Extension Polymorphism

Dharini Balasubramaniam

School of Mathematical and Computational Sciences
University of St Andrews
St Andrews
Fife
KY16 9SS
Scotland

Abstract

Any system that models a real world application has to evolve to be consistent with its changing domain. Dealing with evolution in an effective manner is particularly important for those systems that may store large amounts of data such as databases and persistent languages. In persistent programming systems, one of the important issues in dealing with evolution is the specification of code that will continue to work in a type safe way despite changes to type definitions.

Polymorphism is one mechanism which allows code to work over many types. Inclusion polymorphism is often said to be a model of type evolution. However, observing type changes in persistent systems has shown that types most commonly exhibit additive evolution. Even though inclusion captures this pattern in the case of record types, it does not always do so for other type constructors. The confusion of subtyping, inheritance and evolution often leads to unsound or at best, dynamically typed systems. Existing solutions to this problem do not completely address the requirements of type evolution in persistent systems.

The aim of this thesis is to develop a form of polymorphism that is suitable for modelling additive evolution in persistent systems. The proposed strategy is to study patterns of evolution for the most generally used type constructors in persistent languages and to define a new relation, called extension, which models these patterns. This relation is defined independent of any existing relations used for dealing with evolution. A programming language mechanism is then devised to provide polymorphism over this relation. The polymorphism thus defined is called extension polymorphism.

This thesis presents work involving the design and definition of extension polymorphism and an implementation of a type checker for this polymorphism. A proof of soundness for a type system which supports extension polymorphism is also presented.

Acknowledgements

Many people have contributed in different ways to this thesis. I would particularly like to thank the following for their role:

My supervisor Ron Morrison for his advice, guidance, encouragement and good humour throughout this project and for his dedication and drive which made this thesis complete

My second supervisor Richard Connor for encouraging my interest in type systems and for his invaluable help and advice during the project

Graham Kirby for his time and effort in meticulously reading the draft of the thesis and suggesting many improvements

Past and present members of the persistent programming research group in St Andrews, Quintin Cutts, Graham Kirby, Dave Munro and Stephan Scheuerl, for their contributions to various aspects of the thesis

My mother and my brother for their constant support and understanding

The secretary of the Computer Science Division, Helen Bremner, for always being helpful, kind and patient

Frédérique Stintzy for being a great friend and for all those cups of tea

Graham Smith, Johannes Courtial, Clare Jarvis and Shane and Debbie Bonetti for their friendship

Contents

1 Introduction

1.1 Persistent Evolution

Systems that fail to evolve will atrophy and eventually die. The cost of failing to evolve can be gauged by the resources being invested in interfacing with legacy systems. Databases are systems which store potentially large amounts of data that model real world entities. This data may be used for computations by application programs which model real world operations. Database schemata or meta-data organise the data into logically related groups. Since the real world being modelled is constantly changing, the data and its uses also change. These changes are reflected in the database in the form of changes to data, application programs and schemata. The focus of this thesis is the evolution of database schemata.

Schema evolution [SZ87, Cla92, Rod92, MCC+93, Cla94, Odb94, Rod95, MM96] can be additive, subtractive or descriptive. Additive evolution models more semantic knowledge than was previously available. Subtractive evolution models less semantic knowledge than before while descriptive evolution models the same knowledge in a different manner. Changing the schemata of a database while maintaining the consistency of the data and programs belonging to them with the semantics of change has proved to be a difficult problem [Zdo86, Cla92, MCC+93, Odb94].

As explained in [AM95], database programming languages have unified data models and type systems and some approximate equivalences can be recognised, as shown in Figure 1.1 below.

Databases	Programming Languages
data model	type system
schema	type expression
database	variable
database extent	value

Figure 1.1 : Equivalences between Databases and Programming Languages

Thus, the core concepts in both databases and programming languages serve similar purposes as illustrated in the table. Data models and type systems, schemata and type expressions, databases and variables and database extent and values model equivalent features in their respective domains.

In contrast to databases, type evolution in programming languages does not normally cause a major problem as there is no extant data described by the type. In the case of programming languages data only exists during the execution of a program.

Persistent programming languages [ABC+83, Coc83, AM86, AMP86, AM95, MBC+96b] seek to eliminate the distinction between database systems and traditional programming languages by providing a single system that deals with both the storage and the manipulation of data. With persistent languages, as in the case of databases, when types evolve to reflect changes in the application domain, both programs that use them and the data that belong to them are affected by the changes. Thus the issue of evolution of types in persistent systems is important since it can potentially concern both large amounts of valuable data and a variety of types which may evolve in various ways.

One solution to this problem is to make use of a mechanism that will allow the same programs to keep working in a type safe manner despite changes to types. That is the subject of this thesis.

1.2 Polymorphism

Polymorphism [Str67, Mil78, CW85] is one mechanism which provides programs with the ability to work over changing types. A polymorphic type system is one in which values and variables can have more than one type. This can be contrasted with the more traditional monomorphic systems in which each value belongs to at most one type.

Polymorphism preserves static typing while providing more flexibility and expressive power than is possible in a monomorphic type system with static typing. A piece of code exhibiting polymorphism can operate over a number of types provided they conform to some common structure. The structure required depends on the kind of polymorphism in use.

Parametric polymorphism [Mil78, CW85] and inclusion polymorphism [Car84, CW85, CL90] are two widely used kinds of polymorphism. Parametric polymorphism uses type parameters to provide the common structure whilst inclusion polymorphism makes use of a hierarchy provided by a subtyping relation and a programming language algebra to support it. Inclusion polymorphism in particular is often used to capture type evolution.

1.3 The History of Type Evolution

Traditionally record types have been the most commonly used database type structures. When record types evolve in database systems, it is usually by the addition of new fields or a similar change to one or more of the field types. Cardelli's subtyping relation [Car84] completely captures this form of additive evolution in records and is used in the rest of the thesis as the basis for inclusion polymorphism.

In object oriented programming languages [DT88], there are two ways of creating a new class from an existing class : addition of new fields and over-riding or redefining existing ones. Inheritance is one mechanism which supports such code reuse. The evolution of a subclass from a superclass is similar to the evolution of record types in databases. In order to ensure type safety, the type of a field that is inherited by a subclass is required to be in the subtyping relation with the type of its counterpart in the superclass. Due to this property, the meanings of inheritance, evolution and subtyping have often been confused and the terms are used interchangeably.

However it is not always desirable for subtyping and subclassing to coincide when method types are taken into consideration. A binary method of an object of type o is a function that has at least one argument of type o. It is binary since it acts over at least two objects of the same type: the argument and the object itself. When methods with one or more parameters are inherited by a subclass they may be specialised to take and return values of smaller types. But the subtyping rule for functions is contravariant on the argument type i.e. it requires the argument type of a subtype to be greater than that of the supertype. Thus subtyping fails to capture the specialisation often needed by inheritance of method types. Despite this incompatibility, the existence of a subtyping relation is often claimed between a superclass and its subclasses where it is not the case. In these contexts, the subtyping relation is forced to capture a situation it is not intended to model. This leads to unsound or, at best, dynamically type checked systems.

