_1 \rightarrow D'} \in Emb$, $\phi_{D_2 \rightarrow D'} \in Emb$ and for any $D'' \in Dom$ if $\phi_{D_1 \rightarrow D''} \in Emb$ and $\phi_{D_2 \rightarrow D''} \in Emb$ then $\phi_{D' \rightarrow D''} \in Emb$.*
5. *For any $\phi \in Emb$, both ϕ and ϕ^R are computable, i.e. there is an algorithm to compute $\phi(d)$ and $\phi^R(d')$ for any given $d \in dom(\phi)$ and $d' \in dom(\phi^R)$.*

Emb defines the relation on *Dom* such that D_1 and D_2 are related iff there is $\phi_{D_1 \rightarrow D_2} \in Emb$. This is intended to model an ordering on description types. Condition 1 means that there is only one way to interpret the ordering between two description domains. Moreover,

Proposition 4.6 *The relation defined by Emb is a partial order with the pairwise bounded join property.*

Proof From condition 2 and 3, the relation is reflexive and transitive. For anti-symmetricity, suppose $\phi_{X \rightarrow Y} \in Emb$ and $\phi_{Y \rightarrow X} \in Emb$ for some $X, Y \in Dom$. Apply condition 4 to X (as D_1) and Y (as D_2). Since $\phi_{X \rightarrow X} \in Emb$ and $\phi_{Y \rightarrow Y} \in Emb$, both X and Y satisfy the property of D'

in condition 4. Then the uniqueness of D' in condition 4 implies $X = Y$. The pairwise bounded join property is an immediate consequence of condition 4. ▮

Definition 4.10 (Models of Database Type Systems) *Let $(T, \leqslant)$ be a database type system. A database domain (Dom, Emb) is a model of $(T, \leqslant)$ if there is a mapping $\mu : T \rightarrow Dom$ such that for any $\sigma_1, \sigma_2 \in T$, $\sigma_1 \leqslant \sigma_2$ iff $\phi_{\mu(\sigma_1) \rightarrow \mu(\sigma_2)} \in Emb$.*

Remember that on description domains we imposed the conditions that the ordering is decidable and least upper bounds are computable. Combined with the computability condition on embeddings and projections, they guarantee that join and projection defined as

$$join_{(\sigma_1 \times \sigma_2) \rightarrow (\sigma_1 \sqcup \sigma_2)}(d_1, d_2) = \phi_{\sigma_1 \rightarrow (\sigma_1 \sqcup \sigma_2)}(d_1) \sqcup_{D_{(\sigma_1 \sqcup \sigma_2)}} \phi_{\sigma_2 \rightarrow (\sigma_1 \sqcup \sigma_2)}(d_2) \quad (4.1)$$

$$project^{\sigma_1}_{\sigma_2 \rightarrow \sigma_1}(d) = \psi_{\sigma_2 \rightarrow \sigma_1}(d) \quad (4.2)$$

are always computable functions. This means that if a database type system has a model, then join and projection are available as computable functions with the following polymorphic types:

$$join : (\sigma_1 \times \sigma_2) \rightarrow \sigma_1 \sqcup \sigma_2 \qquad \text{for all } \sigma_1, \sigma_2 \text{ such that } \sigma_1 \sqcup \sigma_2 \text{ exists} \quad (4.3)$$

$$project^{\sigma_1} : \sigma_1 \rightarrow \sigma_2 \qquad \text{for all } \sigma_1, \sigma_2 \text{ such that } \sigma_1 \leqslant \sigma_2 \quad (4.4)$$

The relational join and the relational projection are special cases of the above functions on flat tuple structures. Moreover, from the previous results, we have:

Theorem 4.3 *The set of flat description types with the information ordering $\leqslant$ is a database type system. The pair of the set of relational domains and the set of up-coercions*

$$(\{(D_\sigma, \sqsubseteq) | \sigma \textit{ is a flat description type}\}, \{\phi_{\sigma_1 \rightarrow \sigma_2} | \sigma_1 \leqslant \sigma_2\})$$

is a database domain and a model of the poset of flat description types. ▮

We therefore claim that the notions of database type systems and database domains are a proper generalization of the relational model. The advantage of our characterization is that it is independent of the actual structures of types and objects. This allows us to generalize the relational model to a wide range of complex data structures, even those that include recursively defined types and objects. In the next section we construct a database type system and its database domain, which I believe to be rich enough to cover virtually all proposed representations of complex database objects.

4.4 A Type System for Complex Database Objects

In addition to finite structures representable by finite terms, we would like to allow recursively defined structures, which naturally appear in descriptions of real-word entities. As demonstrated by Aït-Kaci [6], an appropriate formalism are ***regular trees***, which provides a sufficiently rich yet computationally feasible representation for recursive data structures. We therefore develop our type system and its domain using regular trees. However, this generality creates a slight technical complication that we cannot use induction to define structures and to prove properties. This may yield less intuitive definitions and might decrease the readability of the rest of the paper. In order to prevent this situation, for major definitions and properties, we give equivalent inductive characterizations on finite trees. They will not be used in the subsequent development and we shall omit the proofs of their equivalence to the original definitions restricted to finite trees. They can be proved by usual structural induction.

4.4.1 Set of Description Types

We begin with types. Using regular trees (definition 2.10), the set of types for complex database objects is defined as follows:

Definition 4.11 (Set of Description Types) *The set of description type constructors is the set $\boldsymbol{F}_\tau = \{Record, Variant, Set\} \cup \mathcal{B}$. A description type is a regular tree $\sigma \in \boldsymbol{R}(\boldsymbol{F}_\tau)$ satisfying the following conditions:*

1. *if $\sigma(a) = Set$ then $\{l \in \mathcal{L} | a \cdot l \in dom(\sigma)\} = \{elm_1\}$,*
2. *if $a \cdot elm_i \in dom(\sigma)$ for some $a \in \mathcal{L}^*$ then $\sigma(a) = Set$,*
3. *if $\sigma(a) \in \mathcal{B}$, then the set $\{l \in \mathcal{L} | a \cdot l \in dom(\sigma)\}$ is empty.*

A description type σ is finite if it is finite as a tree. The set of all description types and the set of all finite description types are denoted by $\boldsymbol{Dtype}^\infty$ and $\boldsymbol{Dtype}$ respectively. *Record*, *Variant* and *Set* represent the record, the variant and the set type constructors respectively. Condition 1 restricts set types to be homogeneous sets. Let $\sigma_1, \ldots, \sigma_n \in \boldsymbol{Dtype}^\infty$. We use the following notations:

$$\begin{array}{rcl} [l_1 : \sigma_1, \ldots, l_n : \sigma_n] & \text{for} & Record(l_1 = \sigma_1, \ldots, l_n = \sigma_n), \\ \langle l_1 : \sigma_1, \ldots, l_n : \sigma_n \rangle & \text{for} & Variant(l_1 = \sigma_1, \ldots, l_n = \sigma_n), \\ \{\!\!\{\sigma\}\!\!\} & \text{for} & Set(elm_1 = \sigma) \end{array}$$

unit	$= []$
point	$= [X\text{-}cord : int,\ Y\text{-}cord : int]$
intlist	$= (rec\ v.\,\langle Cons : [Head : int,\ Tail : v],\ Nil : unit\rangle)$
object	$= [Name : string,\ Age : int]$
person	$= (rec\ p.\,[Name : string,\ Age : int,\ Parents : \{\!\{p\}\!\}])$
employee	$= (rec\ e.\,[Name : string,\ Age : int,\ Parents : \{\!\{person\}\!\},$ $Salary : int,\ Boss : e])$
student	$= [Name : string,\ Age : int,\ Parents : \{\!\{person\}\!\},\ Courses : \{\!\{string\}\!\}]$
working-student	$= [Name : string,\ Age : int,\ Parents : \{\!\{person\}\!\},$ $Courses : \{\!\{string\}\!\},\ Salary : int,\ Boss : employee]$
flights	$= \{\!\{[Flight : [F\text{-}id : int,\ Date : string],\ Plane : string]\}\!\}$
flown-by	$= \{\!\{[Plane : string,\ Pilots : \{\!\{[Name : string,\ Emp\text{-}id : int]\}\!\}]\}\!\}$
schedule-data	$= \{\!\{[Flight : [F\text{-}id : int,\ Date : string],\ Plane : string,$ $Pilots : \{\!\{[Name : string,\ Emp\text{-}id : int]\}\!\}]\}\!\}$

Figure 4.3: Examples of Description Types

Similar shorthands are adopted in term representations of regular trees. Figure 4.3 shows examples of description types in term representation. In this example, as well as in all other examples we will show later, identifiers such as *unit* are used purely as syntactic shorthands to avoid repetitions and have no significance themselves. As seen in these examples, infinite trees correspond to recursively defined types.

The set of finite description types $\boldsymbol{Dtype}$ coincides with the following inductively defined set $\boldsymbol{Dtype}^o$:

1. $b \in \boldsymbol{Dtype}^o$ for any $b \in \mathcal{B}$,
2. $[l_1 : \sigma_1, \ldots, l_n : \sigma_n] \in \boldsymbol{Dtype}^o$ if $\sigma_1, \ldots, \sigma_n \in \boldsymbol{Dtype}^o$ and $l_1, \ldots, l_n \in \mathcal{L}\,(n \geq 0)$,
3. $\langle l_1 : \sigma_1, \ldots, l_n : \sigma_n\rangle \in \boldsymbol{Dtype}^o$ if $\sigma_1, \ldots, \sigma_n \in \boldsymbol{Dtype}^o$ and $l_1, \ldots, l_n \in \mathcal{L}\,(n \geq 0)$,
4. $\{\!\{\sigma\}\!\} \in \boldsymbol{Dtype}^o$ if $\sigma \in \boldsymbol{Dtype}^o$

where l_i is not a label of the form elm_i.

On the set $\boldsymbol{Dtype}^\infty$, we define the following ordering to capture the ordering of the containment of the structures:

Definition 4.12 (Information Ordering on $\boldsymbol{Dtype}^\infty$) *Let $\sigma_1, \sigma_2 \in \boldsymbol{Dtype}^\infty$. The information ordering $\leqslant$ on $\boldsymbol{Dtype}^\infty$ is the relation defined as: $\sigma_1 \leqslant \sigma_2$ iff $dom(\sigma_1) \subseteq dom(\sigma_2)$ and for any*

$$\begin{array}{rcl}
\textit{unit} & \leqslant & \textit{point} \\
\textit{unit} & \leqslant & \textit{object} \\
\textit{object} & \leqslant & \textit{person} \\
\textit{person} & \leqslant & \textit{employee} \\
\textit{person} & \leqslant & \textit{student} \\
\textit{employee} & \leqslant & \textit{working-student} \\
\textit{student} & \leqslant & \textit{working-student} \\
\textit{flights} & \leqslant & \textit{schedule-data} \\
\textit{flown-by} & \leqslant & \textit{schedule-data}
\end{array}$$

Figure 4.4: Examples of Ordering on Description Types

$a \in dom(\sigma_1)$, $\sigma_1(a) = \sigma_2(a)$ and if $\sigma_1(a) =$ Variant then $\{l \in \mathcal{L} | a \cdot l \in dom(\sigma_1)\} = \{l \in \mathcal{L} | a \cdot l \in dom(\sigma_2)\}$.

This ordering can be regarded as a special case of the subsumption ordering on Aït-Kaci's ψ-terms [6]. The condition on variant nodes means that in order for two variant types to be ordered, they must have the same set of variants. The intuition behind this condition is that if a variant type σ_1 has a component $l : \sigma_1'$ and σ_2 has no l-component, then for a value v of the type σ_1 corresponding to the component $l : \sigma_1'$ there is no value v' of the type σ_2 that is related in structure to v and therefore σ_1 and σ_2 are not related. Figure 4.4 shows examples of the information ordering on $\boldsymbol{Dtype}^{\infty}$ among the description types defined in figure 4.3.

The ordering $\leqslant$, when restricted to the set of finite description types $\boldsymbol{Dtype}$, coincides with the following inductively defined relation $\leqslant^o$:

$$\begin{array}{rcl}
b & \leqslant^o & b \text{ for all } b \in \mathcal{B} \\
[l_1 : \sigma_1, \ldots, l_n : \sigma_n] & \leqslant^o & [l_1 : \sigma_1', \ldots, l_n : \sigma_n', \ldots] \text{ if } \sigma_i \leqslant^o \sigma_i' \ (i \leq n) \\
\{\!\{\sigma\}\!\} & \leqslant^o & \{\!\{\sigma'\}\!\} \text{ if } \sigma \leqslant^o \sigma' \\
\langle l_1 : \sigma_1, \ldots, l_n : \sigma_n \rangle & \leqslant^o & \langle l_1 : \sigma_1', \ldots, l_n : \sigma_n' \rangle \text{ if } \sigma_i \leqslant^o \sigma_i' \ (i \leq n)
\end{array}$$

From the inductive characterization of $\leqslant$, it is easy to check that $(\boldsymbol{Dtype}, \leqslant)$ is a poset with pairwise bounded join property, $\leqslant$ is decidable and least upper bounds (if they exist) are effectively computable. The following two propositions show that these properties still hold for general description types. Their proof can be constructed from the proof of the similar properties shown in [6]. Since the proofs involve general techniques we will repeatedly use in proofs of various properties of the type system, we include their detailed proofs.

Proposition 4.7 *$(\boldsymbol{Dtype}^\infty, \leqslant)$ is a poset with the pairwise bounded join property.*

Proof For any $\sigma \in \boldsymbol{Dtype}^\infty$, clearly $\sigma \leqslant \sigma$. Let $\sigma_1, \sigma_2, \sigma_3$ be any elements in $\boldsymbol{Dtype}^\infty$. Suppose $\sigma_1 \leqslant \sigma_2$ and $\sigma_2 \leqslant \sigma_1$. Then $dom(\sigma_1) = dom(\sigma_2)$ and for all $a \in dom(\sigma_1)$, $\sigma_1(a) = \sigma_2(a)$. Therefore $\sigma_1 = \sigma_2$. Suppose $\sigma_1 \leqslant \sigma_2$ and $\sigma_2 \leqslant \sigma_3$. Then $dom(\sigma_1) \subseteq dom(\sigma_2) \subseteq dom(\sigma_3)$. Let a be any element in $dom(\sigma_1)$. $\sigma_1(a) = \sigma_2(a) = \sigma_3(a)$. Suppose $\sigma_1(a) = Variant$. Then $\{l \in \mathcal{L} | a \cdot l \in dom(\sigma_1)\} = \{l \in \mathcal{L} | a \cdot l \in dom(\sigma_2)\} = \{l \in \mathcal{L} | a \cdot l \in dom(\sigma_1)\}$. Therefore $\sigma_1 \leqslant \sigma_3$ and hence $\leqslant$ is a partial order on $\boldsymbol{Dtype}^\infty$.

For the pairwise bounded join property, suppose σ_1, σ_2 have an upper bound. Let σ' be the tree defined as $dom(\sigma') = dom(\sigma_1) \cup dom(\sigma_2)$ and for $a \in dom(\sigma)$,

$$\sigma'(a) = \begin{cases} \sigma_1(a) & \text{if } a \in dom(\sigma_1) \\ \sigma_2(a) & \text{otherwise.} \end{cases}$$

Then for any $a \in dom(\sigma_1)$, $\sigma_1(a) = \sigma'(a)$ and if $\sigma_1(a) = Variant$ then $\{l \in \mathcal{L} | a \cdot l \in dom(\sigma_1)\} = \{l \in \mathcal{L} | a \cdot l \in dom(\sigma')\}$. Therefore $\sigma_1 \leqslant \sigma'$. For any $a \in dom(\sigma_2) \setminus dom(\sigma_1)$, by the definition of σ', $\sigma_2(a) = \sigma'(a)$ and $\{l \in \mathcal{L} | a \cdot l \in dom(\sigma_2)\} = \{l \in \mathcal{L} | a \cdot l \in dom(\sigma')\}$. Suppose $a \in dom(\sigma_1) \cap dom(\sigma_2)$. Since σ_1, σ_2 have an upper bound, $\sigma_1(a) = \sigma_2(a) = \sigma'(a)$ and $\{l \in \mathcal{L} | a \cdot l \in dom(\sigma_2)\} = \{l \in \mathcal{L} | a \cdot l \in dom(\sigma_1)\} = \{l \in \mathcal{L} | a \cdot l \in dom(\sigma')\}$. Therefore $\sigma_2 \leqslant \sigma'$. Since $dom(\sigma_1) \subseteq dom(\sigma)$ and $dom(\sigma_2) \subseteq dom(\sigma)$, $dom(\sigma') \subseteq dom(\sigma)$. Let $a \in dom(\sigma')$. If $a \in dom(\sigma_1)$ then $\sigma'(a) = \sigma_1(a) = \sigma(a)$ otherwise $a \in dom(\sigma_2)$ then $\sigma'(a) = \sigma_2(a) = \sigma(a)$. Suppose $\sigma'(a) = Variant$. Then we have $\{l \in \mathcal{L} | a \cdot l \in dom(\sigma')\} = \{l \in \mathcal{L} | a \cdot l \in dom(\sigma_i)\} = \{l \in \mathcal{L} | a \cdot l \in dom(\sigma)\}$ where $i = 1$ if $a \in dom(\sigma_1)$ otherwise $i = 2$. Therefore $\sigma' \leqslant \sigma$ and hence σ' is the least upper bound of σ_1, σ_2. ∎

The proof of the following proposition defines an algorithm to compute the least upper bound of consistent description types.

Proposition 4.8 *The ordering $\leqslant$ on $\boldsymbol{Dtype}^\infty$ is decidable and for any description types σ_1, σ_2, it is decidable whether σ_1, σ_2 have an upper bound or not and if they have an upper bound then their least upper bound is effectively computable.*

Proof Let $M_{\sigma_1} = (Q_1, s_1, \boldsymbol{F}_\tau, \delta_1, o_1)$, $M_{\sigma_2} = (Q_2, s_2, \boldsymbol{F}_\tau, \delta_2, o_2)$ be Moore machines representing σ_1 and σ_2 respectively. Let $M = (Q, s, \boldsymbol{F}, \delta, o) = M_{\sigma_1} \times M_{\sigma_2}$, the product machine (definition 2.12) of M_{σ_1} and M_{σ_2}.

We show that $\sigma_1 \leqslant \sigma_2$ iff M has the following properties: for any reachable state q in M,

1. q is either $q = (q_1, q_2)$ for some $q_1 \in Q_1, q_2 \in Q_2$ or $q = (\$, q_2)$ for some $q_2 \in Q_2$,

2. if $q = (q_1, q_2), q_1 \in Q_1, q_2 \in Q_2$ then $o(q) = (f, f)$ for some $f \in \boldsymbol{F}_\tau$,

3. if $q = (q_1, q_2), q_1 \in Q_1, q_2 \in Q_2$, $o(q) = (\mathit{Variant}, \mathit{Variant})$ and $\delta(q, l) = q'$ for some l then $q' = (q_1', q_2')$ for some $q_1' \in Q_1, q_2' \in Q_2$.

By lemma 2.5, the condition 1 is equivalent to $dom(\sigma_1) \subseteq dom(\sigma_2)$ and the condition 2, 3 are respectively equivalent to the two conditions of the definition of the ordering $\leqslant$.

Next we show that if σ_1, σ_2 have an upper bound then M has the following properties: for any reachable state q in M,

1. $o(q)$ is one of the forms: $(f, f), (f, \$), (\$, f)$ for some $f \in \boldsymbol{F}_\tau$,

2. if $q = (q_1, q_2), q_1 \in Q_1, q_2 \in Q_2$, $o(q) = (\mathit{Variant}, \mathit{Variant})$ and $\delta(q, l) = q'$ then $q' = (q_1', q_2')$ for some $q_1' \in Q_1, q_2' \in Q_2$.

Suppose σ_1, σ_2 has an upper bound. Then for any $a \in dom(\sigma_1) \cap dom(\sigma_2)$, $\sigma_1(a) = \sigma_2(a) = f$ for some $f \in \boldsymbol{F}_\tau$. By the property of product machine, this implies $\delta^*(s, pair^*(a, a)) = (f, f)$, which establishes the property 1. Suppose $\sigma_1(a) = \sigma_2(a) = \mathit{Variant}$ then by the definition of $\leqslant$, for all $l \in \mathcal{L}$, $a \cdot l \in dom(\sigma_1)$ iff $a \cdot l \in dom(\sigma_2)$. By the property of product machines, this implies the second condition.

Finally we show that if M satisfies the above condition then we can construct a Moore machine representing the least upper bound of σ_1, σ_2. Suppose M satisfies the above two conditions. Define $M_{\sigma_1 \sqcup \sigma_2}$ as the Moore machine $(Q, s, \boldsymbol{F}, \delta', o')$ where

1. $Q, s, \boldsymbol{F}$ are same as those of M,

2. δ' is defined as $\delta'(q, l) = q'$ iff $\delta(q, (l, l)) = q'$ or $\delta(q, (l, \$)) = q'$ or $\delta(q, (\$, l)) = q'$,

3. o' is defined as $o'(q) = f$ if $o(q)$ is one of the forms $(f, f), (f, \$), (\$, f)$.

Since by the definition of product machine, at most one of $\delta(q, (l, l)), \delta(q, (l, \$))$ or $\delta(q, (\$, l))$ is defined, δ' is well defined. By lemma 2.5 and the definition of $M_{\sigma_1 \sqcup \sigma_2}$, $dom(M_{\sigma_1 \sqcup \sigma_2}(s)) = dom(M_1(s_1)) \cup dom(M_2(s_2))$. By the definition of M and $M_{\sigma_1 \sqcup \sigma_2}$, for all $a \in dom(M_1(s_1))$, $\delta_1^*(s_1, a) = q, o_1(q) = f$ iff $\delta'^*(s, a) = (q, x), o'((q, x)) = f$ for some x, and if $o_1(q) = \mathit{Variant}$ then $\delta_1(q, l)$ is defined iff $\delta((q, x), l)$ is defined. Therefore $M_{\sigma_1 \sqcup \sigma_2}(s)$ satisfies the other two conditions of the definition of $\leqslant$ and hence $M_1(s_1) \leqslant M'(s)$. Similarly $M_2(s_2) \leqslant M'(s)$. Let σ be any upper bound of σ_1, σ_2. Since for $a \in dom(M'(s))$ either $a \in M_1(s_1)$ and $M'(s)(a) = M_1(s_1)(a)$ or $a \in dom(M_2(s_2))$ and $M'(s)(a) = M_2(s_2)(a)$, $M'(s) \leqslant \sigma$ follows from $M_1(s_1) \leqslant \sigma$ and $M_2(s_2) \leqslant \sigma$.

Since the product machine and the machine $M_{\sigma_1 \sqcup \sigma_2}$ are effectively constructed, we have proved the proposition. ▮

Combining proposition 4.7 and 4.8, we have:

Theorem 4.4 *($\boldsymbol{Dtype}^{\infty}, \leqslant$) is a database type system.* ▮

The following are examples of least upper bounds of description types defined in figure 4.3:

$$\begin{aligned} \mathit{employee} \sqcup \mathit{student} &= \mathit{working\text{-}student}, \\ \mathit{flights} \sqcup \mathit{flown\text{-}by} &= \mathit{schedule\text{-}data}. \end{aligned}$$

From examples shown in figure 4.4 and the above examples, we can see that $\leqslant$ is a generalization of the information ordering on types in the relational models to complex structures including recursive structures represented by infinite trees.

4.4.2 Universe of Descriptions

In order to construct a model of ($\boldsymbol{Dtype}^{\infty}, \leqslant$), we first define a set of possible descriptions.

Definition 4.13 (Universe of Descriptions) *The set of description constructors is the set $\boldsymbol{F}_d = \{Record, Inj, Set\} \cup (\bigcup_{b \in \mathcal{B}} D_b) \cup \{null_b | b \in \mathcal{B}\}$. A description is a regular tree $d \in \boldsymbol{R}(\boldsymbol{F}_d)$ satisfying the following conditions: for all $a \in dom(d)$,*

1. *if $d(a) = Set$ then $\{l \in \mathcal{L} | a \cdot l \in dom(d)\} = \{elm_1, \ldots, elm_n\}$ for some $n \geq 0$,*
2. *if $a \cdot elm_i \in dom(d)$ for some $a \in \mathcal{L}^*$ then $d(a) = Set$,*
3. *if $d(a) = Inj$ then the set $\{l \in \mathcal{L} | a \cdot l \in dom(d)\}$ is either a singleton set or the empty set,*
4. *if $d(a) \in D_b$ or $d(a) = null_b$ then the set $\{l \in \mathcal{L} | a \cdot l \in dom(d)\}$ is the empty set.*

A description d is finite if it is finite as a tree. The set of all descriptions and the set of all finite descriptions are denoted by $\boldsymbol{Dobj}^{\infty}$ and $\boldsymbol{Dobj}$ respectively. *Inj* is a variant constructor (injection to a variant type). *Inj* node with no outgoing edge represents null values of variant types.

Let $d_1, \ldots, d_n \in \boldsymbol{Dobj}^{\infty}$. We use the following notations:

$$\begin{aligned} [l_1 = d_1, \ldots, l_n = d_n] \quad &\text{for} \quad Record(l_1 = d_1, \ldots, l_n = d_n), \\ \{\!\{d_1, \ldots, d_n\}\!\} \quad &\text{for} \quad Set(elm_1 = d_1, \ldots, elm_n = d_n). \end{aligned}$$

$$
\begin{array}{ll}
\mathit{Unity} & = [\,] \\[4pt]
\mathit{Point23} & = [\mathit{X\text{-}cord} = 2,\ \mathit{Y\text{-}cord} = 3] \\[4pt]
\mathit{Onelist} & = \mathit{Inj}(\mathit{Cons} = [\mathit{Head} = 1,\ \mathit{Tail} = \mathit{Inj}(\mathit{Nil} = \mathit{Unity})]) \\[4pt]
\mathit{Null\text{-}person} & = (\mathit{rec}\ p.\ [\mathit{Name} = \mathit{null}_{\mathit{string}}, \mathit{Age} = \mathit{null}_{\mathit{int}}, \mathit{Parents} = \{\!\!\{p\}\!\!\}]) \\[4pt]
\mathit{Null\text{-}employee} & = (\mathit{rec}\ e.\ [\mathit{Name} = \mathit{null}_{\mathit{string}}, \mathit{Age} = \mathit{null}_{\mathit{int}}, \mathit{Parents} = \{\!\!\{\mathit{Null\text{-}person}\}\!\!\}, \\
 & \qquad\qquad \mathit{Salary} = \mathit{null}_{\mathit{int}}, \mathit{Boss} = e]) \\[4pt]
\mathit{John} & = [\mathit{Name} = \text{"}\mathit{John\ Smith}\text{"}, \mathit{Age} = 34, \mathit{Parent} = \{\!\!\{\mathit{Null\text{-}person}\}\!\!\}, \\
 & \quad \mathit{Salary} = 23000,\ \mathit{Boss} = \mathit{Null\text{-}employee}] \\[4pt]
\mathit{Mary1} & = [\mathit{Name} = \text{"}\mathit{Mary\ Blake}\text{"}, \mathit{Age} = 21, \mathit{Parent} = \{\!\!\{\mathit{Null\text{-}person}\}\!\!\}, \\
 & \quad \mathit{Courses} = \{\!\!\{\text{"}\mathit{math120}\text{"}, \text{"}\mathit{phil340}\text{"}, \text{"}\mathit{logic110}\text{"}\}\!\!\}] \\[4pt]
\mathit{Mary2} & = [\mathit{Name} = \text{"}\mathit{Mary\ Blake}\text{"}, \mathit{Age} = 21, \mathit{Parent} = \{\!\!\{\mathit{Null\text{-}person}\}\!\!\}, \\
 & \quad \mathit{Salary} = 9000,\ \mathit{Boss} = \mathit{John}] \\[4pt]
\mathit{Mary3} & = [\mathit{Name} = \text{"}\mathit{Mary\ Blake}\text{"}, \mathit{Age} = 21, \mathit{Parent} = \{\!\!\{\mathit{Null\text{-}person}\}\!\!\}, \\
 & \quad \mathit{Courses} = \{\!\!\{\text{"}\mathit{math120}\text{"}, \text{"}\mathit{phil340}\text{"}, \text{"}\mathit{logic110}\text{"}\}\!\!\}, \\
 & \quad \mathit{Salary} = 9000,\ \mathit{Boss} = \mathit{John}] \\[4pt]
\mathit{Flights} & = \{\!\!\{\ [\mathit{Flight} = [\mathit{F\text{-}id} = 001, \mathit{Date} = \text{"}\mathit{8\ Aug}\text{"}],\quad \mathit{Plane} = \text{"}\mathit{Concord}\text{"}], \\
 & \qquad [\mathit{Flight} = [\mathit{F\text{-}id} = 83, \mathit{Date} = \text{"}\mathit{9\ Aug}\text{"}],\quad \mathit{Plane} = \text{"}\mathit{707}\text{"}], \\
 & \qquad [\mathit{Flight} = [\mathit{F\text{-}id} = 116, \mathit{Date} = \text{"}\mathit{10\ Aug}\text{"}],\quad \mathit{Plane} = \text{"}\mathit{747}\text{"}]\}\!\!\} \\[4pt]
\mathit{Flown\text{-}by} & = \{\!\!\{\ [\mathit{Plane} = \text{"}\mathit{Concord}\text{"},\ \mathit{Pilots} = \{\!\!\{\ [\mathit{Name} = \text{"}\mathit{Jones}\text{"}, \mathit{Emp\text{-}id} = 5566]\}\!\!\}], \\
 & \qquad [\mathit{Plane} = \text{"}\mathit{707}\text{"},\quad \mathit{Pilots} = \{\!\!\{\ [\mathit{Name} = \text{"}\mathit{Clark}\text{"}, \mathit{Emp\text{-}id} = 1122], \\
 & \qquad\qquad\qquad\qquad\qquad [\mathit{Name} = \text{"}\mathit{Copely}\text{"}, \mathit{Emp\text{-}id} = 2233], \\
 & \qquad\qquad\qquad\qquad\qquad [\mathit{Name} = \text{"}\mathit{Chin}\text{"}, \mathit{Emp\text{-}id} = 3344]\}\!\!\}], \\
 & \qquad [\mathit{Plane} = \text{"}\mathit{747}\text{"},\quad \mathit{Pilots} = \{\!\!\{\ [\mathit{Name} = \text{"}\mathit{Clark}\text{"}, \mathit{Emp\text{-}id} = 1122], \\
 & \qquad\qquad\qquad\qquad\qquad [\mathit{Name} = \text{"}\mathit{Jones}\text{"}, \mathit{Emp\text{-}id} = 5566]\}\!\!\}]\}\!\!\} \\[4pt]
\mathit{Schedule\text{-}data} & = \{\!\!\{\ [\ \mathit{Plane} = \text{"}\mathit{Concord}\text{"},\ \mathit{Pilots} = \{\!\!\{[\mathit{Name} = \text{"}\mathit{Jones}\text{"}, \mathit{Emp\text{-}id} = 5566]\}\!\!\}, \\
 & \qquad\quad \mathit{Flight} = [\mathit{F\text{-}id} = 001, \mathit{Date} = \text{"}\mathit{8\ Aug}\text{"}]], \\
 & \qquad [\ \mathit{Plane} = \text{"}\mathit{707}\text{"},\quad \mathit{Pilots} = \{\!\!\{[\mathit{Name} = \text{"}\mathit{Clark}\text{"}, \mathit{Emp\text{-}id} = 1122], \\
 & \qquad\qquad\qquad\qquad\qquad [\mathit{Name} = \text{"}\mathit{Copely}\text{"}, \mathit{Emp\text{-}id} = 2233], \\
 & \qquad\qquad\qquad\qquad\qquad [\mathit{Name} = \text{"}\mathit{Chin}\text{"}, \mathit{Emp\text{-}id} = 3344]\}\!\!\}, \\
 & \qquad\quad \mathit{Flight} = [\mathit{F\text{-}id} = 83, \mathit{Date} = \text{"}\mathit{9\ Aug}\text{"}]], \\
 & \qquad [\ \mathit{Plane} = \text{"}\mathit{747}\text{"},\quad \mathit{Pilots} = \{\!\!\{[\mathit{Name} = \text{"}\mathit{Clark}\text{"}, \mathit{Emp\text{-}id} = 1122], \\
 & \qquad\qquad\qquad\qquad\qquad [\mathit{Name} = \text{"}\mathit{Jones}\text{"}, \mathit{Emp\text{-}id} = 5566]\}\!\!\}, \\
 & \qquad\quad \mathit{Flight} = [\mathit{F\text{-}id} = 116, \mathit{Date} = \text{"}\mathit{10\ Aug}\text{"}]]\}\!\!\}
\end{array}
$$

Figure 4.5: Examples of Descriptions

Figure 4.5 shows examples of descriptions.

The set of finite descriptions $\boldsymbol{Dobj}$ coincides with the following inductively defined set $\boldsymbol{Dobj}^o$:

1. $c \in \boldsymbol{Dobj}^o$ for any $c \in B_b, b \in \mathcal{B}$,
2. $null_b \in \boldsymbol{Dobj}^o$ for any $b \in \mathcal{B}$,
3. $[l_1 = d_1, \ldots, l_n = d_n] \in \boldsymbol{Dobj}^o$ if $d_1, \ldots, d_n \in \boldsymbol{Dobj}^o$ and $l_1, \ldots, l_n \in \mathcal{L}\,(n \geq 0)$,
4. $Inj \in \boldsymbol{Dobj}^o$,
5. $Inj(l = d) \in \boldsymbol{Dobj}^o$ if $d \in \boldsymbol{Dobj}^o$ and $l \in \mathcal{L}$,
6. $\{\!\{d_1, \ldots, d_n\}\!\} \in \boldsymbol{Dobj}^o$ if $d_1, \ldots, d_n \in \boldsymbol{Dobj}^o\,(n \geq 0)$.

where l_i is not a label of the form elm_i.

4.4.3 Typing Relation

Description types represent structures of descriptions. A description d *has* a description type σ if d has the structure represented by σ. This relationship is formalized by the *typing relation*:

Definition 4.14 (Typing Relation) *Let $\approx$ be the equivalence relation on $\mathcal{L}$ defined as $l_1 \approx l_2$ iff $l_1 = l_2$ or $l_1 = elm_i, l_2 = elm_j$ for some i, j. Define the consistency relation $:^b$ between $\boldsymbol{F}_d$ and $\boldsymbol{F}_\tau$ as follows: $f :^b g$ iff one of the following holds:*

1. $f = g$,

2. $f = Inj$ and $g = Variant$,

3. $f \in D_g$ and $g \in \mathcal{B}$,

4. $f = null_g$.

The typing relation $d : \sigma$ between $\boldsymbol{Dobj}^\infty$ and $\boldsymbol{Dtype}^\infty$ is defined as: $d : \sigma$ iff for all $a \in dom(d)$,

1. there is some a' such that $a \overset{*}{\approx} a'$, $d(a) :^b \sigma(a')$,

2. if $d(a) = Record$ then $\{l \in \mathcal{L} | a \cdot l \in dom(d)\} = \{l \in \mathcal{L} | a' \cdot l \in dom(\sigma)\}$.

Unity	:	***unit***
Point23	:	***point***
Onelist	:	***intlist***
Null-person	:	***person***
Null-employee	:	***employee***
John	:	***employee***
Mary1	:	***student***
Mary2	:	***employee***
Mary3	:	***working-student***
Flights	:	***flights***
Flown-by	:	***flown-by***
Schedule-data	:	***schedule-data***

Figure 4.6: Examples of Typing Relation

The equivalence relation $\approx$ "ignores" the difference due to the positions $elm_1, \ldots, elm_n$ of occurrences of subtrees in the set constructor $Set(elm_1 = d_1, \ldots, elm_n = d_n)$. $\overset{\bullet}{\approx}$ is the extension of $\approx$ on $\mathcal{L}$ defined in 2.2.3. Figure 4.6 shows examples of typing relations between descriptions defined in figure 4.5 and description types defined in figure 4.3.

When restricted to the set of finite descriptions $\boldsymbol{Dobj}$, the above typing relation coincides with the following relation $:^o$ on $\boldsymbol{Dobj} \times \boldsymbol{Dtype}^\infty$ defined by induction on $\boldsymbol{Dobj}$:

1. $c :^o b$ for all $c \in B_b$,
2. $\boldsymbol{null_b} :^o b$,
3. $[l_1 = d_1, \ldots, l_n = d_n] :^o [l_1 : \sigma_1, \ldots, l_n : \sigma_n]$ if $d_1 :^o \sigma_1, \ldots, d_n :^o \sigma_n$,
4. $Inj :^o \sigma$ for any variant type σ,
5. $Inj(l = d) :^o \langle \ldots, l : \sigma, \ldots \rangle$ if $d :^o \sigma$,
6. $\{\!\{d_1, \ldots, d_n\}\!\} :^o \{\!\{\sigma\}\!\}$ if $d_1 :^o \sigma, \ldots, d_n :^o \sigma$.

Note however that $d \in \boldsymbol{Dobj}$ and $d :^o \sigma$ does not implies that $\sigma \in \boldsymbol{Dtype}$ because of variant types, i.e. rules 4 and 5 in the above definition.