Despite these concerns, covariant subtyping, where the argument type of a function subtype is smaller than that of the supertype, is used in some object oriented systems as it is more intuitive and expressive in this context. Eiffel [Mey92] and O_2 [Zic89, Deu90, LRV90, Deu91, Zic91] use this technique to create more specialised classes. The designers of these systems claim that the

use of covariance for the inheritance of binary methods has not, in practice, caused many problems regarding type safety.

In [Cas95], Castagna explains that covariance and contravariance are two distinct and independent mechanisms which need not be mutually exclusive. Contravariance is used by subtyping relations to allow substitution whereas covariance is a specialisation mechanism which permits substitution in some particular cases. In order to increase expressiveness, both can be integrated in a type-safe manner.

In recent years, a new concept called matching [Bru95, BCC+95, AC96, Bru96] has been introduced to deal with the problem of inheriting binary methods in object oriented systems. In this a matching relation between classes is defined and a subclass which matches its superclass can safely inherit and use its binary methods.

1.4 Motivation

The motivation for the work behind this thesis is to develop a mechanism for dealing with additive type evolution in persistent systems independently from any existing solutions. In particular, it is necessary to remove any ambiguity between inheritance and subtyping in the context of evolution. Even though matching is a useful solution to the problem of inheritance of binary methods, it is only applicable to object types in object oriented programming and does not deal with other type constructors or paradigms.

The aim of this work is to develop a more general mechanism that captures type evolution in most generally used type constructors. The proposed strategy for developing this mechanism is to study patterns of type evolution for different type constructors in persistent systems and to define a new relation, called **extension**, without reference to any other existing ones, to model the most common patterns. A programming language mechanism is then devised to support polymorphism over this relation.

The work presented in this thesis is an attempt at developing a formal model of this relation and an implementation of a type checker for the salient features of the relation and the polymorphism mechanism.

1.5 Outline of Thesis

This thesis consists of eleven chapters. Chapters 2 to 7 provide the basis for the new work and discuss the background topics and related work. Chapter 8 presents the new relation and the polymorphism mechanisms over it. A language supporting extension polymorphism, called Ext, is formally defined and its soundness is proved in Chapter 9. An implementation of a type checker for the features given in Chapters 8 and 9 and related issues are discussed in Chapter 10. The following sections give brief descriptions of the contents of each of these chapters. The conclusions from the work are presented in Chapter 11.

1.5.1 Fundamentals

A core language called Base is introduced in Chapters 2, 3 and 4 as a basis for the work. The type system of Base supports base types such as integers and boolean and the most common type constructors used in programming languages such as records, variants, functions and locations. It does not support any form of polymorphism. A formal definition of Base and a proof of soundness of its type system is given. Base is extended further as necessary in later chapters to illustrate relevant concepts and to incorporate new features.

1.5.2 Background

Chapter 5 presents the background to the main work of the thesis. A detailed discussion of polymorphism, type checking and object oriented programming is given.

The main concepts behind polymorphism and its uses are explained in detail in section 5.1. The main function of a type checker is to match the expected type of an expression with its actual type to ensure type safety. It determines, for example, whether a function has been supplied with the right type of actual parameter and whether a location has been assigned the right type of expression. For a complete piece of code to be correctly typed, every expression and statement in it must be correctly typed. Examples of type checking are given and the properties of type checkers are discussed.

Object oriented programming has emerged as one of most popular paradigms in recent years. In an object oriented system the real world is modelled by objects. Objects can be considered to be instances of abstract data types encapsulating

both state and behaviour. Section 5.3 explains the most important concepts used by object oriented programming.

1.5.3 Schema Evolution

Databases contain data that are logically grouped into schemata. Changes that have to be made to the schemata in a database in order to reflect changes in the application domain are known as schema evolution. Chapter 6 presents an overview of the context and kinds of schema evolution and the problems caused by it. The strategy adopted by the O_2 database system to deal with schema evolution at the data level is also described.

1.5.4 Related Work

There have been various solutions proposed to overcome the problem of type evolution. Chapter 7 describes two of these that are of most relevance to this thesis. Eiffel makes use of covariant subtyping to deal with the inheritance of binary methods in subclassing. A dynamic check is used to ensure type safety.

The concept of matching has been recently introduced to capture the type safe inheritance of binary methods in object oriented languages. TooL [GM95, GM96] and PolyTOIL [BSG95] are two languages which have incorporated matching in their type systems. Chapter 7 also gives an overview of these type systems and describes how matching is used to guarantee type safety in inheritance. A brief account of the techniques used by some other languages is also presented.

1.5.5 Extension Polymorphism

The aim of the work behind this thesis is to develop a means of capturing the most common patterns of type evolution in persistent systems. A new relation, called **extension**, which models additive evolution over records, variants, functions and locations, is introduced. A bounded quantification mechanism is then developed over this relation to provide polymorphism. Chapter 8 explains the design and development of these features in detail.

Extension polymorphism is incorporated into the experimental language Base. The resulting language Ext and its semantics are formally defined in Chapter 9. A proof of soundness for the type system of Ext is also presented.

1.5.6 A Type Checker for Extension Polymorphism

A type checker for the extension relation and the polymorphism mechanism presented in Chapter 8 is implemented in S-algol [Mor79, CM82] and tested. Chapter 10 explains the implementation strategy, type representations and the type checking issues involved in the process.

2 The Base Language

We introduce a core language, Base, which will be used as the basis for introducing parametric, inclusion and extension polymorphism. The type system of Base incorporates some of the most common types found in programming languages and is intended to be our experimental base. Base, as described here, contains no mechanism to provide polymorphism. The language will be extended in Chapter 5 to provide parametric and inclusion polymorphism and in Chapter 8 for extension polymorphism.

For the moment, Base does not support recursion at the type level which means that no type can be defined in terms of itself. This restriction is introduced to simplify the definition since it is known that recursion yields non-trivial anomalies. [Ghe93b] shows that the addition of recursion to System $F_{\leq}$ is not conservative. Conservativity is a property which ensures that any extension to a system preserves existing relations in the system. Thus, when recursion is added to $F_{\leq}$, it is possible to create a type lattice in which previously unrelated types are in the subtyping relation. In the absence of recursion such complications in the prototype language are avoided and various forms of polymorphism can be considered. Section 11.4 incorporates a discussion on the addition of recursion to the language.

A definition of the syntax of Base is presented in sections 2.1 to 2.6 and the typing rules for all the syntactic constructs in Base are given in section 2.7. Chapter 3 describes the semantics of Base and a proof of soundness of its type system is presented in Chapter 4.

2.1 Programs

The set of programs that can be constructed in Base can be defined as :

P ::= D; E

where D is a set of declarations and E is a set of expressions in the language. Thus a program in Base is a set of expressions preceded by a set of declarations. D and E are defined formally in later sections.