From the above inductive characterization of the typing relation, it is easy to check that for any finite description d and any description type σ it is decidable whether $d : \sigma$ or not. This property is essential to develop a type-checking algorithm. Fortunately, this property still holds for general descriptions:

Proposition 4.9 *For any $d \in \boldsymbol{Dobj}^\infty, \sigma \in \boldsymbol{Dtype}^\infty$, the property $d : \sigma$ is decidable.*

Proof Let $M_d = (Q_d, s_d, \boldsymbol{F}_d, \delta_d, o_d)$ and $M_\sigma = (Q_\sigma, s_\sigma, \boldsymbol{F}_\tau, \delta_\sigma, o_\sigma)$ be Moore machines representing d and σ respectively. Let $M = (Q, s, \boldsymbol{F}, \delta, o)$ be the product machine $(M_d \times M_\sigma)/\approx$ where $\approx$ is the equivalence relation on $\mathcal{L}$ defined in definition 4.14. We show that $d : \sigma$ iff M satisfies the following conditions: for any reachable state q,

1. if $q = (q_1, x), q_1 \in Q_1$ then $x \in Q_2$ and $o(q) = (f, g)$ such that $f :^b g$,
2. if $q = (q_1, q_2), q_1 \in Q_1, q_2 \in Q_2$, $o(q) = (\mathit{Record}, \mathit{Record})$ and $\delta(q, l) = q'$ then $l = (l', l'), l' \neq \$$.

By lemma 2.5, M satisfies the condition 1 iff for any $a \in \mathit{dom}(M_1(s_1))$, there is some a' such that $a \overset{*}{\approx} a'$, $\delta_d^*(s_d, a) = q_1$, $\delta_\sigma^*(s_\sigma, a') = q_2$, and $o_d(q_1) :^b o_\sigma(q_2)$. Since M_d, M_σ represent d, σ respectively, this condition is equivalent to the condition 1 of the definition of the typing relation. The equivalences of the condition 2 of the propositions and the condition 2 of the definition of the typing relation are immediate consequences of their definitions.

Since M is effectively constructed and the above property is clearly decidable, the proposition is proved. ∎

4.4.4 Description Domains

For each description type, the typing relation defines the set of descriptions of that type. By defining a proper ordering, we turn this set into a description domain.

Courcelle described [32] the notion of a *coherent* and *simplifiable* relation on $\boldsymbol{Subtrees}(d_1) \times \boldsymbol{Subtrees}(d_2)$ as a relation $\sim$ satisfying the condition that if

$$f(l_1 = d_1, \ldots, l_n = d_n) \sim g(l_1 = d'_1, \ldots, l_n = d'_n)$$

then $f = g$ and $d_i \sim d'_i$ $(1 \leq i \leq n)$. By generalizing this and combining it with Smyth powerdomain preorder, we can generalize the information ordering on flat descriptions to $\boldsymbol{Dobj}^\infty$:

Definition 4.15 (Information Preorder on $\boldsymbol{Dobj}^\infty$) *The information ordering on the set $\boldsymbol{F}_d$ of description constructors is the following partial ordering $\sqsubseteq^b$:*

$$f \sqsubseteq^b g \textit{ iff } f = g \textit{ or } f = null_b \textit{ and } g \in D_b$$

The information preorder $\preceq$ on $\boldsymbol{Dobj}^\infty$ is the relation defined as: $d_1 \preceq d_2$ iff there is a relation $\sim$, called a substructure relation, on $\boldsymbol{Subtrees}(d_1) \times \boldsymbol{Subtrees}(d_2)$ satisfying the following properties:

1. $d_1 \sim d_2$,

2. *if* $d \sim d'$ *then* $d(\epsilon) \sqsubseteq^b d'(\epsilon)$,

3. *if* $d \sim d'$, $d(\epsilon) \in \{Record, Inj\}$ *and* $l \in dom(d)$ *then* $l \in dom(d')$ *and* $d/l \sim d'/l$,

4. *if* $d \sim d'$, $d(\epsilon) = Set$ *then for all* $l \in \{l \in \mathcal{L} | l \in dom(d')\}$ *there is some* $l' \in \{l \in \mathcal{L} | l \in dom(d)\}$ *such that* $d/l' \sim d'/l$.

The relation $\preceq$, when restricted to the set of finite descriptions $\boldsymbol{Dobj}$, coincides with the following inductively defined relation $\preceq^o$:

$$\begin{array}{rcl}
c & \preceq^o & c \text{ for all } c \in B_b, \\
null_b & \preceq^o & c \text{ for all } c \in B_b, \\
null_b & \preceq^o & null_b, \\
[l_1 = d_1, \ldots, l_n = d_n] & \preceq^o & [l_1 = d'_1, \ldots, l_n = d'_n, \ldots] \text{ if } d_i \preceq^o d'_i \ (1 \leq i \leq n), \\
Inj & \preceq^o & Inj, \\
Inj & \preceq^o & Inj(l = d) \text{ for all } d, \\
Inj(l = d) & \preceq^o & Inj(l = d') \text{ if } d \preceq^o d', \\
\{\!\{d_1, \ldots, d_n\}\!\} & \preceq^o & \{\!\{d'_1, \ldots, d'_m\}\!\} \text{ if } \forall d' \in \{d'_1, \ldots, d'_m\}. \exists d \in \{d_1, \ldots, d_n\}. d \preceq^o d'
\end{array}$$

On a substructure relation $\sim$, the following property hold:

Lemma 4.2 *Let* $\sim$ *be a substructure relation on* $\boldsymbol{Subtrees}(d_1) \times \boldsymbol{Subtrees}(d_2)$.
For $d'_1 \in \boldsymbol{Subtrees}(d_1), d'_2 \in \boldsymbol{Subtrees}(d_2)$, *if* $d'_1 \sim d'_2$ *then* $d'_1 \preceq d'_2$.

Proof Immediate consequence of the fact that the restriction of a substructure relation $\sim$ to $\boldsymbol{Subtrees}(d'_1) \times \boldsymbol{Subtrees}(d'_2)$ is also a substructure relation. ▌

We next show that $\preceq$ is a preorder having the desired properties. Rounds' recent work [97] also independently shows a similar result for a certain class of labeled directed graphs.

Proposition 4.10 *The relation* $\preceq$ *is a preorder on* $\boldsymbol{Dobj}^\infty$ *with the pairwise bounded join property.*

Proof The strategy of the following rather long proof is the combination of the technique suggested in [6] to construct a least upper bound of two regular trees by tracing the moves of two Moore machines representing them in "parallel" and the property of Smyth powerdomain preorder shown in [104] that if s_1 and s_2 are finite subset of a a poset then $\{\!\!\{ d_1 \sqcup d_2 | d_1 \in s_1, d_2 \in s_2$ and $d_1 \sqcup d_2$ exists$\}\!\!\}$ is a least upper bound of s_1 and s_2 under the Smyth preorder.

For any description d, the identity relation on $\boldsymbol{Subtrees}(d)$ is a substructure relation and $d \preceq d$. Suppose $d_1 \preceq d_2$ and $d_2 \preceq d_3$. Let $\sim_1$ and $\sim_2$ be substructure relations on $\boldsymbol{Subtrees}(d_1) \times \boldsymbol{Subtrees}(d_2)$ and $\boldsymbol{Subtrees}(d_2) \times \boldsymbol{Subtrees}(d_3)$ respectively. Then the composition of the two relations r_1, r_2 also satisfies the conditions of substructure relation. Therefore $d_1 \preceq d_3$ and $\preceq$ is a preorder.

We next show that $\preceq$ has the pairwise bounded join property by showing the following stronger property: there is an algorithm taking any two descriptions d_1, d_2 that determines whether d_1, d_2 have an upper bound or not and that if d_1, d_2 have an upper bound then computes (one of) their least upper bound. Let $M_{d_1} = (Q_1, s_1, \boldsymbol{F}_d, \delta_1, o_1)$ and $M_{d_2} = (Q_2, s_2, \boldsymbol{F}_d, \delta_2, o_2)$ be Moore machine representing d_1, d_2 respectively. Let $M = (Q, s, \boldsymbol{F}_d, \delta, o)$ be the product machine $(M_1 \times M_2)/\approx$. We say that a state q in M is *consistent* iff it satisfies the condition that if $q = (q_1, q_2)$ for some $q_1 \in Q_1, q_2 \in Q_2$ then $o(q) = (f, g)$ for some $f, g \in \boldsymbol{F}_d$ such that f, g has an upper bound under $\sqsubseteq^b$ and if $o(q) \in \{(Record, Record), (Inj, Inj)\}$ and $\delta(q, (l', l')) = q'$ for some l' then q' is consistent. We first show that if d_1, d_2 has an upper bound then the start state s is consistent. Suppose s is not consistent. Then there is some $a \in \mathcal{L}^*$ satisfying the following conditions: (1) for any proper prefix b of a, $o(s, b) \in \{(Record, Record), (Inj, Inj)\}$, and (2) $\delta^*(s, a) = (q_1, q_2), q_1 \in Q_1, q_2 \in Q_2$, and $o((q_1, q_2)) = (f, g)$ such that $\{f, g\}$ has no upper bound. Now suppose to the contrary that there is some d such that $d_1 \preceq d$ and $d_2 \preceq d$. By the definition of $\preceq$ and lemma 4.2, $d_1/a \preceq d/a$ and $d_2/a \preceq d/a$, which contradicts the fact that $o((q_1, q_2)) = (f, g)$ such that $\{f, g\}$ has no upper bound.

Next we show that if s is consistent then d_1, d_2 has a least upper bound by constructing one. Suppose s is consistent. Define $M' = (Q, s, \boldsymbol{F}_d, \delta', o')$ from M as follows:

1. Q, s are same as M,
2. $\delta'(q, l)$ is defined and equal to q' iff one of the following hold:
   (a) $o(q) \in \{(Record, Record), (Inj, Inj)\}$ and one of the following hold: (i) $\delta(q, (l, l)) = q'$, (ii) $\delta(q, (l, \$)) = q'$ or (iii) $\delta(q, (\$, l)) = q'$,
   (b) $o(q) = (Set, Set)$, $l = elm_i$ and $\delta(q, (elm_j, elm_k)) = q'$ where (elm_j, elm_k) is the i^{th} smallest symbol under the total order $\ll$ on $\mathcal{L}$ in the set $\{(elm_l, elm_n) | \delta(q, (elm_l, elm_n))$ is defined and consistent$\}$,
   (c) $q = (q_1, \$)$ and $l = (l, \$)$ or $q = (\$, q_2)$ and $l = (\$, l)$,

3. o' is defined as

$$o'(q) = \begin{cases} x \sqcup y & \text{if } o(q) = (x, y), x, y \in \boldsymbol{F}_d \text{ and } x \sqcup y \text{ exists} \\ x & \text{if } o(q) = (x, \$) \\ y & \text{if } o(q) = (\$, y) \\ \$ & \text{otherwise.} \end{cases}$$

We show that $M'(s)$ is a least upper bound of d_1, d_2. Let $S_1 = \{M_1(q) | q \in Q_1, q \text{ reachable}\}$, $S_2 = \{M_2(q) | q \in Q_2, q \text{ reachable}\}$, and $S = \{M'(q) | q \in Q, q \text{ reachable}\}$. Then $S_1 = \boldsymbol{Subtrees}(d_1), S_2 = \boldsymbol{Subtrees}(d_2)$ and $S = \boldsymbol{Subtrees}(M'(s))$. Define the relation $\sim_1$ between S_1 and S as $M_1(q) \sim_1 M'(q')$ iff $q' = (q, x)$ for some x. Then it is easily checked that this relation satisfies the conditions of substructure relation and therefore $d_1 \preceq M'(s)$. Similarly $d_2 \preceq M'(s)$. Let d be any upper bound of d_1, d_2. Let $\sim'_1, \sim'_2$ be substructure relations on $\boldsymbol{Subtrees}(d_1) \times \boldsymbol{Subtrees}(d)$ and $\boldsymbol{Subtrees}(d_2) \times \boldsymbol{Subtrees}(d)$ respectively. Define the relation $\sim$ on $S \times \boldsymbol{Subtrees}(d)$ as $M'(q) \sim d'$ iff one of the following hold: (1) $q = (q_1, \$), M_1(q_1) \sim'_1 d'$, (2) $q = (\$, q_2), M_2(q_2) \sim'_2 d'$, or (3) $q = (q_1, q_2), M_1(q_1) \sim'_1 d', M_2(q_2) \sim'_2 d'$. Then $\sim$ satisfies conditions 1,2,3 of the definition of a substructure relation (definition 4.15). For condition 4, suppose $M'(q) \sim d'$ and $M'(q) = Set$. If $q = (q_1, \$)$ or $q = (\$, q_2)$ then condition 4 follows from the fact that $\sim'_1, \sim'_2$ are substructure relations. Suppose $q = (q_1, q_2)$. Then $M_1(q_1) \sim'_1 d'$ and $M_2(q_2) \sim'_2 d'$. If $l \in dom(d')$ for some $l \in \mathcal{L}$, then there is some $l_1, l_2 \in \mathcal{L}$ such that $\delta_1(q_1, l_1) = q'_1, \delta_2(q_2, l_2) = q'_2$, $M_1(q'_1) \sim'_1 d'/l$ and $M_2(q'_2) \sim'_2 d'/l$. By lemma 4.2, $M_1(q'_1) \preceq d'/l$ and $M_2(q'_2) \preceq d'/l$. Let M'_1, M'_2, M'' be respectively Moore machines obtained from M_1, M_2, M' by respectively replacing their start states with $q'_1, q'_2, (q'_1, q'_2)$. Clearly $M_1(q'_1) = M'_1(q'_1)$, $M_2(q'_2) = M'_2(q'_2)$ and $M'' = (M'_1 \times M'_2)/\approx$. Since $M'_1(q'_1)$ and $M_2(q'_2)$ has an upper bound, (q'_1, q'_2) is consistent. By definition, $l_1 = elm_i$ and $l_2 = elm_j$ for some i, j. Then by the definition of M' there is some l' such that $\delta'(q, l') = (q'_1, q'_2)$ and therefore $M'(q)/l' \sim d'/l$.

Since M' is effectively constructed, the proposition was proved. ▮

The above proof also establishes that least upper bounds are effectively computable. For the Moore machine M' defined in the above proof, it can be also easily shown that $d_1 \preceq d_2$ iff M' satisfies the following conditions: for all reachable state q in M', (1) $second(q) \in Q_2$, (2) if $q = (q_1, q_2), q_1 \in Q_1, q_2 \in Q_2$ then $o_1(q_1) \sqsubseteq^b o_2(q_2)$ and if $o(q) = Set$ then for all l_2 such that $\delta_2(q_2, l_2)$ is defined there is some l_1 such that $\delta'(q, (l_1, l_2))$ is defined. Therefore we have:

Proposition 4.11 *The relation $\preceq$ on $\boldsymbol{Dobj}^\infty$ is decidable and least upper bounds (if they exist) are effectively computable.* ▮

The next proposition show that the typing is preserved by least upper bound.

Proposition 4.12 *If $d_1 : \sigma$, $d_2 : \sigma$ and d is a least upper bound of d_1, d_2 then $d : \sigma$.*

Proof Let d_1, d_2 be any descriptions and M' be the Moore machine representing a least upper bound of d_1 and d_2 constructed in the proof of proposition 4.10. By the construction of M', for any $a \in dom(M'(s))$ either there is some $b \in dom(d_1)$ such that $a \overset{*}{\approx} b$ and $d_1(b) \sqsubseteq^b M'(s)(a)$ or there is some $c \in dom(d_2)$ such that $a \overset{*}{\approx} c$ and $d_2(c) \sqsubseteq^b M'(s)(a)$. Since for some $x, y \in \boldsymbol{F}_d$, if $x \sqsubseteq^b y$ and $x :^b f$ for some $f \in \boldsymbol{F}_\tau$ then $y :^b f$, in either case a satisfies the conditions of the definition of the typing relation $d : \sigma$. ∎

Definition 4.16 *For any description type $\sigma \in \boldsymbol{Dtype}^\infty$, the domain D_σ associated with σ is the poset $[(\{d | d : \sigma\}, \preceq)]$.*

Theorem 4.5 *For any σ, D_σ is a description domain.*

Proof We show that D_σ has a bottom element. By definition of D_σ, it is suffices to show the existence of a description d such that $d \preceq d'$ for all $d' \in \{d | d : \sigma\}$. Define a mapping $nullval : \boldsymbol{F}_\tau \longrightarrow \boldsymbol{F}_d$ as

$$nullval(f) = \begin{cases} null_f & \text{if } f \in \mathcal{B} \\ Inj & \text{if } f = Varinat \\ f & \text{otherwise.} \end{cases}$$

For any σ, define the description $Null(\sigma)$ as follows:

1. $a \in dom(Null(\sigma))$ iff $a \in dom(\sigma)$ and there is no proper prefix b of a such that $\sigma(b) = Varinat$, and

2. for all $a \in dom(Null(\sigma))$, $Null(\sigma)(a) = nullval(\sigma(a))$.

From this definition, it is easy to check that $Null(\sigma) : \sigma$ and $Null(\sigma) \preceq d$ for any description $d : \sigma$. Then the theorem follows from propositions 4.10, 4.11, 4.12 and lemma 2.1. ∎

4.4.5 A Model of the Type System

We now define the set of embedding-projection pairs to connect the set of description domains and turn them into a database domain.

For defining functions and properties on D_σ, the following definitions and results are useful. Let $(P_1, \leq_1)$, $(P_2, \leq_2)$ be a preordered sets. A function $f : P_1 \rightarrow P_2$ is *monotone* iff for any $p_1, p_2 \in P_1$, if $p_1 \leq_1 p_2$ then $f(p_1) \leq_2 f(p_2)$. For a monotone function $f : P_1 \rightarrow P_2$, define $[f] : P_1/\!\equiv \rightarrow P_2/\!\equiv$ as $[f]([x]) = [f(x)]$. Since f is monotone, $[f]$ is well defined in the sense that it does not depend on representatives of equivalence classes. It is also clear that $[f]$ is monotone. The following lemma is an immediate consequence of the definition.

Lemma 4.3 *Let $(P_1, \leq_1), (P_2, \leq_2)$ be preordered sets and $f : P_1 \rightarrow P_2$, $g : P_2 \rightarrow P_1$ be monotone functions. If for all $p \in P_1$, $g(f(p)) = p$ and for all $p \in P_2$, $f(g(p)) \leq_2 p$ then $([f], [g])$ is an embedding-projection pair between $[(P_1, \leq_1)]$ and $[(P_2, \leq_2)]$.* ▌

Definition 4.17 *Let $\sigma_1, \sigma_2 \in \boldsymbol{Dtype}^\infty$ such that $\sigma_1 \leqslant \sigma_2$. $\phi_{\sigma_1 \rightarrow \sigma_2}$ is a function from $\{d | d : \sigma_1\}$ to $\{d | d : \sigma_2\}$ defined as follows: $a \in dom(\phi_{\sigma_1 \rightarrow \sigma_2}(d))$ iff either (1) $a \in dom(d)$ or (2) there are some a_1, a_2, b_1 such that $a = a_1 \cdot a_2$ and $a_1 \overset{*}{\approx} b_1$ satisfying:*

1. *$a_1 \in dom(d)$, $d(a_1) = Record$ and for any non empty prefix a_3 of a_2 there is no a_4 such that $a_1 \cdot a_3 \overset{*}{\approx} a_4$, $a_4 \in dom(d)$,*
2. *$b_1 \cdot a_2 \in dom(\sigma)$ and for any proper prefix a_3 of a_2, $\sigma(b_1 \cdot a_3) \neq Variant$,*

and for any $a \in dom(\phi_{\sigma_1 \rightarrow \sigma_2}(d))$,

$$\phi_{\sigma_1 \rightarrow \sigma_2}(d)(a) = \begin{cases} d(a) & \text{if } a \in dom(d) \\ nullval(\sigma_2(b)) & \text{if } a \notin dom(d) \text{ where } b \overset{*}{\approx} a, b \in dom(\sigma) \end{cases}$$

where nullval is the function from $\boldsymbol{F}_\tau$ to $\boldsymbol{F}_d$ defined in the proof of theorem 4.5.

$\psi_{\sigma_2 \rightarrow \sigma_1}$ is a mapping from $\{d | d : \sigma_2\}$ to $\{d | d : \sigma_1\}$ defined as follows: $\psi_{\sigma_2 \rightarrow \sigma_1}(d)$ is the restriction of d such that $a \in dom(\psi_{\sigma_2 \rightarrow \sigma_1})(d)$ iff $a \in dom(d)$ and there is some $b \in dom(\sigma_2)$ such that $a \overset{*}{\approx} b$.

Define

$$\begin{aligned} \boldsymbol{Emb}^\infty &= \{\phi_{\sigma_1 \rightarrow \sigma_2} | \sigma_1, \sigma_2 \in \boldsymbol{Dtype}^\infty, \sigma_1 \leqslant \sigma_2\}, \\ \boldsymbol{Emb} &= \{\phi_{\sigma_1 \rightarrow \sigma_2} | \sigma_1, \sigma_2 \in \boldsymbol{Dtype}, \sigma_1 \leqslant \sigma_2\}, \\ \boldsymbol{Proj}^\infty &= \{\psi_{\sigma_1 \rightarrow \sigma_2} | \sigma_1, \sigma_2 \in \boldsymbol{Dtype}^\infty, \sigma_2 \leqslant \sigma_1\}, \\ \boldsymbol{Proj} &= \{\psi_{\sigma_1 \rightarrow \sigma_2} | \sigma_1, \sigma_2 \in \boldsymbol{Dtype}, \sigma_2 \leqslant \sigma_1\}. \end{aligned}$$

Since for any a, b in $dom(\sigma)$, if $a \overset{*}{\approx} b$ then $a = b$, the above definition of $\phi_{\sigma_1 \rightarrow \sigma_2}$ is well defined.

Proposition 4.13 *For any $\sigma_1, \sigma_2, \sigma_1 \leqslant \sigma_2$, $([\phi_{\sigma_1 \rightarrow \sigma_2}], [\psi_{\sigma_2 \rightarrow \sigma_1}])$ is an embedding-projection pair between D_{σ_1} and D_{σ_2}.*

Proof For any element d such that $d : \sigma_1$, let $d' = \phi_{\sigma_1 \to \sigma_2}(d)$ and $d'' = \psi_{\sigma_2 \to \sigma_1}(d')$. By the definitions of $\phi_{\sigma_1 \to \sigma_2}$ and $d : \sigma_1$, $a \in dom(d)$ iff $a \in dom(d')$ and there is some $b \in dom(\sigma_1)$ such that $a \overset{*}{\approx} b$, and for any $a \in dom(d), d'(a) = d(a)$. By the definition of $\psi_{\sigma_2 \to \sigma_1}$, $a \in dom(d'')$ iff $a \in dom(d')$ and there is some b such that $a \overset{*}{\approx} b$, $b \in dom(\sigma_1)$. Also for any $a \in dom(d''), d''(a) = d'(a)$. Therefore $d = d''$ and hence $\psi_{\sigma_2 \to \sigma_1}(\phi_{\sigma_1 \to \sigma_2})(d) = d$.

For any element d such that $d : \sigma_2$, let $d' = \psi_{\sigma_2 \to \sigma_1}(d)$ and $d'' = \phi_{\sigma_1 \to \sigma_2}(d')$. Define a relation $\sim$ on $\boldsymbol{Subtrees}(d'') \times \boldsymbol{Subtrees}(d)$ as follows: for $d_1 \in \boldsymbol{Subtrees}(d''), d_2 \in \boldsymbol{Subtrees}(d)$, $d_1 \sim d_2$ iff either there is some $a \in dom(d')$ such that $d_1 = d''/a$ and $d_2 = d/a$, or there is some a, b such that $a \notin dom(d')$, $a \overset{*}{\approx} b$, $d_1 = d''/b$ and $d_2 = d/a$. Since $\epsilon \in dom(d')$, $d'' \sim d$. Suppose $d_1 = d''/a, d_2 = d/a$ for some $a \in dom(d')$. By the definitions of $\phi_{\sigma_1 \to \sigma_2}$ and $\psi_{\sigma_2 \to \sigma_1}$, $d''(a) = d'(a) = d(a)$. Suppose $d_1 = d''/b, d_2 = d/a$ for some $a \notin dom(d')$, $a \overset{*}{\approx} b$. Then by the definition of $\phi_{\sigma_1 \to \sigma_2}$, there is some c such that $c \overset{*}{\approx} b, c \in dom(\sigma_2)$ and $d''(b) = nullval(\sigma_2(c))$. By the property of $nullval$, $d''(b) \sqsubseteq^b d(a)$. Therefore in both case $d_1(\epsilon) \sqsubseteq^b d_2(\epsilon)$. The other conditions of substructure relation (condition 3–4) can be easily checked by distinguishing cases whether $a \in dom(d')$ or not and using the property of the typing relation and the definition of $\phi_{\sigma_1 \to \sigma_2}$ in the latter case.

For the monotonicity of $\phi_{\sigma_1 \to \sigma_2}$, let $d_1, d_2 \in \{d | d : \sigma_1\}$ and $d_1' = \phi_{\sigma_1 \to \sigma_2}(d_1)$, $d_2' = \phi_{\sigma_1 \to \sigma_2}(d_2)$. Suppose there is a substructure relation $\sim$ on $\boldsymbol{Subtrees}(d_1) \times \boldsymbol{Subtrees}(d_2)$. Define a relation $\sim'$ on $\boldsymbol{Subtrees}(d_1') \times \boldsymbol{Subtrees}(d_2')$ as follows: $d \sim' d'$ iff either (1) there are a, b such that $d_1/a \sim d_2/b$ and $d = d_1'/a, d' = d_2'/b$ or (2) there are a, b, c such that $d_1/a \sim d_2/b$, $d = d_1'/a \cdot c$, $d' = d_2/b \cdot c$, and for any nonempty prefix d of c $a \cdot d \notin dom(d_1)$, $b \cdot d \notin dom(d_2)$. It can then be checked that $\sim'$ is a substructure relation. For the monotonicity of $\psi_{\sigma_2 \to \sigma_1}$, let $d_1, d_2 \in \{d | d : \sigma_2\}$ and $d_1' = \psi_{\sigma_2 \to \sigma_1}(d_1)$, $d_2' = \psi_{\sigma_2 \to \sigma_1}(d_2)$. Suppose there is a substructure relation $\sim$ on $\boldsymbol{Subtrees}(d_1) \times \boldsymbol{Subtrees}(d_2)$. Define a relation $\sim'$ on $\boldsymbol{Subtrees}(d_1') \times \boldsymbol{Subtrees}(d_2')$ as: $d \sim' d'$ iff there are a, b such that $d_1/a \sim d_2/b$ and $d = d_1'/a, d' = d_2'/b$. Then it is easily verify that $\sim'$ is a substructure relation. Then the proposition follows from lemma 4.3. ▮

For ***Emb*** and ***Proj***, there are inductive definitions. We first define functors (function constructors) for records, variants and sets.

1. *Records.*

   Let $f_1 : \sigma_1^1 \to \sigma_1^2, \ldots, f_n : \sigma_n^1 \to \sigma_n^2$ be any functions and $c_{n+1}, \ldots, c_{n+m}$ be any constants $(n, m \geq 0)$. $[l_1 = f_1, \ldots, l_n = f_n, l_{n+1} = c_{n+1}, \ldots, l_{n+m} = c_{n+m}]$ is the function on records of type $[l_1 : \sigma_1^1, \ldots, l_n : \sigma_n^1]$ defined as

   $$[l_1 = f_1, \ldots, l_n = f_n, l_{n+1} = c_{n+1}, \ldots, l_{n+m} = c_{n+m}]([l_1 = d_1, \ldots, l_n = d_n]) =$$

$$[l_1 = f_1(d_1), \ldots, l_n = f_n(d_n), l_{n+1} = c_{n+1}, \ldots, l_{n+m} = c_{n+m}]$$

and $[l_1 = f_1, \ldots, l_k = f_k, l_{k+1} = \sigma_{k+1}, \ldots, l_n = \sigma_n]$ $(0 \leq k \leq n)$ is the function on records of type $[l_1 : \sigma_1^1, \ldots, l_k = \sigma_k^1, l_{k+1} = \sigma_{k+1}, \ldots, l_n = \sigma_n]$ defined as

$$\begin{aligned}&[l_1 = f_1, \ldots, l_k = f_k, l_{k+1} = \sigma_{k+1}, \ldots, l_n = \sigma_n]([l_1 = d_1, \ldots, l_n = d_n]) = \\ &\quad [l_1 = f_1(d_1), \ldots, l_k = f_k(d_k)].\end{aligned}$$

2. *Variants.*

   Let $f_1 : \sigma_1^1 \to \sigma_1^2, \ldots, f_n : \sigma_n^1 \to \sigma_n^2$ be any functions. $\langle l_1 = f_1, \ldots, l_n = f_n \rangle$ is the function on variants of type $\langle l_1 : \sigma_1^1, \ldots, l_n : \sigma_n^1 \rangle$ defined as

$$\begin{aligned}\langle l_1 = f_1, \ldots, l_n = f_n \rangle(Inj) &= Inj, \\ \langle l_1 = f_1, \ldots, l_n = f_n \rangle(Inj(l_i = d)) &= Inj(l_i = f_i(d))(1 \leq i \leq n).\end{aligned}$$

3. *Sets.*

   Let $f : \sigma_1 \to \sigma_2$ be any function. $\{\!\{f\}\!\}$ is the function on sets of type $\{\!\{\sigma_1\}\!\}$ defined as

$$\{\!\{f\}\!\}(\{\!\{d_1, \ldots, d_n\}\!\}) = \{\!\{f(d_1), \ldots, f(d_n)\}\!\}.$$

Then $\boldsymbol{Emb}$ coincides with the following inductively defined set $\boldsymbol{Emb}^o$:

1. $id_b \in \boldsymbol{Emb}^o$ for any $b \in \mathcal{B}$ where id_b is the identity function on B_b,

2. $[l_1 = \phi_{\sigma_1^1 \to \sigma_1^2}, \ldots l_n = \phi_{\sigma_n^1 \to \sigma_n^2}, l_{n+1} = Null(\sigma_{n+1}), \ldots, l_{n+m} = Null(\sigma_{n+m})] \in \boldsymbol{Emb}^o$ if $\phi_{\sigma_1^1 \to \sigma_1^2}, \ldots, \phi_{\sigma_n^1 \to \sigma_n^2} \in \boldsymbol{Emb}^o$ and $\sigma_{n+1}, \ldots, \sigma_{n+m} \in \boldsymbol{Dtype}$ where $Null(\sigma_i)$ is the mapping defined in theorem 4.5,

3. $\langle l_1 = \phi_{\sigma_1^1 \to \sigma_1^2}, \ldots l_n = \phi_{\sigma_n^1 \to \sigma_n^2} \rangle \in \boldsymbol{Emb}^o$ if $\phi_{\sigma_1^1 \to \sigma_1^2}, \ldots, \phi_{\sigma_n^1 \to \sigma_n^2} \in \boldsymbol{Emb}^o$,

4. $\{\!\{\phi_{\sigma_1 \to \sigma_2}\}\!\} \in \boldsymbol{Emb}^o$ if $\phi_{\sigma_1 \to \sigma_2} \in \boldsymbol{Emb}^o$.

The $\boldsymbol{Proj}$ coincides with the following inductively defined set $\boldsymbol{Proj}^o$:

1. $id_b \in \boldsymbol{Proj}^o$ where id_b is the identity function on B_b,

2. $[l_1 = \psi_{\sigma_1^1 \to \sigma_1^2}, \ldots l_k = \psi_{\sigma_k^1 \to \sigma_k^2}, l_{k+1} = \sigma_{k+1}, \ldots, l_n = \sigma_n] \in \boldsymbol{Proj}^o$ if $\psi_{\sigma_1^1 \to \sigma_1^2}, \ldots, \psi_{\sigma_k^1 \to \sigma_k^2} \in \boldsymbol{Proj}^o$ and $\sigma_{k+1}, \ldots, \sigma_n \in \boldsymbol{Dtype}$

3. $\langle l_1 = \psi_{\sigma_1^1 \to \sigma_1^2}, \ldots l_n = \psi_{\sigma_n^1 \to \sigma_n^2} \rangle \in \boldsymbol{Proj}^o$ if $\psi_{\sigma_1^1 \to \sigma_1^2}, \ldots, \psi_{\sigma_n^1 \to \sigma_n^2} \in \boldsymbol{Proj}^o$,

4. $\{\!\{\psi_{\sigma_1 \to \sigma_2}\}\!\} \in \boldsymbol{Proj}^o$ if $\psi_{\sigma_1 \to \sigma_2} \in \boldsymbol{Proj}^o$.

From the inductive characterization of $\boldsymbol{Emb}$ and $\boldsymbol{Proj}$ it is easy to see that all embeddings and projections between finite types are computable functions. This necessary property still hold for general embeddings and projections.

Proposition 4.14 *Elements of* $\boldsymbol{Emb}^{\infty}$ *and* $\boldsymbol{Proj}^{\infty}$ *are all computable functions.*

Proof We first show for the embeddings in $\boldsymbol{Emb}^{\infty}$. Let $\sigma_1 \leqslant \sigma_2$ and $d : \sigma_1$. Let $M_d = (Q_d, s_d, \boldsymbol{F}_d, \delta_d, o_d)$ and $M_{\sigma_2} = (Q_{\sigma_2}, \boldsymbol{F}_\tau, \delta_{\sigma_2}, o_{\sigma_2})$ be Moore machines representing d, σ_2 respectively. Let $M = (Q, s, \boldsymbol{F}, \delta, o) = (M_d \times M_{\sigma_2})/\approx$ be the product machines modulo the equivalence relation $\approx$ defined in definition 4.14. Define $M' = (Q, s, \boldsymbol{F}_d, \delta', o')$ from M as follows:

1. Q, s are same as M,
2. $\delta'(q, l)$ is defined and equal to q' iff either $\delta(q, (l, l')) = q$ and $l \neq \$$, or $\delta(q, (\$, l)) = q'$ and $o(q) \notin \{(Inj,\ Variant), (\$,\ Variant), (Set, Set)\}$,
3. o' is defined as

$$o'(q) = \begin{cases} f & \text{if } o(q) = (f, g), f \neq \$ \\ nullval(g) & \text{if } o(q) = (\$, g), g \neq \$ \\ \$ & \text{otherwise.} \end{cases}$$

It can then be checked that $M'(s) = \phi_{\sigma_1 \to \sigma_2}(d)$.

For the projections in $\boldsymbol{Proj}^{\infty}$, let $\sigma_2 \leqslant \sigma_1$ and $d : \sigma_1$. Let $M_d = (Q_d, s_d, \boldsymbol{F}_d, \delta_d, o_d)$ and $M_{\sigma_2} = (Q_{\sigma_2}, \boldsymbol{F}_\tau, \delta_{\sigma_2}, o_{\sigma_2})$ be Moore machines representing d, σ_2 respectively. Let $M = (Q, s, \boldsymbol{F}, \delta, o) = (M_d \times M_{\sigma_2})/\approx$. Define $M' = (Q, s, \boldsymbol{F}_d, \delta', o')$ from M as follows:

1. Q, s are same as M,
2. $\delta'(q, l)$ is defined and equal to q' iff $\delta(q, (l, l')) = q$ and $l' \neq \$$.
3. o' is defined as

$$o'(q) = \begin{cases} f & \text{if } o(q) = (f, g), g \neq \$ \\ \$ & \text{otherwise.} \end{cases}$$

Then by lemma 2.5, $M'(s) = \psi_{\sigma_1 \to \sigma_2}(d)$. ∎

We now have the following theorem:

Theorem 4.6 $(\{D_\sigma | \sigma \in \boldsymbol{Dtype}^{\infty}\}, \{[\phi] | \phi \in \boldsymbol{Emb}^{\infty}\})$ *is a database domain and a model of* $(\boldsymbol{Dtype}^{\infty}, \leqslant)$.

Proof By proposition 4.13, for all $\phi \in \boldsymbol{Emb}^{\infty}$, $[\phi]$ is an embedding. Since $\boldsymbol{Dtype}^{\infty}$ is a poset with the pairwise bounded join property, conditions 1 – 4 of a database domain (definition 4.9) are satisfied by the set $\{[\phi] | \phi \in \boldsymbol{Emb}^{\infty}\}$. Condition 5 is shown by proposition 4.14. The mapping $\mu : \boldsymbol{Dtype}^{\infty} \rightarrow \{D_\sigma | \sigma \in \boldsymbol{Dtype}^{\infty}\}$ is given as $\mu(\sigma) = D_\sigma$. ▌

This theorem says that we have successfully completed the constructions of a type system for complex database objects and its semantic domain. The type system allows arbitrarily complex objects constructed by records, variants, finite sets and recursive definition. This demonstrates that our mathematical characterizations of database type systems and their semantic spaces are general enough to provide a semantic formulation of a database domain that is rich enough to cover virtually all existing representations of complex database objects.

Another important implication of the above theorem is its computational contents. It guarantees that for arbitrarily complex types, various properties needed to compute joins and projections and to type-check expressions are always effectively computable. As we will show in the next chapter, these properties enable us to develop a practical programming language that integrates the database type system we have constructed and an ML-style polymorphic type system.

Joins and projections are given by equations (4.1) and (4.2), which are always computable functions. An actual algorithm to compute them can be easily extracted from the proof of the theorem. Moreover, there are generic ways to compute joins and projections. For joins, we have:

Proposition 4.15 *If $d_1 : \sigma_1$, $d_2 : \sigma_2$ are descriptions such that $\sigma = \sigma_1 \sqcup \sigma_2$ then*

$$\phi_{\sigma_1 \rightarrow \sigma}(d_1) \sqcup \phi_{\sigma_2 \rightarrow \sigma}(d_2) = d_1 \sqcup d_2.$$

Proof By the definitions of ϕ and $Null$, $\phi_{\sigma_1 \rightarrow \sigma}(d_1) = d_1 \sqcup Null(\sigma)$ and $\phi_{\sigma_2 \rightarrow \sigma}(d_2) = d_2 \sqcup Null(\sigma)$. Then we have : $\phi_{\sigma_1 \rightarrow \sigma}(d_1) \sqcup \phi_{\sigma_2 \rightarrow \sigma}(d_2)$ is defined iff $(d_1 \sqcup d_2) \sqcup Null(\sigma)$ is defined iff $d_1 \sqcup d_2$ is defined. The equation follows from the fact that $Null(\sigma)$ is the bottom element of the set D_σ. ▌

Since we have shown that least upper bounds are effectively computable, the above result gives a generic way to compute joins. For projections, the definitions of $\psi_{\sigma_1 \rightarrow \sigma_2}$ is already generic in the sense that it does not depend on σ_1. Define the partial function $Proj^{\sigma}$ as follows: for any description d, $Proj^{\sigma}(d)$ is the restriction of d such that $a \in dom(d')$ iff $a \in dom(d)$ and there is some $b \in dom(\sigma)$ such that $a \overset{*}{\approx} b$. Since the definition of $Proj^{\sigma}$ and $\psi_{\sigma_1 \rightarrow \sigma}$ is identical except their domains, we have:

Proposition 4.16 *If d is a description of type σ such that $\sigma' \leqslant \sigma$ then*

$$Proj^{\sigma'}(d) = \psi_{\sigma \to \sigma'}(d).$$

∎

For static type-checking, since join and projection have polymorphic type schemes (4.3) and (4.4), the result types of joins and projections are always determined from the types of their arguments. Moreover, theorem 4.6 guarantees that they are effectively computed. The following are examples of joins of descriptions in figure 4.5:

$$Join(Mary1, Mary2) = Mary3,$$

$$Join(Flights, Flown\text{-}by) = Schedule\text{-}data.$$

The types of the above two joins are *working-student* and *schedule-data* respectively, which are computed from the types of their arguments. This property allows us to develop a static type system.

Chapter 5

A Polymorphic Language for Databases and Object-Oriented Programming

This chapter combines ML type system we have analyzed in chapter 3 and the type system for complex database objects we have constructed in chapter 4 and defines a programming language that achieves the integration of records, variants and database objects. We call the language Machiavelli maintaining the name we gave to the language in [87]. Later in chapters 6 and 7, the language is extended to integrate the other desirable features we discussed in the introduction of the thesis. Some of the results in this chapter were presented in [85].

5.1 Introduction

Machiavelli extends the polymorphic programming language ML with

- labeled record and labeled variants with associated operations field selection, field modification, and case statement, and
- complex database objects and the associated database operations *join* and *projection*.

We work out this extension preserving the ML's central feature of static type-checking, polymorphism and static type inference.

Let us first illustrate how ML type system is extended by simple examples. In section 1.1, we have defined the function *wealthy*:

$$\begin{aligned}\textbf{fun } \mathit{wealthy}(X) \ = \ & \textbf{select } x.\mathit{Name}\\ & \textbf{where } x \ \in X\\ & \textbf{with } x.\mathit{Salary} > 100000;\end{aligned}$$

for which Machiavelli infers the following type information

$$\mathit{wealthy} : \{\!\{[(s_1)\mathit{Name} : s_2, \mathit{Salary} : \mathit{int}]\}\!\} \rightarrow \{\!\{s_2\}\!\}.$$

As we have explained, the above type expression means that *wealthy* is a function that takes a homogeneous set of records, each of the type $[(s_1)Name : s_2, Salary : int]$, and returns a homogeneous set of values of the type s_2. s_2 is a ***description type variables*** representing an arbitrary description type we have constructed in section 4.4 and $[(s_1)Name : s_2, Salary : int]$ is a ***conditional type variable*** which intuitively represents an arbitrary record type that contains $Name : s_2$ and $Salary : int$ fields.

The function *wealthy* is ***polymorphic*** with respect to the description type variable s_2 of the values in the *Name* field (as representable in ML) but is also polymorphic with respect to the conditional type variable $[(s_1)Name : s_1, Salary : int]$. In this second form of polymorphism, *wealthy* can be thought of as a "method" in the sense of object-oriented programming languages where methods associated with a class may be inherited by a subclass, and thus applied to objects of that subclass. This second form of polymorphism is illustrated by the following example. Suppose we implement person objects by expressions of the type

$$[\mathit{Name} : \mathit{string}, \mathit{Age} : \mathit{int}]$$

and want to define a function *increment_age* which takes a person object and returns a new person object whose *Age* is incremented by one. The function is written in Machiavelli as follows:

$$\textbf{fun } \mathit{increment\_age} \ p \ = \ \textbf{modify}(p, \mathit{Age}, p.\mathit{Age} + 1)$$

where **modify** is the primitive that modifies (or updates) a record at a specified field with a specified value. Machiavelli finds the following type for this function:

$$\mathit{increment\_age} : [(s)\mathit{Age} : \mathit{int}] \rightarrow [(s)\mathit{Age} : \mathit{int}].$$

This says that *increment_age* is a function which takes a record of any type containing $Age : int$ field and retuns a record of the same type. By this mechanism, Machiavelli achieves the similar goal to the system for ***multiple inheritance*** originally proposed by Cardelli [24]. For example, suppose we implement employee objects by expressions of the type:

$$[Name : string, Age : int, Salary : int].$$

Since the type $[Name : string, Age : int, Salary : int]$ is an instance of $[(s)Age : int]$, the following type is an instance of the type of *increment_age*:

$$[Name : string, Age : int, Salary : int] \rightarrow [Name : string, Age : int, Salary : int]$$

and therefore *increment_age* can also be applied to an employee object returning an employee object.

In the rest of this section, we review existing approaches to integrate records, variants and complex database objects in a static type system and outline our strategy.

5.1.1 Records and Variants for Object-oriented Programming

There are numerous arguments on the features of object-oriented programming. Here we will not go into the argument on the "essence" of object-oriented programming but instead we concentrate on the following features:

- method inheritance,
- user definable class hierarchies based on an inheritance relation,
- data abstraction,

which I believe to be the major contributions of object-oriented programming. Among these features, the method inheritance mechanism is a form of polymorphism in the sense that it allow methods to be applied to various structures sharing certain common properties. In this chapter we only consider this feature. I believe that this mechanism is a basic feature of a type system that should be integrated in the polymorphic core of the language. The other two features will be treated in the next chapter by extending Machiavelli with a new construction for classes.

Method Inheritance by Subtyping

Perhaps the first serious attempt to integrate method inheritance in a static type system was [24] by Cardelli. He argued that the essential feature of "objects" can be captured by labeled records and labeled variants. Of course records and variants alone do not achieve the features of object-oriented programming such as classes and data abstraction. However, we can agree with this view under the interpretation that these data structures enable us to implement the essential features of object-oriented programming. Indeed in object-oriented programming languages such as Smalltalk [44], objects are implemented by a set of "instance variables" associated with "states" which can be naturally regarded as labeled record structures.

It is straightforward to add labeled records and labeled variants to a *simple* type system. Indeed many programming languages such as Pascal satisfactorily integrate them into a static type system. In such a type system objects can be implemented using record types and variant types. For example, the following types can be regarded as types for classes *person* and *employee*:

$$\begin{aligned} person &= [Name : string, Age : int], \\ mployee &= [Name : str, Age : int, Salary : int]. \end{aligned}$$

Methods can be implemented by functions on those types. For example, the following function

$$\textbf{fun } name(x : person) = x.Name$$

extracts the value of $Name$ from a person object and can be regarded as a method of the class *person*, where $r.l$ denotes the selection of the l field from the record r. Note that in a simply typed language, it is mandatory to specify the type of a formal parameter as in the above example. An obvious drawback to these simple type systems is that they do not support method inheritance. Since the body of the function *name* is also meaningful to objects of the type *employee*, we would also like to use this method for objects of the type *employee*. However, a simple type discipline does not allow such application. As a result, we are forced to define the same function for the type *employee*.

Cardelli observed that the method inheritance such as those in the above example is related to the structures of record types and variant types and proposed a type system that supports method inheritance [24]. He defined the *subtype relation*, which is based on the following relation on record types and variant types:

$$[l_1 : \tau_1, \ldots, l_n : \tau_n, \ldots] \leq [l_1 : \tau_1', \ldots, l_n : \tau_n'] \text{ if } \tau_i \leq \tau_i' \ (1 \leq i \leq n),$$

$$\langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle \leq \langle l_1 : \tau_1', \ldots, l_n : \tau_n', \ldots \rangle \text{ if } \tau_i \leq \tau_i' \ (1 \leq i \leq n).$$

The complete subtype relation is obtained by "lifing" these relation to function types by the following rule:

$$\text{if } \tau_1 \leq \tau_2 \text{ and } \tau_3 \leq \tau_4 \text{ then } \tau_2 \rightarrow \tau_3 \leq \tau_1 \rightarrow \tau_4.$$

He then defined a type system where the following rule is derivable:

$$(\text{TRANS}) \qquad \frac{e : \tau}{e : \tau'} \qquad \text{if } \tau \leq \tau'.$$

Because of this rule, a function defined on a type can be applied to values of all its subtypes. For example, the function *name* defined above can be applied not only to objects of the type *person* but also to objects of the type *employee* because $employee \leq person$ and any object o that has the typing $o : employee$ also has the typing $o : person$. Cardelli and Wegner extended this type system to integrate polymorphism using explicit type abstraction [27] (where the rule (TRANS) is an inference rule).

However, the type system of this kind that have been so far proposed suffer from the problem called "loss of type information". The problem was first pointed out in [27] in the context of Cardelli's original type system. To see the problem, consider the method that extracts *Name* field. In the following examples, we use λ notation to represent functions. If the type of *Name* field is *string*, then the method is implemented by the following function in [24]:

$$name1 \ \equiv \ \lambda x : [Name : string].\, x.Name.$$

Any types that are subtype of the type $[Name : string]$ inherit this method. Moreover, the result type is *string* as we expect. Now consider the case where the type of *Name* field is itself a record type such as $[Fn : string, Ln : string]$. The method is implemented by the following function:

$$name2 \ \equiv \ \lambda x : [Name : [Fn : string, Ln : string]].\, x.Name.$$

Again any types that are subtype of the type $[Name : [Fn : string, Ln : string]]$ inherit this method. However, the result type is not always the one we expect. For example, consider the following application:

$$name2([Name = [Fn = "Joe", Mi = "M", Ln = "Doe"]).$$

Since the type of $[Name = [Fn = "Joe", Mi = "M", Ln = "Doe"]$ is $[Name : [Fn : string, Mi : string, Ln : string]]$ which is a subtype of the type $[Name : [Fn : string, Ln : string]]$, the above application is well typed. Since *nema2* is a function that extracts *Name* field form a record, we expect this application to yield the value $[Fn = "Joe", Mi = "M", Ln = "Doe"]$. We therefore expect the type of this application to be $[Fn : string, Mi : string, Ln : string]$. However, in his system, the actual result type is $[Fn : string, Ln : string]$. This example shows that an application of an inherited method from a super type sometimes "loses" type information. Since the language is strongly typed, this also means the serious problem of loss of information of object itself. In the above example, the Mi filed of the object returned by the inherited method $name2$ can never be accessed.

In [27] a new construction, ***bounded quantification***, is introduced, which is a generalized form of type abstraction [94]. The introduction of type abstraction obviously enhances the expressive power of the language. However, the introduction of bounded quantification does not eliminate the problem of loss of type information. In the first place, the terms we have just examined are still terms of the language. Secondly, although the introduction of type parameters allows us to apply a term to an appropriate type, it does not guarantee that the term will not be prone to loss of type information. For example, in the new type system, the function that extract $Name$ field is implemented by the following polymorphic term:

$$pname \equiv \Lambda t_2.\Lambda t_1 \leq [Name : t_2].\lambda x : t_1.x.Name$$

where Λt is type abstraction and $\Lambda t \leq \tau$ is ***bounded*** type abstraction. By instantiating the type variable t_2 with $[Ln : string, Fn : string]$, we have the term:

$$pname1 \equiv \Lambda t_1 \leq [Name : [Ln : string, Fn : string]]\lambda x : t_1.x.Name$$

with the type

$$\forall t_1 \leq [Name : [Ln : string, Fn : string]].t_1 \rightarrow [Ln : string, Fn : string].$$

t_1 can be instantiated with any type that is a subtype of $[Name : [Ln : string, Fn : string]]$. Take $[Name : [Ln : string, Mi : sting, Ln : string]]$ and we have the term

$$\lambda x : [Name : [Ln : string, Mi : sting, Ln : string]].x.Name$$

whose type is

$$[Name : [Ln : string, Mi : sting, Ln : string]] \rightarrow [Ln : string, Ln : string].$$

Then by applying it to the object $[Name = [Ln = "Joe", Mi = "M", Ln = "Doe"]]$, we have an object of the type $[Ln : string, Ln : string]$. But since $pname$ is a function that extracts $Name$ field, we expect the result of the above application to yield $[Ln = "Joe", Mi = "M", Ln = "Doe"]$. We apparently lost the type information of $Mi : string$.

A precise analysis of the loss of type information phenomenon is certainly desirable. Meanwhile, the following analysis may contribute towards understanding part of the problem. Using the rule (TRANS) to type-check the terms containing field selection $e.l$ seems to fail to reflect the precise operational behavior of this program construction, which is

$$[\ldots, l = e, \ldots].l \Rightarrow e.$$

A typing rule that would fit this is:

$$\text{(dot)} \qquad \frac{e : \tau_1}{e \cdot l : \tau_2} \quad \text{where } \tau_1 \text{ is a record type containing } l : \tau_2.$$

The associated condition coincides with the subtype relation $\tau_1 \leq [l : \tau_2]$ *only if* τ_2 has no proper subtype. However, if τ_2 is a type that has proper subtypes such as record types, then the condition associated with the typing rule (dot) is strictly stronger than the subtype relation. In the original type system [24], this may help explaining why, for example, *name*2 exhibits loss of type information, but *name*1 does not. An appropriate relation to type-check terms containing field selection seems not the subtype relation but the *filed inclusion relation* among record types.

The same mismatch between the operational behavior of $e.l$ and the rule (TRANS) remains in the new type system [27]. As we have seen in the example of *pnema*, the bounded quantification does not correct the mismatch. Moreover, the bounded quantification does not provide an alternative to the rule (TRANS) to type-check terms containing filed selection. In [27] it is suggested that "now that we have bounded quantifiers, we could remove the other mechanism [of inheritance], requiring exact matching of types on parameter passing...". This should mean that the rule (TRANS) could be removed, since this is the only rule to achieve non-exact matching of types on parameter passing in the type system of [27]. If we do this, however, we will lose much of the power of representing inheritance, as shown below:

Proposition 5.1 *Let M be a term in FUN [27] that contain $x.l$ where x is a free variable in e. If $\lambda x : \tau_1 . M$ has a typing derivation in the type system of FUN then either τ is not a type variable or the derivation contains the application of the rule (*TRANS*).*

Proof A typing judgement in FUN is a formula of the form $C, A \rhd M : \tau$ where A is a type assignment and C is a subtype assumption. For the complete definition of the type system of FUN, readers are referred to [27].

The proposition is proved by the following property of the type system. Let A be a type assignment such that $A(x) = t$ for some type variable t. Then any derivation for $C, A \rhd x.l : \tau$ must contains the application of the rule (TRANS). ∎

This implies that without the rule (TRANS), functions containing field selection $e.l$ cannot be polymorphic even with the existence of bounded quantification and therefore if the rule(TRANS) is removed then we lose most of the power to represent inheritance.

The presence of the rule (TRANS) also raises some problems with primitive operations that are important for database programming such as join and equality. Here we consider the treatment of equality. There are at least two forms of equality that are commonly used in programming languages. One is the identity between run-time objects in store. Examples include Amber's equality primitive [23] and *eq* predicate in Lisp. In this view $[Name = "Joe"] = [Name = "Joe"]$ is presumably false, and the best we could say about equality is that it is a function of type $\tau_1 \times \tau_2 \longrightarrow bool$. Another form is the "structural" equality which tests whether two expressions denote the same value or not. Examples of this equality include the equality primitive = in Standard ML and *equal* in Lisp. For this form of equality, we would expect a type error if it is applied to values of different types. This is the type checking rule adopted in Standard ML. However, with the existence of the rule (TRANS), this desirable type discipline cannot be enforced. For example, consider the untyped term $\lambda x.\, x = [Name = "Joe"]$. In ML the type $[Name : string] \longrightarrow bool$ is inferred for this term, but with the addition of (TRANS), it is legitimate to apply this function to $[Name = "Joe", Age = 21]$ - something that in the untyped case we would expect to raise a run-time type error!

Method Inheritance by Type Inference

Wand observed [111] that the method inheritance is a mechanism to capture the polymorphic nature of operations associated with records and variants and can be achieved by ML style polymorphism through type inference. If a type system can infer the most general type for operations on record such as field selection, then those operations can be applied to any records to which the application is type correct. The major problem in accomplishing this idea is that basic operations on records and variants do not have a principal type-scheme and therefore the conventional type inference algorithm (theorem 3.2) cannot be directly applicable to these operations. The problem is seen in Standard ML[1], which integrates labeled records but cannot find a type for functions whose arguments are partial matches for records such as

```
fun name [Name = x,...] = x
```

[1] In Standard ML, the syntax for records is $\{l = e, \ldots, l = e\}$. Here we use the syntax $[l = e, \ldots, l = e]$.

where "..." in $[Name = x, ...]$ is the patter in Standard ML that matches any sequence of fields that do not contain *Name* field. Although Standard ML compiler reports a type error for the above definition, this function is well typed in the sense that it has a typing. For example, if we specify a type of the argument as in:

fun name ([Name = x,...] : [Name : str, Age : int]) = x

then Standard ML compiler find the following correct type for the function:

$[Name : string, Age : int] \rightarrow string.$

The problem is that there is no *principal* typing scheme because of the condition associated with the field selection.

Wand [111] tried to solve this problem by decomposing type expressions into two languages, TE and RE, respectively called type expressions and row expressions. (His language also contains labeled variants. Here we restrict attention to labeled records. Labeled variants can be understood similarly.) TE and RE are defined as follows:

$$\begin{aligned} TE &::= t \mid \iota \mid TE \rightarrow TE \mid [RE], \\ RE &:= \gamma \mid \emptyset \mid RE(l : TE) \end{aligned}$$

where t, γ are variables ranging over TE, RE respectively and $\emptyset$ is a constant symbol denoting the empty row. RE satisfies the following equality rules:

$$R(l : T_1)(l : T_2) = R(l : T_2) \tag{5.1}$$

$$R(l_1 : T_1)(l_2 : T_2) = R(l_2 : T_2)(l_1 : T_1) \tag{5.2}$$

His type system contains the following typing rules for records.

$$(with) \quad \frac{\mathcal{A} \vdash e_1 : [\rho] \qquad \mathcal{A} \vdash e_2 : \tau}{\mathcal{A} \vdash e_1 \; with \; l := e_2 : [\rho(l : \tau)]}$$

$$(dot') \quad \frac{\mathcal{A} \vdash e : [\rho(l : \tau)]}{\mathcal{A} \vdash e.l : \tau}$$

where τ and ρ ranges over arbitrary types and rows respectively. The rule for $e.l$ is now represented without any condition. The necessary condition is correctly captured by the equality rule (5.1). Therefore the primitive operators $\ldots \; with \; l := \ldots$ and $\ldots .l$ do have principal type schemes $[\gamma] \times t \rightarrow [\gamma(l : t)]$ and $[\gamma(l : t)] \rightarrow t$ respectively in TE and RE. Since RE equipped with the equality rule

(5.1), any ground instances of these type schemes are types of these operators. However, RE is no longer freely generated. Because of this fact, equations between TE and RE in general do not have most general solution. This reflects the fact that in his language there is a typable term that has no principal typing scheme. For example, the following term has a typing under his set of type inference rules but has no principal typing scheme:

$$(\lambda x.\, \lambda y.\, z\, (y(x \textit{ with } l := 1))\, (y\, [l = 1])\,)\, [l = \textit{true}].$$

His type inference algorithm produces the following equations on RE to infer the type of y:

$$[\gamma(l : \textit{int})] = [\emptyset(l : \textit{int})]$$

which does have a solution but has no most general one. Because of this fact, his unification-based type inference algorithm cannot find a solution and reports a type error.

We overcome this problem by extending the notion of typing schemes to allow conditions on type variables. In our system, the term $\lambda x.\, x.l$ has the following *principal* conditional typing scheme:

$$\emptyset \triangleright \lambda x.\, x.l \;:\; [(t_1)l : t_2] \rightarrow t_2$$

where $[(t_1)l : t_2]$ is a type variable t_1 associated with the condition that substitutions of t_1 are restricted to those θ such that $\theta(t_1)$ is a record type containing $l : \theta(t_2)$ field. By this mechanism we solve the type inference problem for records and variants. Several other solutions have been also proposed [38, 105, 25, 63, 26, 93]. An advantage of our method is that it allows us to extend ML type system uniformly to a wide range of data structures and operations including complex database objects and associated operations join and projection.

5.1.2 Integrating Database Objects

We turn our attention to the problem of integrating complex database objects and associated operations into programming languages. Since our objective of this subsection is to identify the problems, we will not attempt to give a comprehensive survey. Readers are referred to [10] for an excellent survey in this area.

Historically database systems are developed relatively independently of programming languages. Many database systems were built as stand alone systems with a special data manipulation languages often called ***database query languages***. A typical example is SQL [8] for System R [9]. Obvious problem of these languages is that they are extremely limited in expressive power because of the lack of general programming capability, which also makes it difficult to integrate databases with other applications.

In order to overcome this disadvantages, several *embedded languages* – the languages that embed a database query language as subroutine calls in a general purpose programming language – have been developed. The problem of this ad hoc solution is that the interface between a database and a host language is usually limited to primitive types such as integers and strings. The database structures such as records and relations are not recognized by the host language. As a result, database programming cannot make full use of the expressive power and the type-checking capability of the host language.

The designers of certain database programming languages, notably Pascal-R [99] and Galileo [7] have recognized this mismatch problem and have implemented languages in which a database can be directly represented in there type systems. Type checking in both of the languages is static and the database types are relatively simple and elegant extensions to the existing type systems of the programming languages on which they are based. Galileo also allows complex database objects and incorporates a form of inheritance based on the subtype rule we have just analyzed. However, these languages do not support database operations such as join and projection. Database programming is done by using special iteration primitives (**for each** ... **in** ... **do** in Pascal-R and **for** ... **in** ... **with** ... **do** in Galileo). Such iteration primitives are of course useful and necessary. But we would also have database operations on "bulk" of data such as join and projection. As we have argued in section 4.1, these operations are extremely useful in many database programming. As we will see in chapter 7 they are also useful to represent object-oriented databases. These language also do not integrate polymorphism and static type inference – the other essential features of good programming languages. To my knowledge, no attempt has been made to integrate database structures into a type system with polymorphism and static type inference.

In the previous chapter, we have solved the problem of constructing a type system for complex database objects with generalized join and projection. In this chapter we solve the problem of integrating them into an ML style type system. In order to achieve this goal we first extend the type system of ML to include the database type system we have developed in the previous chapter. We then define a set of constants for constructing and manipulating complex database objects and extend the proof system for typings to include these new constants. For the extended language, we develop a type inference algorithm. It turns out that our proposal of *conditional typing schemes* we have explained in the previous subsection is general enough to include structures and operations for databases.

5.2 Definition of Machiavelli

5.2.1 Types and Description Types

The set of *Type* (ranged over by τ) of Machiavelli is the set of regular trees denoted by the following term representations (subsection 2.2.3):

$$\tau ::= b \mid \tau \rightarrow \tau \mid [l : \tau, \ldots, l : \tau] \mid \langle l : \tau, \ldots, l : \tau \rangle \mid \{\!\{\tau\}\!\} \mid (rec\ v.\, \tau(v))$$

where b stands for base types. By the interpretation defined in subsection 2.2.3, each expression denotes a regular tree. The set of regular trees denoted by the following language is exactly the set $\boldsymbol{Dtype}^{\infty}$ of description types we have constructed in section 4.4:

$$\sigma ::= \ b \mid [l : \sigma, \ldots, l : \sigma] \mid \langle l : \sigma, \ldots, l : \sigma \rangle \mid \{\!\{\sigma\}\!\} \mid (rec\ v.\, \sigma(v)).$$

We apply directly the ordering we have defined on $\boldsymbol{Dtype}^{\infty}$ to those type expressions. For convenience, we assume a set of special labels $\#1, \ldots, \#i, \ldots$ and use the following shorthand:

$$\tau_1 \times \cdots \times \tau_n \Leftrightarrow [\#1 : \tau_1, \ldots, \#n : \tau_n].$$

5.2.2 Raw Terms

We first define the set *Consts* of constants of Machiavelli to represent records, variants and complex database objects and associated operations.

For labeled records, labeled variants and sets, the necessary constants are the following:

1. Record constructor constants:

$$record^{(l_1, \ldots, l_n)} : \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow [l_1 : \tau_1, \ldots, l_n : \tau_n]$$

   for all finite sequence of (distinct) labels $(l_1, \ldots, l_n)$ and for all types $\tau_1, \ldots, \tau_n$.

2. Variant constructor constants:

$$variant^{l} : \tau_1 \rightarrow \tau_2$$

   for all labels l and all pair of types τ_1, τ_2 such that τ_2 is a variant type containing $l : \tau_1$, i.e. τ_2 is a regular tree such that $\tau_2(\epsilon) = Variant$, $l \in dom(\tau_2)$ and $\tau_2/l = \tau_1$.

3. Set constructor constants:

$$set^{n} : \underbrace{\sigma \rightarrow \cdots \rightarrow \sigma}_{n} \rightarrow \{\!\{\sigma\}\!\} \ (n \text{ argument curried function})$$

   for all description type σ and integer n.

In order to represent cyclic (recursive) database descriptions, we define the set of *description constructor constants*. This requires several steps. The set of *constructor expressions* (ranged over by c) is given by the following syntax:

$$c ::= x \mid Record^{(l_1,\ldots,l_n)}(c_1,\cdots,c_n) \mid Variant^{l}(c) \mid Set^{n}(c_1,\cdots,c_n) \mid (rec\ x.\,c).$$

x in $(rec\ x.\,c)$ is a bound variable. As for lambda terms, we write $FV(c)$ for the set of free variables in c and $c[c_1/x]$ for the constructor expression obtained from c by replacing x by c_1 with necessary bound variable renaming. A constructor expression c has a description type σ under a type assignment $\mathcal{A}$, denoted by $\mathcal{A}\ \rhd c : \sigma$, if it is derived by the following typing rules:

(VAR) $\quad \mathcal{A} \rhd x : \sigma \quad$ if $\mathcal{A}(x) = \sigma$

(RECORD) $$\frac{\mathcal{A} \rhd c_1 : \sigma_1 \ \cdots\ \mathcal{A} \rhd c_n : \sigma_n}{\mathcal{A} \rhd Record^{(l_1,\ldots,l_n)}(c_1,\cdots,c_n) : [l_1 : \sigma_1,\ldots,l_n : \sigma_n]}$$

(VARIANT) $$\frac{\mathcal{A} \rhd c : \sigma_1}{\mathcal{A} \rhd Variant^{l}(c) : \sigma_2}$$ if σ_2 is a variant type containing $l : \sigma_1$

(SET) $$\frac{\mathcal{A} \rhd c_1 : \sigma \ \cdots\ \mathcal{A} \rhd c_n : \sigma}{\mathcal{A} \rhd Set^{n}(c_1,\cdots,c_n) : \{\!|\sigma|\!\}}$$

(REC) $$\frac{\mathcal{A}\{x := \sigma\} \rhd c : \sigma}{\mathcal{A} \rhd (rec\ x.\,c) : \sigma}$$

We write $\mathbf{CE} \vdash \mathcal{A} \rhd c : \sigma$ if $\mathcal{A} \rhd c : \sigma$ is derivable in the above proof system. For this typing system the following properties are proved as in lemma 3.5 and lemma 3.8:

Lemma 5.1 *If* $\mathbf{CE} \vdash \mathcal{A} \rhd c : \sigma$ *then* $\mathbf{CE} \vdash \mathcal{A}\restriction^{FV(c)} \rhd c : \sigma$. *If* $\mathbf{CE} \vdash \mathcal{A} \rhd c : \sigma$ *and* $x \notin dom(\mathcal{A})$ *then* $\mathbf{CE} \vdash \mathcal{A}\{x := \sigma'\} \rhd c : \sigma$ *for any* σ'. ∎

Lemma 5.2 *If* $\mathbf{CE} \vdash \mathcal{A}\{x := \sigma_1\} \rhd c : \sigma_2$ *and* $\mathbf{CE} \vdash \mathcal{A} \rhd c' : \sigma_1$ *then* $\mathbf{CE} \vdash \mathcal{A} \rhd c[c'/x] : \sigma_2$. ∎

By this lemma and the rule (REC), we have:

Proposition 5.2 *If* $\mathbf{CE} \vdash \mathcal{A} \rhd (rec\ x.\,c) : \sigma$ *then* $\mathbf{CE} \vdash \mathcal{A} \rhd c[(rec\ x.\,c)/x] : \sigma$. ∎

A constructor expression c is *proper* if it is not a variable and if $c \equiv (rec\ x.\,c')$ then c' is one of the form $Record^{(l_1,\ldots,l_n)}(c_1,\cdots,c_n)$, $Varinat^{l}(c_1)$ or $Set^{n}(c_1,\cdots,c_n)$. The set of description constructor constants is now defined as the set:

$$\{c^{(x_1,\ldots,x_n)} : \sigma_1 \to \cdots \to \sigma_n \to \sigma |$$
$$FV(c) = \{x_1, \ldots, x_n\},\ c \text{ is proper and } \boldsymbol{CE} \vdash \{x_1 := \sigma_1, \ldots, x_n := \sigma_n\} \rhd c : \sigma\}$$

For cyclic description constructor constants, the following property hold.

Proposition 5.3 *If* $(rec\ x.\, c)^{(x_1,\ldots,x_n)} : \sigma$ *is a description constructor constant then so is* $c[(rec\ x.\, c)/x]^{(x_1,\ldots,x_n)} : \sigma$.

Proof By $FV((rec\ x.\, c)) = FV(c[(rec\ x.\, x)/x])$ and proposition 5.2. ∎

This property corresponds to unfolding of a recursive definition.

Cyclic description constructors should not be confused with fixed point constructors such as $(fix\ x\ e)$ in ML [78]. Cyclic descriptions denote regular trees of descriptions we have developed in chapter 4. As an example, consider the cyclic description:

$$((rec\ x.\, Record^{(head,tail)}(h, x))^{(h)}\ 1) : (rec\ v.\, [Head : int, Tail : v]).$$

This denotes a regular tree represented by a finite graph and therefore the evaluation of this expression always terminates under any evaluation strategy. On the other hand

$$fix\ x.\, [Head = 1, Tail = x]$$

should denotes a fixed point of the function $\lambda x.\, [Head = 1, Tail = x]$ (which is also definable in Machiavelli using a fixed point combinator). If the interpretation of the function abstraction is "strict" then the latter denotes "bottom". Operationally this corresponds to the fact that the evaluation of the latter expression diverges under the "call-by-value" evaluation. As we have pointed out in section 3.5, fixed point combinators are definable in our language and therefore we do not need to include fixed point constructors (or equivalently special constants $Y : (\tau \to \tau) \to \tau$). We will comment more on this topic when we will discuss recursive function definitions in Machiavelli (subsection 5.6.1).

Finally we define constants for operations on these data structures:

1. Field selectors:

$$select^l : \tau_1 \to \tau_2$$

for all l and pair of types τ_1, τ_2 such that τ_1 is a record type containing $l : \tau_2$.

2. Field modification:
$$modify^l : \tau_1 \rightarrow \tau_2 \rightarrow \tau_1$$
for all l and pair of types τ_1, τ_2 such that τ_1 is a record type containing $l : \tau_2$.

3. Case analysis for variants:
$$case^{(l_1,\ldots,l_n)} : \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle \rightarrow (\tau_1 \rightarrow \tau) \rightarrow \cdots \rightarrow (\tau_n \rightarrow \tau) \rightarrow \tau$$
for all finite sequences of (distinct) $l_1, \ldots, l_n$ and all types $\tau_1, \ldots, \tau_n, \tau$.

4. Set unions:
$$union : \{\!\{\sigma\}\!\} \rightarrow \{\!\{\sigma\}\!\} \rightarrow \{\!\{\sigma\}\!\}$$
for all description types σ.

5. Cartesian product of sets:
$$prod^n : \{\!\{\sigma_1\}\!\} \rightarrow \cdots \rightarrow \{\!\{\sigma_n\}\!\} \rightarrow \{\!\{\sigma_1 \times \cdots \times \sigma_n\}\!\}$$
for all n and description types $\sigma_1, \ldots, \sigma_n$.

6. Mapping a function over a set:
$$map : (\sigma_1 \rightarrow \sigma_2) \rightarrow \{\!\{\sigma_1\}\!\} \rightarrow \{\!\{\sigma_2\}\!\}$$
for all description types σ_1, σ_2.

7. Join of descriptions:
$$join : \sigma_1 \rightarrow \sigma_2 \rightarrow \sigma_3$$
for all description types $\sigma_1, \sigma_2, \sigma_3$ such that $\sigma_3 = \sigma_1 \sqcup \sigma_2$.

8. Consistency check for two descriptions:
$$con : \sigma_1 \rightarrow \sigma_2 \rightarrow bool$$
for all description types σ_1, σ_2 such that $\sigma_1 \sqcup \sigma_2$ exists.

9. Projection of descriptions:
$$proj^{\sigma} : \sigma_1 \rightarrow \sigma$$
for all description types σ, σ_1 such that $\sigma \leqslant \sigma_1$.

10. Equality on descriptions:
$$eq : \sigma \rightarrow \sigma \rightarrow bool.$$

We define the set *Consts* of constants of Machiavelli to be the set of all the above typed constants (i.e. pairs of a constant symbol and a type) and the set of typed atomic values and standard primitives on atomic types (such as addition on integers and conditional on boolean).

Definition 5.1 (Set of Raw Terms of Machiavelli) *The set of raw terms of Machiavelli (ranged over by e) is the set given by the following syntax:*

$$e ::= c \mid x \mid \lambda x.\, e \mid (e\ e) \mid \textbf{let } x = e \textbf{ in } e \textbf{ end}$$

where c stands for the set of symbols $\{c \mid \exists\tau.\, c : \tau \in Consts\}$.

Note that this definition is an instance of the definition of the set of raw terms of ML (definition 3.21).

For the raw terms, we use the following syntactic shorthands:

$[l_1 = e_1, \ldots, l_n = e_n]$	$\Leftrightarrow$	$record^{(l_1,\ldots,l_n)}(e_1, \cdots, e_n)$,
$(e_1, \ldots, e_n)$	$\Leftrightarrow$	$record^{(\#1,\ldots,\#1)}(e_1, \cdots, e_n)$,
$e.l$	$\Leftrightarrow$	$select^{l}(e)$,
$\langle l = e \rangle$	$\Leftrightarrow$	$variant^{l}(e)$,
$(\textbf{case } e \textbf{ of } l_1 \Rightarrow e_1, \ldots, l_1 \Rightarrow e_1)$	$\Leftrightarrow$	$case^{(l_1,\ldots,l_n)}(e)(e_1)\cdots(e_n)$,
$(\textbf{rec } x.\, e)$	$\Leftrightarrow$	$(\textbf{rec } x.\, c^{(x_1,\ldots,x_n)})(e_1)\ldots(e_n)$ where $e \equiv c^{(x_1,\ldots,x_n)}[e_1/x_1, \ldots, e_n/x_n]$.

In examples we also use the following syntactic sugar for case statements:

$$(case\ e\ of\ \langle l_1 = x_1\rangle \Rightarrow e_1, \ldots, \langle l_n = x_n\rangle \Rightarrow e_n) \Leftrightarrow$$
$$(case\ e\ of\ l_1 \Rightarrow \lambda x_1.\, e_1, \ldots, l_n \Rightarrow \lambda x_n.\, e_1).$$

It should be noted that new notations we have introduced above are not new term constructors. The abstract syntax of raw terms remans the same (definition 3.21). Later we will give an alternative definition where those new notations are introduced as raw terms constructors.

5.2.3 The Proof System for Typings

Since the definition of raw terms and types are instances of the general definitions of ML we gave in section 3.5, the definition for the proof system for typings of Machiavelli is the same as that of ML (definition 3.22). We repeat the typing derivation system below:

(CONST) $\quad \mathcal{A} \rhd c : \tau \quad$ if $c : \tau \in \mathit{Consts}$

(VAR) $\quad \mathcal{A} \rhd x : \tau \quad$ if $\mathcal{A}(x) = \tau$

(APP) $$\frac{\mathcal{A} \rhd e_1 : \tau_1 \rightarrow \tau_2 \qquad \mathcal{A} \rhd e_2 : \tau_1}{\mathcal{A} \rhd (e_1\ e_2) : \tau_2}$$

(ABS) $$\frac{\mathcal{A}\{x := \tau_1\} \rhd e : \tau_2}{\mathcal{A} \rhd \lambda x.\, e : \tau_1 \rightarrow \tau_2}$$

(LET) $$\frac{\mathcal{A} \rhd e_1[e_2/x] : \tau \qquad \mathcal{A} \rhd e_2 : \tau'}{\mathcal{A} \rhd \textbf{let } x = e_2 \textbf{ in } e_1 \textbf{ end} : \tau}$$

We write $MC \vdash \mathcal{A} \rhd e : \tau$ if $\mathcal{A} \rhd e : \tau$ is derivable.