2.2 Types

The set of types in Base, ranged over by T, is given by the following syntax :

T ::= **int** | **bool** | **unit** | **decl** | { $l_1 : T, \ldots, l_n : T$ } | [$l_1 : T, \ldots, l_m : T$] |

fun($T \rightarrow T$) | **loc**(T) | **any** | t

where

Type	Interpretation
int	integers
bool	boolean values
unit	trivial type
decl	sets of bindings consisting of identifiers and their meanings
{ $l_1 : T_1, \ldots, l_n : T_n$ }	labelled cartesian cross products or records; $l_1, \ldots, l_n$ are distinct labels
[$l_1 : T_1, \ldots, l_m : T_m$]	labelled disjoint sums or variants; $l_1, \ldots, l_m$ are distinct labels
fun($T_1 \rightarrow T_2$)	functions and procedures
loc(T)	locations of type T
any	infinite union
t	type identifier

Figure 2.1 : Types in the Base Language

The base types provided are integers, booleans and two trivial types ***unit*** and ***decl*** which contain no elements. ***unit*** is the type of expressions that do not evaluate to a value of any other type and ***decl*** is the type of declarations. Records provide a means for collecting fields of different types into one structure. Variants offer the choice of any one of a predefined collection of fields of different types. Functions take a value of the argument type and return a value of the result type. Mutability is explicitly modelled in this type system and hence the type constructor ***loc***. Any value in the value space of Base can be an element of the infinite union type ***any***. While the rest of the types correspond to values in the value space, ***decl*** is used to type declarations, ***unit*** is the type of void constructs in Base, and variants and ***any*** provide abstractions over types. Types can also be type identifiers introduced by type declarations explained in the next section.

2.3 Declarations

The set of possible declarations in the base language can be defined by

D ::= **type** t **is** T | **let** x = E | D ; D

where

Syntax	Interpretation
type t **is** T	binds the type identifier t to the type expression T
let x = E	binds the identifier x to the expression E
D ; D	sequences declarations

Figure 2.2 : Declarations

Thus, Base allows type and identifier declarations and the sequencing of any of these declarations.

2.4 Expressions

The set of expressions in the base language is given by the following syntax :

E ::= n | b | E + E | E - E | ~ E | E **or** E | E **and** E | E = E |

{ l_1 = E, . . . , l_n = E } | E.l |

[l_i : E] : T | **project** E **as** X **onto** l : T **in** E **else** E |

fun(x : T → T) E | E(E) | x |

^ E | E := E | @ E |

inject(E, T) | **project** E **as** X **onto** T **in** E **else** E |

if E **then** E **else** E | E ; E | **begin** D ; E **end**

where

Syntax	Interpretation
n	integer literal
b	boolean literal
E_1 + E_2	sum of two integer expressions
E_1 - E_2	difference between two integer expressions
~ E	boolean negation
E_1 **or** E_2	logical or
E_1 **and** E_2	logical and
E_1 = E_2	test of equality
{ l_1 = E_1, . . . , l_n = E_n }	record constructor
E.l	field selection from a record
[l_i : E_i] : T	variant constructor
project E **as** X **onto** l : T **in** E_1 **else** E_2	projection from a variant onto a label with projected value bound to X in the expression E_1
fun(x : $T_1 \rightarrow T_2$) E	function expression
E_1(E_2)	function application
x	identifier
^ E	location that contains E
E_1 := E_2	assignment
@ E	value contained in location E
inject(E, T)	injection into **any**
project E **as** X **onto** T **in** E_1 **else** E_2	projection from a value of **any** with projected value bound to X in the expression E_1
if E **then** E_1 **else** E_2	conditional
E_1 ; E_2	sequence of expressions
begin D ; E **end**	block definition

Figure 2.3 : Base Language Expressions

The atomic expressions *n* and *b* stand for literals of base types ***int*** and ***bool*** respectively. Integers have the operators for addition and subtraction, + and -,

defined over them. Booleans have the logical *not*, *or* and *and* operators defined over them. In addition to these, both integers and booleans have an equality operation.

Record expressions are constructed by associating an expression with each field of the record. A dot operator is provided to dereference a record field.

A variant expression can be constructed by associating an expression with the chosen field and specifying the variant type. This type information is necessary to identify the exact type to which the variant expression belongs since many variants may have the same labels. The project clause is provided to dereference the current value of a variant. This operation projects the value onto the label specified. If the projection is successful then the result of the operation is the expression following the ***in*** keyword. The identifier specified following the ***as*** keyword will have the value of the variant and the type T associated with it within this expression. If the label does not match the variant then the ***else*** expression is chosen and the identifier will have the value and the type of the variant.

A function value is created by specifying the formal parameter, argument and result types and the expression for the function body. A function can be applied by supplying it with an actual parameter within round brackets.

Locations have three operations defined over them. The ^ operator creates a location containing the expression following it. The infix assignment operator takes a location value and an expression and assigns the latter to the former. The value contained in a location can be dereferenced by the @ operator.

A value of type ***any*** can be created by performing an inject operation on an expression along with its type. The projection operation over ***any*** is very similar to the one for variants. In this case, the value is projected onto the type specified.

The infix *;* operator allows the specification of a sequence of expressions just as it is used for sequencing declarations. The keywords ***begin*** and ***end*** mark the beginning and end of a block definition. The ***if then else*** clause provides the means to specify two way choices. If the boolean expression E is true then the result of the whole conditional expression is E_1 otherwise it is E_2.

2.5 Abbreviations

The following abbreviations in notation are introduced:

Abbreviation	Stands for	Explanation
$\{ l_i : T_i \}_{(i = 1, n)}$	$\{ l_1 : T_1, \ldots, l_n : T_n \}$	a record type containing fields l_1 to l_n with corresponding types T_1 to T_n
$\{ l_k : T_k \}^+$	$\{ \ldots, l_k : T_k, \ldots \}$	a record type containing at least the field l_k with type T_k
$\{ l_i = E_i \}_{(i = 1, n)}$	$\{ l_1 = E_1, \ldots, l_n = E_n \}$	a record expression containing fields l_1 to l_n with expressions E_1 to E_n associated with them
$[l_i : T_i]_{(i = 1, m)}$	$[l_1 : T_1, \ldots, l_m : T_m]$	a variant type containing branches l_1 to l_m with corresponding types T_1 to T_m
$[l_k : T_k]^+$	$[\ldots, l_k : T_k, \ldots]$	a variant type containing at least the branch l_k with type T_k

Figure 2.4 : Abbreviations

2.6 Naming Conventions

The conventions specified below are adopted for naming different constructs in Base in the sections to follow.

- expression variables belong to the set { x, y, z }
- type variables belong to the set { s, t, u, v }
- types belong to the set { S, T, U, V }

2.7 Typing Rules

The concept of environments is introduced to determine the types of the constructs in Base. In the following typing rules for programs, declarations and expressions, environments are lists of bindings. π denotes the environments where identifiers are bound to their types (x : T) and τ stands for the

environments in which type identifiers are bound to type expressions (t = T). A_1::b::A_2 is used to represent a list A which contains a binding b. A++B is used to denote the concatenation of two lists of bindings A and B. Both π and τ are global environments and support block structure. The notation α.i is used to represent the identifier i contained in environment α.

The function *typeDecl* takes a list of bindings between type identifiers and type expressions and adds them to the environment τ. Similarly, function *idDecl* takes a list of bindings between identifiers and types as its arguments and updates the environment π with the new bindings. Bindings are represented as pairs and the notation < x, T > is used to denote a pair value consisting of x and T. (b_1, . . . , b_n) is a list containing bindings b_1 to b_n.

The typing rules make use of a set *Type* which is a set of strings defined by T in section 2.2. If a type S can be generated by this definition then S is a member of *Type*. Thus *Type* is a set of all well-formed types in Base. Similarly all expressions e and e_i in the type rules belong to the set of all well-formed expressions, *Expression*, defined by E in section 2.4.