5.3 Alternative Presentation of Raw Terms and Typings

The above presentation of raw terms has the technical advantage that it significantly simplifies the presentations of the typing inference system, type inference algorithm and the semantics of the language. In particular, many results about the semantics can be directly applied to Machiavelli without re-proving them by adding cases for new term constructors. However, programming languages are usually defined using raw term constructors, which may yield a more intuitive and readable definition. For this reason, we give an alternative presentation for the set of raw terms using raw term constructors:

$$\begin{aligned} e \ ::= \ & c^{\tau} \mid x \mid \lambda x.\, x \mid (e\ e) \mid \textbf{let } x = e \textbf{ in } e \textbf{ end} \mid \\ & [l = e, \ldots, l = e] \mid e.l \mid \textbf{modify}(e, l, e) \mid \\ & \langle l = e \rangle \mid (\textbf{case } e \textbf{ of } l \Rightarrow e, \ldots, l \Rightarrow e) \mid \\ & \{\!\!\{ e, \ldots, e \}\!\!\} \mid \textbf{union}(e, e) \mid \textbf{prod}(e, \ldots, e) \mid \textbf{map}(e, e) \mid \\ & \textbf{join}(e, e) \mid \textbf{project}(e, \sigma) \mid \textbf{con}(e, e) \mid \textbf{eq}(e, e) \mid (\textbf{rec } v.\, e(v)) \end{aligned}$$

where c^τ stands for atomic constants and operations on base types and $e(v)$ in $(\mathbf{rec}\ v.e(v))$ stands for a raw term possibly containing symbol v.

For the alternative presentation of the set of raw terms using raw term constructors, the equivalent proof system can be given by the following set of rules:

(CONST) $\quad \mathcal{A} \rhd c^\tau : \tau \quad$ if $c : \tau \in Consts$

(VAR) $\quad \mathcal{A} \rhd x : \tau \quad$ if $\mathcal{A}(x) = \tau$

(APP) $$\frac{\mathcal{A} \rhd e_1 : \tau_1 \rightarrow \tau_2 \qquad \mathcal{A} \rhd e_2 : \tau_1}{\mathcal{A} \rhd (e_1\ e_2) : \tau_2}$$

(ABS) $$\frac{\mathcal{A}\{x := \tau_1\} \rhd e : \tau_2}{\mathcal{A} \rhd \lambda x.e : \tau_1 \rightarrow \tau_2}$$

(LET) $$\frac{\mathcal{A} \rhd e_1[e_2/x] : \tau \qquad \mathcal{A} \rhd e_2 : \tau'}{\mathcal{A} \rhd \mathbf{let}\ x = e_2\ \mathbf{in}\ e_1\ \mathbf{end} : \tau}$$

(RECORD) $$\frac{\mathcal{A} \rhd e_1 : \tau_1, \ldots, \mathcal{A} \rhd e_n : \tau_n}{\mathcal{A} \rhd [l_1 = e_1, \ldots, l_n = e_n] : [l_1 : \tau_1, \ldots, l_n : \tau_n]}$$

(DOT) $$\frac{\mathcal{A} \rhd e : \tau_1}{\mathcal{A} \rhd e.l : \tau_2} \quad \text{if } \tau_1 \text{ is a record type containing } l : \tau_2$$

(MODIFY) $$\frac{\mathcal{A} \rhd e_1 : \tau_1 \qquad \mathcal{A} \rhd e_2 : \tau_2}{\mathcal{A} \rhd \mathbf{modify}(e_1, l, e_2) : \tau_1} \quad \text{if } \tau_1 \text{ is a record type containing } l : \tau_2$$

(VARIANT) $$\frac{\mathcal{A} \rhd e : \tau_1}{\mathcal{A} \rhd < l = e >: \tau_2} \quad \text{if } \tau_2 \text{ is a variant type containing } l : \tau_1$$

(CASE) $$\frac{\mathcal{A} \rhd e : \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle \qquad \mathcal{A} \rhd e_1 : \tau_1 \rightarrow \tau \cdots \mathcal{A} \rhd e_n : \tau_n \rightarrow \tau}{\mathcal{A} \rhd (\mathbf{case}\ e\ \mathbf{of}\ l_1 \Rightarrow e_1, \ldots, l_n \Rightarrow e_n) : \tau}$$

(SET) $$\frac{\mathcal{A} \rhd e_1 : \sigma \cdots \mathcal{A} \rhd e_n : \sigma}{\mathcal{A} \rhd \{\!\{e_1, \ldots, e_n\}\!\} : \{\!\{\sigma\}\!\}}$$

(UNION) $$\frac{\mathcal{A} \rhd e_1 : \{\!\{\sigma\}\!\} \qquad \mathcal{A} \rhd e_2 : \{\!\{\sigma\}\!\}}{\mathcal{A} \rhd \mathbf{union}(e_1, e_2) : \{\!\{\sigma\}\!\}}$$

(PROD) $$\frac{\mathcal{A} \rhd e_1 : \{\!\{\sigma_1\}\!\} \cdots \mathcal{A} \rhd e_n : \{\!\{\sigma_n\}\!\}}{\mathcal{A} \rhd \mathbf{prod}(e_1, \ldots, e_n) : \{\!\{\sigma_1 \times \cdots \times \sigma_n\}\!\}}$$

(MAP) $$\frac{\mathcal{A} \rhd e_1 : \sigma_1 \rightarrow \sigma_2 \qquad \mathcal{A} \rhd e_2 : \{\!\{\sigma_1\}\!\}}{\mathcal{A} \rhd \mathbf{map}(e_1, e_2) : \{\!\{\sigma_2\}\!\}}$$

(JOIN) $$\frac{\mathcal{A} \rhd e_1 : \sigma_1 \qquad \mathcal{A} \rhd e_2 : \sigma_2}{\mathcal{A} \rhd \mathbf{join}(e_1, e_2) : \sigma_3} \qquad \text{if } \sigma_3 = \sigma_1 \sqcup \sigma_2$$

(PROJECT) $$\frac{\mathcal{A} \rhd e : \sigma_1}{\mathcal{A} \rhd \mathbf{project}(e_1, \sigma_2) : \sigma_2} \qquad \text{if } \sigma_2 \leqslant \sigma_1$$

(CON) $$\frac{\mathcal{A} \rhd e_1 : \sigma_1 \qquad \mathcal{A} \rhd e_2 : \sigma_2}{\mathcal{A} \rhd \mathbf{con}(e_1, e_2) : \mathit{bool}} \qquad \text{if } \sigma_1 \sqcup \sigma_2 \text{ exists}$$

(EQ) $$\frac{\mathcal{A} \rhd e_1 : \sigma \qquad \mathcal{A} \rhd e_2 : \sigma}{\mathcal{A} \rhd \mathbf{eq}(e_1, e_2) : \mathit{bool}}$$

(REC) $$\frac{\mathcal{A}\{v := \sigma\} \rhd e(v) : \sigma}{\mathcal{A} \rhd (\mathbf{rec}\ v\ e(v)) : \sigma}$$

The reader is encouraged to check that the set of all derivable typings are indeed isomorphic to those in the proof system based on the constructor constants. In examples that follows we use this representation of raw terms, but we continue to use the previous representation of raw terms and typings in definitions of formal property of Machiavelli.

5.4 Type Inference Problem

In our view, terms of ML are typing schemes representing sets of typings (definition 3.10). As we have analyzed in section 3.2, type-checking and type inference depend on the existence of a principal typing scheme for any typable raw terms. In order to preserve this property, it was needed for constants to have a principal typing scheme (assumption 3.1). As we have seen in section 5.1, however, field selection select^l does not have a principal typing scheme. The type inference algorithm for ML is therefore not directly applicable to our language.

The reason that select^l does not have a principal typing scheme is the condition associated with the typing rule. The conventional notion of type-schemes cannot represent the set of all types of the forms $\tau_1 \rightarrow \tau_2$ such that τ_1 is a record type containing $l : \tau_2$ field. The same problem exists for the constants *variant*^{l}, *join* and *proj*$^\sigma$ (and description constructor constants containing them). The family of description constructor constants are restricted to description types, which is also a form of condition that cannot be represented by conventional type-schemes. We solve this problem by extending the notion of typing schemes to allow *conditions* on substitutions.

5.4.1 Conditional Typing schemes

Let *Tvar* be a set of type variables (ranged over by t).

Definition 5.2 *The set of type-schemes, Tscheme ranged over by ρ, of Machiavelli is the set of regular trees represented by the following syntax:*

$$\rho ::= t \mid b \mid \rho \rightarrow \rho \mid [l : \rho, \ldots, l : \rho] \mid \langle l : \rho, \ldots, l : \rho \rangle \mid (\mathbf{rec}\ v.\, \rho(v)).$$

On substitutions of type-schemes, we define the following five forms of conditions.

Definition 5.3 *A condition is one of the following formula:*

1. $dtype(\rho)$,
2. $[\rho_1 \ni l : \rho_2]$,
3. $\langle \rho_1 \ni l : \rho_2 \rangle$,
4. $\rho_1 = jointype(\rho_2, \rho_3)$,
5. $lessthan(\sigma, \rho)$.

The first condition states that ρ should be a description type. The other four conditions represent the conditions associated with the constant functions *select*^{l}, *variant*^{l}, *join* and *proj*$^\sigma$. We call these four forms of conditions *record*, *variant*, *join* and *projection-conditions* respectively. σ in a projection condition $lessthan(\sigma, \rho)$ is called a *target type* of the projection condition. The meanings of these conditions are defined by the following *satisfiability relation*:

Definition 5.4 *A substitution θ satisfies a condition c:*

1. *if $c \equiv dtype(\rho)$ and $\theta(\rho) \in \mathbf{Dtype}^\infty$,*
2. *if $c \equiv [\rho_1 \ni l : \rho_2]$ and $\theta(\rho_1)$ is a record type containing $l : \theta(\rho_2)$, (i.e. $\theta(\rho_1)$ is a regular tree such that $\theta(\rho_1)(\epsilon) = Record$, $l \in dom(\theta(\rho_1))$ and $\theta(\rho_1)/l = \theta(\rho_2)$),*

3. *if $c \equiv \langle \rho_1 \ni l : \rho_2 \rangle$ and $\theta(\rho_1)$ is a variant type containing $l : \theta(\rho_2)$,*

4. *if $c \equiv (\rho_1 = jointype(\rho_2, \rho_3))$ and $\theta(\rho_1) \in \boldsymbol{Dtype}^\infty$, $\theta(\rho_2) \in \boldsymbol{Dtype}^\infty$, $\theta(\rho_3) \in \boldsymbol{Dtype}^\infty$ and $\theta(\rho_1) = \theta(\rho_1) \sqcup \theta(\rho_3)$, where $\sqcup$ is the least upper bound of the ordering on $\boldsymbol{Dtype}^\infty$ we have defined in subsection 4.4.1,*

5. *if $c \equiv lessthan(\sigma, \rho)$ and $\theta(\rho) \in \boldsymbol{Dtype}^\infty$, $\sigma \leqslant \theta(\rho)$, where $\leqslant$ is the ordering on $\boldsymbol{Dtype}^\infty$ we have defined in section 4.4.1.*

A substitution θ satisfies a set of condition C iff θ satisfies each $c \in C$.

The condition of the form $dtype(\rho)$ is similar to the condition associated with the *equality types* in Standard ML[2] and can be implicitly represented by introducing distinct type variables and meta-symbols for description types. For this purpose we define the set of *description type-schemes*, a subset of type-schemes that represents only description types. We divide the set $Tvar$ of type variables into two sets; the set of *unconditional type variables* (ranged over by u) and the set of *description type variables* (ranged over by s). We continue to use t for a type variable ranging over arbitrary elements in $Tvar$. The set of description type-schemes (ranged over by δ) is the set of regular trees represented by the following syntax:

$$\delta ::= s \mid b \mid [l : \delta, \ldots, l : \delta] \mid \langle l : \delta, \ldots, l : \delta \rangle \mid (rec\ v.\, \delta(v)).$$

In what follows, we implicitly regard δ as a type-scheme associated with the condition $dtype(\delta)$ and write $\theta(\delta)$ assuming that θ is a substitution satisfying the condition $\theta(\delta) \in \boldsymbol{Dtype}^\infty$. This is tacitly done in Standard ML implementation to incorporate equality primitive in ML's original type inference mechanism. For example, we have the following typing scheme in Standard ML:

$$\textbf{fn}\ x \Rightarrow \textbf{fn}\ y \Rightarrow x = y : {''a} \rightarrow {''a} \rightarrow bool$$

where $''a$ is an *equality type variable*, which is regarded as a type variable with the condition that substitutions are restricted to those θ such that $\theta(''a)$ is a type that does not contain function type constructor.[3]

Definition 5.5 (Conditional Typing scheme) *A conditional typing scheme is a formula of the form $C, \Sigma \rhd e : \rho$ such that for any ground substitution θ for C, Σ and ρ, if θ satisfies C then $MC \vdash \theta(\Sigma) \rhd e : \theta(\rho)$. A conditional typing scheme $C, \Sigma \rhd e : \rho$ is principal if for any typing $\mathcal{A} \rhd e : \tau$ there is a substitution θ such that θ satisfies C and $\mathcal{A}\restriction^{dom(\Sigma)} = \theta(\Sigma)$, $\tau = \theta(\rho)$.*

[2]This similarity was pointed out to me by Robin Milner

[3]With the existence of reference types, the precise condition is that $\theta(''a)$ does not contain function type constructor outside the scope of any reference type constructor.

The following theorem extends theorem 3.2 for ML.

Theorem 5.1 *There is an algorithm $\mathcal{CTS}$ which, given any raw term e, yields either failure or (C, Σ, ρ) such that if $\mathcal{CTS} = (C, \Sigma, \rho)$ then $C, \Sigma \rhd e : \rho$ is a principal conditional typing scheme otherwise e has no typing.*

Proof We assume that the unification algorithm $\mathcal{U}$ (section 2.2) on regular trees is extended to an algorithm to unify a finite sequence of regular trees simultaneously. Such an algorithm can be easily constructed by the unification algorithm that unifies two regular trees. As before we consider Σ as a tree by using a linear order on variables (as in theorem 3.2).

We first show that all constants have a principal conditional typing scheme. For constants other than description constructor constants, we have the following principal conditional typing schemes representing their set of associated types:

$$\begin{array}{ll}
MC \vdash & \emptyset, \emptyset \rhd c^{\tau} : \tau \text{ (for all atomic constants and operations on base types)},\\
MC \vdash & \emptyset, \emptyset \rhd record^{(l_1,\dots,l_n)} : u_1 \to \cdots u_n \to [l_1 : u_1, \dots, l_n : u_n],\\
MC \vdash & \{\langle u_1 \ni l : u_2\rangle\}, \emptyset \rhd variant^{l} : u_2 \to u_1,\\
MC \vdash & \emptyset, \emptyset \rhd set^{n} : s \to \cdots s \to \{\!\{s\}\!\} \text{ (n arguments)},\\
MC \vdash & \{[u_1 \ni l : u_2]\}, \emptyset \rhd select^{l} : u_1 \to u_2,\\
MC \vdash & \{[u_1 \ni l : u_2]\}, \emptyset \rhd modify^{l} : u_1 \to u_2 \to u_1,\\
MC \vdash & \emptyset, \emptyset \rhd case^{(l_1,\dots,l_n)} : \langle l_1 : u_1, \dots, l_n : u_n\rangle \to (u_1 \to u) \to \cdots \to (u_n \to u) \to u,\\
MC \vdash & \emptyset, \emptyset \rhd union : \{\!\{s\}\!\} \to \{\!\{s\}\!\} \to \{\!\{s\}\!\},\\
MC \vdash & \emptyset, \emptyset \rhd prod^{n} : \{\!\{s_1\}\!\} \to \cdots \to \{\!\{s_n\}\!\} \to \{\!\{s_1 \times \cdots \times s_n\}\!\},\\
MC \vdash & \emptyset, \emptyset \rhd map : (s_1 \to s_2) \to \{\!\{s_1\}\!\} \to \{\!\{s_2\}\!\},\\
MC \vdash & \{s_3 = jointype(s_1, s_2)\}, \emptyset \rhd join : s_1 \to s_2 \to s_3,\\
MC \vdash & \{s_3 = jointype(s_1, s_2)\}, \emptyset \rhd con : s_1 \to s_2 \to bool,\\
MC \vdash & \{lessthan(\sigma, s)\}, \emptyset \rhd proj^{\sigma} : s \to \sigma,\\
MC \vdash & \emptyset, \emptyset \rhd eq : s \to s \to bool.
\end{array}$$

By the definition of these constants and the satisfiability of conditions, all the above are clearly principal conditional typing schemes.

We next show the same property for description constructor constants. We first define the algorithm $\mathcal{CE}$ to compute a conditional typing scheme of constructor expressions:

Algorithm $\mathcal{CE}$

$$\mathcal{CE}(c) = (C, \Sigma, \delta) \text{ where}$$

```
(1) Case $c \equiv x$:
        $C = \emptyset$
        $\Sigma = \{x := s\}$ ($s$ fresh)
        $\delta = s$

(2) Case $c \equiv Record^{(l_1,\ldots,l_n)}(c_1, \cdots, c_n)$:
    let
        $(C_1, \Sigma_1, \delta_1) = \mathcal{CE}(c_1)$
            $\vdots$
        $(C_n, \Sigma_n, \delta_n) = \mathcal{CE}(c_n)$
        $X = dom(\Sigma_1) \cup \cdots \cup dom(\Sigma_n)$
        $\Sigma_1' = \Sigma_1\{x_1^1 := s_1^1, \ldots, x_k^1 := s_k^1\}$ ($s_i^1, 1 \leq i \leq k$ fresh)
            where $\{x_1^1, \ldots, x_k^1\} = X \setminus dom(\Sigma_1)$
            $\vdots$
        $\Sigma_n' = \Sigma_1\{x_1^n := s_1^n, \ldots, x_l^n := s_l^n\}$ ($s_i^n, 1 \leq i \leq l$ fresh)
            where $\{x_1^n, \ldots, x_l^n\} = X \setminus dom(\Sigma_n)$
        $\theta = \mathcal{U}(\Sigma_1', \ldots, \Sigma_n')$
    in
        $C = \theta(C_1 \cup \cdots \cup C_n)$
        $\Sigma = \theta(\Sigma_1')$
        $\delta = [l_1 : \theta(\delta_1), \ldots, l_n : \theta(\delta_n)]$

(3) Case $c \equiv Variant^l(c')$:
    let
        $(C_1, \Sigma_1, \delta_1) = \mathcal{CE}(c')$
    in
        $C = C_1 \cup \{\langle s \ni l : \delta_1 \rangle\}$ ($s$ fresh)
        $\Sigma = \Sigma_1$
        $\delta = s$

(4) Case $e \equiv Set^n(c_1, \cdots, c_n)$:
    let
    $(C_1, \Sigma_1, \delta_1) = \mathcal{CE}(c_1)$
        $\vdots$
    $(C_n, \Sigma_n, \delta_n) = \mathcal{CE}(c_n)$
    $X = dom(\Sigma_1) \cup \cdots \cup dom(\Sigma_n)$
    $\Sigma'_1 = \Sigma_1\{x^1_1 := s^1_1, \ldots, x^1_k := s^1_k\}$ ($s^1_i, 1 \leq i \leq k$ fresh)
        where $\{x^1_1, \ldots, x^1_k\} = X \setminus dom(\Sigma_1)$
        $\vdots$
    $\Sigma'_n = \Sigma_1\{x^n_1 := s^n_1, \ldots, x^n_l := s^n_l\}$ ($s^1_i, 1 \leq i \leq l$ fresh)
        where $\{x^n_1, \ldots, x^n_l\} = X \setminus dom(\Sigma_n)$
    $\theta = \mathcal{U}((\Sigma'_1, \delta_1), \ldots, (\Sigma'_n, \delta_n))$
in
    $C = \theta(C_1 \cup \cdots \cup C_n)$
    $\Sigma = \theta(\Sigma'_1)$
    $\delta = \{\!\{\theta(\delta_1)\}\!\}$

(5) Case $c \equiv (rec\ x.\, c_1)$:
  let
    $(C_1, \Sigma_1, \delta_1) = \mathcal{CE}(c_1)$
  in
    if $x \in dom(\Sigma_1)$ then
        let
            $\theta = \mathcal{U}(\Sigma_1(x), \delta_1)$
        in
            $C = \theta(C_1)$
            $\Sigma = \theta(\Sigma_1\restriction^{dom(\Sigma_1)\setminus\{x\}})$
            $\delta = \theta(\delta_1)$
    else
        $C = C_1$
        $\Sigma = \Sigma_1$
        $\delta = \delta_1$
```

For this algorithm, the following property holds, whose proof is similar to that of lemma 3.4:

Lemma 5.3 *If* $\mathcal{CE}(c) = (C, \Sigma, \delta)$ *then* $dom(\Sigma) = FV(c)$. ▮

The principal conditional typing scheme for a description constructor constant $c^{(x_1,\ldots,x_n)}$ is then given as:

$$C, \emptyset \rhd c^{(x_1,\ldots,x_n)} : \Sigma(x_1) \to \cdots \to \Sigma(x_n) \to \delta$$

where $(C, \Sigma, \delta) = \mathcal{CE}(c)$.

We show that the above algorithm computes a principal conditional typing scheme for all description constructor constants. By definition of $c^{(x_1,\ldots,x_n)}$ and lemma 5.1, 5.3, it suffices to show that (1) if $\boldsymbol{CE} \vdash \mathcal{A} \rhd c : \sigma$ then $\mathcal{CE}(c) = (C, \Sigma, \delta)$ and there is some substitution θ such that θ satisfies C and $\theta(\delta) = \sigma$, $\theta(\Sigma) = \mathcal{A}\restriction^{dom(\Sigma)}$, and (2) if $\mathcal{CE}(c) = (C, \Sigma, \delta)$ then for any substitution θ ground for C and Σ if θ satisfies C, then $\boldsymbol{CE} \vdash \theta(\Sigma) \rhd c : \theta(\delta)$. Since the set of constructor expressions are inductively defined terms (not regular trees), we can show the above properties by induction on the structure of c. The basis is trivial. The induction step is by cases. In the following proof, $C, C_i, \ldots, \Sigma, \Sigma_i \ldots, \rho, \rho_i \ldots, \theta$ refer to those described in the algorithm $\mathcal{CE}$.

1. Case $c \equiv Record^{(l_1,\ldots,l_n)}(c_1, \cdots, c_n)$: Suppose $\boldsymbol{CE} \vdash \mathcal{A} \rhd Record^{(l_1,\ldots,l_n)}(c_1, \cdots, c_2) : \sigma$ for some $\mathcal{A}$ and σ. Then by the typing rules, σ must be of the form $[l_1 : \sigma_1, \ldots, l_n : \sigma_n]$. By the induction hypothesis, for each $1 \leq i \leq n$, $\mathcal{CE}(c_i) = (C_i, \Sigma_i, \delta_i)$ succeeds and there is some η_i that satisfies C_i and $\eta_i(\Sigma_i) = \mathcal{A}\restriction^{dom(\Sigma_i)}$, $\eta_i(\delta_i) = \sigma_i$. Let $\eta'_i = \eta_i\restriction^{FTV((C_i,\Sigma_i,\delta_i))}$ where $FTV(X)$ is the set of type variables in X. Then $\eta'_i(\Sigma_i) = \mathcal{A}\restriction^{dom(\Sigma_i)}$, $\eta'_i(\delta_i) = \sigma_i$. By the definition of $\mathcal{CE}$ the sets $FTV((C_i, \Sigma_i, \delta_i))$ of type variables are all disjoints. Therefore $\eta = \eta'_1 \cup \cdots \eta'_n$ (as graph) is a well defined substitution and satisfies the following properties: for all $1 \leq i \leq n$, η satisfies C_i, $\eta(\Sigma_i) = \mathcal{A}\restriction^{dom(\Sigma_i)}$ and $\eta(\delta_i) = \sigma_i$. Then by the property of unification, $\mathcal{U}(\Sigma'_1, \ldots, \Sigma'_n)$ in the algorithm succeeds and return θ such that $\eta \precsim \theta$. Let η' be a substitution such that $\eta = \eta' \circ \theta$. Then η' satisfies $\theta(C_1 \cup \cdots \cup C_n)$, $\eta'(\theta(\Sigma'_1)) = \mathcal{A}\restriction^{dom(\Sigma)}$, $\eta'(\theta(\delta_i)) = \sigma_i$ (for all $1 \leq i \leq n$) and therefore $\eta'(\delta) = \sigma$.

   Conversely suppose $\mathcal{CE}(Record^{(l_1,\ldots,l_n)}(c_1, \cdots, c_n)) = (C, \Sigma, \delta)$. Then by the definition of $\mathcal{CE}$ $\mathcal{CE}(c_i) = (C_i, \Sigma_i, \delta_i)$ for all $1 \leq i \leq n$ and there is a substitution θ such that $C = \theta(C_1 \cup \cdots \cup C_n)$, $\Sigma\restriction^{dom(\Sigma_i)} = \theta(\Sigma_i)$, $\delta = [l_1 : \theta(\delta_i), \ldots, l_n : \theta(\delta_n)]$. Let η be any substitution ground for C, Σ that satisfies C. Then $\eta \circ \theta$ is a substitution ground for all $C_i, \Sigma_i, 1 \leq i \leq n$ that satisfies all $C_i, 1 \leq i \leq n$. Then by the induction hypothesis and by lemma 5.1, $\boldsymbol{CE} \vdash \eta(\Sigma) \rhd c_i : \eta(\theta(\delta_i))$. By the rule (RECORD), $\boldsymbol{CE} \vdash \eta(\Sigma) \rhd c : \eta(\delta)$.

2. Case $c \equiv Variant^l(c')$: Suppose $\boldsymbol{CE} \vdash \mathcal{A} \rhd Variant^l(c') : \sigma$ for some $\mathcal{A}, \sigma$. Then by the typing rules, σ must be a variant type containing $l : \sigma_1$ such that $\boldsymbol{CE} \vdash \mathcal{A} \rhd c_1 : \sigma_1$. By the induction hypothesis, $\mathcal{CE}(c') = (C_1, \Sigma_1, \delta_1)$ succeeds and there is a substitution η ground for C_1, Σ_1 that satisfies C_1 and $\eta(\Sigma_1) = \mathcal{A}\restriction^{dom(\Sigma_1)}$, and $\eta(\delta_1) = \sigma_1$. Since s introduce in the

algorithm is fresh, $\eta' = \eta\{s := \sigma\}$ is a substitution satisfies all the above three conditions. But since σ is a variant type containing $l : \sigma_1$, η' also satisfies C and $\eta'(\Sigma) = \mathcal{A}\restriction^{dom(\Sigma)}$, $\eta'(\delta) = \sigma$.

Conversely, suppose $\mathcal{CE}(Variant^l(c')) = (C, \Sigma, \delta)$. By the definition of $\mathcal{CE}$, $\mathcal{CE}(c') = (C_1, \Sigma, s)$ such that $C = C_1 \cup \{(s \ni l : \delta_1)\}$. Let η be any substitution ground for C, Σ that satisfies C. Then η satisfies C_1 and $\eta(s)$ is a variant type containing $l : \eta(\delta_1)$. By the induction hypothesis, $\boldsymbol{CE} \vdash \eta(\Sigma) \rhd c' : \eta(\delta_1)$. Since $\eta(s)$ is a variant type containing $l : \eta(\delta_1)$, by the rule (VARIANT), $\boldsymbol{CE} \vdash \eta(\Sigma) \rhd Variant^l(c') : \eta(s)$.

3. Case $c \equiv Set^n(c_1, \cdots, c_n)$: Similar to the case for $e \equiv Record^{l_1, \cdots, l_n}(c_1, \cdots, c_n)$.

4. Case $c \equiv (rec\ x.\, c_1)$: Suppose $\boldsymbol{CE} \vdash \mathcal{A} \rhd (rec\ x.\, c_1) : \sigma$. Then by the typing rules, $\boldsymbol{CE} \vdash \mathcal{A}\{x := \sigma\} \rhd c_1 : \sigma$. By the induction hypothesis, $\mathcal{CE}(c_1) = (C_1, \Sigma_1, \delta_1)$ succeed and there is some η ground for C_1, Σ_1 that satisfies C_1 such that $\eta(\Sigma_1) = \mathcal{A}\{x := \sigma\}\restriction^{dom(\Sigma_1)}$, $\theta(\delta_1) = \sigma$. Suppose $x \in dom(\Sigma_1)$ then $\eta(\Sigma_1(x)) = \sigma = \eta(\delta_1)$. Therefore the unification in the algorithm $\mathcal{U}(\Sigma_1(x), \delta_1)$ succeeds and returns θ such that $\eta \precsim \theta$. Let $\eta' \circ \theta = \eta$. Then η' satisfies $\theta(C_1)$, $\eta'(\theta(\Sigma_1\restriction^{dom(\Sigma_1)\setminus\{x\}})) = \mathcal{A}\restriction^{dom(\theta(\Sigma\restriction^{dom(\Sigma_1)\setminus\{x\}}))}$, and $\eta'(\theta(\delta_1)) = \sigma$. Suppose $x \notin dom(\Sigma_1)$ then $\eta(\Sigma_1) = \mathcal{A}\{x := \sigma\}\restriction^{dom(\Sigma_1)} = \mathcal{A}\restriction^{dom(\Sigma_1)}$ and $\eta(\delta_1) = \sigma$.

   Conversely suppose $\mathcal{CE}((rec\ x.c_1)) = (C, \Sigma, \delta)$. By the definition of $\mathcal{CE}$, $\mathcal{CE}(c_1) = (C_1, \Sigma_1, \delta_1)$. Suppose $x \in dom(\Sigma_1)$. Then $\theta = \mathcal{U}(\Sigma_1(x), \delta_1)$ and $C = \theta(C_1)$, $\Sigma = \theta(\Sigma_1\restriction^{dom(\Sigma_1)\setminus\{x\}})$, $\delta = \theta(\delta_1)$. Let η be any substitution ground for Σ, C that satisfies C. Then $\eta \circ \theta$ satisfies C_1. By the induction hypothesis $\boldsymbol{CE} \vdash \eta \circ \theta(\Sigma_1) \rhd c_1 : \eta \circ \theta(\delta_1)$. But $\theta(\Sigma_1(x)) = \theta(\delta_1) = \delta$. Thus $\boldsymbol{CE} \vdash \eta(\Sigma\{x := \delta\}) \rhd c_1 : \eta(\delta)$, i.e. $\boldsymbol{CE} \vdash \eta(\Sigma)\{x := \eta(\delta)\}) \rhd c_1 : \eta(\delta)$. Then by the rule (REC), $\boldsymbol{CE} \vdash \eta(\Sigma) \rhd (rec\ x.\, c_1) : \eta(\delta)$. Suppose $x \notin dom(\Sigma_1)$. Then $\Sigma = \Sigma_1$, $C = C_1$, and $\delta = \delta_1$. Let η be any substitution ground for Σ, C that satisfies C. By the induction hypothesis $\boldsymbol{CE} \vdash \eta(\Sigma) \rhd c_1 : \eta(\delta)$. By lemma 5.1, $\boldsymbol{CE} \vdash \eta(\Sigma)\{x := \eta(\delta)\} \rhd c_1 : \eta(\delta)$. Then by the rule (REC), $\boldsymbol{CE} \vdash \eta(\Sigma) \rhd (rec\ x.\, c_1) : \eta(\delta)$.

We now define the algorithm to compute a conditional principal typing scheme for general raw terms:

Algorithm $\mathcal{CTS}$:

$$\mathcal{CTS}(e) = (C, \Sigma, \rho) \text{ where}$$

(1) Case $e \equiv c$: Given above.

```
(2) Case $e \equiv x$:
      $C = \emptyset$,
      $\Sigma = \{x := u\}$,
      $\rho = u$.

(3) Case $e \equiv (e\ e)$:
    let
      $(C_1, \Sigma_1, \rho_1) = \mathcal{CTS}(e_1)$
      $(C_2, \Sigma_2, \rho_2) = \mathcal{CTS}(e_2)$
      $\Sigma_1' = \Sigma_1\{x_1 := u_1^1, \ldots, x_n := u_n^1\}$ where
         $\{x_1, \ldots, x_n\} = dom(\Sigma_2) \setminus dom(\Sigma_1)$ $(u_1^1, \ldots, u_n^1$ fresh$)$
      $\Sigma_2' = \Sigma_2\{y_1 := u_1^2, \ldots, y_m := u_m^2\}$ where
         $\{y_1, \ldots, y_m\} = dom(\Sigma_1) \setminus dom(\Sigma_2)$ $(u_1^2, \ldots, u_m^2$ fresh$)$
      $\theta = \mathcal{U}((\Sigma_1', \rho_1), (\Sigma_2', \rho_2 \rightarrow u))$ $(u$ fresh$)$
    in
      $C = \theta(C_1) \cup \theta(C_2)$,
      $\Sigma = \theta(\Sigma_1')$,
      $\rho = \theta(u)$.

(4) Case $e \equiv \lambda x.\, e_1$:
    let
      $(C_1, \Sigma_1, \rho_1) = \mathcal{CTS}(e_1)$
    in
      if $x \in dom(\Sigma_1)$ then
         $C = C_1$,
         $\Sigma = \Sigma\restriction^{dom(\Sigma)\setminus\{x\}}$,
         $\rho = \Sigma_1(x) \rightarrow \rho_1$
      else
         $C = C_1$,
         $\Sigma = \Sigma$.
         $\rho = u \rightarrow \rho_1$ $(u$ fresh$)$.

(5) Case $e \equiv$ **let** $x = e_1$ **in** $e_2$ **end**:
    let
```

$(C_1, \Sigma_1, \rho_1) = \mathcal{CTS}(e_1)$

$(C_2, \Sigma_2, \rho_2) = \mathcal{CTS}(e_2[e_1/x])$

$\Sigma_1' = \Sigma_1\{x_1 := u_1^1, \ldots, x_n := u_n^1\}$ where

$\quad \{x_1, \ldots, x_n\} = dom(\Sigma_2) \setminus dom(\Sigma_1)$ $(u_1^1, \ldots, u_n^1$ fresh)

$\Sigma_2' = \Sigma_2\{y_1 := u_1^2, \ldots, y_m := u_m^2\}$ where

$\quad \{y_1, \ldots, y_m\} = dom(\Sigma_1) \setminus dom(\Sigma_2)$ $(u_1^2, \ldots, u_m^2$ fresh)

$\theta = \mathcal{U}(\Sigma_1', \Sigma_2')$

in

$C = \theta(C_1) \cup \theta(C_2)$,

$\Sigma = \theta(\Sigma_1')$,

$\rho = \theta(\rho_2)$.

For the correctness of the algorithm, we need to show that (1) if $\boldsymbol{MC} \vdash \mathcal{A} \rhd e : \tau$ then $\mathcal{CTS}(e) = (C, \Sigma, \rho)$ and there is a substitution θ that satisfies C and $\mathcal{A}\upharpoonright^{dom(\Sigma)} = \theta(\Sigma)$, $\tau = \theta(\rho)$ and (2) if $\mathcal{CTS}(e) = (C, \Sigma, \rho)$ then for any ground substitution θ for C, Σ, ρ if θ satisfies C then $\boldsymbol{MC} \vdash \theta(\Sigma) \rhd e : \theta(\rho)$. This is proved by induction on the structure of e. We have already proved for the case for constants. The case of x is identical to the proof for ML (theorem 3.2). Since the cases for $\lambda x. e_1$ and $(e_1\ e_2)$ does not create new conditions, these cases are proved similar to the corresponding cases in ML. The necessary new properties are implied by the corresponding induction hypothesis. ∎

5.4.2 Satisfiability of Conditions

We have shown that for any raw term e if e is typable then we can effectively construct a principal conditional typing scheme. This result is, however, not enough for complete static type-checking since a principal conditional typing scheme $C, \Sigma \rhd e : \rho$ constructed by the algorithm may not have an instance. This may happen because the set of conditions C may not be satisfiable. As an example, the algorithm computes the following conditional typing scheme for the raw term $e \equiv \lambda x. (x.l + 1, x.l\ and\ true)$:

$$\{[u \ni l : int], [u \ni l : bool]\}, \emptyset \rhd e : u \to (int \times bool)$$

where set of conditions is clearly unsatisfiable and therefore the raw term has no typing. In such a case the type system should report a type error. In the above example, it is rather trivial to detect the unsatisfiability. For general terms, however, we need to develop an algorithm to simplify a set of conditions and to test their satisfiability.

For a set of conditions C, we write $Sat(C)$ for the set $\{\theta|\theta \text{ satisfies } C\}$.

Definition 5.6 *A set of conditions C_1 is a refinement of C_2 if there is a substitution θ such that $Sat(C_2) = \{\eta \circ \theta | \eta \in Sat(C_1)\}$. θ is called a refinment substitution for C_2.*

The following property is an immediate consequence of the definition:

Proposition 5.4 *If $C, \Sigma \rhd e : \rho$ is a principal conditional typing scheme and C' is a refinement of C with a refinement substitution θ, then $C', \theta(\Sigma) \rhd e : \theta(\rho)$ is also a principal conditional typing scheme.* ▮

We would like to develop an algorithm which, given a set of conditions C, decides whether C is satisfiable or not and if it is satisfiable then computes a refinement of C and a refinement substitution θ.

If a set of conditions C only contains record conditions or variant conditions then such an algorithm exists.

Definition 5.7 *A set of conditions C is in record-variant normal form if the following properties hold:*

1. *if $[\rho \ni l : \rho'] \in C$ then ρ is a type variable,*
2. *if $\langle \rho \ni l : \rho' \rangle \in C$ the ρ is a type variable,*
3. *if $[\rho \ni l : \rho_1] \in C, [\rho \ni l : \rho_2] \in C$ then $\rho_1 = \rho_2$,*
4. *if $\langle \rho \ni l : \rho_1 \rangle \in C, \langle \rho \ni l : \rho_2 \rangle \in C$ then $\rho_1 = \rho_2$,*
5. *if $[\rho_1 \ni l_1 : \rho_2] \in C, \langle \rho_3 \ni l_2 : \rho_4 \rangle$ then $\rho_1 \not\equiv \rho_3$.*

Lemma 5.4 *If C does not contain join conditions or projection conditions and in record-variant normal form then C is satisfiable.*

Proof Consider the substitution θ satisfying the following properties: $dom(\theta) = \{t|\exists \rho. [t \ni l : \rho] \in C\} \cup \{t|\exists \rho. \langle t \ni l : \rho \rangle \in C\}$ and for all $t \in dom(\theta)$ if there is some $[t \ni l : \rho] \in C$ then $\theta(t) = [l_1 : \theta(\rho_1), \ldots, l_n : \theta(\rho_n)]$ where $\{(l_1, \rho_1), \ldots, (l_n, \rho_n)\} = \{(l, \rho)|[t \ni l : \rho] \in C\}$ otherwise there is some $\langle t \ni l : \rho \rangle \in C$ then $\theta(t) = \langle l_1 : \theta(\rho_1), \ldots, l_n : \theta(\rho_n) \rangle$ where $\{(l_1, \rho_1), \ldots, l_n, \rho_n)\} = \{(l, \rho)|\langle t \ni l : \rho \rangle \in C\}$. Since the above equations on θ defines a *regular system* satisfying *Greibach condition* [32], such θ always exists. It is clear that θ satisfies C. ▮

Proposition 5.5 *If C does not contain join conditions or projection conditions then C is satisfiable iff there is a refinement C' of C that is in record-variant normal form. Moreover, there is an algorithm which computes a refinment C' of C that is in record-variant normal form if one exists otherwise reports the unsatisfiability of C.*

Proof We prove the proposition by defining an algorithm. Define the transformation relation $\Longrightarrow$ on pairs of a set of conditions and a substitution as follows:

$$(C \cup \{[[\ldots, l : \rho_1, \ldots] \ni l : \rho_2]\}, \theta) \Longrightarrow (\theta'(C), \theta' \circ \theta) \text{ where } \theta' = \mathcal{U}(\rho_1, \rho_2) \tag{5.3}$$

$$(C \cup \{\langle\langle \ldots, l : \rho_1, \ldots\rangle \ni l : \rho_1\rangle\}, \theta) \Longrightarrow (\theta'(C), \theta' \circ \theta) \text{ where } \theta' = \mathcal{U}(\rho_1, \rho_2) \tag{5.4}$$

$$(C \cup \{[\rho \ni l : \rho_1], [\rho \ni l : \rho_2]\}, \theta) \Longrightarrow (\theta'(C \cup \{[\rho \ni l : \rho_1]\}), \theta' \circ \theta) \text{ where } \theta' = \mathcal{U}(\rho_1, \rho_2) \tag{5.5}$$

$$\begin{aligned}&(C \cup \{\langle\rho \ni l : \rho_1\rangle, \langle\rho \ni l : \rho_2\rangle\}, \theta) \Longrightarrow \\ &\qquad (\theta'(C \cup \{\langle\rho \ni l : \rho_1\rangle\}), \theta' \circ \theta) \text{ where } \theta' = \mathcal{U}(\rho_1, \rho_2)\end{aligned} \tag{5.6}$$

Let $\overset{*}{\Longrightarrow}$ be the transitive reflexive closure of $\Longrightarrow$. Define the algorithm $\mathcal{RV}$ as:

$$\mathcal{RV}(C) = \begin{cases} (C', \theta) & \text{if } (C, id) \overset{*}{\Longrightarrow} (C', \theta) \text{ and } C' \text{ is in record-variant normal form} \\ & \text{where } id \text{ is the identity substitution,} \\ \textit{unsatisfiable} & \text{otherwise} \end{cases}$$

(by assuming some ordering on applications of rules). Since the transformation $\Longrightarrow$ strictly reduces the number of conditions in C, for any given (C, θ), $\mathcal{RV}(C)$ always terminates.

We show the properties of $\mathcal{RV}$ needed to prove the theorem. By lemma 5.4, proposition 5.4 and by the definition of $\mathcal{RV}$, it suffices to show the following two properties: (1) if $\mathcal{RV}(C) = (C', \theta)$ then C' is a refinement of C with θ a refinement substitution and (2) if $\mathcal{RV}(C) = \textit{unsatisfiable}$ then C is not satisfiable.

For the first property, by the definition of $\mathcal{RV}$, it suffices to show that if $(C_1, \theta_1) \Longrightarrow (C_2, \theta_2)$ then C_2 is a refinement of C_1 with a refinement substitution θ' such that $\theta_2 = \theta' \circ \theta_1$. We show this property for each rule of $\Longrightarrow$.

1. The case for the rule (5.3):

$$(C \cup \{[[\ldots, l : \rho_1, \ldots] \ni l : \rho_2]\}, \theta) \Longrightarrow (\theta'(C), \theta' \circ \theta) \text{ where } \theta' = \mathcal{U}(\rho_1, \rho_2).$$

   Suppose η satisfies $C \cup \{[[\ldots, l : \rho_1, \ldots] \ni l : \rho_2]\}$. Then $\eta(\rho_1) = \eta(\rho_2)$. Therefore there is some η_1 such that $\eta = \eta_1 \circ \theta'$ and η_1 satisfies $\theta'(C \cup \{[[\ldots, l : \rho_1, \ldots] \ni l : \rho_2]\})$ and therefore it also satisfies $\theta'(C)$. Conversely suppose η satisfies $\theta'(C)$. Since $\theta'(\rho_1) = \theta'(\rho_2)$, η also satisfies $\theta'(C \cup \{[[\ldots, l : \rho_1, \ldots] \ni l : \rho_2]\})$. Therefore $\eta \circ \theta'$ satisfies $C \cup \{[[\ldots, l : \rho_1, \ldots] \ni l : \rho_2]\}$. This implies that $\theta'(C)$ is a refinement of $C \cup \{[[\ldots, l : \rho_1, \ldots] \ni l : \rho_2]\}$ with a refinement substitution θ'.

2. The case for the rule (5.5):

$$(C \cup \{[\rho \ni l : \rho_1], [\rho \ni l : \rho_2]\}, \theta) \Longrightarrow (\theta'(C \cup \{[\rho \ni l : \rho_1]\}), \theta' \circ \theta) \text{ where } \theta' = \mathcal{U}(\rho_1, \rho_2).$$

Suppose η satisfies $C \cup \{[\rho \ni l : \rho_1], [\rho \ni l : \rho_2]\}$. Then $\eta(\rho_1) = \eta(\rho_2)$. Therefore there is some η_1 such that $\eta = \eta_1 \circ \theta'$ and η_1 satisfies $\theta'(C \cup \{[\rho \ni l : \rho_1], [\rho \ni l : \rho_2]\})$. Then η_1 also satisfies $\theta'(C \cup \{[\rho \ni l : \rho_1]\})$. Conversely suppose η satisfies $\theta'(C\{[\rho \ni l : \rho_1]\})$. Since $\theta'(\rho_1) = \theta'(\rho_2)$, η also satisfies $\theta'(C \cup \{[\rho \ni l : \rho_1], [\rho \ni l : \rho_2]\})$. Therefore $\eta \circ \theta'$ satisfies $C \cup \{[\rho \ni l : \rho_1], [\rho \ni l : \rho_2]\}$. This implies that $\theta'(C \cup \{[\rho \ni l : \rho_1]\})$ is a refinement of $C \cup \{[\rho \ni l : \rho_1], [\rho \ni l : \rho_2]\}$ with a refinement substitution θ'.

3. The cases for the rule (5.4) and (5.6) are similar to (5.3) and (5.5) respectively.

We next show the second property. Suppose $\mathcal{RV}(C) = \mathit{unsatisfiable}$. By the definition of $\mathcal{RV}$, there is some C' that is a refinement of C and C' satisfies one of the following properties:

1. $[[\ldots, l : \rho_1, \ldots] \ni l : \rho_2] \in C'$ and there is no substitution θ such that $\theta(\rho_1) = \theta(\rho_2)$,
2. $\langle\langle \ldots, l : \rho_1, \ldots\rangle \ni l : \rho_1\rangle \in C'$ and there is no substitution θ such that $\theta(\rho_1) = \theta(\rho_2)$,
3. $\{[\rho \ni l : \rho_1], [\rho \ni l : \rho_2]\} \subseteq C'$ and there is no substitution θ such that $\theta(\rho_1) = \theta(\rho_2)$,
4. $\{\langle \rho \ni l : \rho_1\rangle, \langle \rho \ni l : \rho_2\rangle\} \subseteq C'$ and there is no substitution θ such that $\theta(\rho_1) = \theta(\rho_2)$,
5. $\langle \rho \ni l : \rho'\rangle \in C'$ and ρ is either a record type or a set type,
6. $[\rho \ni l : \rho'] \in C'$ and ρ is either a variant type or a set type,
7. $[\rho \ni l_1 : \rho_1] \in C'$ and $\langle \rho \ni l_2 : \rho_2\rangle \in C'$ for some ρ.

The first 4 cases correspond to the cases where one of unifications in $\mathcal{RV}(C)$ fails. Others correspond to the cases where the result of the transformation of C in $\mathcal{RV}(C)$ is not in record-variant normal form. In each case C' is not satisfiable. Since C' is a refinement of C, C is not satisfiable. ∎

Since join conditions and projection conditions are created only if an expression contains *join*, *con* or *proj*$^\sigma$, the above result establishes the complete type checking and type inference procedure for expressions that do not contain these three primitives. This result can be extended to projections on finite description types, as shown in the following proposition.

Proposition 5.6 *Let C be a set of conditions that does not contain join condition and the target type of each projection condition is finite. If C is satisfiable then there is a refinement C' of C that contains only record conditions and variant conditions. Moreover, there is an algorithm that computes a refinment C' of C if C is satisfiable otherwise reports the unsatisfiability of C.*

Proof As before we prove the proposition by defining an algorithm by a transformation relation. Let $\Longrightarrow$ be the transformation relation on pairs of a set of conditions and a substitution as follows:

$$(C \cup \{lessthan(b, \rho)\}, \theta) \Longrightarrow (\theta'(C), \theta' \circ \theta) \text{ where } \theta' = \mathcal{U}(b, \rho)$$

$$\begin{aligned}&(C \cup \{lessthan([l_1 : \sigma_1, \ldots, l_n : \sigma_n], \rho)\}, \theta) \Longrightarrow \\ &\quad (C \cup \{[\rho \ni l_1 : s_1], \ldots, [\rho \ni l_n : s_n], lessthan(\sigma_1, s_1), \ldots, lessthan(\sigma_n, s_n)\}, \theta) \\ &\quad (s_1, \ldots, s_n \text{ fresh})\end{aligned}$$

$$\begin{aligned}&(C \cup \{lessthan(\langle l_1 : \sigma_1, \ldots, l_n : \sigma_n \rangle, \rho)\}, \theta) \Longrightarrow \\ &\quad (\theta'(C \cup \{lessthan(\sigma_1, s_1), \ldots, lessthan(\sigma_n, s_n)\}), \theta' \circ \theta) \\ &\quad \text{where } \theta' = \mathcal{U}(\rho, \langle l_1 : s_1, \ldots, l_n : s_n \rangle), (s_1, \ldots, s_n \text{ fresh})\end{aligned}$$

$$\begin{aligned}&(C \cup \{lessthan(\{\!\{\sigma_1\}\!\}, \rho)\}, \theta) \Longrightarrow \\ &\quad (\theta'(C \cup \{lessthan(\sigma_1, s), \theta' \circ \theta) \text{ where } \theta' = \mathcal{U}(\rho, \{\!\{s\}\!\}), (s \text{ fresh})\end{aligned}$$

Define the algorithm $\mathcal{FP}$ as:

$$\mathcal{FP}(C) = \begin{cases} (C', \theta) & \text{if } (C, id) \overset{*}{\Longrightarrow} (C', \theta) \text{ and } C' \text{ has no projection condition,} \\ \mathit{unsatisfiable} & \text{otherwise} \end{cases}$$

(by assuming some ordering on applications of rules).

Define the weight of a projection condition $lessthan(\sigma, \rho)$ as the height of σ (i.e. the maximum number of the nesting level of type constructors in σ). Since σ is finite, each projection condition has a finite weight. Moreover each transformation step always decreases the following complexity measure: the multiset of weights of projection conditions in C under the multiset ordering. Therefore the above algorithm always terminates. The correctness of the algorithm follows from the definition of the satisfiability of $lessthan(\sigma, \rho)$ (see the inductive definition of $\leqslant$ in subsection 4.4.1). ∎

We then have:

Theorem 5.2 *If C does not contain join conditions and the target type of each projection condition is finite then C is satisfiable iff there is a refinement C' of C that is in record-variant normal form. Moreover, there is an algorithm that computes a refinement C' and a refinement substitution θ if one exists otherwise reports the unsatisfiability of C.*

Proof By applying proposition 5.6 followed by proposition 5.5. ▌

This result establishes the complete type inference for raw terms that do not contain joins and all projections are those on finite description types. Moreover, there is a compact representation for a conditional typing scheme for such a raw term. Since if a set of conditions is in record-variant normal form then all conditions are conditions on type variables, conditions can be integrated into type-schemes by extending type-schemes to include type variables with conditions. Define *conditional type-schemes* (ranged over by T) as the regular trees represented by the following syntax:

$$\begin{aligned} T \quad ::= \quad & t \mid [(t)l : T, \ldots, l : T] \mid (t \leqslant \sigma) \mid \langle (t)l : T, \ldots, l : T \rangle \mid b \mid \\ & T \to T \mid [l : T, \ldots, l : T] \mid \langle l : T, \ldots, l : T \rangle \mid (\textbf{rec } v.\, T(v)). \end{aligned}$$

$[(t)l_1 : T_1, \ldots, l_n : T_n]$ is a type variable t associated with the set of conditions $\{[t \ni l_1 : T_1], \ldots, [t \ni l_n : T_n]\}$. Similarly for $\langle (t)l : T, \ldots, l : T \rangle$. $(t \leqslant \sigma)$ is a type variable t associated with the condition $\mathit{lessthan}(t, \sigma)$. We call those type variables *conditional type variables*. Conditional typing schemes for raw terms that do not contain joins and all projections are those on finite types can then be represented by these conditional type-scheme. I leave to the reader the mechanical transformation of a set of conditions to conditional type variables. The following are examples of principal conditional typing schemes using the above representation:

$$\begin{aligned} \boldsymbol{MC} \vdash \quad & \emptyset \rhd \textbf{fn}(x) \Rightarrow x.l \;:\; [(u_1)l : u_2] \to u_2, \\ \boldsymbol{MC} \vdash \quad & \emptyset \rhd \textbf{fn}(x) \Rightarrow \textbf{modify}(x, l, x.l + 1) \;:\; [(u_1)l : \mathit{int}] \to \mathit{int}, \\ \boldsymbol{MC} \vdash \quad & \emptyset \rhd \textbf{fn}(x) \Rightarrow (x.l_1, (x.l_1).l_2) \;:\; [(u_1)l_1 : [(u_2)l_2 : u_3]] \to ([(u_2)l_2 : u_3] \times u_3). \end{aligned}$$

The examples we have seen in the beginning of this chapter are conditional typing schemes using these compact representations. In examples that follows we use these compact representations for conditions whenever appropriate.

When a set of conditions contains join conditions or projection conditions with infinite target types, however, the problem of deciding whether it is satisfiable or not becomes a difficult problem. Indeed the following result implies that it is an NP-hard problem.

Theorem 5.3 *It is NP hard to decide whether a given raw term e has a typing or not.*

Proof The proof is by reduction from MONOTONE 3SAT [43]:

> Given a 3CNF Boolean formula whose clauses consist of either all negated literals (called negative clauses) or all un-negated literals (called positive clauses), test whether there is a truth assignment.

Let $F = \{c_1, \ldots, c_m\}$ be the given set of clauses and $\{x_1, \ldots, x_n\}$ be the set of all literals that appear (either un-negated or negated) in F. We construct a term e^F such that F has a truth assignment iff e^F has a typing. We use the following constants: $f : int \rightarrow int$, $g : bool \rightarrow bool$. We use four variables $x_{true}, x_{false}, x_{int}, x_{bool}$ for each literal x, one label $\#x$ for each x and one label $\#c$ for each c and labels $l, \#1, \#2, \#3, \#4$. For each x, let M^x be the term

$$\begin{aligned} M^x \equiv [\#1 &= f(\mathbf{join}(x_{true}, x_{int}).l), \\ \#2 &= g(\mathbf{join}(x_{false}, x_{bool}).l), \\ \#3 &= \mathbf{join}(x_{true}, x_{false}).l, \\ \#4 &= \mathbf{join}(x_{int}, x_{bool}).l]. \end{aligned}$$

For each clause c, if c consists of un-negated literals $\{x, y, z\}$ then let N^c be the term

$$N^c \equiv f(\mathbf{join}(\mathbf{join}(x_{true}, y_{true}), z_{true}).l)$$

otherwise c consists of negated literals $\{\overline{x}, \overline{y}, \overline{z}\}$ then let N^c be the term

$$N^c \equiv g(\mathbf{join}(\mathbf{join}(x_{false}, y_{false}), z_{false}).l).$$

Now define the desired term e as the following record:

$$e^F \equiv [\#x_1 = M^{x_1}, \ldots, \#x_n = M^{x_n}, \#c_1 = N^{c_1}, \ldots, \#c_m = N^{c_m}].$$

The translation from F to e is clearly polynomial.

We next show the desired property of e^F. Suppose e^F has a typing $\mathcal{A} \vdash e^F : \tau$. By the typing rule, both M^x and N^c have a typing under $\mathcal{A}$. By the definition of M^x, if M^x has a typing under $\mathcal{A}$ then either $\mathcal{A}(x_{true})$ is a record type containing the field $l : int$ or $\mathcal{A}(x_{false})$ is a record type containing the field $l : bool$ and not both. Define a truth assignment $\mathcal{M}$ such that $\mathcal{M}(x) = true$ iff $\mathcal{A}(x_{true})$ is a record type containing $l : int$ field. By the definition of N^c and the typing rule for *join*, for a positive clause $\{x, y, z\}$, if $N^{\{x,y,z\}}$ has a typing under $\mathcal{A}$ then at least one of $\mathcal{A}(x_{true}), \mathcal{A}(y_{true}), \mathcal{A}(z_{true})$ has the field $l : int$ and for a negative clause $\{\overline{x}, \overline{y}, \overline{z}\}$, if $N^{\{\overline{x},\overline{y},\overline{z}\}}$ has a typing under $\mathcal{A}$ then at least one of $\mathcal{A}(x_{false}), \mathcal{A}(y_{false}), \mathcal{A}(z_{false})$ has the field $l : bool$. By the definition of $\mathcal{M}$ this implies that $\mathcal{M}$ satisfies F.

Conversely suppose F is satisfied by an assignment $\mathcal{M}$. Define a type assignment $\mathcal{A}$ as follows: if $\mathcal{M}(x) = true$ then $\mathcal{A}(x_{true}) = [l : int]$, $\mathcal{A}(x_{false}) = []$, $\mathcal{A}(x_{int}) = []$, $\mathcal{A}(x_{bool}) = [l : bool]$ otherwise $\mathcal{A}(x_{true}) = []$, $\mathcal{A}(x_{false}) = [l : bool]$, $\mathcal{A}(x_{int}) = [l : int]$, $\mathcal{A}(x_{bool}) = []$. It is then easy to check that e has the following type under $\mathcal{A}$:

$$[\#x_1 : \tau_1, \ldots, \#x_n : \tau_n, \#c_1 : \tau'_1, \ldots, \#c_m : \tau'_m]$$

where τ_i is $[\#1 : int, \#2 : bool, \#3 : int, \#4 : bool]$ if $\mathcal{M}(x_i) = true$ otherwise $[\#1 : int, \#2 : bool, \#3 : bool, \#4 : int]$ and $\tau'_j = int$ if c_j is positive clause otherwise $\tau'_j = bool$ ∎

Worse yet, we do not know whether the type inference problem for the entire language is decidable or not. There is, however, a practical solution to this problem. The strategy is to "delay" the satisfiability checking of join conditions and projection conditions (of infinite target types) until all type variables are instantiated, i.e. we delay the satisfiability checking of conditions associated with functions containing *join* and $proj^{\sigma}$ for infinite σ until they are applied to actual arguments. If a set of conditions does not contain type variables then the satisfiability can be decided by checking the ordering relation and computing the least upper bound of description types, which have been shown to be decidable (proposition 4.11). There is, however, one problem in this approach because of variants. Like ML's *nil*, variants are polymorphic values and therefore their type variables may never be instantiated. We solve this problem by imposing the following restriction:

> The programmer must supply the type specifications for variants if they are arguments of the functions *join* or $proj^{\sigma}$ with infinite σ (directly or indirectly through function abstractions/applications).

Intuitively, this states that the "actual" arguments to the functions *join* and $proj^{\sigma}$ (with infinite σ) must have a ground type. It is interesting to note that the same problem arises with the combination of the reference types and type variables and the above restriction is the same as the one adopted in Standard ML implementation [71]. We say that a conditional typing scheme is a *program scheme* if it is of the form $C, \emptyset \rhd e : T$ such that T does not contain function type constructor. If a conditional typing scheme is a program scheme then join conditions and projection conditions are respectively of the forms $\rho = jointype(\sigma_1, \sigma_2)$ and $lessthan(\sigma_1, \sigma_2)$, both of which are reduced to true or false. Therefore only remaining type variables in a program scheme are those created for variants. Therefore the above restriction is imposed by treating a program scheme $C, \Sigma \rhd e : T$ as a type error if C contain (irreducible) type variables.

We now define the set of terms Machiavelli as follows:

Definition 5.8 (Terms of Machiavelli) *A term of Machiavelli is a conditional typing scheme such that the set of conditions is in record-variant normal form and if it is a program-scheme then its does not contain join condition or projection condition with infinite target types.*

5.5 Semantics of Machiavelli

In this section, we define a denotational semantics and an operational semantics of Machiavelli. For this purpose, we need to identify the subset of terms that represent descriptions we have constructed in section 4.4. The set of *description terms* (ranged over by d) is the following set of raw terms that have a typing:

$$d ::= c^b \mid record^{(l_1,\ldots,l_n)}(d)\cdots(d) \mid set^n(d)\cdots(d) \mid variant^l(d) \mid c^{(x_1,\ldots,x_n)}(d)\cdots(d).$$

A description term denotes a regular tree in $\boldsymbol{Dobj}^\infty$. We write $D(d)$ for the element in $\boldsymbol{Dobj}^\infty$ denoted by d.

5.5.1 Denotational Semantics of Machiavelli

To give a denotational semantics of Machiavelli, we extend the semantic framework for ML we have developed in chapter 3. Let xMachiavelli be the typed lambda calculus $T\Lambda^+$ we have defined in section 3.5 with the set $Type$ of types of Machiavelli and the set of constants $\{c^\tau | c : \tau \in Consts\}$ where $Consts$ is the set of constants of Machiavelli.

Definition 5.9 (Models of xMachiavelli) *An extensional extended frame $(\mathcal{F}, \bullet, \mathcal{C})$ is a model of xMachiavelli if the following conditions hold:*

1. *it satisfies the definition of abstract models (definitions 3.24 and 3.25) of $T\Lambda^+$,*
2. *for each description type σ, $F_\sigma \supseteq D_\sigma$ where D_σ is the description domain of σ we have constructed in section 4.4,*
3. *for each description term d of type σ, $[\![\emptyset \rhd d : \sigma]\!]\varepsilon \in D_\sigma$,*
4. *the following equations are valid (we omit types):*

$$(1) \qquad d_1 = d_2 \qquad \textit{if } D(d_1) = D(d_2)$$

(2) $[l_1 = e_1, \ldots, l_i = e_i, \ldots, l_n = e_n].l = e_i$

(3) $modify^{l_i}\ [l_1 = e_1, \ldots, l_i = e_i, \ldots, l_n = e_n]\ e =$
$[l_1 = e_1, \ldots, l_i = e, \ldots, l_n = e_n]$

(4) $(\textbf{case}\ \langle l_i = e\rangle\ \textbf{of}\ l_1 \Rightarrow e_1, \ldots, l_i \Rightarrow e_i, \ldots, l_n \Rightarrow e_n) = (e_i\ e)$

(5) $union\ \{\!\{e_1^1, \ldots, e_n^1\}\!\}\ \{\!\{e_1^2, \ldots, e_m^2\}\!\} = \{\!\{e_1^1, \ldots, e_n^1, e_1^2, \ldots, e_m^2\}\!\}$

(6) $prod^n\ \{\!\{e_1^1, \ldots, e_{k_1}^1\}\!\} \cdots \{\!\{e_1^n, \ldots, e_{k_n}^n\}\!\} =$
$\{\!\{(e_1^1, e_1^2, \ldots, e_1^n), (e_1^1, e_1^2, \ldots, e_2^n), \ldots (e_{k_1}^1, e_{k_2}^2, \ldots, e_{k_n}^n)\}\!\}$

(7) $map\ e\ \{\!\{e_1, \ldots, e_n\}\!\} = \{\!\{(e\ e_1), \ldots, (e\ e_n)\}\!\}$

(8) $join\ d_1\ d_2 = d_3 \quad if\ D(d_3) = D(d_1) \sqcup D(d_2)$

(9) $con\ d_1\ d_2 = true \quad if\ D(d_1) \sqcup D(d_2)\ exists$

(10) $con\ d_1\ d_2 = false \quad if\ D(d_1) \sqcup D(d_2)\ does\ not\ exist$

(11) $proj^\sigma(d_1) = d_2 \quad if\ D(d_2) = Proj^\sigma(D(d_1))$

(12) $eq\ d_1\ d_2 = true \quad if\ D(d_1) \sqsubseteq D(d_2)\ and\ D(d_2) \sqsubseteq D(d_1)$

(13) $eq\ d_1\ d_2 = false \quad if\ either\ D(d_1) \not\sqsubseteq D(d_2)\ or\ D(d_2) \not\sqsubseteq D(d_1)$

where $Proj^\sigma$ in (11) is the function on $\bigcup \boldsymbol{Dobj}^\infty$ defined in subsection 4.4.5.

The relationship between typings of Machiavelli and terms of xMachiavelli remains the same as that between ML and $T\Lambda^+$ and theorem 3.6 holds. This implies that the definition of the semantics of typings of ML (definition 3.18) applies directly to those of Machiavelli.

The semantics of Machiavelli terms relative to a model of xMachiavelli is then defined as follows. For a given pair of a type assignment scheme Σ and a set of conditions C, the set of *admissible type assignments under C and Σ*, denoted by $TA(C, \Sigma)$, is the set $\{\mathcal{A} | dom(\Sigma) \subseteq dom(\mathcal{A}), \exists\theta.\, \theta$ satisfies C and $\mathcal{A}\restriction^{dom(\Sigma)} = \theta(\Sigma)\}$. Under a given type assignment $\mathcal{A}$, the set $TP(\mathcal{A}, C, \Sigma \rhd e : \rho)$ of the types associated with a term $C, \Sigma \rhd e : \rho$ is the set $\{\tau | \exists\theta.\, \theta$ satisfies C, and $(\mathcal{A}\restriction^{dom(\Sigma)}, \tau) = \theta(\Sigma, \rho)\}$. Then the semantics $\mathcal{M}[\![C, \Sigma \rhd e : \rho]\!]$ of a Machiavelli term $C, \Sigma \rhd e : \rho$ relative to a model $\mathcal{M}$ is the function taking a type assignment $\mathcal{A} \in TA(C, \Sigma)$ and an environment $\varepsilon \in Env^{\mathcal{M}}(\mathcal{A})$ that returns an element in $\Pi\tau \in TP(\mathcal{A}, C, \Sigma \rhd e : \rho)$. D_τ defined as follows:

$$\mathcal{M}[\![C,\Sigma \rhd e : \rho]\!]\mathcal{A}\varepsilon = \{(\tau, \mathcal{M}[\![\mathcal{A} \rhd e : \tau]\!]^{\mathrm{ML}}\varepsilon) | \tau \in TP(\mathcal{A},\Sigma \rhd e : \rho)\}.$$

Note that the definition works also for terms with unsatisfiable conditions. In such a case the denotation is the emptyset.

As an example, the semantics of the constant *join*:

$$\mathcal{M}[\![\{s_3 = jointype(s_1, s_2)\}, \emptyset \rhd \mathbf{join} : s_1 \rightarrow s_2 \rightarrow s_3]\!]\mathcal{A}\varepsilon$$

is a set of functions $f : F_{\sigma_1} \rightarrow F_{\sigma_2} \rightarrow F_{\sigma_3}$ for all triples of $\sigma_1, \sigma_2, \sigma_3$ such that $\sigma_3 = \sigma_1 \sqcup \sigma_3$. Now if we restrict its domains $F_{\sigma_1} F_{\sigma_2}$ to $D_{\sigma_1}, D_{\sigma_2}$ then we get exactly the set of joins we have defined in subsection 4.4.4. Similar property holds for projections. This confirms that we have successfully integrated the database domain we have developed in chapter 4 into an ML-style type system.

5.5.2 Operational Semantics

This section gives an operational semantics of Machiavelli by an evaluation relation on conditional typing schemes. There are several evaluation strategies for operational semantics of functional programming languages. Here we only give an operational semantics based on the "call-by-value" evaluation strategy that "stops" at function abstraction. An operational semantics based on the "call-by-name" strategy is simpler to specify and can be easily defined by changing some of the rules given below.

We first define the immediate reduction relation $\longrightarrow$ on raw terms.

(I1) $((\lambda x.\, e)\ d) \longrightarrow e[d/x]$

(I2) $((\lambda x.\, e_1)\ (\lambda y.\, e_2)) \longrightarrow e_1[(\lambda y.\, e_2)/x]$

(I3) $$\frac{e_1 \longrightarrow e_2}{(e\ e_1) \longrightarrow (e\ e_2)}$$

(I4) $$\frac{e_1 \longrightarrow e_2}{(e_1\ d) \longrightarrow (e_2\ d)}$$

(I5) $$\frac{e_1 \longrightarrow e_2}{(e_1\ (\lambda x.\, e)) \longrightarrow (e_2\ (\lambda x.\, e))}$$

(II) $$\frac{e_i \twoheadrightarrow e_i'}{c\ e_1 \cdots e_i \cdots e_n \twoheadrightarrow c\ e_1 \cdots e_i' \cdots e_n} \quad \text{for any constant } c$$

(III1) $[l_1 = e_1, \ldots, l_i = e_i, \ldots, l_n = e_n].l \twoheadrightarrow e_i$

(III2) $(\mathbf{rec}\ x.\, e).l \twoheadrightarrow (e[(\mathbf{rec}\ x.\, e)/x]).l$

(IV1) $modify^l\ [l_1 = e_1, \ldots, l_i = e_i, \ldots, l_n = e_n]\ e \twoheadrightarrow$
$[l_1 = e_1, \ldots, l_i = e, \ldots, l_n = e_n]$

(IV2) $modify^l\ (rec\ x.\, e_1)\ e_2 \twoheadrightarrow modify^l\ e_1[(rec\ x.\, e_1)/x]\ e_2$

(V1) $(\mathbf{case}\ \langle l_i = e\rangle\ \mathbf{of}\ l_1 \Rightarrow e_1, \ldots, l_i \Rightarrow e_i, \ldots, l_n \Rightarrow e_n) \twoheadrightarrow (e_i\ e)$

(V2) $(\mathbf{case}\ (rec\ x.\, e)\ \mathbf{of}\ l_1 \Rightarrow e_1, \ldots, l_n \Rightarrow e_n) \twoheadrightarrow$
$(\mathbf{case}\ e[(rec\ x.\, e)/x]\ of\ l_1 \Rightarrow e_1, \ldots, l_n \Rightarrow e_n)$

(VI1) $union\ \{\!\{e_1^1, \ldots, e_n^1\}\!\}\ \{\!\{e_1^2, \ldots, e_m^2\}\!\} \twoheadrightarrow \{\!\{e_1^1, \ldots, e_n^1, e_1^2, \ldots, e_m^2\}\!\}$

(VI2) $union\ (rec\ x.\, e_1)\ e_2 \twoheadrightarrow union\ e_1[(rec\ x.\, e_1)/x]\ e_2$

(VI3) $union\ e_1\ (rec\ x.\, e_2) \twoheadrightarrow union\ e_1\ e_2[(rec\ x.\, e_2)/x]$

(VII1) $prod^n\ \{\!\{e_1^1, \ldots, e_{k_1}^1\}\!\} \cdots \{\!\{e_1^n, \ldots, e_{k_n}^n\}\!\} \twoheadrightarrow$
$\{\!\{(e_1^1, e_1^2, \ldots, e_1^n), (e_1^1, e_1^2, \ldots, e_2^n), \ldots (e_{k_1}^1, e_{k_2}^2, \ldots, e_{k_n}^n)\}\!\}$

(VII2) $prod^n e_1\ \cdots\ (rec\ x.\, e_i) \cdots\ e_n \twoheadrightarrow prod^n e_1\ \cdots\ e_i[(rec\ x.\, e_i)/x] \cdots\ e_n$

(VIII1) $map\ e\ \{\!\{e_1, \ldots, e_n\}\!\} \twoheadrightarrow \{\!\{(e\ e_1), \ldots, (e\ e_n)\}\!\}$

(VIII2) $map\ e_1\ (rec\ x.\, e_2) \twoheadrightarrow map\ e_1\ e_2[(rec\ x.\, e_2)/x]$

(IX) $join\ d_1\ d_2 \twoheadrightarrow d_3$ if $D(d_3) = D(d_1) \sqcup D(d_2)$

(X) $con\ d_1\ d_2 \twoheadrightarrow true$ if $D(d_1) \sqcup D(d_2)$ exists

(X) $con\ d_1\ d_2 \twoheadrightarrow false$ if $D(d_1) \sqcup D(d_2)$ does not exist

(XI) $proj^\sigma\ d_1 \twoheadrightarrow d_2$ if $D(d_2) = Proj^\sigma(D(d_1))$

(XII) $eq\ d_1\ d_2 \twoheadrightarrow true$ if $D(d_1) \sqsubseteq D(d_2)$ and $D(d_2) \sqsubseteq D(d_1)$

(XII) $\quad eq\ d_1\ d_2 \twoheadrightarrow false \quad$ if $D(d_1) \not\subseteq D(d_2)$ or $D(d_2) \not\subseteq D(d_1)$

The evaluation relation on conditional typing schemes is defined as the reflexive transitive closure of this immediate reduction relations.

5.6 Syntactic Shorthands

This section defines several useful syntactic shorthands.

5.6.1 Recursive Function

In ML like languages, recursive functions are defined by the special term constructor (**fix** $x.\,e$). This is regarded as a shorthand for $(Y\ (\mathbf{fn}(x) \Rightarrow e))$ where Y is the symbol of the following set of constants denoting fixed point operators:

$$\{Y^{(\tau\to\tau)\to(\tau\to\tau)\to\tau\to\tau} \mid \text{for all } \tau\}.$$

We did not include these fixed point constants because fixed point combinators are definable, as we have seen in section 3.5. Under an operational semantics based on the call-by-name evaluation strategy, both Y_{curry} and Y_{turing} (defined in section 3.5) provide a desired operational behavior. However, they do not have the desired property under an operational semantics based on the call-by-value evaluation strategy such as the one we have defined for Machiavelli. Indeed it is easily checked that for any raw term e, both $(Y_{curry}\ e)$ and $(Y_{turing}\ e)$ do not terminate under such an operational semantics.

Under the call-by-value evaluation, the desired operational rules for Y constant for defining recursive functions should be:

(Y1) $\quad ((Y\ \lambda x.\,e)\ d) \twoheadrightarrow (e[(Y\ \lambda x.\,e)/x]\ d),$

(Y2) $\quad ((Y\ \lambda x.\,e_1)\ \lambda y.\,e_2) \twoheadrightarrow (e_1[(Y\lambda x.\,e_1)/x]\ \lambda y.\,e_2).$

In [90] Plotkin gave the following fixed point combinator:

$$Y_{plotkin} = \lambda f.\,(\lambda x.\,\lambda y.\,f(x\ x)y)(\lambda x.\,\lambda y.\,f(x\ x)y).$$

which realizes the above behavior under the call-by-value evaluation rules for function application (rules (I1) - (I5)). The extra η-redexes are essential to get the desired behavior. Note also that

$Y_{plotkin}$ has the following principal conditional typing scheme:

$$\vdash \emptyset, \emptyset \triangleright Y_{plotkin} : (u \to u) \to (u \to u) \to u \to u$$

This combinator can also be used under the call-by-name evaluation rules. We therefore define the following syntactic shorthand for recursion:

$$(\mathbf{fix}\ x.\, e) \Leftrightarrow (Y_{plotkin}\ (\lambda x.\, e)).$$

5.6.2 Value Bindings and Function Definitions

As a practical programming language Machiavelli should provide a form of binding mechanism that binds names to Machiavelli terms. As is done in Standard ML, we regard the following binding

$$\mathbf{val}\ id = e;\ \cdots$$

as a syntactic shorthand for the following fragment:

$$\mathbf{let}\ id = e\ \mathbf{in}\ \cdots\ \mathbf{end}.$$

For function definition, we further adopt the following shorthand:

$$\mathbf{fun}\ f\ x_1 \cdots x_n = e;$$

for

$$\mathbf{val}\ f = (\mathbf{fix}\ f\ \lambda x_1 \ldots \lambda x_n.\, e);.$$

Note that the two occurrences of f in the above definition are different. The first one is the name to which the body of the expression is bound and the second one is a bound variable.