In the following type rules, all variables are in italics and all Base expressions are in `Courier` font. The keywords of Base are in bold.

2.7.1 Programs

$$\frac{\phi,\ \phi \vdash \mathrm{D} : \textbf{decl}\ \ \mathrm{typeDecl}(\ \Omega\ )\ \mathrm{idDecl}(\ \Psi\ ) \qquad \phi{::}\Omega,\ \phi{::}\Psi \vdash \mathrm{e} : \mathrm{T}}{\phi,\ \phi \vdash \texttt{D ; e} : \mathrm{T}}\ [\text{program}]$$

If declaration D causes the lists of bindings Ω and Ψ to be added to the type and identifier environments which were previously empty and given these bindings, the type of expression e can be deduced to be T then the type of a program `D ; e` is also T.

2.7.2 Declarations

$$\frac{\tau \vdash \mathrm{T} \in \mathrm{Type}}{\tau \vdash \textbf{type t is T} : \textbf{decl} \quad \mathrm{typeDecl}(\ (\ <\ \mathrm{t},\ \mathrm{T}\ >\ )\ )}\ [\text{typeDecl}]$$

If τ has a new binding associating the type identifier t to type expression T then the declaration **`type t is T`** is of type **decl** and will have the effect of updating τ with the new binding.

$$\frac{\tau,\ \pi \vdash \mathtt{e} : \mathrm{T}}{\tau,\ \pi \vdash \mathbf{let}\ \mathtt{x}\ =\ \mathtt{e} : \mathbf{decl}\quad \mathrm{idDecl}(\ (\ <\ \mathrm{x},\ \mathrm{T}\ >\ )\ )} \qquad \text{[idDecl]}$$

If it can be deduced that an expression e is of type T then the declaration **let** x = e is of type **decl** and will have the effect of updating π with the binding x : T.

$$\frac{\tau,\ \pi \vdash \mathrm{D}_1 : \mathbf{decl}\ \ \mathrm{Decl}_1 \quad \tau{+}{+}\Omega_1,\ \pi{+}{+}\Psi_1 \vdash \mathrm{D}_2 : \mathbf{decl}\ \ \mathrm{Decl}_2}{\tau,\ \pi \vdash \mathtt{D}_1\ \mathtt{;}\ \mathtt{D}_2 : \mathbf{decl}\ \ \mathrm{typeDecl}(\ \Omega_1\ {+}{+}\ \Omega_2\ )\ \ \mathrm{idDecl}(\ \Psi_1\ {+}{+}\ \Psi_2\ )} \qquad \text{[declSeq]}$$

where Decl_1 stands for typeDecl(Ω_1) and idDecl(Ψ_1) and Decl_2 stands for typeDecl(Ω_2) and idDecl(Ψ_2).

If declaration D_1 causes changes Ω_1 and Ψ_1 to be introduced to the environments τ and π, and when these changes have been incorporated into the environments, declaration D_2 causes changes Ω_2 and Ψ_2 to be added then the declaration sequence operation D_1; D_2 has the effect of causing changes $\Omega_1{+}{+}\Omega_2$ and $\Psi_1{+}{+}\Psi_2$ respectively.

2.7.3 Expressions

The type rules for expressions belonging to each base type and type constructor in the language are given below.

2.7.3.1 Integers

$$\frac{\mathrm{n}\ \in\ \mathrm{Integer}}{\mathtt{n} : \mathbf{int}} \qquad \text{[intValue]}$$

If n belongs to the set of integer literals then n is of type **int**.

$$\frac{\tau,\ \pi \vdash \mathtt{e}_1 : \mathbf{int} \quad \tau,\ \pi \vdash \mathtt{e}_2 : \mathbf{int}}{\tau,\ \pi \vdash \mathtt{e}_1\ +\ \mathtt{e}_2 : \mathbf{int}} \qquad \text{[intAdd]}$$

If expressions $\mathtt{e}_1$ and $\mathtt{e}_2$ are of type **int** then the expression $\mathtt{e}_1$ + $\mathtt{e}_2$ is also of type **int**.

$$\frac{\tau,\ \pi \vdash \mathtt{e}_1 : \mathbf{int} \quad \tau,\ \pi \vdash \mathtt{e}_2 : \mathbf{int}}{\tau,\ \pi \vdash \mathtt{e}_1\ -\ \mathtt{e}_2 : \mathbf{int}} \qquad \text{[intSub]}$$

If expressions e_1 and e_2 are of type **int** then the expression `e1 - e2` is also of type **int**.

$$\frac{\tau, \pi \vdash e_1 : \textbf{int} \quad \tau, \pi \vdash e_2 : \textbf{int}}{\tau, \pi \vdash e_1 = e_2 : \textbf{bool}} \qquad \text{[intEq]}$$

If expressions e_1 and e_2 are of type **int** then the expression `e1 = e2` is of type **bool**.

2.7.3.2 Booleans

$$\frac{b \in \text{Boolean}}{b : \textbf{bool}} \qquad \text{[boolValue]}$$

If b belongs to the set of boolean literals then b is of type **bool**.

$$\frac{\tau, \pi \vdash e : \textbf{bool}}{\tau, \pi \vdash \sim e : \textbf{bool}} \qquad \text{[negation]}$$

If expression e is of type **bool** then the expression ~e is of type **bool**.

$$\frac{\tau, \pi \vdash e_1 : \textbf{bool} \quad \tau, \pi \vdash e_2 : \textbf{bool}}{\tau, \pi \vdash e_1 \ \textbf{or} \ e_2 : \textbf{bool}} \qquad \text{[or]}$$

If expressions e_1 and e_2 are of type **bool** then the expression `e1 or e2` is also of type **bool**.

$$\frac{\tau, \pi \vdash e_1 : \textbf{bool} \quad \tau, \pi \vdash e_2 : \textbf{bool}}{\tau, \pi \vdash e_1 \ \textbf{and} \ e_2 : \textbf{bool}} \qquad \text{[and]}$$

If expressions e_1 and e_2 are of type **bool** then the expression `e1 and e2` is also of type **bool**.

$$\frac{\tau, \pi \vdash e_1 : \textbf{bool} \quad \tau, \pi \vdash e_2 : \textbf{bool}}{\tau, \pi \vdash e_1 = e_2 : \textbf{bool}} \qquad \text{[boolEq]}$$

If expressions e_1 and e_2 are of type **bool** then the expression `e1 = e2` is of type **bool**.

2.7.3.3 Records

$$\frac{\forall i \in \{1...n\}\ (\ l_i \in \text{labels} \quad \tau, \pi \vdash e_i : T_i\ )}{\tau, \pi \vdash \{\ l_i = e_i\ \}_{(i\ =\ 1,\ n)} : \{\ l_i : T_i\ \}_{(i\ =\ 1,\ n)}} \qquad \text{[recValue]}$$

If l_1 to l_n belong to the set of labels and expressions e_1 to e_n are of type T_1 to T_n respectively then the record expression formed by $\{\ l_i = e_i\ \}_{(i\ =\ 1,\ n)}$ is of type $\{\ l_i : T_i\ \}_{(i = 1, n)}$

$$\frac{\tau, \pi \vdash e : \{\ l : T\ \}^+}{\tau, \pi \vdash e.l : T} \qquad \text{[recDeref]}$$

If e is a record expression with at least the field l of type T then the dereference expression `e.l` is of type T.

2.7.3.4 Variants

$$\frac{\tau, \pi \vdash e_i : T_i \quad \tau, \pi_1 :: x : T :: \pi_2 \vdash x : [\ l_i : T_i\ ]^+}{\tau, \pi \vdash [\ l_i\ =\ e_i\ ]\ :\ T : [\ l_i : T_i\ ]^+} \qquad \text{[varValue]}$$

If T is a variant type with at least a label l_i of type T_i and expression e_i has type T_i then the type of the expression `[ li = ei ] : T` is a variant type with at least a label l_i of type T_i.

$$\frac{\tau, \pi \vdash e : [\ l_i : T_i\ ]^+ \quad \tau, \pi_1 :: x : T_i :: \pi_2 \vdash e_1 : T \quad \tau, \pi \vdash e_2 : T}{\tau, \pi \vdash \textbf{project}\ e\ \textbf{as}\ x\ \textbf{onto}\ l_i : T_i\ \textbf{in}\ e_1\ \textbf{else}\ e_2\ : T} \qquad \text{[varProj]}$$

If the type of expression e is a variant with at least a label l_i which is of type T_i, expression e_1 is of type T if the identifier x is of type T_i and expression e_2 is of type T then the project operation on the variant type, **`project`** `e` **`as`** `x` **`onto`** `li : Ti` **`in`** `e1` **`else`** `e2`, has type T.

2.7.3.5 Functions

$$\frac{\tau, \pi_1 :: x : T_1 :: \pi_2 \vdash e : T_2}{\tau, \pi \vdash \textbf{fun}(\ x\ :\ T_1 \rightarrow T_2\ )\ e : \textbf{fun}(\ T_1 \rightarrow T_2)} \qquad \text{[funValue]}$$

If τ and π with a binding $x : T_1$ imply that expression e is of type T_2 then the function expression **fun** (x : $T_1 \rightarrow T_2$) e is of type **fun** ($T_1 \rightarrow T_2$).

$$\frac{\tau, \pi \vdash e_1 : T_1 \quad \tau, \pi \vdash e : \textbf{fun}(T_1 \rightarrow T_2)}{\tau, \pi \vdash e(e_1) : T_2} \qquad \text{[funApp]}$$

If expression e_1 is of type T_1 and expression e is of a function type **fun** ($T_1 \rightarrow T_2$) then the expression e(e_1) is of type T_2.

2.7.3.6 Identifiers

$$\frac{}{\tau, \pi_1 :: x : T :: \pi_2 \vdash x : T} \qquad \text{[id]}$$

If π contains a binding associating the identifier x with type T then x is of type T.

2.7.3.7 Locations

$$\frac{\tau, \pi \vdash e_2 : T \quad \tau, \pi \vdash e_1 : \textbf{loc}(T)}{\tau, \pi \vdash e_1 := e_2 : \textbf{unit}} \qquad \text{[assign]}$$

If expression e_2 is of type T and expression e_1 is of a location type **loc**(T) then the assignment expression e_1 := e_2 is of type **unit**.

$$\frac{\tau, \pi \vdash e : T}{\tau, \pi \vdash \hat{}e : \textbf{loc}(T)} \qquad \text{[locValue]}$$

If expression e is of type T then the expression ^e is of type **loc**(T).

$$\frac{\tau, \pi \vdash e : \textbf{loc}(T)}{\tau, \pi \vdash @e : T} \qquad \text{[locDeref]}$$

If expression e is of type **loc**(T) then the dereference expression @e is of type T.

2.7.3.8 Infinite Union

$$\frac{\tau,\ \pi \vdash \text{e} : \text{T}}{\tau,\ \pi \vdash \texttt{inject(}\ \text{e},\ \text{T}\ \texttt{)} : \textbf{any}} \qquad \text{[anyInj]}$$

If expression e is of type T then the expression **`inject( e, T )`** is of type **any**.

$$\frac{\tau,\ \pi \vdash \text{e} : \textbf{any} \quad \text{T}_1 \in \text{types} \quad \tau,\ \pi_1 :: x : \text{T}_1 :: \pi_2 \vdash \text{e}_1 : \text{T} \quad \tau,\ \pi \vdash \text{e}_2 : \text{T}}{\tau,\ \pi \vdash \textbf{project}\ \text{e}\ \textbf{as}\ x\ \textbf{onto}\ \text{T}_1\ \textbf{in}\ \text{e}_1\ \textbf{else}\ \text{e}_2\ : \text{T}} \qquad \text{[anyProj]}$$

If expression e is of type **any**, T_1 is a valid type in Base, expression e_1 is of type T if the identifier x is of type T_1 and expression e_2 is also of type T then the project expression on **any,** **`project e as x onto`** T_1 **`in`** e_1 **`else`** e_2, is of type T.

2.7.3.9 Sequence

$$\frac{\tau,\ \pi \vdash \text{e}_1 : \textbf{unit} \quad \tau,\ \pi \vdash \text{e}_2 : \text{T}}{\tau,\ \pi \vdash \text{e}_1\ \texttt{;}\ \text{e}_2 : \text{T}} \qquad \text{[seq]}$$

If expression e_1 is of type **unit** and expression e_2 is of some type T then the sequence of expressions e_1 `;` e_2 is also of type T.

2.7.3.10 Block

$$\frac{\tau,\ \pi \vdash \text{D} : \textbf{decl} \quad \text{typeDecl}(\ \Omega\ ) \quad \text{idDecl}(\ \Psi\ ) \quad \tau + + \Omega,\ \pi + + \Psi \vdash \text{e} : \text{T}}{\tau,\ \pi \vdash \textbf{begin}\ \text{D}\ \texttt{;}\ \text{e}\ \ \textbf{end} : \text{T}} \qquad \text{[block]}$$

If declaration D of type **decl** causes changes Ω and Ψ to environments τ and π respectively and if, after incorporating the changes to the environments, the expression e is of type T then the block definition **`begin D ; e end`** is also of type T.

2.7.3.11 Conditional

$$\frac{\tau, \pi \vdash e : \textbf{bool} \quad \tau, \pi \vdash e_1 : T \quad \tau, \pi \vdash e_2 : T}{\tau, \pi \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : T} \qquad \text{[if]}$$

If expression e is of type **bool** and expressions e_1 and e_2 are of the same type T then the conditional expression **`if`** e **`then`** e_1 **`else`** e_2 is also of type T.

2.8 Summary

An experimental base language called Base has been formally defined in this chapter and the typing rules for its constructs are presented. Base includes some of the most common type constructors used by programmers but does not incorporate recursion or any form of polymorphism. Later chapters will use Base as the basis for introducing different kinds of polymorphism.

3 Semantics of the Base Language

The syntax of Base was formally defined in the previous chapter. We now present a semantic context for the types in Base and a semantics for its constructs.

3.1 A Semantic Context for Types

The semantic context for the types which can be used to construct values in the value space of Base along with the notation used for them in semantics is given in Figure 3.1 below. The notation ⟦ T ⟧ is used to represent the meaning of type T.