5.6.3 Database Operations

An important part of database programming is query processing. One of common structures of query processing is so called *select-join-project* query [35]. For such query processing, we define the following shorthand whose syntax follows from SQL [8] and *list comprehension* in Miranda [107]:

select $e1$
where $x_1 \in S_1, \ldots, x_n \in Sn$
with e_2

for

```
let
   fun result x =
       let
          val $x_1 = x.\#1$;
          ⋮
          val $x_n = x.\#n$;
       in
          $e_1$
       end
   fun pred x =
       let
          val $x_1 = x.\#1$;
          ⋮
          val$x_n = x.\#n$
       in
          $e_2$
       end
in
   filter pred (map(result, prod($S_1, ...S_n$)))
end
```

where *filter* is a polymorphic selection function which, given a boolean valued function P and a set S, selects all those elements e from S such that $P(e) = true$. The following is one implementation of *filter*:

```
fun filter P S =
     map($\lambda x. x.Val$,
          join($\{\![ [Pred = true] ]\!\}$,
               map($\lambda x. [Value = x, Pred = P(x)], S$)
```

with the following principal conditional typing scheme:

$$\boldsymbol{MC} \vdash \emptyset, \emptyset \rhd \mathit{filter} : (s \rightarrow \mathit{bool}) \rightarrow \{\![s]\!\} \rightarrow \{\![s]\!\}.$$

5.7 Programming Examples

This section shows examples. We show them by simulating interactive session in Machiavelli. We write

```
-> expr ;
-> val id = expr ;
```

for an expression and a binding entered by the programmer and

```
>> val id = expr :   ρ where {c1,...,cn}
```

for the output (as a principal conditional typing scheme) computed by Machiavelli. `where` clause describes the unresolved conditions. If the last input is an expression then the system assumes `val it =` . In these examples we use the following notations:

`fn(x) => e`	$\Leftrightarrow$	$\lambda x. e$,
`{e1,...}`	$\Leftrightarrow$	$\{\!\{e1, \ldots\}\!\}$,
`x <- s`	$\Leftrightarrow$	$x \in s$.

Following Standard ML, we write `'a`, `'b` etc for unconditional type variables and `"a`, `"b` etc for description type variables (which roughly correspond to equality type variables in Standard ML).

The first example shown in figure 5.1 is a simple session in Machiavelli involving records and variants. Form this example, we can see that Machiavelli extends ML with records and variants preserving its features of static type inference and polymorphism.

Figure 5.2 shows a simple example involving *join* and *project*. `Join3` computes the join of three (joinable) descriptions. `where` clause represents the conditions associated with the two `join` expressions. If `r1,r2,r3` are three joinable flat relations, then `Join3(r1,r2,r3)` is exactly the natural join of the three.

Figure 5.3 shows an example of a database containing non-flat records, variants, and nested sets assuming that `parts`, `suppliers` and `supplied_by` have been already defined. With the availability of generalized join and projection, we can immediately write programs that manipulate such databases. Figure 5.4 show some simple query processing for the example database in figure 5.3. From this example, one can see that join and projection in Machiavelli faithfully extend the natural join and projection in the relational model to complex objects.

```
-> val joe = [Name = "Joe", Age = 21];
>> val joe = [Name = "Joe", Age = 21] : [Name:string, Age:int]
-> val helen = [Name = [Fn = "Helen", Ln = "Smith"], Age = 31];
>> val helen =  [Name = [Fn = "Helen", Ln = "Smith"], Ag e= 31]
   : [Name:[Fn:string, Ln:string], Age:int]

-> fun name(p) = p.Name;
>> val name = fn : [('a)Name:'b] -> 'b
-> fun increment_age(x) = modify(x, Age, x.Age + 1);
>> val increment_age = fn : [('a) Age:int] -> [('a) Age:int]

-> name(joe);
>> val it = "Joe" : string
-> name(helen);
>> val it = [Fn= "Helen", Ln = "Smith"] : [Fn:string, Ln:string]
-> increment_age(joe);
>> val it = [Name="Joe", Age=22] : [Name:string, Age:int]
-> increment_age(helen);
>> val it = [Name = [Fn = "Helen", Ln = "Smith"], Ag e= 32]
   : [Name:[Fn:string, Ln:string], Age:int]

-> val john  = [Name="John", Age=21,
               Status=<Consultant = [Address="Philadelphia",
                                     Telephone=2221234]>];
>> val john = [Name="Joe", Age=21
              Status=<Consultant = [Address="Philadelphia",
                                    Telephone=2221234]>]
   : [Name:string, Age:int,Status:<('a) Consultant:[Address:string,
                                                     Telephone:int]>]
-> val mary  = [Name="Mary", Age=31,
               Status=<Employee = [Office=278, Extension=4895]>];
>> val mary = [Name="Mary", Age=31
              Status=<Employee = [Office=278, Extension=4895]>];
   : [Name:string, Age:int,Status:<('a) Employee:[Office:string,
                                                  Extension:int]>]

-> fun phone(x) = (case x.Status of <Employee=y> => y.Extension,
                                   <Consultant=y> => y.Telephone);
>> val phone = fn
   : [('a) Status:<Employee:[('b) Extension:'d],
                   Consultant:[('c) Telephone:'d]>]  -> 'd
-> phone(john);
>> 2221234 : int
-> phone(mary);
>> 4895 : int
```

Figure 5.1: A Simple Session in Machiavelli

```
-> val fun Join3(x,y,z) = join(x,join(y,z));
>> val Join3 = fn : ("a * "b * "c) -> "d
   where { "d = "a lub "e, "e = "b lub "c }
-> Join3([Name="Joe"],[Age=21],[Office=278]);
>> val it = [Name="Joe",Age=21,Office=278]
   : [Name:string,Age:int,Office:int]
-> project(it,[Name:string]);
>> val it = [Name="Joe"] : [Name:string]
```

Figure 5.2: A Simple Example Involving *join* and *project*

```
-> parts;
>> val it =
     {[Pname="bolt",P#=1,Pinfo=(BasePart of [Cost=5])],
         ...
      [Pname="engine",P#=2189,
       Pinfo=(CompositePart of [SubParts={[P#=1,Qty=189],...},
                                AssemCost=1000])],
         ...
     }
    : {[Pname:str,P#:int,
        Pinfo:<BasePart:[Cost:int],
               CompositePart:[SubParts:{[P#:int,Qty:int]},
                              AssemCost:int]>]}
 -> suppliers;
 >> val it =
      {[Sname="Baker",S#=1,City="Paris"],
       ...
      }
    : {[Sname:string,S#:int, City:string]}

 -> supplied_by;
 >> val it =
      {[P#=1,Suppliers={[S#=1],[S#=12],....}],
        ...
      }
    : {[P#:int,Suppliers:{[S#:int]}]}
```

Figure 5.3: A Part-Supplier Database in Generalized Relational Model

```
(* Select all base parts *)
-> join(parts,{[Pinfo=(BasePart of [])]});
>> val it =
     {[Pname="bolt",P#=1,Pinfo=(BasePart of [Cost=0.05])],
          ...
     }
    : {[Pname:str,P#:int,
        Pinfo:<BasePart:[Cost:int],
               CompositePart:[SubParts:{[P#:int,Qty:int]},
                              AssemCost:int]>]}

(* List part names supplied by "Baker" *)
-> select x.Pname
   where x <- join(parts,supplied_by)
   with Join3(x.Suppliers,suppliers,{[Sname="Baker"]}) <> {};
>> {"bolt",...} : {str}
```

Figure 5.4: Some Simple Queries

```
-> fun Closure R =
       let
         fun member (e,S) = filter((fn(x) => x=e), S) <> {}
         val r = select [A=x.A,B=y.B]
                 where x <- R, y <- R
                 with (x.B = y.A) andalso not(member([A=x.A,B=y.B],R))
       in
         if r = {} then R else Closure(union(R,r))
       end;
>> Closure = fn : {[A:"a,B:"b]} -> {[A:"a,B:"b]}
```

Figure 5.5: A Simple Implementation of Polymorphic Transitive Closure

The most important feature of Machiavelli is that these data structures and operations are all "first-class citizens" in the language. This eliminates the problem of "impedance mismatch" we discussed in chapter 1. Data and operations can be freely mixed with other features of the language including recursion, higher-order functions, polymorphism. This allows us to write powerful query processing programs relatively easily. The type correctness of programs is then automatically checked at compile time. Moreover, the resulting programs are in general polymorphic and can be shared in many applications. Figure 5.5 shows a simple implementation of a polymorphic transitive closure function. By using a renaming operation (which is definable in Machiavelli), this function can be used to compute the transitive closure of any binary relation. Figure 5.6 shows query processing on the example database using polymorphic functions. The function cost takes a part record as argument and computes the total cost of the part. Without proper integration of the

```
(* A function that computes the total cost of a part *)
-> fun cost(p) =
     (case p.Pinfo of
        BasePart of x=>x.Cost,
        CompositePart of x=>
          x.AssemCost +  hom((fn(y)=>y.SubpartCost * y.Qty),+,0,
                           select [SubpartCost=cost(z),Qty=w.Qty]
                           where w <- x.SubParts, z <- parts
                           with z.P#=w.P#));
>> val cost = fn
   : [('a) Pinfo:<BasePart:[('c) Cost:int],
                  CompositePart:[('d) SubParts:{[('e) P#:int,Qty:int]},
                                  AssemCost:int]>]
        -> int

(* select names of "expensive" parts *)
-> fun expensive_parts(partdb,n) =
     select x.Pname
     where x <- partdb
     with cost(x) > n;
>> val expensive_parts = fn :
   : ({[('a) Pinfo:<BasePart:[('c) Cost:int],
                   CompositePart:[('d) SubParts:{[('e) P#:int,Qty:int]},
                       AssemCost:int]>]},
      int) -> {str}

-> expensive_parts(parts,1000);
>> val it = {"engine",...} : {str}
```

Figure 5.6: Query Processing Using Polymorphic Functions

Chapter 6

Parametric Classes for Object-Oriented Programming

This chapter extends the type system of Machiavelli to include user definable class hierarchies with multiple inheritance declarations. This extension achieves the integration of ML style parametric abstract data types and explicitly defined inheritance hierarchy. The extended type system is sound with respect to the type system of Machiavelli and still has a static type inference algorithm. Some of the results of this chapter were presented in [86]

6.1 Introduction

The idea that is fundamental in object-oriented programming is that each data element (object) belongs to a class and can only be manipulated by methods defined in classes. Moreover, classes are organized by an explicit *inheritance* hierarchy defined by the programmer. The methods that are applicable to an object are not only the ones defined in its own class but also those defined in its all *super-classes*. This mechanism elegantly combines *data abstraction* and *method inheritance*. In particular, inheritance is controlled by the programmer enabling him to develop a taxonomical organization that reflects the intended semantics.

The type system of Machiavelli we have developed in the previous chapter does allow method inheritance by ML style polymorphism but lacks both data abstraction and user control of inheritance. The method inheritance of Machiavelli relies on the explicit structure of record and variant types; inheritance is derived from the polymorphic nature of operations on records and variants. Because of this nature, the type system cannot prevent unintended manipulation of objects based on the knowledge of their implementation details nor can it prevent misuses of methods through a coincidence of implementations. For example, suppose we implement the class *person* by the type $[Name : string, Age : int]$ and define a method *minor* with the polymorphic type $[(u)Age : int] \rightarrow bool$ which determines whether a person's age is less than 21 or not. Machiavelli's polymorphism allows *minor* to be applied not only to objects of the class *person* but also to objects of, say, the class *employee* implemented by the type $[Name : string, Age : int, Salary : int]$, as we expected. However, this method can also be applied to objects of *any* class whose implementation type happens to have an $Age : int$ field. For example, an application might contain the class *pet* implemented by the type $[Name : string, Age : int, Owner : string]$. The method *minor* is equally well applicable to objects of the class *pet* but we want to prevent such applications. In order to represent object-oriented systems, we would like to add an abstraction mechanism with multiple inheritance to the type system of Machiavelli so that the programmer can "hide" implementations of objects and control method sharing.

A well known mechanism for data abstraction in ML style type system is *abstract data type* implemented, for example, in Standard ML and Miranda. An abstract data type is a type associated with a set of user defined functions. Outside of its declaration, the type system treats an abstract data type and its associated functions as if they were a primitive type with an associated set of primitive operations. This mechanism successfully hides the actual implementation of an abstract data type. Moreover, in those languages, abstract data types can be parameterized by types, allowing "generic" definitions. For example, the following fragment of Standard ML code defines a generic set type:

```
abstype 'a set = Set of 'a list  with
  val emptyset = Set nil;
  fun singleton x = Set [x] ;
  fun union s1 s2 = Set (s1@s2) ;
      ...
end;
```

Intuitively this definition defines a family of abstract set types τ **set** for all instances τ of `'a`. The methods **singleton**, **union**, etc. are shared by all these instances types.

A drawback to this approach is that it does not combine data abstraction with inheritance in the same sense that object-oriented programming languages do this. ML style abstract data types do not allow method inheritance even if we extend the underlying type system to the type system of Machiavelli. To illustrate the problem, consider the following abstract data type definition as an implementation of the class **person** (assuming the Machiavelli type system in Standard ML syntax):

```
abstype person = Person of [Name : string, Age : int] with
  fun make_person n a = Person [Name = n, Age = a];
  fun name (Person p) = p.Name;
  fun age (Person p) = p.Age;
  fun increment_age (Person p) = Person(modify(p,Age,p.Age + 1));
end;
```

Now suppose we are to implement the class **employee** by the type `[Name:string, Age:int, Salary:int]`. Since the method **name**, **age** and **increment_age** defined in the class **person** are also applicable to the above type in the type system of Machiavelli, we would like them to be shared by the class **employee**. However, there is no mechanism to allow such sharing in ML style abstract data types. As a result, we are forced to repeat the identical definitions for these methods in the declaration of the class **employee**. For the same reason that we preferred a polymorphic type system to a simple type system, we would like to extend ML style abstract data types with inheritance declarations.

Galileo [7] integrates inheritance and class hierarchy in a static type system by combining the subtype relation (see the analysis of subtypes in section 5.1.1) and abstract data type declarations. However, Galileo does not support polymorphism nor type inference. Jategaonkar and Mitchell suggest [63] the possibility of using their type inference method to extend ML's abstract data types to support inheritance. Here we provide a formal system that achieves the integration of ML style abstract data types and multiple inheritance as an extension of the type system of Machiavelli we have developed in the previous chapter. Moreover, our proposal achieves a proper integration of multiple inheritance in object-oriented programming and type parameterization in ML style abstract data types. As a remark, the class declarations, which can be regarded as a generalization of ML's abstract data types, appear to have no immediate connection with the notion of abstract types as existential types proposed by Mitchell and Plotkin [80].

As an example, the class *person* can be implemented in our language by the following class definition:

class $person = [Name : string, Age : int]$ *with*

  fun $make\_person\ n\ a = [Name = n, Age = a] : string \rightarrow int \rightarrow person;$

```
        fun name p = p.Name : sub → string;
        fun age p = p.Age : sub → int;
        fun increment_age p = modify(p, Age, p.Age + 1) : sub → sub;
    end
```

Outside of the definition, the actual structure of objects of the type *person* is hidden and can only be manipulated through the explicitly defined set of interface functions (methods).

As in Miranda's abstract data types, we require the programmer to specify the type (type-scheme) of each method. The keyword **sub** in the type specifications of methods is a special type variable representing all possible subclasses of the class being defined. It is to be regarded as an assertion by the programmer (which may later prove to be inconsistent with a subclass definition) that a method can be applied to values of any subclass. For example, we may define a subclass

```
    class employee = [Name : string, Age : int, Salary : int] isa person
    with
        fun make_employee n a = [Name = n, Age = a, Salary = 0]
            : string → int → employee;
        fun add_salary e s = modify(e, Salary, e.Salary + s) : sub → int → sub;
        fun salary(e) = e.Salary : sub → int
    end
```

which inherits the methods *name*, *age* and *increment_age*, but not *make_person* from the class *person* because there is no **sub** in the type specification of *make_person*. For reasons that will emerge later we have given the complete record type required to implement *employee*, not just the additional fields we need to add to the implementation of *person*. It is possible that for simple record extensions such as these we could invent a syntactic shorthand that is more in line with object-oriented languages. Continuing in the same fashion we could define classes

```
    class student = [Name : string, Age : int, Grade : string] isa person
    with
        ⋮
    end

    class research_student = [Name : string, Age : int, Salary : int; Grade : string]
    isa {employee, student}
```

```
with
    ⋮
end
```

The second of these illustrates the use of multiple inheritance.

The type system we are presenting can statically check the type correctness of these class definitions containing multiple inheritance declarations. Moreover, the type system always infers a principal conditional typing scheme for expressions containing methods defined in classes. For example, for the following function

$$\textbf{fun } raise\_salary(p) = add\_salary(p, salary(p)/10)$$

which raise the salary of an object approximately by 10%, the type system infers the following principal conditional typing scheme:

$$\emptyset \triangleright raise\_salary : (t < employee) \rightarrow (t < employee)$$

where $(t < employee)$ is a new form of ***conditional type variable*** representing arbitrary subclasses of *employee*. By this type inference mechanism, the type system achieves a proper integration of ML style polymorphism and inheritance. The above function can be applied to objects of any subclass of *employee*. The type correctness of such applications is statically checked.

To demonstrate the use of type parameters, consider how a class for lists might be constructed. We start from a class which defines a "skeletal" structure for lists.

```
class $pre\_list$ = (rec $t.\langle Empty : nil, List : [Tail : t]\rangle$)
with
    val $nil = \langle Empty = Nil\rangle$ : sub;
    fun $tl\ l$ = (case $l$ of
                          $\langle Empty = y\rangle \Rightarrow \ldots error \ldots$,
                          $\langle List = z\rangle \Rightarrow z.Tail$)
        : sub → sub;
    fun $null\ l$ = (case $l$ of
                            $\langle Empty = y\rangle \Rightarrow true$,
                            $\langle List = z\rangle \Rightarrow false$)
        : sub → $bool$;
end
```

By itself, the class *pre_list* is useless for it provides no method for constructing non-empty lists. We may nevertheless derive a useful subclass from it.

```
class $list(u) = (\textbf{rec}\ t.\ \langle Empty : nil, List : [Head : u, Tail : t]\rangle)$ isa $pre\_list$
with
     fun $cons\ h\ t = \langle List = [Head = h, Tail = t]\rangle : u \rightarrow \textbf{sub} \rightarrow \textbf{sub}$;
     fun $hd\ l =$ (case $l$ of
                          $\langle Empty = y\rangle \Rightarrow \ldots error \ldots,$
                          $\langle List = z\rangle \Rightarrow z.Head)$
          $: \textbf{sub} \rightarrow u$
     ⋮
end
```

which is a class for polymorphic lists much as they appear in ML. Separating the definition into two parts may seem pointless here but we may be able to define other useful subclasses of *pre_list*. Moreover, since u may itself be a record type, we may be able to define further useful subclasses of *list*. We will show more examples in section 6.7. The type correctness of these parametric class declarations is also statically checked by the type system and the type inference also extends to methods of parametric classes.

In the following sections we provide the syntax and typing rules for classes that extend Machiavelli type system and show that the extended language is correct with respect to the underlying Machiavelli type system and provide the necessary results to show that there is a type inference algorithm.

6.2 Raw Terms, Types, and Type-schemes

We assume that there are a set of *class constructor symbols* (ranged over by c) and a set of *method names* (ranged over by m). The set of raw terms of the extended language is the set obtained from the set of raw terms of Machiavelli (definition 5.1) by extending with the set of method names:

$$e ::= m \mid c \mid x \mid \lambda x.\, e \mid (e\ e).$$

We continue to use the syntactic shorthands defined in section 5.2 and, in examples, we use the representation of raw terms using term constructors defined in section 5.3.

The set of types of the extended language is the set of regular trees represented by the following syntax:

$$\tau ::= b \mid [l : \tau, \ldots, l : \tau] \mid \langle l : \tau, \ldots, l : \tau\rangle \mid \tau \rightarrow \tau \mid c(\tau, \ldots, \tau) \mid (\mathbf{rec}\ v.\tau(v)).$$

The set of type-schemes is also extended with classes:

$$\rho ::= t \mid b \mid [l : \rho, \ldots, l : \rho] \mid \langle l : \rho, \ldots, l : \rho\rangle \mid \rho \rightarrow \rho \mid c(\rho, \ldots, \rho) \mid (\mathbf{rec}\ v.\rho(v)).$$

We call type-schemes of the form $c(\rho, \ldots, \rho)$ *class schemes.*

6.3 Syntax of Class Definitions

We write $c(\overline{t}\,)$ and $c(\overline{\rho})$ for $c(t_1, \ldots, t_k)$ and $c(\rho_1, \ldots, \rho_k)$ for some k. We also write $[\overline{t} := \overline{\rho}, \ldots]$ for the substitution $[t_1 := \rho_1, \ldots, t_k := \rho_k, \ldots]$ where $\overline{t} = t_1, \ldots, t_k$ and $\overline{\rho} = \rho_1, \ldots, \rho_k$.

A *class definition* has the following syntax:

class $c(\overline{t}\,) = \rho$ **isa** $\{c_1(\overline{\rho_{c_1}}), \ldots, c_n(\overline{\rho_{c_n}})\}$ **with**
  val $m_1 = e_1$: M_1;
  $\vdots$
  val $m_n = e_n$: M_n
end.

$c(\overline{t}\,)$ is the class scheme being defined by this declaration. $\overline{t}$ in $c(\overline{t})$ are type parameters of the class c. ρ is the *implementation type-scheme* of the class $c(\overline{t}\,)$. $\{c_1(\overline{\rho_{c_1}}), \ldots, c_n(\overline{\rho_{c_n}})\}$ is the set of class schemes of *immediate super-classes* from which $c(\overline{t}\,)$ directly inherits methods. If this set is empty then **isa** declaration is omitted. If this set is a singleton set then we omit the braces { and }. Each m_i is the name of method implemented by the code e_i. For method definitions, we use the syntactic shorthand for recursive function definition (section 5.6). M_i is a *method type* specifying the type of m_i, whose syntax is given below:

$$M \quad ::= \quad \mathbf{sub} \mid t \mid b \mid [l : M, \ldots, l : M] \mid \langle l : M, \ldots, l : M\rangle \mid M \rightarrow M \mid c(M, \ldots, M).$$

sub is a distinguished type variable ranging over all subclasses of the class being defined. Note that we restrict method types to be finite types. This is necessary to ensure the decidability of type-checking of class definitions.

We require a class definition to satisfy the following restrictions:

1. all type variables in the definition are contained in the type parameters $\bar{t}$ of the class scheme being defined, and
2. the implementation type-scheme ρ is not a type variable.

These restriction are needed to construct a consistent proof system for parametric classes.

A *class context* $\mathcal{D}$ is a finite sequence of class definitions:

$$\mathcal{D} ::= \emptyset \mid \mathcal{D}; D.$$

A class definition containing type variables is a generic definition of a class. Continuing our interpretation that a type-scheme is a representation of its all ground instances (section 3.2), we regard a generic class definition as a representation of the set of all its ground instances obtained by instantiating $\bar{t}$ with types. The set of type variables $\bar{t}$ are form of bound variables whose scope is the body of the definition for $c(\bar{t})$. Therefore the definition **class** $c(\bar{t}) \ldots$ **end** is equivalent to the one obtained from it by renaming type variables $\bar{t}$.

These declarations are forms of bindings for which we need some mechanism to resolve naming conflict such as visibility rules and explicit name qualifications. Here we ignore this complication and assume that method names and class constructor names are unique in a given class context.

The special type variable **sub** that appears in a method type specifications denotes the set of all possible subclasses that the programmer will declare later. This can be regarded as a form of ***bounded quantification*** proposed by Cardelli and Wegner [27]. The method type M containing **sub** corresponds to $\forall \mathbf{sub} \leq c(\bar{t}\,).\, M$ where $c(\bar{t}\,)$ is the class being defined. The relation $\leq$ is the ***subsumption*** relation induced by the **isa** declarations:

Definition 6.1 *The **subsumption** relation $\mathcal{D} \vdash c_1(\overline{\rho_1}) \leq c_2(\overline{\rho_2})$ induced by $\mathcal{D}$ is the smallest relation containing:*

1. $\mathcal{D} \vdash c(\bar{t}\,) \leq c(\bar{t}\,)$ *if $\mathcal{D}$ contains a class definition of the form* **class** $c(\bar{t}) = \rho \;\cdots$ **end**.
2. $\mathcal{D} \vdash c_1(\overline{t_1}) \leq c_2(\overline{\rho_2})$ *if $\mathcal{D}$ contains a class definition of the form*
   class $c_1(\overline{t_1}) = \rho$ **isa** $\{\ldots, c_2(\overline{\rho_2}), \ldots\}$ **with** $\cdots$ **end**.
3. $\mathcal{D} \vdash c_1(\overline{\rho_1}) \leq c_2(\overline{\rho_2})$ *if $\mathcal{D} \vdash c_1(\overline{\rho_1'}) \leq c_2(\overline{\rho_2'})$ and $(\overline{\rho_1}, \overline{\rho_2})$ is an instance of $(\overline{\rho_1'}, \overline{\rho_2'})$.*
4. $\mathcal{D} \vdash c_1(\overline{\rho_1}) \leq c_2(\overline{\rho_2})$ *if $\mathcal{D} \vdash c_1(\overline{\rho_1}) \leq c_3(\overline{\rho_3})$ and $\mathcal{D} \vdash c_3(\overline{\rho_3}) \leq c_2(\overline{\rho_2})$ for some $c_3(\overline{\rho_3})$.*

The combination of multiple inheritance and type parameterization requires certain restriction on **isa** declarations.

Definition 6.2 *A class context $\mathcal{D}$ is* coherent *if $\mathcal{D} \vdash c_1(\overline{\rho_1}) \leq c_2(\overline{\rho_2})$ and $\mathcal{D} \vdash c_1(\overline{\rho_1}) \leq c_2(\overline{\rho_2'})$ then $\overline{\rho_2} = \overline{\rho_2'}$.*

We require a class context to be coherent. This condition is necessary to develop a type inference algorithm for the extended language. Even if some other formulation of classes in a statically typed polymorphic language is preferable to the system proposed here, I believe that similar issues will arise.

Lemma 6.1 *For a given class context $\mathcal{D}$, it is decidable whether $\mathcal{D}$ is coherent or nor.*

Proof Let $c_1, \ldots, c_n$ be any sequence such that $\mathcal{D}$ contains a class definition of the form

$$\textbf{class } c_i(\overline{t_i}) \cdots \textbf{isa } \{\ldots, c_{i+1}(\overline{\rho_{i+1}}), \ldots\} \cdots \textbf{ end } (1 \leq i \leq n-1).$$

Define $\overline{\rho_i'}$ inductively as follows: $\overline{\rho_1'} = \overline{t_1}$, $\overline{\rho_i'} = \overline{\rho_i}[\overline{t_{i-1}} := \overline{\rho_{i-1}'}]$ $(2 \leq i \leq n)$. Now define $\overline{\rho_{(c_1,\ldots,c_n)}}$ as $\overline{\rho_n'}$. By the definition of subsumption, $\mathcal{D} \vdash c(\overline{\rho}) \leq c'(\overline{\rho'})$ iff there is a sequence $c_1, \ldots, c_n$ such that $\mathcal{D}$ contains **class** $c_i(\overline{t_i}) \cdots$ **isa** $\{\ldots, c_{i+1}(\overline{\rho_{i+1}}), \ldots\} \cdots$ **end** $(1 \leq i \leq n-1)$, $c = c_1, c' = c_n$ and $\overline{\rho'} = \overline{\rho_{c_1,\ldots,c_n}}[\overline{t_1} := \overline{\rho}]$. Therefore $\mathcal{D}$ is coherent iff for each pair (c, c'), $\overline{\rho_{(c_1,\ldots,c_n)}}$ are all identical for all sequences $c_1, \ldots, c_n$ such that $\mathcal{D}$ contains **class** $c_i(\overline{t_i}) \cdots$ **isa** $\{\ldots, c_{i+1}(\overline{\rho_{i+1}}), \ldots\} \cdots$ **end** $(1 \leq i \leq n-1)$, and $c = c_1, c' = c_n$. Since $\mathcal{D}$ is finite, the above condition can be effectively checked. ∎

We say that a subsumption relation $\mathcal{D} \vdash c_1(\overline{\rho_1}) \leq c_2(\overline{\rho_2})$ is *more general* than $\mathcal{D} \vdash c_1(\overline{\rho_1'}) \leq c_2(\overline{\rho_2'})$ if $(\overline{\rho_1'}, \overline{\rho_2'})$ is an instance of $(\overline{\rho_1}, \overline{\rho_2})$. A subsumption relation $\mathcal{D} \vdash c_1(\overline{\rho_1}) \leq c_2(\overline{\rho_2})$ is *principal* if it is more general than all provable subsumption relations between c_1 and c_2. Under the coherency condition, the subsumption relation has the following property:

Lemma 6.2 *If $\mathcal{D}$ is coherent and $\mathcal{D} \vdash c_1(\overline{\rho_1}) \leq c_2(\overline{\rho_2})$ then there is a principal subsumption relation $\mathcal{D} \vdash c_1(\overline{t_1}) \leq c_2(\overline{\rho_2'})$. Moreover, there is an algorithm which, given a coherent class context $\mathcal{D}$ and a pair c_1, c_2, returns either $(\overline{t}, \overline{\rho})$ or failure such that if it retuns $(\overline{t}, \overline{\rho})$ then $\mathcal{D} \vdash c_1(\overline{t}) \leq c_2(\overline{\rho})$ is a principal subsumption relation between c_1, c_2 otherwise there is no subsumption relation between c_1 and c_2.*

Proof The algorithm is defined as follows. Let c, c' be a given pair of class names. If there is no sequence $c_1, \ldots, c_n$ such that $\mathcal{D}$ contains **class** $c_i(\overline{t_i}) \cdots$ **isa** $\{\ldots, c_{i+1}(\overline{\rho_{i+1}}), \ldots\} \ldots$ **end**, $1 \leq i \leq n-1$, $c_1 = c$, $c_n = c'$, then report failure. Otherwise pick one such sequence $c_1, \ldots, c_n$ and return $(\overline{t}, \overline{\rho_{(c_1,\ldots,c_n)}})$, where $\overline{\rho_{(c_1,\ldots,c_n)}}$ is defined in the proof of the previous lemma. If the algorithm reports failure then there is no sequence $c_1, \ldots, c_n$ satisfying the above conditions. By the subsumption rules, it implies that there is no subsumption relation between c and c'. Suppose the algorithm returns $(\bar{t}, \bar{p})$. Then by the subsumption rules this implies $\mathcal{D} \vdash c_1(\bar{t}) \leq c'(\bar{p})$. Let $\mathcal{D} \vdash c_1(\overline{\rho_1}) \leq c'(\overline{\rho_2})$ be any provable subsumption relation. By the rule 3 of subsumption relation, $\mathcal{D} \vdash c_1(\overline{\rho_1}) \leq c'(\bar{p}[\bar{t} := \rho_1])$ is also provable. Since $\mathcal{D}$ is coherent, $\overline{\rho_2} = \bar{p}[\bar{t} := \rho_1]$. ∎

6.4 Proof System for Class Definitions and Typings

The extended type system has the following forms of judgements:

$$MC^+ \vdash \mathcal{D} \qquad \mathcal{D} \text{ is a well typed class context,}$$
$$MC^+ \vdash \mathcal{D}, \mathcal{A} \rhd e : \tau \qquad \text{the typing } \mathcal{D}, \mathcal{A} \rhd e : \tau \text{ is derivable.}$$

where $\mathcal{A}$ stands for type assignments. The proof systems for those two forms of judgements are defined simultaneously.

Let D be a class definition of the form **class** $c(\bar{t}) = \rho_c \cdots$ **end**. D induces the *tree substitution* ϕ_D on type-schemes. For finite type-schemes, $\phi_D(\rho)$ is defined by induction on the structure of ρ as follows:

$$\begin{aligned}
\phi_D(b) &= b, \\
\phi_D(t) &= t, \\
\phi_D(f(\rho_1, \ldots, \rho_n)) &= f(\phi_D(\rho_1), \ldots, \phi_D(\rho_n)) \text{ for any type constructor } f \text{ s.t. } f \neq c, \\
\phi_D(c(\bar{p})) &= \rho_c[\bar{t} := \phi_D(\bar{p})].
\end{aligned}$$

Since ρ_c is not a type variable, ϕ_D is a *non-erasing second-order substitution* on trees [32] which extend uniquely to regular trees. See [32] for the technical details. Since regular trees are closed under second-order substitution [32], $\phi_D(\rho)$ is a well defined type-scheme.

Definition 6.3 (Proof System for Class Definitions) *The rule for $MC^+ \vdash \mathcal{D}$ is defined by induction on the length of $\mathcal{D}$:*

1. *The empty class context is a well typed class context, i.e.* $\vdash \emptyset$.
2. *Suppose* $\vdash \mathcal{D}$. *Let D be the following class definition:*

   class $c(\bar{t}\,) = \rho$ **isa** $\{c_1(\overline{\rho_{c_1}}), \ldots, c_n(\overline{\rho_{c_n}})\}$
   with

val $m_1 = e_1 : M_1$;

$\vdots$

val $m_n = e_n : M_n$;

end.

Then $MC^+ \vdash \mathcal{D}; D$ *if the following conditions hold:*

(a) it is coherent,

(b) if a class name c' *appear in some of* $\rho, \rho_{c_1}, \ldots, \rho_{c_n}$ *then* $\mathcal{D}$ *contains a definition of the form* **class** $c'(\overline{t'}) \cdots$ **end**,

(c) $MC^+ \vdash \mathcal{D}, \emptyset \rhd e_i : \tau$ *for any ground instance* τ *of* $\phi_D(M_i[\mathbf{sub} := \rho])$,

(d) for any method **val** $m = e_m : M_m$ *defined in some definition of class* $c'(\overline{t'})$ *in* $\mathcal{D}$ *such that* $\mathcal{D} \vdash c(\overline{t}\,) \le c'(\overline{\rho'})$, $MC^+ \vdash \mathcal{D}, \emptyset \rhd e_m : \tau$ *for any ground instance* τ *of* $M_m[\overline{t'} := \overline{\rho'}, \mathbf{sub} := \rho]$.

We have already discussed the necessity of the condition (a). The necessity of the condition (b) is obvious. The condition (c) states that each method defined in the definition of the class $c(\overline{t}\,)$ is type consistent with its own implementation. The condition (d) ensures that all methods of all super classes that are already defined in $\mathcal{D}$ are also applicable to the class $c(\overline{t}\,)$. This is done by checking the type consistency of each method e_m defined in a super class against the type-scheme obtained from M_m by instantiating its type variables with type-schemes specified in **isa** declaration in the definition of the class $c(\overline{t})$ and replacing the variable **sub** with the implementation type-scheme ρ of the class $c(\overline{t}\,)$.

Definition 6.4 (Proof System for Typings) *The proof system for typings of the extended language is the one obtained from the proof system for Machiavelli (section 5.2.3, definition 3.22) by changing typing formula* $\mathcal{A} \rhd e : \tau$ *to* $\mathcal{D}, \mathcal{A} \rhd e : \tau$ *and adding the following rule:*

(METHOD) $MC^+ \vdash \mathcal{D}, \mathcal{A} \rhd m : \tau$

if $MC^+ \vdash \mathcal{D}$ *and there is a method* **val** $m = e : M$ *of a class* $c(\overline{t}\,)$ *in* $\mathcal{D}$ *such that* τ *is an instance of* $M[\overline{t} := \overline{\rho}, \mathbf{sub} := c'(\overline{t'})]$ *for some* $\mathcal{D} \vdash c'(\overline{t'}) \le c(\overline{\rho}\,)$.

6.5 Soundness of the Type System

We show the correctness of the type system for the extended language with respect to the type system of Machiavelli.

Let $\mathcal{D}$ be a given class context and τ be a type. The *exposure of τ under $\mathcal{D}$*, denoted by $expose_{\mathcal{D}}(\tau)$, is the type given by the following inductive definition on the length of $\mathcal{D}$:

1. if $\mathcal{D} = \emptyset$ then $expose_{\mathcal{D}}(\tau) = \tau$,

2. if $\mathcal{D} = \mathcal{D}'; D$ then $expose_{\mathcal{D}}(\tau) = expose_{\mathcal{D}'}(\phi_D(\tau))$.

Intuitively, $expose_{\mathcal{D}}(\tau)$ is the type obtained from τ by recursively replacing all its classes by their implementation type-schemes. Since $expose_{\mathcal{D}}$ is a composition of second-order-substitutions which is also a second-order substitution, the following property follows from their general properties [32]:

Lemma 6.3 *Let $\mathcal{D}$ be any class context and f be any type constructors other than class names appears in $\mathcal{D}$. $expose_{\mathcal{D}}(f(\tau_1, \ldots, \tau_n)) = f(expose_{\mathcal{D}}(\tau_1), \ldots, expose_{\mathcal{D}}(\tau_n))$.* ▮

We extend *expose* to any syntactic structures that contain types. The above property also extends to such syntactic structures.

The *unfold* of a raw term e under a class context $\mathcal{D}$, denoted by $unfold_{\mathcal{D}}(e)$, is the raw term given by the following inductive definition:

1. if $\mathcal{D} = \emptyset$ then $unfold_{\mathcal{D}}(e) = e$,

2. if $\mathcal{D} = \mathcal{D}'$;**class** ... **with**

   val $m_1 = e_1 : M_1$;

   $\vdots$

   val $m_n = e_n : M_n$

   end,

   then $unfold_{\mathcal{D}}(e) = unfold_{\mathcal{D}'}(e[e_1/m_1, \ldots, e_n/m_n])$.