Type	Semantic Context	Denotation of Semantic Context
int	the set of integers	Integer
bool	the set of boolean values (true, false)	Boolean
unit	the empty set	Unit
decl	the set of bindings caused by a declaration	List(Pair (identifier, meaning)) Abbreviated to Decl
$\{ l_1 : T_1, \ldots, l_n : T_n \}$	the set of tuples with fields l_1 to l_n	Record($l_1 : ⟦ T_1 ⟧, \ldots,$ $l_n : ⟦ T_n ⟧$)
fun($T_1 \rightarrow T_2$)	the set of functions from $⟦ T_1 ⟧$ to $⟦ T_2 ⟧$	Function($⟦ T_1 ⟧ \rightarrow ⟦ T_2 ⟧$)
loc(T)	the set of locations that contain ⟦ T ⟧	Location(⟦ T ⟧)

Figure 3.1 : Semantic Context of Value Constructors

The semantics of type constructors which merely provide an abstraction over other types rather than creating new values in the value space are given in Figure 3.2. The meta-type *typeRep* used in this table stands for type representations of any type in Base.

Type	Context	Denoted by
$[\![[l_1 : T_1, \ldots, l_m : T_m]]\!]$	the set of values being one of $[\![T_1]\!]$ to $[\![T_m]\!]$ for labels l_1 to l_m	Pair (label, $[\![T_n]\!]$) where T_n is one of T_1 to T_m
$[\![$ **any** $]\!]$	the set of all the values in the language coerced into a general type	Pair (typeRep, $[\![T]\!]$) where T can be any type in Base

Figure 3.2 : Semantic Context of Type Abstractions

A variant type represents a union of the sets of values belonging to the types corresponding to all the labels. Since a variant value at any time can only be a value of one of the component types, it can be represented as a label - value pair and its type accordingly. Since ***any*** is an infinite union type incorporating all legal values in Base of any type, values of this type can be represented as a pair of type representation indicating the specific type and value.

3.2 Meta-operations

The meta-operations defined over the value constructors described in Figure 3.1 are given below in Figure 3.3.

Semantic Type	Meta-operations
Integer	add : (Integer, Integer → Integer) subtract : (Integer, Integer → Integer) intEquals : (Integer, Integer → Boolean)
Boolean	andOp : (Boolean, Boolean → Boolean) orOp : (Boolean, Boolean → Boolean) notOp : (Boolean → Boolean) boolEquals : (Boolean, Boolean → Boolean)
Unit	
Decl	

Record ($l_1 : T_1, \ldots, l_n : T_n$) Abbreviated to: Record ($l_k : T_k)_{(k = 1, n)}$	mkRec : (List(Pair($label_i$, T_i)) $\rightarrow$ Record ($l_k : T_k)_{(k = 1, n)}$) $getL_1$: (Record ($l_k : T_k)_{(k = 1, n)} \rightarrow T_1$) . . . $getL_n$: (Record ($l_k : T_k)_{(k = 1, n)} \rightarrow T_n$)
Function ($T_1 \rightarrow T_2$)	mkFun : (expression, variable, environment $\rightarrow$ Function ($T_1 \rightarrow T_2$)) apply : (Function ($T_1 \rightarrow T_2$), $T_1 \rightarrow T_2$)
Location (T)	mkLoc : (T $\rightarrow$ Location(T)) put : (Location(T), T $\rightarrow$ Unit) get : (Location(T) $\rightarrow$ T)

Figure 3.3 : Meta-operations for Type Constructors

Integers have the meta-operations *add*, *subtract* and *intEquals* defined over them. They are used to perform integer addition, integer subtraction and the test for integer equality respectively. Boolean values are provided with *andOp*, *orOp*, *notOp* and *boolEquals* which are used for logical and, logical or, boolean negation and the test for boolean equality.

There are no meta-operations defined for *Unit* and *Decl* since no values are constructed of these types.

The meta-operations provided for records are a *mkRec* function that takes a list of pairs consisting of labels and expressions and returns a record with fields corresponding to the pairs in the list and 'get functions' for each label of the record which take a record and return the value associated with the label.

Functions have a *mkFun* meta-operation that takes the expression of function body, the parameter variable and the environment in which the expression is to be evaluated and returns a function. The *apply* operation takes a function and a parameter value and returns a value of the result type.

Mutable types have three meta-operations associated with them. The *mkLoc* function takes a value of type T and returns a location which contains the value. The *put* operation takes a location containing T and a value of type T and assigns the value to the location. Since there is no value to be returned the result type is Unit. The *get* operation takes a location and returns the value contained in the location.

The meta-operations for type abstractions are given below in Figure 3.4.

Semantic Type	Meta-operations
Pair (label, ⟦ T ⟧) for variants	mkPair : (label, ⟦ T ⟧ → Pair (label, ⟦ T ⟧)) fst : (Pair (label, ⟦ T ⟧) → label) snd : (Pair (label, ⟦ T ⟧) → ⟦ T ⟧)
Pair (typeRep, ⟦ T ⟧) for any	mkPair : (typeRep, ⟦ T ⟧ → Pair (typeRep, ⟦ T ⟧)) fst : (Pair (typeRep, ⟦ T ⟧) → typeRep) snd : (Pair (typeRep, ⟦ T ⟧) → ⟦ T ⟧)

Figure 3.4 : Meta-operations for Type Abstractions

Since variants are modelled as a pair of label and value, the meta-operations on these structures are those available to pairs. A *mkPair* operation, which takes a label and a value belonging to the type of the label and returns a pair, is provided as the constructor. In order to dereference the pair, two operations, *fst* and *snd*, are defined which return the first and the second elements of the pair.

The meta-operations associated with ***any*** are very similar to those for variants since ***any*** is also modelled as a pair. In this case the *mkPair* operation takes a type representation and a value of the type and returns a pair consisting of both. The *fst* and *snd* operations are again defined to access the first and second elements of the pair.

3.3 A Semantics for Base

Given the semantic context and meta-operations specified in Figures 3.1 to 3.4, the semantics of the language can now be defined. $⟦ e ⟧_{Env}$ is used to denote the meaning of a well-typed expression e and Env is the environment against which meanings of expressions are defined. $Env_{x = V \in T}$ is used to indicate that in this environment, identifier x has meaning V belonging to type T associated with it. Similarly Env_A is used to mean that the environment incorporates any changes that might have been caused by an expression or declaration A. ϕ stands for an environment which contains no bindings. The function *getTypeRep* used for infinite unions takes any legal type in Base and returns its type representation as a *typeRep*. Function *if*, used to define the semantics of variants, infinite union and conditionals, takes a Boolean and the meanings of two expressions and returns one of the meanings depending on the Boolean value.

3.3.1 Programs

$[\![D ; e]\!]_{\phi} = [\![e]\!]_{Env_D}$ where $Env_D = [\![D]\!]_{\phi}$ D1

3.3.2 Declarations

$[\![$ **type** t **is** T $]\!]_{Env} = \phi$ D2

$[\![$ **let** x = e $]\!]_{Env}$ = mkList(< x, $[\![e]\!]_{Env}$ >) D3

$[\![D_1 ; D_2]\!]_{Env} = [\![D_1]\!]_{Env} ++ [\![D_2]\!]_{Env_{D_1}}$ D4

3.3.3 Integers

$[\![n]\!]_{Env} = n$ D5

where n is the semantic meaning (integer value n) of the syntactic form n.