$unfold_{\mathcal{D}}(e)$ is the raw term obtained from e by recursively replacing all method names defined in $\mathcal{D}$ with their implementations. The following property is an immediate consequence of the definition:

Lemma 6.4 *For any class context $\mathcal{D}$,*

$$\begin{aligned}
unfold_{\mathcal{D}}(c) &= c, \\
unfold_{\mathcal{D}}(x) &= x, \\
unfold_{\mathcal{D}}((e_1\ e_2)) &= (unfold_{\mathcal{D}}(e_1)\ unfold_{\mathcal{D}}(e_2)), \\
unfold_{\mathcal{D}}(\lambda x.\, e) &= \lambda x.\, unfold_{\mathcal{D}}(e). \ ▮
\end{aligned}$$

Theorem 6.1 *If* $MC^+ \vdash \mathcal{D}, \mathcal{A} \rhd e : \tau$ *then* $MC \vdash expose_{\mathcal{D}}(\mathcal{A}) \rhd unfold_{\mathcal{D}}(e) : expose_{\mathcal{D}}(\tau)$.

Proof Proof is by induction on the length of $\mathcal{D}$. The basis hold since if $\mathcal{D} = \emptyset$ then $MC^+ \vdash$ reduced to $MC \vdash$ and $expose_{\mathcal{D}}, expose_{\mathcal{D}}$ are both identity functions.

The proof for the induction step is by induction on the structure of e. Let $\mathcal{D}$ be the following class context:

$\mathcal{D}'$;**class** $c(\overline{t}) = \rho \ldots$ **with**

  val $m_1 = e_1 : M_1$;

  $\vdots$

  val $m_n = e_n : M_n$

 end.

The cases other than m follows directly from the above two lemmas and the induction hypothesis (in terms of the structure of e). Suppose $MC^+ \vdash \mathcal{D}, \emptyset \rhd m : \tau$. Then by definition of the extended proof system for typings (i.e. the rule (METHOD)), there is a method **val** $m = e : M$ in a definition of a class $c_1(\overline{t_1})$ in $\mathcal{D}$ such that τ is an instance of $M[\overline{t_1} := \overline{\rho_1}, \textbf{sub} := c_2(\overline{t_2})]$ for some $\mathcal{D} \vdash c_2(\overline{t_2}) \leq c_1(\overline{\rho_1})$. We need to prove that $MC \vdash \emptyset \rhd unfold_{\mathcal{D}}(m) : expose_{\mathcal{D}}(\tau)$. We distinguish the following cases:

1. Case $c_1 = c_2 = c$: m must be one of the m_i in the definition of c and $\overline{t_2} = \overline{\rho_1}$, which is a renaming of $\overline{t}$. By the definitions of $unfold_{\mathcal{D}}$ and $expose_{\mathcal{D}}$, $unfold_{\mathcal{D}}(m_i) = unfold_{\mathcal{D}'}(e_i)$ and $expose_{\mathcal{D}}(\tau) = expose_{\mathcal{D}'}(\phi_D(\tau))$. But since $\overline{t_2} = \overline{\rho_1}$, which is a renaming of $\overline{t}$, and M_i is finite, by the inductive definition of ϕ_D, $\phi_D(\tau)$ is an instance of $\phi_D(M_i[\textbf{sub} := \rho])$. Since $MC^+ \vdash \mathcal{D}, \emptyset \rhd m : \tau$ implies $MC^+ \vdash \mathcal{D}$, by the definition of $MC^+ \vdash \mathcal{D}$, $MC^+ \vdash \mathcal{D}', \emptyset \rhd e_i : \tau'$ for any instance τ' of $\phi_D(M_i[\textbf{sub} := \rho])$. In particular, $MC^+ \vdash \mathcal{D}', \emptyset \rhd e_i : \phi_D(\tau)$. By the induction hypothesis (of the main induction), $MC \vdash \emptyset \rhd unfold_{\mathcal{D}'}(e_i) : expose_{\mathcal{D}'}(\phi_D(\tau))$. Therefore $MC \vdash \emptyset \rhd unfold_{\mathcal{D}}(m) : expose_{\mathcal{D}}(\tau)$.

2. Case $c_2 = c$ and $c_1 \neq c$: By the definitions of $unfold_{\mathcal{D}}$ and $expose_{\mathcal{D}}$, $unfold_{\mathcal{D}}(m) = unfold_{\mathcal{D}'}(m)$ and $expose_{\mathcal{D}}(\tau) = expose_{\mathcal{D}'}(\phi_D(\tau))$. Since τ is an instance of $M[\overline{t_1} := \overline{\rho_1}, \textbf{sub} := c_2(\overline{t_2})]$, $c_2 = c$, and none of $M, \overline{\rho}, \overline{\rho_1}$ contain c_2, $\phi_D(\tau)$ is an instance of $M_i[\overline{t_1} := \overline{\rho_1}, \textbf{sub} := \rho]$. But since $MC^+ \vdash \mathcal{D}, \emptyset \rhd m : \tau$ implies $MC^+ \vdash \mathcal{D}$, by the definition of $MC^+ \vdash \mathcal{D}$, $MC^+ \vdash \mathcal{D}', \emptyset \rhd m : \tau'$ for any instance τ' of $M[\overline{t_1} := \overline{\rho_1}, \textbf{sub} := \rho]$. In particular, $MC^+ \vdash \mathcal{D}', \emptyset \rhd m : \phi_D(\tau)$. Then by the induction hypothesis (of the main induction), $MC \vdash \emptyset \rhd unfold_{\mathcal{D}'}(m) : expose_{\mathcal{D}'}(\phi_D(\tau))$. Therefore $MC \vdash \emptyset \rhd unfold_{\mathcal{D}}(m) : expose_{\mathcal{D}}(\tau)$.

3. Case $c_1 \neq c$ and $c_2 \neq c$: Then $\mathit{unfold}_{\mathcal{D}}(m) = \mathit{unfold}_{\mathcal{D}'}(m)$, $\mathit{expose}_{\mathcal{D}}(\tau) = \mathit{expose}_{\mathcal{D}'}(\tau)$ and $\boldsymbol{MC} \vdash \emptyset \rhd \mathit{unfold}_{\mathcal{D}'}(m) : \mathit{expose}_{\mathcal{D}'}(\tau)$ implies $\boldsymbol{MC} \vdash \emptyset \rhd \mathit{unfold}_{\mathcal{D}}(m) : \mathit{expose}_{\mathcal{D}}(\tau)$. The desired result follows from the induction hypothesis (of the main induction).

By the definition of subsumption relation, $c_1 = c$ and $c_2 \neq c$ contradict the assumption that $\mathcal{D} \vdash c_2(\overline{t_2}) \leq c_1(\overline{\rho_1})$. Therefore we have exhausted all cases. ∎

This theorem establishes the correctness of the type system with respect to the type system of the core language. In particular, since the type system of the core language prevents all run-time type errors, a type correct program in the extended language cannot produce run-time type error.

The converse of this theorem, of course, does not hold, but we would not expect it to hold, for one of the advantages of data abstraction is that it allows us to distinguish two methods that may have the same implementation. As an example, suppose $\mathcal{D}$ contains definitions for the classes *car* and *person* whose implementation type-schemes coincide and *person* has a method *minor* which determines whether a person is younger than 21 or not. By the coincidence of the implementations, $\vdash \emptyset \rhd \mathit{expose}(\mathit{minor}(c)) : \mathit{bool}$ for any *car* object c. But $\vdash \mathcal{D}, \mathcal{A} \rhd \mathit{minor}(c) : \mathit{bool}$ is not provable unless we declare (by a sequences of **isa** declarations) that *car* is a subclass of *person*. This prevents illegal use of a method via a coincidence of the implementation type-schemes.

6.6 Type Inference for the Extended Language

We now solve the type inference problem for the extended language by defining an algorithm to compute a principal conditional typing schemes. For this purpose we introduce a new form of conditions.

Definition 6.5 *A subsumption condition is a formula of the form* $isa(\rho, c(\overline{\rho}))$. *A substitution* θ *satisfies the subsumption condition under the class context* $\mathcal{D}$ *if* $\mathcal{D} \vdash \theta(\rho) \leq \theta(c(\overline{\rho}))$.

Note that the satisfiability for subsumption conditions is defined relative to a class context.

The notion of conditional typing schemes is now extended to include subsumption conditions.

Definition 6.6 *A conditional typing scheme for the extended language is a formula of the form* $\mathcal{D}, C, \Sigma \rhd e : \rho$ *such that for any substitution* θ *that is ground for* Σ, C *and* ρ *and that satisfies* C *under the class context* $\mathcal{D}$, $\boldsymbol{MC}^{+} \vdash \mathcal{D}, \theta(\Sigma) \rhd e : \theta(\rho)$.

The definition for principal conditional typing schemes is the same as before.

Since the typing judgment $MC^{+} \vdash \mathcal{D}, \mathcal{A} \rhd e : \tau$ implies the well typedness $MC^{+} \vdash \mathcal{D}$ of the class context $\mathcal{D}$, in order to define a type inference algorithm we also need to define a type-checking algorithm for class contexts. A subtle complication in defining these algorithm is that we need to develop them simultaneously, since the two forms of judgements are mutually dependent. We solve this problem in the following two stages:

1. to develop an algorithm that computes a principal conditional typing scheme *under a type correct class context* $\mathcal{D}$ (proposition 6.1); then
2. to develop an algorithm to decide whether a class context is well typed or not using the above result (proposition 6.2).

Proposition 6.1 *For any raw term e, and any class context $\mathcal{D}$ if e has a typing under $\mathcal{D}$ then it has a principal conditional typing scheme. Moreover, there is an algorithm which, given any raw term and any well typed class context $\mathcal{D}$, computes a principal conditional typing scheme if one exists otherwise reports failure.*

Proof If $\mathcal{D}$ is not well typed then by definition there is no typing of e uner $\mathcal{D}$. Suppose $\mathcal{D}$ is well typed. The algorithm to compute a principal conditional typing scheme is obtained from the algorithm $\mathcal{CTS}$ (theorem 5.1) for Machiavelli by adding a parameter $\mathcal{D}$ and the following case for methods:

$$\mathcal{CTS}(\mathcal{D}, e) = (C, \Sigma, \rho) \text{ where}$$

```
(6) Case $e \equiv m$:
    let
        **val** $m = e : M$ be the method defined in the class definition for $c(\overline{t})$ in $\mathcal{D}$
    in
        $C = \{(isa(t_2, c(\overline{t_1}))\}$ ($\overline{t_1}, t_2$ fresh),
        $\Sigma = \emptyset$,
        $\rho = M[\overline{t} := \overline{t_1}, \mathbf{sub} := t_2]$
```

Other cases are the same as before. In particular, they do not depend on $\mathcal{D}$.

Since ρ returned in the case above is a type-scheme (i.e. it does not contain **sub**) and a set of conditions denotes the conjunction of all its elements, the correctness of the algorithm for the above case follows immediately from the definition of the condition $isa(\rho, c(\overline{p}))$. The proof of the correctness of the other cases is same as the proof in theorem 5.1. ∎

Proposition 6.2 *For any class context $\mathcal{D}$, it is decidable whether $\boldsymbol{MC}^+ \vdash \mathcal{D}$ or not.*

Proof The proof is by induction on the length of $\mathcal{D}$. Basis is trivial. For the induction step, let $\mathcal{D}$ be the following class context:

$\mathcal{D}'$;**class** $c(\overline{t}) = \rho \ldots$ **with**

  val $m_1 = e_1 : M_1$;

  ⋮

  val $m_n = e_n : M_n$

 end.

In order to decide whether $\boldsymbol{MC}^+ \vdash \mathcal{D}$, we need to check the four conditons (a) – (d) of definition 6.3. We have shwon by lemma 6.1 the decidability of condition (a). The decidability of condition (b) is obvious. We show that condition (c) is decidable. Since M_i is finite, $\phi_D(M_i[\textbf{sub} := \rho])$ is effectively computable by the inductive definition of ϕ_D. By induction hypothesis, it is decidable whether $\boldsymbol{MC}^+ \vdash \mathcal{D}$ or not. Then by proposition 6.1, we can compute a principal conditional typing scheme $\boldsymbol{MC}^+ \vdash \mathcal{D}, C, \Sigma \rhd e_i : \rho$. We can then decide whether $\boldsymbol{MC}^+ \vdash \mathcal{D}, \emptyset, \emptyset \rhd e_i : \phi_D(M_i[\textbf{sub} := \rho])$ is a conditional typing scheme or not by checking whether there is a substitution θ that satisfies C and $\theta(\Sigma, \rho) = (\emptyset, \phi_D(M_i[\textbf{sub} := \rho])$. This implies the decidability of condition (c). Since the subsumption condition is decidable (lemma 6.2), the decidability of condition (d) is shown similarly to condition (c). ∎

We now have a complete type inference algorithm:

Theorem 6.2 *There is an algorithm which, given any raw term e and any class context $\mathcal{D}$, computes a principal conditional typing scheme of e under $\mathcal{D}$ if one exists otherwise reports failure.*

Proof The algorithm is first checks the well typedness of the class context $\mathcal{D}$ by using proposition 6.2 and if it is well typed then computes a principal conditional typing shceme by using proposition 6.1 otherwise reports failure. The correctness of the algorithm follows from the definition of typings. ∎

In subsection 5.4.2 we noted that the existence of a conditional typing scheme is not enough for complete type-checking because a set of conditions may not be satisfiable. Subsumption conditions we have introduced in this section is another source of unsatisfiability. To see this consider the following examples. Suppose $\mathcal{D}$ contains the class definitions for *person* and *car* (without type parameters) with the method

fun $make\_person(n, a) = \ldots : (string * int) \rightarrow person$

for *person* and the method

fun $fuel(c) = \ldots : \mathbf{sub} \rightarrow string$

for *car*. Now consider the expression: $fuel(make\_person("Joe", 21))$. This expression has the following conditional typing scheme:

$$MC^{+} \vdash \mathcal{D}, \{isa(person, car)\}, \emptyset \triangleright fuel(make\_person("Joe", 21)) \ : \ string$$

which has no instance because of the unsatisfiable condition $isa(person, car)$ (unless *person* is defined to be a subclass of *car* in $\mathcal{D}$). As we have done for Machiavelli we develop a method to detect these inconsistency of subsumption conditions by transforming them into a simpler form.

Definition 6.7 *A set of condition C is in weak subsumption normal form if the following properties hold:*

1. *if $isa(\rho_1, c(\overline{\rho_2})) \in C$ then ρ is a type variable,*

2. *if $isa(t, c_1(\overline{\rho_1})) \in C$ and $isa(t, c_2(\overline{\rho_2})) \in C$ then $c_1 \neq c_2$.*

Proposition 6.3 *If a set of conditions C is satisfiable under a given coherent class context $\mathcal{D}$, then there is a set C' of conditions which is a refinement of C and is in weak subsumption normal form. Moreover, there is an algorithm which, given a set of conditions C and a coherent class context $\mathcal{D}$, computes either a pair (C', θ) or unsatisfiable such that if it returns (C', θ) then C' is in weak subsumption normal form and is a refinment of C with θ a refinement substitution otherwise, C is unsatisfiable under $\mathcal{D}$.*

Proof We first define a set of transformation rules on pairs of a set of conditions and a substitution:

$$(C \cup \{isa(c_1(\overline{\rho_1}), c_2(\overline{\rho_2}))\}, \theta) \Longrightarrow$$

$(\theta'(C), \theta' \circ \theta)$ where $\theta' = \mathcal{U}((\overline{\rho_1}, \overline{\rho_2}), (\overline{\rho_3}, \overline{\rho_4}))$ such that $\mathcal{D} \vdash c_1(\overline{\rho_3}) \leq c_2(\overline{\rho_4})$

is a principal subsumption relation between c_1 and c_2 under $\mathcal{D}$,

$$(C \cup \{isa(t, c_1(\overline{\rho_1})), isa(t, c_1(\overline{\rho_2}))\}, \theta) \Longrightarrow$$
$$(\theta'(C \cup \{isa(t, c_1(\overline{\rho_1}))\}), \theta' \circ \theta) \text{ where } \theta' = \mathcal{U}(\overline{\rho_1}, \overline{\rho_2})$$

We now define the algorithm $\mathcal{SUB}$ as:

$$\mathcal{SUB}(C) = \begin{cases} (C', \theta) & \text{if } (C, id) \overset{*}{\Longrightarrow} (C', \theta) \text{ and } C' \text{ is in weak subsumption normal} \\ & \text{form} \\ \mathit{unsatisfiable} & \text{otherwise} \end{cases}$$

(by assuming some ordering on applications of the rules).

Since the both transformation rules strictly reduce the number of conditions in C, the algorithm always terminates. The correctness of the algorithm is proved similar to the proof of proposition 5.5 using lemma 6.2 and the coherent assumption on $\mathcal{D}$. ∎

We can then extend theorem 5.2 with subsumption conditions:

Theorem 6.3 *Let C be a set of condition that does not contain join conditions and all projection conditions are those that have a finite target type. If C is satisfiable then there is a refinment C' of C that is in record-variant normal form and in weak subsumption normal form. Moreover, there is an algorithm to compute C' and a refinement substitution θ if they exist.*

Proof Let $\Longrightarrow$ be the transformation relation obtained by combining the transformation rules defined in the proof of proposition 5.5 and the proof of proposition 6.3. Now define the algorithm $\mathcal{RVPS}$ as follows:

$$\mathcal{RVPS}(C) = \begin{cases} (C'', \theta) & \text{if } C' = \mathcal{FP}(C), (C', id) \overset{*}{\Longrightarrow} (C'', \theta) \text{ such that} C'' \text{ is in} \\ & \text{record-variant normal form and in weak subsumption normal form} \\ \mathit{failure} & \text{otherwise} \end{cases}$$

Since all the rules defined in the proof of proposition 5.5 and the proof of proposition 6.3 strictly reduce the number of conditions, $\Longrightarrow$ also strictly reduces the number of conditions and the algorithm always terminates. The correctness of the algorithm follows from theorem 5.2 and proposition 6.3. ∎

Similar to record and variant conditions, subsumption conditions in weak subsumption normal form also have a compact representation. We extend the set of conditional type-schemes defined in subsection 5.4.2 to include representations of subsumption conditions:

$$\begin{aligned} T \quad ::= \quad & t \mid \langle (t) l : T, \ldots, l : T \rangle \mid [(t) l : T, \ldots, l : T] \mid (t \leqslant \sigma) \mid (t \; isa(c_1(\overline{T_1}), \ldots, c_n(\overline{T_n}))) \mid \\ & b \mid T \to T \mid [l : T, \ldots, l : T] \mid \langle l : T, \ldots, l : T \rangle \mid (\mathbf{rec}\ v.\, T(v)) \end{aligned}$$

where $(t\ isa((c_1(\overline{T_1}), \ldots, c_n(\overline{T_n})))$ is a type variable associated with the set of subsumption conditions $isa(t, c_i(\overline{T_i})), 1 \leq i \leq n$. Conditional typing schemes for raw terms that do not contain joins and projections on infinite target types can be represented by these conditional type-schemes.

Note however that theorem 6.3 is weaker than theorem 5.2. Unlike the record-variant normal form (definition 5.7), the fact that C is in weak subsumption normal form does not implies that all subsumption conditions in C are satisfiable. To see this consider the following example. Suppose $\mathcal{D}$ consists of the two class definitions *student* and *employee* with the method

fun *grade s* $= \ldots :$ **sub** $\rightarrow$ *int*

for *student* and the method

fun *salary e* $= \ldots :$ **sub** $\rightarrow$ *int*

for *employee*. Now consider the following function definition:

fun *saraly_and_grade p* $= (salary(p), grade(p))$.

This function has the following conditional typing scheme:

$$MC^+ \vdash \mathcal{D}, \emptyset, \emptyset \rhd saraly\_and\_grade : (t\ isa(student, employee)) \rightarrow (int * int).$$

But since there is no class that is a subclsss of both *student* and *employee*, the above typing scheme has no instance.

As a formal system it is easy to fix this problem by adding the following condition to that of weak subsumption normal form of a set C of conditions under a class context $\mathcal{D}$:

> if $isa(t, c_1(\overline{p_1})) \in C$, $isa(t, c_2(\overline{p_2})) \in C, \ldots,$ $isa(t, c_n(\overline{p_n})) \in C$ then there is some $c(\overline{p})$ and a substitution θ such that $\mathcal{D} \vdash c(\theta(\overline{p})) \leq c_i(\theta(\overline{p_i}))$ for all $1 \leq i \leq n$.

Since the number of classes defined in $\mathcal{D}$ is finite, the above condition is shown to be decidable by checking against all classes defined in $\mathcal{D}$ by using lemma 6.2. However, in practice, $\mathcal{D}$ might become very large and such test might be prohibitively expensive. We think that the weak subsumption normal form is satisfactory for practical purpose. The only remaining subsumption conditions are on function types. This means that the existence of weak subsumption normal form guarantees that all applications of methods to objects are type correct. This solution is similar to the one we have adopted for the satisfiability checking of join conditions and projection conditions with an infinite target type (subsection 5.4.2). Also note that this problem does not arise in type-checking a class context because of explicit type specifications of methods.

6.7 Further Examples

In section 6.1, we defined the classes ***person*** and ***employee***. The sequence of the two definitions is indeed a type correct class context in our type system. Figure 6.1 shows an example of an interactive session involving these class definition in our prototype implementation. `('a < person)` and `('a < employee)` are bounded type variables. As seen in this example, the system displays the set of all inherited method for each type correct class definition.

Let us look briefly at some further examples of how type parameterization can interact with inheritance. At the end of section 6.1 we defined a polymorphic list class *list*(a). We could immediately use this by implicit instantiation of a. For example, the function

fun $sum\ l =$ **if** $null(l)$ **then** 0 **else** $hd(l) + sum(tl(l))$

will be given the type $list(int) \rightarrow int$, as would happen in ML. However we can instantiate the type variable a in other ways. For example, we could construct a class

class $genintlist(b) =$ (**rect**. $\langle Empty : nil, List : [Head : [Ival : int, Cont : b], Tail : t]\rangle$)
isa $list([Ival : int, Cont : b])$
with
⋮
end

which could be used, say, as the implementation for a "bag" of values of type b. In this case all the methods of *pre_list* and *list* are inherited. However, we might also attempt to create a subclass of *list* with the following declaration in which we directly extend the record type of the *List* variant of the implementation:

class $genintlist(b) =$ (**rec** t. $\langle Empty : nil, List : [Head : int, Cont : b, Tail : t]\rangle$)
isa $list(int)$
with
⋮
end

```
-> class person = [Name:string,Age:int]
   with
      :
   end;

>> class person with
      make_person :  (string*int) -> person
      name :  ('a < person) -> string
      age :  ('a < person) -> int
      increment_age :  ('a < person) -> ('a < person)

-> class employee = [Name:string,Age:int,Salary:int]
   with
   :
   end;

>> class employee isa person with
      make_employee :  (string*int) -> employee
      add_salary :  (('a < employee)*int) -> (a' < employee)
      salary :  ('a < employee) -> int
   inherited methods:
      name :  ('a < person) -> string
      age :  ('a < person) -> int
      increment_age :  ('a < person) -> ('a < person)

-> val joe = make_person("Joe",21);
>> val joe = _ :  person

-> val helen = make_employee("Helen",31)
>> val helen = _ :  employee

-> age(joe);
>> 21 :  int

-> val helen = increment_age(helen);
>> val helen = _ :  employee

-> age(helen);
>> 32 :  int

-> fun wealthy e = salary(e) > 100000;
>> val wealth = fn :  ('a < employee) -> bool
```

Figure 6.1: A Simple Interactive Session with Classes

In this class, all the methods of *pre_list* could be inherited but the method *cons* of *list*(a) cannot be inherited because the implementation type-scheme of *genintlist*(b) is incompatible with any of the possible types of *cons*. In this case, the type checking for class definition fails and the type system reports an error.

6.8 Limitations and Implementation

First, we should point out that the language we have proposed differs in some fundamental ways from object-oriented languages in the Smalltalk tradition. A static type system does not fit well with *late binding* – a feature of many object-oriented languages. One reason to have late binding might be to implement *overriding* of methods. It is possible that some form of overloading could be added to the language to support this.

Another limitation is the restriction we imposed on inheritance declarations in connection to type parameters. We required that if a class $c(t_1,\ldots,t_n)$ is a subtype of both $c'(\tau_1,\ldots,\tau_n)$ and $c'(\tau'_1,\ldots,\tau'_n)$ then $\tau_j = \tau'_j$ for all $1 \leq i \leq n$. This is needed to preserve the existence of principal conditional typing schemes for all typable raw terms. This disallows certain type consistent declarations such as:

```
class $C_1(t) = \rho$ with
      fun $m\ x = m(x) : \mathbf{sub} \rightarrow t$
end;

class $C_2 = \rho'$ isa $\{C_1(int), C_1(bool)\}$ with
      $c = e : C_2$
      ⋮
end.
```

This definition is type consistent in any implementation type-schemes ρ, ρ' but creates a problem that terms like $m(c)$ do not have a principal conditional typing scheme. However, we believe that the condition is satisfied by virtually all natural declarations. Note that in the above example the result type of the method m is the free type variable t without any dependency of its domain type **sub** which reflects the property that the method m does not terminates on any input. I could not construct any natural example that is type consistent but does not satisfy this coherent condition.

Form a practical perspective, checking the type-correctness of a class definition with **isa** declaration requires the consistency checking of all methods of all super-types already defined. A naive way to do this would involve recursively unfolding definitions of types and method and then type-checking the resulting raw term in the type system of the core language, which will be prohibitively expensive when the class hierarchy become large. This problem is avoided using the existence of a principal conditional typing scheme for any typable raw term in the extended language. At the time of a definition of each method, we can save its principal conditional typing scheme. The type correctness of the method against a newly defined subclass can then be checked by checking whether the required method type is an instance of its principal conditional type-scheme or not. This eliminates repeated type-checking of method bodies but still requires checking of type correctness against the set of all inherited methods. This can be also avoided. The set of all possible implementation type-schemes of subclass of a class can be represented by a single principal conditional type-scheme. As an example, consider the example of *person* we defied in the introduction. The most general conditional type-scheme of the type variable **sub** in the definition of *person* can be computed as $[(t)\ Name : string, Age : int]$. Using this property, the type correctness of a subclass declaration can be checked by checking that the implementation type-scheme is an instance of $[(t)\ Name : string, Age : int]$ without checking the consistencies of each method. While the number of inherited methods might become very large, we expect that the number of super-classes is relatively small even in development of a large system and therefore that this strategy yields an efficient implementation of a static type-checking of large class hierarchy.

Chapter 7

Object-identities and Views for Object-oriented Databases

This chapter extends Machiavelli with reference types to represent database objects with "identities" and describes a method to represent object-oriented databases.

7.1 Introduction

As we have demonstrated through examples in section 5.7, Machiavelli provides a suitable medium to represent the relational and other complex object models in ML style type system. In these models, database objects are pure values in the sense that two objects are equal iff they denote a same regular tree (remember the definition of *eq* primitive in Machiavelli in section 5.5). In those value-based database systems, real-world objects are represented by sets of their attribute values. Moreover, the information about a single real-world object such as a person might be stored in various places. Query processing is done by manipulating these values using join, projection and other operations.

In contrast to those value-based approach, many other data models have been developed based on the intuitively appearing idea that a real-world entity should be directly represented by a single database object. Perhaps the first well established model based on this idea is the entity-relationship model [28]. Many recent proposals such as [46, 74, 67] also integrate the features of object-oriented programming, forming the increasingly popular area of "semantic" data models and object-oriented databases. See [12, 58] for surveys in this area.

There have been arguments [103, 12] that object-oriented databases provide better solutions to problems of database programming than those provided by value-based systems. It is however apparent that there are many applications for which value-based systems provide simpler and more elegant solutions. As an example, recent development of a formalism in natural language processing called *feature structures* [97] strongly suggests that the databases for linguistic information are best represented in a value-based system. On the other hand many ideas developed in object-oriented databases such as object identities and extents of classes have obvious practical benefits. I believe that the real problem we should solve is to integrate these two features in a unified type system so that the programmer can enjoy both advantages of the two approaches. There have been also argued [12, 66] that value-based database systems such as the relational model and its extensions do not well fit a type system of a programming language. Through chapter 4 to chapter 5, we have just shown the opposite by integrating very general complex objects into a polymorphic type system of a programming language. In this chapter we present a method to integrate the features of object-oriented database programming into our type system.

There have been a number of arguments on the properties of object-oriented databases [65, 17, 12]. Here rather than adding to this philosophical discussion, we restrict our attention to the notions of *object identities* and *extents*. When combined with the central features of object-oriented programming, they provide what we believe to be the desired features of programming with object-oriented databases. We have integrated central features of object-oriented programming in Machiavelli in chapter 5 and 6. In this chapter we analyze object identities and extents and propose a method to represent them in Machiavelli.

The notion of object identities is based on the intuitive idea that database objects should model real-world entities that change their attributes while maintaining their "identities". The properties of objects with identities can be summarized as follows:

1. two objects are equal if and only if they are identical (i.e. they are created by the same instance of the creation operation),
2. an object has a set of attribute values that can be changed without affecting its identity,
3. an object is referred and accessed independently of the values of its attributes.

The practical importance of these properties is that they nicely represent sharing and mutability, which are rather cumbersome to represent in a pure value-based system. Objects with identities are usually implemented by maintaining a special value space such as "object identifier" [67] and "key" [7]. Objects are referred and accessed by those special values. User are required to create objects so that their identifying values are unique. The system enforces the uniqueness requirement.

The notion of *extent* is related to the notion of classes in object-oriented programming we have analyzed and integrated in Machiavelli in chapter 6. A class in object-oriented programming can be regarded as an association of a (hidden) structure defining the internal representation of objects and a set of operations defining their external behavior. Such classes are hierarchically organized by inheritance relation supporting method sharing. For example, if we define a class *point* with a method $move(p, x, y)$ that displaces a point p by co-ordinates x and y, we can define a subclass, *circle*, of *point* and expect that the same method, *move*, can be applied to instances of the class *circle*. In object-oriented databases, the notion of classes not only represents these inheritance relation but also imposes a relationship on the sets of objects of classes. For example, when we say an *employee isa person* in a object-oriented database system, as in object-oriented languages we expect *employee* to be a subclass of *person* in that every method of *person* is applicable to instances of *employee* but we also mean that, in a given database, the set of *employee* objects is a subset of the set of *person* objects. We call a set of objects of a class an *extent* of a class. Note that the notion of extents are relative to a database (or a subset of a database depending on the context). Since it is natural and sometimes necessary to maintain multiple databases having a same class structure, it is desirable to allow multiple extents for a single class.

An important implication of this notion is that classes in object-oriented databases cannot be directly modeled by types. This contrasts with object-oriented programming whose classes are accurately represented in the type system we have constructed in chapter 6. To clarify this issue, let us look at a sketch of a database query in which this subset assumption is made.

1. Obtain the set S of *students* in the database.

2. Perform some complicated restriction of S, e.g. find the subset S' of S whose Age is below the average Age of S.

3. Obtain the subset E of S of *employees* in S'.

4. Print out some information about E, e.g. print the names and ages of people in E with a given salary range.

Since each object is a model of a real-world entity, the above query certainly make sense and the process (3) might be done by "intersecting" the set of *employee* objects and S. Different from the typing relation between values and types, an object o belongs to a class c does not means that the entire structure of o is specified by c but it should mean that c specifies some *partial information* of the structure of o. A *person* object might be an *employee*. It also suggests that an object-oriented database require some form of heterogeneous collection of objects.

Programming such queries are certainly possible in a dynamically typed language, and can easily be handled in object-oriented database languages based on dynamic type-checking. Of course this situation can be also handled in a statically typed system by maintaining disjoint sets of *employees* and *persons* instances, each of which is uniformly typed, and at the same time by maintaining a "natural" embedding (injective map) from the set of *employees* to the set of *persons* as a typed function in the language. This is what is done in the relational model through (seldom implemented) foreign key constraints. Implementing the query then involves join at line 3 in order to find the employees who have a key that is in the relation containing the selected persons.

Galileo handles this problem by introducing special types called "classes". For each class-type, Galileo maintains a unique extent associated with a class. An object of a class-type is required to have a key that is unique not only among the set of all objects of that class but also among the set of all objects of all its subclasses. The system then maintain the inclusion relation of extents associated with classes. This approach has a disadvantage that we can have only one extent for each class. This creats a problem when we want to define generic methods which are applicable to many extentns sharing common structures. For example, if we want to maintain a separate extent of students for each department then we are forced to define a separate class for each department. This means that even if the structure and required set of methods are identical for all those departments we are forced to repeat class and method definitions for each department.

In this chapter, we introduce the reference types to represent objects with identities. We then introduce the notion of *views* to represent extents and demonstrate that Machiavelli type system can represent object-oriented databases. The justification of these notions are, however, intuitive and ad hoc. Development of a formal theory for object identities and extents is left to future investigation.

7.2 Reference Types

We claim that the properties of object identities we have described are accurately captured by *reference* types (or *pointer* types) that are implemented in many typed programming language including Standard ML [47].

We extend Machiavelli type system by adding the type constructor *ref*, i.e. if τ is a type then *ref*(τ) is a type. The set of raw terms are also extended with three constant functions:

$$\begin{aligned} ref &: \tau \rightarrow ref(\tau) \text{ (reference creation)} \\ ! &: ref(\tau) \rightarrow \tau \text{ (de-referencing)} \\ := &: ref(\tau) \rightarrow \tau \rightarrow nil \text{ (update)} \end{aligned}$$

where *nil* is a trivial base type containing only one value *Nil*. Following the syntax of Standard ML, we use the notation $e_1 := e_2$ for $:=(e_1)(e_2)$.

As has been known and was pointed out in [71], a general use of reference in ML style type system can create type inconsistency not detected by static type-checking. The following example is given in [71]:

```
let
   val $x = ref\,\lambda x.\, x$
in
   $(x := \lambda x.\, x + 1, !x\ true)$
end.
```

If the type system treat the constant *ref* as an ordinary constant then the type system infer the type *bool* for the above expression but cause a runtime type error if the evaluation of pair $(x := \lambda x.\, x + 1, !x\ true)$ is left to right. Here we adopt the solution proposed in [71] that the "actual" argument to the function *ref* must have a ground type. This condition is the same as the one we imposed on the function *join* and $proj^{\sigma}$ (with infinite σ) and therefore can be enforced by the same method (subsection 5.4.2).

Two references are equal iff they are created by the same invocation of *ref*. For example, $ref(3) = ref(3)$ is *false* but **let val** $a = ref(3)$ **in** $a = a$ **end** is *true*. A reference is created with a value which can be arbitrary complex and can be changed without changing the value of reference itself. For example, if we create a department object (with identity) as the following reference value:

val $d = ref([Dname = "Sales", Building = 45]);$

and from this we create two employee objects:

val $emp_1 = ref([Name = "Jones", Department = d]);$
val $emp2 = ref([Name = "Smith", Department = d]);$

then these two employees have the object d in the *Department* field not the value $[Dname = "Sales", Building = 45]$. The following change of values of d:

```
let
   val $d = (!emp_1).Department$
in
   $d := modify(!d, Building, 67)$
end
```

will be reflected in the department as seen from emp_2 and the expression

$$(!((!emp_2).Department)).Building$$

is evaluated to 67.

7.3 Views for Representing Extents

The way we capture the notion of view in our language is through coercions or *views*. The type of an object will, in general, be a reference to a rather complicated type, say *PersonObj*. A database (or a part of it) will consist of a set D of such objects, i.e. a value of type $\{\!\{PersonObj\}\!\}$. A view of D is a set of relatively simple records in which we "reveal" a part of the structure of each member of D so that we can apply operations on database objects we have already developed. For example, $\{\!\{[Name : string, Id : PersonObj]\}\!\}$ and $\{\!\{[Name : string, Age : int, Id : PersonObj]\}\!\}$ can be the types of views of the set D. But notice that within these records we have kept a distinguished Id field that contains the object itself, and this field, being a reference type, can also be treated as an "identity" or key when we have a set of objects. Because of the presence of this field, we can define generalized set operations on views even though they are of different types. In fact we have already seen one such operation, join. When applied to views it is an operation that takes the intersection of sets of identities, but produces a result that has a join type and gives us the union of the "methods". In fact we shall simply define a *class associated with extents* as any record type that contains an Id field, which will be assumed to be some reference type.