$[\![e_1 + e_2]\!]_{Env}$ = add($[\![e_1]\!]_{Env}$, $[\![e_2]\!]_{Env}$) D6

$[\![e_1 - e_2]\!]_{Env}$ = subtract($[\![e_1]\!]_{Env}$, $[\![e_2]\!]_{Env}$) D7

$[\![e_1 = e_2]\!]_{Env}$ = intEquals($[\![e_1]\!]_{Env}$, $[\![e_2]\!]_{Env}$) D8

3.3.4 Booleans

$[\![b]\!]_{Env} = b$ D9

where b is the semantic meaning (boolean value b) of the syntactic form b.

$[\![\sim e]\!]_{Env}$ = notOp($[\![e]\!]_{Env}$) D10

$[\![e_1$ **or** $e_2]\!]_{Env}$ = orOp($[\![e_1]\!]_{Env}$, $[\![e_2]\!]_{Env}$) D11

$[\![e_1$ **and** $e_2]\!]_{Env}$ = andOp($[\![e_1]\!]_{Env}$, $[\![e_2]\!]_{Env}$) D12

$[\![e_1 = e_2]\!]_{Env}$ = boolEquals($[\![e_1]\!]_{Env}$, $[\![e_2]\!]_{Env}$) D13

3.3.5 Records

$[\![\{ l_i = e_i \}_{(i = 1, n)}]\!]_{Env} =$
mkRec(< l_1 , $[\![e_1]\!]_{Env}$ > ++ . . . ++ < l_n , $[\![e_n]\!]_{Env}$ >) D14

$[\![\text{e.l}]\!]_{Env} = \text{getL}([\![\text{e}]\!]_{Env})$ D15

3.3.6 Variants

$[\![[\text{l}_i : \text{e}_i] : [\text{l}_k : \text{T}_k]_{(k = 1, m)}]\!]_{Env} = \text{mkPair}(\text{l}_i, [\![\text{e}_i]\!]_{Env})$ D16

$[\![$ **project** e **as** x **onto** l : T **in** e_1 **else** $\text{e}_2]\!]_{Env} =$
if(sameLabel(fst($[\![\text{e}]\!]_{Env}$), l), $[\![\text{e}_1]\!]_{Env_{x = \text{snd}([\![\text{e}]\!]_{Env}) \in [\![\text{T}]\!]}}$, $[\![\text{e}_2]\!]_{Env}$)
D17

3.3.7 Functions

$[\![$ **fun** (x : $\text{T}_1 \rightarrow \text{T}_2$) e $]\!]_{Env}$ = mkFun(e, x, Env) D18

$[\![\text{e}_1(\text{e}_2)]\!]_{Env} = \text{apply}([\![\text{e}_1]\!]_{Env}, [\![\text{e}_2]\!]_{Env})$ D19

3.3.8 Identifiers

$[\![\text{x}]\!]_{Env_{x = V \in T}} = \text{V}$ D20

3.3.9 Locations

$[\![\text{\^{} e}]\!]_{Env} = \text{mkLoc}([\![\text{e}]\!]_{Env})$ D21

$[\![\text{x := e}]\!]_{Env} = \text{put}([\![\text{x}]\!]_{Env}, [\![\text{e}]\!]_{Env})$ D22

$[\![\text{@ e}]\!]_{Env} = \text{get}([\![\text{e}]\!]_{Env})$ D23

3.3.10 Infinite Union

$[\![$ **inject** (e, T) $]\!]_{Env}$ = mkPair(getTypeRep(T), $[\![\text{e}]\!]_{Env}$) D24

$[\![$ **project** e **as** x **onto** T **in** e_1 **else** $\text{e}_2]\!]_{Env} =$
if(sameTypeRep(fst($[\![\text{e}]\!]_{Env}$), getTypeRep(T)),
$[\![\text{e}_1]\!]_{Env_{x = \text{snd}([\![\text{e}]\!]_{Env}) \in [\![\text{T}]\!]}}$, $[\![\text{e}_2]\!]_{Env}$)
D25

3.3.11 Sequence

$[\![e_1 ; e_2]\!]_{Env} = [\![e_2]\!]_{Env_{e_1}}$ D26

3.3.12 Block

$[\![$ **begin** D ; e **end** $]\!]_{Env} = [\![e]\!]_{Env_D}$ D27

3.3.13 Conditional

$[\![$ **if** e **then** e_1 **else** $e_2]\!]_{Env}$ = if($[\![e]\!]_{Env}$, $[\![e_1]\!]_{Env}$, $[\![e_2]\!]_{Env}$)

D28

3.4 Type Mapping

Languages which support both structural type equivalence and the use of type identifiers require a rewrite rule to reduce a type expression to its canonical form. A canonical form in this context is the expanded version of a type which contains no type identifiers. Consider the following type definitions in Figure 3.5. The type ***string*** used below is a base type representing a string of characters.

type ageType **is loc**(**int**)

type person **is** { name : **string** , age : ageType }

type expandedPerson **is** { name : **string** , age : **loc**(**int**) }

Figure 3.5 : The Canonical Form of a Type

The definition of type *person* contains a type identifier *ageType.* Type *expandedPerson* is the canonical form of the type person since it defines the same type and the type identifier *ageType* in its definition has been replaced by the corresponding type expression.

The mapping of a type identifier t to its type expression T is denoted by $t \triangleright T$. For this mapping to be valid, the type environment τ should contain the binding $< t, T >$. The notation $\overline{T}$ is used in this thesis to denote the canonical form of a type T. $T \triangleright \overline{T}$ if every type identifier in T has been replaced by the type

expression associated with it in τ. The canonical form of any type expression can be determined by the recursive application of the following rules:

- $\{ l_i : T_i \}_{(i = 1, n)} \triangleright \{ l_i : \overline{T}_i \}_{(i = 1, n)}$
- $[l_i : T_i]_{(i = 1, m)} \triangleright [l_i : \overline{T}_i]_{(i = 1, m)}$
- **fun**$(T_1 \rightarrow T_2) \triangleright$ **fun**$(\overline{T}_1 \rightarrow \overline{T}_2)$
- **loc**$(T) \triangleright$ **loc**$(\overline{T})$

It should be noted that both T and $\overline{T}$ represent the same set of values. However, while τ may be needed to evaluate the meaning of T, the meaning of $\overline{T}$ can be evaluated independently. Thus, $[\![T]\!]_\tau = [\![\overline{T}]\!]_\phi$.

3.5 Summary

A semantic context for the type constructors in Base has been presented. Type constructors which create new values in the value space have been differentiated from those which merely provide an abstraction over other types. Meta-operations which are suitable for representing operations over the type constructors in their respective semantic contexts are defined. These meta-operations are then used to provide a semantics for the syntactic constructs in Base. The semantics introduced here is used in the next chapter to provide a proof of soundness of the typing rules for Base.