As an example, a part of the database could be a collection of "person" objects modeling the set of persons in a university. Among persons, some are students and others are employees. Such subsets naturally form a taxonomic hierarchy or *class* structure. Figure 7.1 shows a simple example. Note that the arrows not only represent inheritance of properties but also actual set inclusions. We use variant types to represent structures of objects that share common properties (e.g. being a

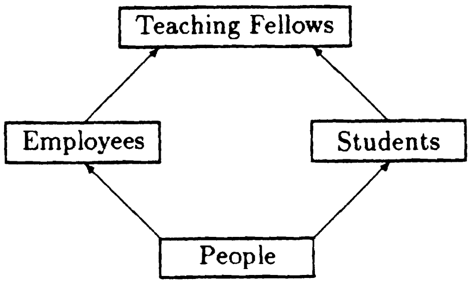

Figure 7.1: A Simple Class Structure

person) but differ in special properties. The example is then represented by the following types in Machiavelli.

$$\begin{aligned}
PersonObj = \mathit{ref}([&Name : string, Salary : \langle None : nil, Value : int\rangle,\\
&Advisor : \langle None : nil, Value : PersonObj\rangle,\\
&Class : \langle None : nil, Value : string\rangle])
\end{aligned}$$

$$Person = [Name : string, Id : PersonObj]$$

$$Student = [Name : string, Advisor : PersonObj, Id : PersonObj]$$

$$Employee = [Name : string, Salary : Integer, Id : PersonObj]$$

$$\begin{aligned}
TeachingFellow = [&Name : string, Salary : Integer, Advisor : PersonObj,\\
&Class : String, Id : PersonObj]
\end{aligned}$$

The reference type *PersonObj* is the type of person objects. The type *Person*, *Employee* and *TeachingFellow* are types of person objects *viewed* as persons, employees and teaching fellows respectively. For example, a person object is viewed as (or more precisely can be coerced to) an employee if it has name and salary attributes. A database would presumably contain a set of person objects, i.e. a set of type $\{\!\{PersonObj\}\!\}$, and views of any set of this type can be constructed in Machiavelli by the following definitions:

fun $PersonView(S) =$

```
        select $[Name = (!x).Name, Id = x]$
        where $x \in S$
        with $true$;

    fun $EmployeeView(S) =$
        select $[Name = (!x).Name, Salary = value(((!x).Salary), Id = x]$
        where $x \in S$
        with(case $(!x).Saraly$ of $< Value = x > \Rightarrow true, < None = y > \Rightarrow false$);

    fun $StudentView(S) =$
        select $[Name = (!x).Name, Advisor = value((!x).Advisor), Id = x]$
        where $x \in S$
        with (case $(!x).Advisor$ of $< Value = x > \Rightarrow true, < None = y > \Rightarrow false$);

    fun $TFView(S) =$
        select $join(x, [Course = value((!x).Course)])$
        where $x \in join(StudentView(S), Employee(S))$
        with (case $(!x).Course$ of $< Value = x > \Rightarrow true, < None = y > \Rightarrow false$);
```

where *value* is the function defined as:

```
    fun $value(v) =$ (case $v$ of $< Value = x > \Rightarrow x, < None = y > \Rightarrow Error$);
```

The types inferred for these functions will be quite general, but the following are the instances that are important to us in the context of this example.

$PersonView : \{\!\{PersonObj\}\!\} \rightarrow \{\!\{Person\}\!\}$
$EmployeeView : \{\!\{PersonObj\}\!\} \rightarrow \{\!\{Employee\}\!\}$
$StudentView : \{\!\{PersonObj\}\!\} \rightarrow \{\!\{Student\}\!\}$
$TFView : \{\!\{PersonObj\}\!\} \rightarrow \{\!\{TeachingFellow\}\!\}$

In the definition of *TFView*, the join of two views models both the intersection of the two classes and the inheritance of methods. If σ_1, σ_2 are types of classes, then $\sigma_1 \leqslant \sigma_2$ implies that $project^{\sigma_1}(View_{\sigma_2}(S)) \subseteq View_{\sigma_1}(S)$ where $View_{\sigma_1}$ and $View_{\sigma_2}$ denote the corresponding viewing functions on classes σ_1 and σ_2. This property guarantees that the join of two views corresponds to the intersection of the two. We therefore define:

fun $view\_intersection\ V_1\ V_2 = join(V_1, V_2)$.

The property of the ordering on types and Machiavelli's polymorphism also supports the inheritance of methods. For example, suppose we have a database *persons*. Then

$$view\_intersect(StudentView(persons), EmployeeView(persons))$$

always represents the set of objects that are both student and employee. Moreover, the type of the intersection is the join of the types of *StudentView(persons)* and *EmployeeView(persons)* and therefore methods defined on *StudentView(persons)* and *EmployeeView(persons)* are automatically inherited by Machiavelli's type inference mechanism. Here are some examples of query processing in a interactive session in Machiavelli:

```
(* New view of people who are both Student and Employees *)
-> val supported_student =
       view_intersection(StudentView(persons),EmployeeView(persons));
>> val supported_student = { .... }
   : {[Name:string, Salary:int, Advisor:PersonObj, Id:PersonObj]}

(* Names of students who earn more than their advisors *)
-> select x.Name
   where x <- supported_student, y<-EmployeeView(persons)
   with x.Advisor=y.Id andalso x.Salary > y.Salary;
>> val it = { ... } : {string}
```

Dual to the join which corresponds to the intersection of classes, we can define the "union" of extents in Machiavelli. The result type of the union of two views should be the type "common" to both of the types. Using our ordering on description types, this can be represented by the following typing rule:

$$view\_union : \{\!\{\delta_1\}\!\} \times \{\!\{\delta_2\}\!\} \rightarrow \{\!\{\delta_1 \sqcap \delta_2\}\!\}$$

The appropriate definition for the *view_union* can then be given as:

$$view\_union(s_1, s_2) = union(proj^{\delta_1 \sqcap \delta_2}(s_1), proj^{\delta_1 \sqcap \delta_2}(s_2))$$

which is reduced to the standard set-theoretic union when $\delta_1 = \delta_2$. This operation can be used to give a union of classes of different type. For example,

$$view\_union(StudentView(person), EmployeeView(person))$$

correspond to the union of students and employees. On such a set, one can only safely apply methods that are defined both on students and employees. As with join, this constraint is automatically maintained by Machiavelli's type system simply because the result type is $\{\!\{Person\}\!\}$.

In addition one can easily define the "membership" operation on objects of disparate type.:

$$\textbf{fun}\ view\_member(x, S) = join(\{\!\{x\}\!\}, S) \neq \{\}$$

$view\_member(x, S) = true$ iff there is some member of s of S such that x and s have a common identity. In this fashion it is possible to extend a large catalog of set-theoretic operations to classes.

It is interesting to note that this approach, when considered as a data model, has some similarities with that proposed in the IFO model [3]. The database consists of a collection of sets of different types of which a set of type *PersonObj* in our example, would be one. Subclasses ("specializations" in IFO) correspond to views. However, unions of these cannot be formed directly, because the *Id* fields will have different types. The correct way to form a union (IFO's "generalizations") would be to exploit a variant type.

Chapter 8

Conclusion and Topics for Further Research

This thesis has proposed a programming language for databases and objet-oriented programming. The language has a static type system with static type inference and ML style polymorphism. The type system uniformly integrates complex database objects, central features of object-oriented programming and ML style polymorphism. Various complex models including complex object models and object-oriented databases can be directly represented in the type system. This allows database programmers to share benefits of ML style type system and useful features of object-oriented programming such as multiple inheritance and data abstraction.

This thesis has achieved this goal by extending ML style type system to structures and operations for databases and object-oriented programming. I have analyzed the syntax and semantics of ML and constructed a framework for denotational semantics for ML polymorphism and axiomatized the equational theory that correspond to the semantics. I have also constructed a theory of types for database objects and proposed a concrete type system for complex database objects that is rich enough to represent virtually all proposed database objects. By combining the analysis of ML and the type system for database objects, I have defined the polymorphic core of the proposed language and developed a type inference algorithm. The core has then been extended to include user definable classes for object-oriented programming. I have also presented a method to represent object-oriented databases in the language.

As in many endeavors, the work presented in this thesis is not complete with respect to its ultimate goal. There are many topics that remain to be investigated before a truly satisfactory programming language for data intensive applications can become a reality. The rest of this chapter lists some of these topics that can be regarded as continuations of this work.

8.1 Semantics

One of my belief underlying this study is that construction of a clean mathematical semantics is essential to understand existing systems, to extend existing systems and to integrate various different systems. The following investigations should be useful for a better understanding of the type system proposed in this thesis and for further extensions.

8.1.1 A Concrete Models for Machiavelli

In section 3.3, I defined the notion of models and equational theories for ML like languages and proved the soundness and completeness theorem. The notion of models for Machiavelli was defined based on this result. By the completeness theorem for the simply typed lambda calculus, we know that there is a model. But it is also interesting to construct a syntax free model for Machiavelli. By the definition of models of Machiavelli, it is enough to construct a model for the explicitly typed language xMachiavelli. An appropriate formalism might be the domain theory. In [91] Plotkin constructed a domain theoretic model for a variant of the simply typed lambda calculus with recursion. In this model, base types are interpreted as flat cpos (complete partial orders) and function types are interpreted as continuous function spaces. In this language the only type constructor is the function type constructor, which corresponds to the continuous function space construction on cpos. In xMachiavelli, types correspond to regular trees generated by various type constructors. Since a regular tree is specified by an equation, one way to construct a domain theoretic model for xMachiavelli might therefore be to translate an equation on regular trees to an equation on cpos and to interpret a type as the solution of the corresponding equation over cpos.

8.1.2 Call-by-Value Semantics for ML Polymorphism

My analysis of a denotational semantics for ML polymorphism was based on the standard model theory and the standard equational theory for lambda calculus. It assumed β-equality both in equational theories and in semantic spaces (through a condition on semantic mappings). As I pointed out in section 3.1, however, these theories do not agree with the operational semantics based on the "call-by-value" evaluation. Although the operational semantics is sound with respect to the denotational semantics in the sense that if a term is evaluated to another term then they have the same meaning, the denotational semantics is not ***computationally adequate*** [76] for those operational semantics and therefore the full abstraction result presented in subsection 3.6 has little help to them. It is therefore desirable to develop another frameworks for equational theory and denotational semantics that correspond to the call-by-value evaluation. Under our view of ML polymorphism, we need to develop a model theory and equational theory for the simply typed lambda calculus. Once we have those theories, I hope it might not be hard to develop the corresponding theories for ML like language using similar strategy I developed in section 3.3.

8.1.3 Semantics of Class Declarations

Another interesting question is a semantics of class definitions. A definition of a class determines a subset of types that are compatible with the set of methods (i.e. the set of raw lambda terms that implement the methods). This suggests that a class definition could be regarded as a form of existential type $\exists \mathbf{sub} : K.(M_1 \times \ldots \times M_n)$ where K denotes the subset of types that are compatible with the set of methods and $M_1, \ldots, M_n$ are the types of the methods defined in the class definition. This is a form of ***bounded existential types*** introduced in [27] but differs from theirs in that the kind K reflects directly the implementations of methods. Semantics of such types should explain not only the functionality of the set of methods (as was done in [80]) but also the structure of a kind K determined by a set of raw lambda terms.

8.1.4 Semantics of Object Identities

In section 7.2, I claimed that the properties of objet identities were accurately captured by reference types. However, the justification was intuitive and ad hoc. Although the notion of object identities is intuitively clear and appealing, the precise formulation of the semantics of objects with identities constitutes a challenge. A uniform and elegant integration of objects with indentities into a programming language type system may need an analysis analogous to the one I did for complex database values in chapter 4.

8.2 Extensions of the Language

I believe that the structures and operations available in the language proposed in this thesis is a good approximation to a sufficient set for practical programming. However, there are many features that might be useful.

8.2.1 General Set Operation

The set of available operations on the set data types in the language are *union*, *map*, *prod*, *join* and *proj*. I demonstrated that they enables us to define a general selection function, which was named *filter*, and therefore seem sufficient for query processing in the style of relational algebra. However, determining whether they are sufficient for general programming with sets or not needs more investigation. In particular, some application might want an operation that maps a set to a sequence. With the existence of references, such operation is definable. The following is one example:

```
fun $f$ $(S : \{\!\{s\}\!\}) =$
    let
        val $lref = (ref\, nil) : ref(list(s))$
        fun $f\ e = lref := cons\,e(!lref)$
        val $dummy = map\ f\ S$
    in
        $!lref$
    end;
```

The type specifications $S : \{\!\{s\}\!\}$ and $(ref\ nil) : ref(list(s))$ are required to prevent the problem associated with the interaction between polymorphism and references (see section 7.2). The type $list(s)$ and the function *cons* are easily defined.

A better approach would be to introduce a general elimination operator, which we call *hom* here, for sets analogous to the operations on lists such as the "pump" operation in FAD [13] and the "fold" or "reduce" in many functional languages. One possible definition for *hom* is:

$$\begin{aligned} hom(f, op, z, \{\!\{\}\!\}) &= z, \\ hom(f, op, z, \{\!\{e_1, e_2, \ldots, e_n\}\!\}) &= op(f(e_1), op(f(e_2), \ldots, op(f(e_n), z)\ldots)). \end{aligned}$$

It seems that virtually all useful operations are definable by using *hom*, *union* and *join*. For example, the following are definitions for *map* and *filter*:

```
fun $map\ f\ S = hom(\lambda x.\ \{\!\{x\}\!\}, union, \{\!\{\}\!\}, S)$,
fun $filter\ p\ S = hom(\lambda x.$ if $p\ x$ then$\{\!\{x\}\!\}$ else $\{\!\{\}\!\}, union, S)$.
```

In addition to these examples, *hom* can be used to define the cartesian product (*prod*) of sets, set difference, membership in a set, and the powerset (the set of subsets) of a set.

A problem of introducing *hom* as a primitive operation on sets in the definition of the language is that the result of this operation will in general depend on the order in which the elements of the set are encountered and therefore the above definition of *hom* does not completely specifies the behavior of *hom* unless the third argument *op* is an associative commutative operation and the first argument *f* does not cause side-effect. It seems impossible to determine statically these properties for each application of *hom*. If it is the case then we will be forced to leave the effect of *hom* as implementation dependent if *op* and *f* do not satisfy the above property.

8.2.2 Heterogeneous Sets

As I analyzed in section 7.1, an object-oriented database require some form of heterogeneous collection of objects. I solved this problem by encoding heterogeneous collection in a (homogeneous) set of variants. Combining with *views*, it has almost achieved the goal of having static type system that deals properly with collections of heterogeneous objects. Using *join* and *proj*, queries on views looked the way we want them to look in object-oriented databases. However the construction of complicated "catch-all" variant types (like *PersonObj* in section 7.3) and the definition of views – even though they could be automatically generated from a semantic data model schema – is something we would prefer to avoid. It would be more natural if the schema were directly represented in the type system.

One radical solution would be to introduce the heterogeneous set type constructor $\{\!\{\sigma_1, \ldots, \sigma_n\}\!\}$. In order to build a type system based on this idea, we need to decide the meaning of the typing judgement $\{e_1, e_2, ..., e_n\} : \{\tau_1, \tau_2, ..., \tau_m\}$. There are at least the following three possibilities:

1. for each e_i there is some τ_j such that $e_i : \tau_j$,

2. for each τ_i there is some e_j such that $e_j : \tau_i$,

3. the conjunction of the two.

Unfortunately none of them provide a satisfactory interpretation. The first interpretation implies the set expressions have more than one types, yielding the same problem as found in subtype based systems (section 5.1). The other two interpretations do not support necessary operations on sets including intersection and difference. For example, under either of the two interpretations,

$S_1 : \{\tau_1, ..., \tau_n\}$ and $S_2 : \{\tau_1, ..\tau_n\}$ does not imply $S_1 \cap S_2 : \{\tau_1, ..., \tau_n\}$. The second interpretation is also unsafe when combined with function application.

A more promising approach would be to introduce some notion of *partial type information* for sets. One way to specify partial type information is to specify a set of possible types. In the theory of types, sets of types are sometimes called ***kinds*** [70] and are treated as themselves objects. The special set constructor can then be characterized as an operator which takes a kind and returns a type. In particular, the following kinds representing various sets of record types seems particularly useful in object-oriented databases

$$\kappa ::= K(\tau) \mid (l : \kappa, \ldots, l : \kappa)$$

where $K(\tau)$ is the kind correspond to the singleton set of τ and $(l_1 : \kappa_1, \ldots, l_n : \kappa_n)$ denote the set of all record types containing *at least* all the fields $l_1, \ldots, l_n$ of types specified by respective kinds. This relation can be easily formalized by kinding rule, which also induce a partial order on kinds.

The set of types can be extended by partial types:

$$\tau ::= T(\kappa) \mid \ldots$$

where $T(\kappa)$ is the type with partial type information represented by the kind κ. Since our objective is to integrate heterogeneous sets, the following introduction rule seems the only necessary introduction rule for partial types.

$$(\text{HSET}) \qquad \frac{e : \sigma}{\{\!\{e\}\!\} : \{\!\{T(K(\sigma))\}\!\}}$$

Possible elimination rules are:

$$(\text{COERCE}) \qquad \frac{e : T(K(\sigma))}{e : \sigma}$$

$$(\text{DOT}) \qquad \frac{e : T((\ldots, l : \kappa, \ldots))}{e.l : T(\kappa)}$$

Union and intersection can be generalied to these partially specified set types.

$$(\text{HUNION}) \qquad \frac{e_1 : \{\!\{\kappa_1\}\!\} \qquad e_2 : \{\!\{\kappa_2\}\!\}}{union(e_1, e_2) : \{\!\{\kappa_1 \sqcup \kappa_2\}\!\}}$$

$$(\text{HINTERSECT}) \qquad \frac{e_1 : \{\!\{\kappa_1\}\!\} \qquad e_2 : \{\!\{\kappa_2\}\!\}}{intersection(e_1, e_2) : \{\!\{\kappa_1 \sqcap \kappa_2\}\!\}}$$

where $\kappa_1 \sqcup \kappa_2$ and $\kappa_1 \sqcap \kappa_2$ denote respectively the least upper bound and greatest lower bound of kinds under the set inclusion ordering.

8.3 Communicating to Existing Databases

Of course, a good database programming language should not only be able to manipulate databases that conform to its own type system but others as well. In particular, most current object-oriented database languages do not have any static type-checking, but we would still like to deal with them in the same way that we have dealt with uniformly typed classes. This is possible through use of *dynamic* values. A dynamic value [1] is one which carries its type description with it. Functions exist for interrogating this type description and for coercing dynamic values back to ordinary typed values. Let us assume that dynamic values also behave like references in that two dynamic values are equal only if they were created by the same invocation of the function *Dynamic*, which creates values of type *dynamic*.

We can now view an external database as a single large set of dynamic values, i.e. it has the type $\{\!\{dynamic\}\!\}$. In the same fashion that we generated views in chapter 7, we can generate views (probably by some external procedures) based on dynamic. Thus an employee view of the database might be a class of type

$$\{\!\{[Name : string, Salary : int, Id : dynamic]\}\!\}$$

and a department view could be a class of type

$$\{\!\{[Dname : string, Building : string, Id : dynamic]\}\!\}$$

with the "intersection" of these classes being empty. Once this has been done we can write programs to manipulate these structures in the type-safe way I have advocated throughout this thesis even though the underlying database does not have any imposed type constraints. The implementation of views (in addition we would need procedures to perform updates) must, of course, respect the projection property I described earlier. But I also hope that, for a given object-oriented database system, building these views will be straightforward and could be carried out by generating them automatically.

8.4 Implementation Issues

There are a number of important ways in which Machiavelli needs to be augmented to make it a viable programming language for databases and object-oriented programming. The most important of these is the implementation of persistence and efficient evaluation algorithm for expressions – especially those containing records and sets. I believe that these issues are orthogonal to Machiavelli type system and many existing approaches can be adopted. On the other hand I also feel that we do not need to deal in great detail with the efficiency of the whole range of database structures. My hope is that Machiavelli can be parasitic on already implemented database management systems and will serve as a medium for communication between heterogeneous systems and, in particular, that it will allow us to achieve a clean integration of already implemented relational and object-oriented systems.

With the future efforts towards efficient implementations, I hope that Machiavelli (or some language like it) will become a practical programming language for various databases, object-oriented system and other data intensive applications.

Appendix A

Abstract Syntax of Machiavelli

The following is the summary of the abstract syntax of Machiavelli. The syntax is given as a syntax free grammar. The top level objects are declarations denoted by *decl*. Key words and lexical tokens are written in typewriter font. Optional constructs are enclosed in square bracket ([ ]). ϵ denote the empty string. We use the following syntactic classes:

id	the set of identifiers
atomics	atomic tokens representing values of base types
constfun	builtin functions on base types such as addition and conditionals.

decl `::=` *classdecls* | *binding* | *expr*

classdecls `::=` ϵ | *classdecl* | *classdecls*

classdecl `::=` `class` `=` *id* `=` *type* `isa` {*ctype*,...,*ctype*}
methoddecls
`end`

methoddecls `::=` ϵ | *binding* `:` *methodtype* `;` *methoddecls*

binding `::=` `val` *id* `=` *expr* | `fun` *id* *id* $\cdots$ *id* `=` *expr*

expr `::=` *integer* | *boolean* | *string* | *real* |
`(fn` *id* `=>` *expr*`)` | *expr* *expr* | `let` *id* `=` *expr* `in` *expr* `end` |

```
        [id = expr, ..., id = expr] | expr.id |
        <id = expr> |
        (case expr of <id = id =>expr,...,
                      <id = id> =>expr
                      [,other =>expr]) |
        {expr, ..., expr} | union(expr,expr) | hom(expr, expr, expr, expr) |
        join(expr, expr) | proj(expr,type) | con(expr, expr) | eq(expr, expr) |
        (rec id.  expr) | (expr) | (expr:type)

type ::= 'id | "id | int | bool | string | real |
        [(id) id:type, ..., id:type] | <(id)id:type, ..., id:type> |
        (id > dtype) | (id isa ctype, ..., ctype) |
        [id:type, ..., id:type] | <(id) id:type, ..., id:type> |
        type -> type | id(type, ..., type)

mtype ::= sub | 'id | "id | int | bool | string | real |
         [id:mtype, ..., id:mtype] | <(id) id:mtype, ..., id:mtype> |
         mtype -> mtype | id(mtype, ..., mtype)

ctype ::= 'id | "id | int | bool | string | real | (id isa ctype, ..., ctype) |
         [id:ctype, ..., id:ctype] | <(id) id:ctype, ..., id:ctype> |
         ctype -> ctype | id(ctype, ..., ctype)

dtype ::= int | bool | string | real |
         [id:dtype, ..., id:dtype] | <id:dtype, ..., id:dtype> |
         (rec id dtype)
```

Bibliography

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *Proc. 16th ACM Symposium on Principles of Programming Languages*, 1989.

[2] S. Abiteboul and N. Bidoit. Non first normal form relations to represent hierarchically organized data. In *Proc. ACM Symposium on Principles of Database Systems*, Waterloo, Ontario, Canada, 1984.

[3] S. Abiteboul and R. Hull. IFO: A formal semantic database model. *ACM Transactions on Database Systems*, 12(4):525–565, December 1987.

[4] S. Abiteboul and R. Hull. Restructuring hierarchical database objects. *Theoretical Computer Science*, 62:3 – 38, 1988. Special issue dedicated to the 2^{nd} International Conference on Database Theory.

[5] A. V. Aho and J.D. Ullman. Universality of data retrieval languages. In *Proc. 6th ACM Symposium on Principles of Programming Languages*, pages 110–120, 1979.

[6] H. Aït-Kaci. *A Lattice Theoretic Approach to Computation based on Calculus of Partially Ordered Type Structures*. PhD thesis, University of Pennsylvania, 1985.

[7] A. Albano, L. Cardelli, and R. Orsini. Galileo: A strongly typed, interactive conceptual language. *ACM Transactions on Database Systems*, 10(2):230–260, 1985.

[8] M. M. Astrahan and Chamberlin D. D. Implementation of a structured english query language. *Communications of the ACM*, 18(10):580–587, 1975.

[9] M.M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Waston. System R: a relational approach to database management. *ACM Transactions on Database Systems*, 1(2):97–137, June 1976.

[10] M.P. Atkinson and O.P. Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, June 1987.

[11] L. Augustsson. A compiler for lazy ML. In *Symposium on LISP and Functional Programming*, pages 218–227. ACM, 1984.

[12] F. Bancilhon. Object-oriented database systems. In *Proc. ACM Symposium on Principles of Database Systems*, Austin, Texas, March 1988. ACM SIGART-SIGMOD-SIGACT.

[13] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. FAD, a powerful and simple database language. In *Proc. Intl. Conf. on Very Large Data Bases*, pages 97–105, 1988.

[14] F. Bancilhon and S. Khoshafin. A calculus for complex objects. In *Proc. ACM Symposium on Principles of Database Systems*, 1986.

[15] H.P. Barendregt. *The Lambda Caluculus*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984. revised edition.

[16] J. Biskup. A formal approach to null values in database relations. In *Advances in Data Base Theory Vol 1*. Prenum Press, New York, 1981.

[17] T. Bloom and S. Zdonik. Issues in the design of object-oriented database programming languages. In *OOPSLA*, pages 441–451, 1987.

[18] R.J. Brachman and J.G. Schmolze. An overview of the kl-one knowledge representation system. *Cognitive Science*, 9(3), 1985.

[19] V. Breazu-Tannen and A. R. Meyer. Lambda calculus with constrained types (extended abstract). In R. Parikh, editor, *Proceedings of the Conference on Logics of Programs, Brooklyn, June 1985*, pages 23–40. *Lecture Notes in Computer Science*, Vol. 193, Springer-Verlag, 1985.

[20] K. B. Bruce, A. R. Meyer, and J. C. Mitchell. The semantics of second-order lambda calculus. *Information and Computation*, 1988. to appear.

[21] K. B. Bruce and P. Wegner. An algebraic model of subtypes in object-oriented languages. *SIGPLAN Notices*, 21(10):163–172, October 1986.

[22] P. Buneman, A. Jung, and A. Ohori. Using powerdomains to generalize relational databases. *Theoretical Computer Science*, To Appear, 1989. Available as a technical report from Department of Computer and Information Science, University of Pennsylvania.

[23] L. Cardelli. Amber. Technical Memorandum TM 11271-840924-10, AT&T Bell Laboratories, 1984.

[24] L. Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types, Lecture Notes in Computer Science 173*. Springer-Verlag, 1984.

[25] L. Cardelli. Quest. Technical report, Digital Equipment Corporation, 130 Lytton Avenue, Palo Alto, CA 94301, 1988.

[26] L. Cardelli and J. Mitchell. Semantic methods for object-oriented languages. Unpublished lecture notes for tutorial given at the Conference on Object-Oriented Programming Systems, Languages, and Applications, September 1988.

[27] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surverys*, 17(4):471–522, December 1985.

[28] P. P. Chen. The entity-relationship model: Toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[29] E. F. Codd. A relational model for large shared databank. *Communications of the ACM*, 13(6):377–387, 1970.

[30] M. Coppo. Completeness of type assignment in continuous lambda models. *Theoretical Computer Science*, 29:309–324, 1984.

[31] M. Coppo. A completeness theorem for recursively defined types. In G. Goos and J. Hartmanis, editors, *Automata, Languages and Programming, 12th Colloquium, LNCS 194*, pages 120–129. Splinger-Verlag, July 1985.

[32] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

[33] O. Dahl and K. Nygaard. Simula, an Algol-based simulation language. *Communications of the ACM*, 9:671–678, 1966.

[34] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. 9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[35] C.J. Date. *An Introduction to Database System Vol 1*. Addison-Wesley, third edition, 1981.

[36] P.C. Fischer and S.J. Thomas. Operators for non-first-normal-form relations. In *Proc. IEEE COMPSAC*, 1983.

[37] H. Friedman. Equations between functionals. In *Lecture Notes in Mathematics 453*, pages 22–33. Springer-Verlag, 1973.

[38] Y-C. Fuh and P. Mishra. Type inference with subtypes. In *Proceedings of ESOP '88*, pages 94–114, 1988. Springer LNCS 300.

[39] M. R. Garey and D. S. Johnson. *Computer and Intractability, A Guide to the Theory of NP-Completeness.* Freeman, New York, 1979.

[40] G. Gierz, H.K. Hofmann, K. Keimel, J.D. Lawson, M. Mislove, and D.S. Scott. *A Compendium of Continuous Lattices.* Springer-Verlag, 1980.

[41] Susanna Ginali. *Iterative Algebraic Theories, Infinite Trees and Program Schemata.* PhD thesis, University of Chicago, 1976.

[42] J.-Y. Girard. Une extension de l'interpretation de gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et théorie des types. In *Second Scandinavian Logic Symposium.* Springer-Verlag, 1972.

[43] E. M. Gold. Complexity of automaton identification from given data. Unpublished manuscript, 1974.

[44] A. Goldberg and D. Robson. *SMALLTALK-80. The language and its implementation.* Addison-Wesley, 1983.

[45] S. Gorn. Explicit definitions and linguistic dominoes. In J. Hart and S. Takasu, editors, *Systems and Computer Science*, pages 77–105. University of Toronto Press, 1967.

[46] N. Hammer and D. McLeod. Database description with SMD: a semantic database model. *ACM Transactions on Database Systems*, 6(3):351–386, September 1981.

[47] R. Harper, D. B. MacQueen, and R. Milner. Standard ML. LFCS Report Series ECS-LFCS-86-2, Department of Computer Science, University of Edinburgh, March 1986.

[48] R. Harper, R. Milner, and M. Tofte. The definition of Standard ML (version 2). LFCS Report Series ECS-LFCS-88-62, Department of Computer Science, University of Edinburgh, August 1988.

[49] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and λ-Calculus.* Cambridge University Press, 1986.

[50] R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. American Mathematical Society*, 146:29–60, December 1969.

[51] R. Hindley. The completeness theorem for typing λ-terms. *Theoretical Computer Science*, 22:1–17, 1983.

[52] J. Hopcroft and J. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, 1979.

[53] J.E. Hopcroft and R.M. Karp. An algorithm for testing the equivalence of finite automata. Technical Report TR-71-114, Department of Computer Science, Cornell University, Ithaca, NY., 1971.

[54] P. Hudak and P. (editors) Wadler. Report on the programming language Haskell, a non-strict, purely functional language, version 1.0. Technical report, University of Glasgow, 1989. Pre-release Draft for FPCA '89.

[55] G Huet. *Résolution d'équations dans les langages d'ordre 1,2,...ω.* PhD thesis, University Paris, 1976.

[56] R. Hull. Relative information capacity of simple relational database schemata. *SIAM J. Computing*, 15(3):856–886, August 1986.

[57] R. Hull. *Databases*, chapter 5 A survey of theoretical research on typed complex database objects, pages 193–253. Academic Press, 1987.

[58] R. Hull and R. King. Semantic database modeling: Survey, applications and research issues. *Computing Surveys*, 19(3), September 1987.

[59] R. Hull and C.K. Yap. The format model: a theory of database organization. *Journal of the ACM*, 31(3):518–537, 1984.

[60] J.H. Ichbiah, J.G.P. Barnes, J.C. Heliard, B. Krieg-Bruckner, O. Roubine, and B.A. Wichmann. Rationale of the design of the programming language Ada. *SIGPLAN Notices*, 14(6), 1979.

[61] T. Imielinski and W. Lipski. Incomplete information in relational databases. *Journal of ACM*, 31(4):761–791, October 1984.

[62] G Jaeschke and H. Schek. Remarks on algebra on non first normal forn relations. In *Proc. ACM Symposium on Principles of Database Systems*, pages 124–138, Los Angeles, March 1982.

[63] L. A. Jategaonkar and J.C. Mitchell. ML with extended pattern matching and subtypes. In *Proc. ACM Conference on LISP and Functional Programming*, pages 198–211, Snowbird, Utah, July 1988.

[64] K. Jensen and N. Wirth. *PASCAL user manual and report.* Springer-Verlag, Berlin, second edition, 1975.

[65] S.N. Khoshafian and G.p. Copeland. Object identity. In *Proc. OOPSLA*, pages 406–416, September 1986.

[66] C. Lecluse and P. Richard. Modeling inhertitance and genericity in object-oriented databases. In *Proc.* 2^{nd} *International Conference on Database Theory*, pages 223–238, 1988.

[67] C. Lecluse, P. Richard, and F. Velez. O_2, an object-oriented data model. In *Proceedings of ACM SIGMOD Conference*, pages 424–434, 1988.

[68] Y. Lien. On the equivalence of database models. *Journal of the ACM*, 29(2):333–362, April 1982.

[69] W. Lipski. On semantic issues connected with incomplete information databases. *ACM Transactions on Database Systems*, 4(3):262–296, September 1979.

[70] N. Maccracken. *An Investigation of a Programming Language with a Polymorphic Type Structure.* PhD thesis, Syracuse University, 1979.

[71] D. MacQueen. References and weak polymoprhism. Note in Standard ML of New Jersey Distribution Package, 1988.

[72] D.B. MacQueen, G.D. Plotkin, and Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1/2):95–130, 1986.

[73] D. Maier. *The Theory of Relational Databases.* Computer Science Press, 1983.

[74] D. Maier. A logic for objects. In *Proceedings of Workshop on Deductive Database and Logic Programming*, Washington, D.C., August 1986.

[75] D Maier. Why database languages are a bad idea. In F. Bancilhon and P. Buneman, editors, *Workshop on Database Programming Languages.* Addison-Wesley, 1989. To Appear.

[76] A.R. Meyer and S.S. Cosmadakis. Semantical paradigms. In *Proc. IEEE Symposium on Logic in Computer Science*, July 1988.

[77] R. Milner. Fully abstract models of typed λ-calculi. *Theoretical Computer Science*, 4:1–22, 1977.

[78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[79] J. C. Mitchell and R. Harper. The essence of ML. In *Proc. 15th ACM Symposium on Principles of Programming Languages*, pages 28–46, San Diego, California, January 1988.

[80] J. C. Mitchell and G. D. Plotkin. Abstract types have existential types. In *Proc. 12th ACM Symposium on Principles of Programming Languages*, pages 37–51, New Oreans, January 1985.

[81] K. Mulmuley. A semantic characterization of full abstraction for typed lambda calculus. In *Proc. 25-th IEEE Symposium on Fundations of Computer Science*, pages 279–288, 1984.

[82] A. Ohori. Denotational semantics of relational databases. Master's thesis, Department of Computer and Information Science, University of Pennsylvania, 1986.

[83] A. Ohori. Semantics of types for database objects. *Theoretical Computer Science, Special issue dedicated to 2^{nd} International Conference on Database Theory (To Appear)*, 1989. Available as a technical report form University of Pennsylvania.

[84] A. Ohori. A simple semantics for ML polymorphism. In *Proceedings of ACM/IFIP Conference on Functional Programming Languages and Computer Architecture*, pages 281–292, London, England, September 1989.

[85] A. Ohori and P. Buneman. Type inference in a database programming language. In *Proc. ACM Conference on LISP and Functional Programming*, pages 174–183, Snowbird, Utah, July 1988.

[86] A. Ohori and P. Buneman. Static type inference for parametric classes. In *Proceedings of ACM OOPSLA Conference*, pages 445–456, New Orleans, Louisiana, October 1989.

[87] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database programming in Machiavelli – a polymorphic language with static type inference. In *Proceedings of the ACM SIGMOD conference*, pages 46–57, Portland, Oregon, May – June 1989.

[88] G. Özsoyoğlu, Z. M. Özsoyoğlu, and V. Matos. Extended relational algebra and relational calculus with set-valued attributes and aggregate functions. *ACM Transactions on Database Systems*, 12(4):566–608, 1987.

[89] Z. Özsoyoğlu and L. Yuan. A new normal form for nested relations. *ACM Transactions on Database Systems*, 12(1):111–136, March 1987.

[90] G. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[91] G. Plotkin. $t\omega$ as a universal domain. *Journal of Computer and System Sciences*, 17(2):209–236, 1978.

[92] G.D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[93] D. Rémy. Typechecking records and variants in a natural extension of ml. In David MacQueen, editor, *ACM Conference on Principles of Programming Languages*, 1989.

[94] J.C. Reynolds. Towards a theory of type structure. In *Paris Colloq. on Programming*, pages 408–425. Springer-Verlag, 1974.

[95] A.M. Roth, H.F. Korth, and A. Silberschatz. Extended algebra and calculus for ¬1nf relational databases. Technical Report TR-84-36, Department of Computer Sciences, The University of Texas at Austin, 1984. revised 1985.

[96] M. A. Roth, H. F. Korth, and A. Silberschatz. Extended algebra and calculus for nested relational databases. *ACM Transactions on Database Systems*, 13(4):389–417, 1988.

[97] W. Rounds. Set values for unification-based grammar formalisms and logic programming. CSLI Technical Report CSLI-88-129, Center for Study of Language and Information, June 1988.

[98] D.A. Schmidt. *Denotational Semantics, A Methodology for Language Development.* Allyn and Bacon, 1986.

[99] J.W. Schmidt. Some high level language constructs for data of type relation. *ACM Transactions on Database Systems*, 5(2), 1977.

[100] E. Sciore. Null values, updates, and incomplete information. Technical report, Department of Electrical Engineering and Computer Science, Princeton University, 1979.

[101] D. Scott. Domains for denotational semantics. In *ICALP*, July 1982.

[102] S.M. Shieber. An introduction to unification-based approaches to grammar. In *Proc. 23rd Annual Meeting of the Association for Computational Linguistics*, 1985.

[103] K. E. Smith and S. B. Zdonik. Intermedia: A case study of the difference between relational and object-oriented database systems. In *Proc. of OOPSLA Conference*, pages 452–465, October 1987.

[104] M.B. Smyth. Power domains. *Journal of Computer and System Sciences*, 16(1):23–36, 1978.

[105] R. Stansifer. Type inference with subtypes. In *Proc. 15th ACM Symposium on Principles of Programming Languages*, pages 88–97, 1988.

[106] R.D. Tennent. *Principles of Programming Languages.* Prentice-Hall, 1981.

[107] D.A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science 201*, pages 1–16. Springer-Verlag, 1985.

[108] J.D. Ullman. *Principle of Database Systems.* Pittman, second edition, 1982.

[109] A. *et al.* van Wijngaarden. Report on the algorithmic language Algol 68. *Numerische Mathematik*, 14:79–218, 1969.

[110] M. Wand. A types-as-sets semantics for Milner-style polymorphism. In *Proc. 11th ACM Symposium on Principles of Programming Languages*, pages 158–164, January 1984.

[111] M. Wand. Complete type inference for simple objects. In *Proceedings of the Second Annual Symposium on Logic in Computer Science*, pages 37–44, Ithaca, New York, June 1987.

[112] C. Zaniolo. Database relation with null values. *Journal of Computer and System Sciences*, 28(1):142–166, 1984.

Index