4 A Proof of Soundness for Base

Given a framework in which meanings can be assigned to expressions, the typing of an expression e of type T is sound if the meaning of e is in the set of meanings of T. The soundness of the typing rules for Base presented in section 2.7 can be proved by structural induction [FS91, Sch94] based on the following definition of soundness :

$\tau, \pi \vdash e : T \Rightarrow [\![e]\!]_{Env} \in [\![T]\!]_{\tau}$

where $\forall i \in Env, \exists i \in \pi . [\![Env.i]\!] \in [\![\pi.i]\!]$

This definition states that if the type of a well typed expression e can be deduced to be T from the environments τ (which contains the bindings between type identifiers and the types they stand for) and π (which contains the bindings between identifiers and their types) then the meaning of e evaluated with respect to an environment Env belongs to the meaning of T with respect to τ under the condition that for every identifier i in Env there is a corresponding i in π such that the meaning of i in Env belongs to the meaning of i in π. Since Env is a list of bindings between identifiers and their meanings, this condition ensures the consistency between the different environments used in the proof. It should also be noted here that given the definition of Decl in Figure 3.1, Env belongs to Decl.

The meanings of types in Base have been defined in Figures 3.1 and 3.2 and the meanings of expressions in Base can be evaluated using the semantics specified in section 3.4.

The proof of the whole type system can be broken down into proofs for programs, declarations and the different kinds of expressions that can be formed in the language. Sections 4.1 to 4.4 consider programs, declarations, base cases for the expressions and non-atomic expressions respectively.

4.1 Programs

The proof obligation for any program in Base can be stated as

$\phi, \phi \vdash p : T \Rightarrow [\![p]\!]_{\phi} \in [\![T]\!]_{\phi}$

This is a specialisation of the soundness rule given earlier for any expression. Since every binding that can be used by a program is contained within it, the soundness of the program has to be proven from empty premises.

Expression $\phi, \phi \vdash D ; e : T$

To be proved $[\![D ; e]\!]_\phi \in [\![T]\!]_\phi$

Inductive Hypotheses $[\![D]\!]_\phi \in [\![$ **decl** $]\!]_\phi$ and $[\![e]\!]_{Env_D} \in [\![T]\!]_\tau$

since D : **decl** and e : T with D added to τ and π

by type rule [program]

Inductive Step

$[\![D ; e]\!]_\phi = [\![e]\!]_{Env\ D}$ by D1

But $[\![e]\!]_{Env_D} \in [\![T]\!]_\tau$ by hypotheses

Therefore, $[\![D ; e]\!]_\phi \in [\![T]\!]_\tau$

However, $[\![D ; e]\!]_\phi \in [\![\overline{T}]\!]_\phi$ where $T \triangleright \overline{T}$

4.2 Declarations

The declarations are considered below.

Expression $\tau, \pi \vdash$ **type** t **is** T : **decl**

To be proved $[\![$ **type** t **is** T $]\!]_{Env} \in [\![$ **decl** $]\!]_\tau$

Inductive Hypothesis $T \in$ Type by type rule [typeDecl]

Inductive Step

$[\![$ **type** t **is** T $]\!]_{Env} = \phi$ by D2

Since, in this case, ϕ stands for the environment Env without any bindings,

$\phi \in [\![$ **decl** $]\!]_\tau$ by the definition of Env and Decl

Therefore, $[\![$ **type** t **is** T $]\!]_{Env} \in [\![$ **decl** $]\!]_\tau$

Expression $\tau, \pi \vdash$ **let** x = e : **decl**

To be proved $[\![$ **let** x = e $]\!]_{Env} \in [\![$ **decl** $]\!]_\tau$

Inductive Hypothesis $[\![e]\!]_{Env} \in [\![T]\!]_\tau$ by type rule [idDecl]

Inductive Step

$[\![$ **let** x = e $]\!]_{Env} = Env_{x = [\![e]\!]_{Env}}$ by D3

But $Env_{x = [\![e]\!]_{Env}} \in [\![$ **decl** $]\!]_\tau$ by the definition of Env and Decl

Therefore, $[\![$ **let** x = e $]\!]_{Env} \in [\![$ **decl** $]\!]_\tau$

Expression $\tau, \pi \vdash D_1 ; D_2 : \textbf{decl}$

To be proved $[\![D_1 ; D_2]\!]_{Env} \in [\![\textbf{decl}]\!]_\tau$

Inductive Hyp. $[\![D_1]\!]_{Env} \in [\![\textbf{decl}]\!]_\tau$ and $[\![D_2]\!]_{Env_{D_1}} \in [\![\textbf{decl}]\!]_\tau$

since $\tau, \pi \vdash D_1 : \textbf{decl}$ and $\tau', \pi' \vdash D_2 : \textbf{decl}$ where τ' and π' stand for τ and π with D_1 added to them

by type rule [declSeq]

Inductive Step

$[\![D_1 ; D_2]\!]_{Env} = [\![D_1]\!]_{Env} ++ [\![D_2]\!]_{Env_{D_1}}$ by D4

But $[\![D_1]\!]_{Env} \in [\![\textbf{decl}]\!]_\tau$ and $[\![D_2]\!]_{Env_{D_1}} \in [\![\textbf{decl}]\!]_\tau$ by hypotheses

$[\![D_1]\!]_{Env} ++ [\![D_2]\!]_{Env_{D_1}} \in [\![\textbf{decl}]\!]_\tau$ by the definition of Env

Therefore, $[\![D_1 ; D_2]\!]_{Env} \in [\![\textbf{decl}]\!]_\tau$

4.3 Base Cases

The two base cases for the proof are considered below.

Case 1 Expression $\tau, \pi \vdash n : \textbf{int}$

$[\![n]\!]_{Env} = n$ by D5

$n \in$ Integer by type rule [intValue]

$[\![\textbf{int}]\!]_\tau$ = Integer from Table 3.1

Therefore $[\![n]\!]_{Env} \in [\![\textbf{int}]\!]_\tau$

Case 2 Expression $\tau, \pi \vdash b : \textbf{bool}$

$[\![b]\!]_{Env} = b$ by D9

$b \in$ Boolean by type rule [boolValue]

$[\![\textbf{bool}]\!]_\tau$ = Boolean from Table 3.1

Therefore $[\![b]\!]_{Env} \in [\![\textbf{bool}]\!]_\tau$

4.4 Expressions

4.4.1 Integers

Expression $\tau, \pi \vdash e_1 + e_2 : \textbf{int}$

To be proved $[\![e_1 + e_2]\!]_{Env} \in [\![\textbf{int}]\!]_\tau$

Inductive Hypotheses	$[\![e_1]\!]_{Env} \in [\![\textbf{int}]\!]_\tau$ and $[\![e_2]\!]_{Env} \in [\![\textbf{int}]\!]_\tau$	
	since $\tau, \pi \vdash e_1 : \textbf{int}$ and $\tau, \pi \vdash e_2 : \textbf{int}$	
		by type rule

[intAdd]

Inductive Step

$[\![e_1 + e_2]\!]_{Env} = add([\![e_1]\!]_{Env}, [\![e_2]\!]_{Env})$	by D6
$[\![\textbf{int}]\!]_\tau$ = Integer	from Table 3.1
$[\![e_1]\!]_{Env} \in$ Integer and $[\![e_2]\!]_{Env} \in$ Integer	by hypotheses

Since add is a function that takes two Integer values and returns an Integer, $add([\![e_1]\!]_{Env}, [\![e_2]\!]_{Env}) \in$ Integer

i.e. $add([\![e_1]\!]_{Env}, [\![e_2]\!]_{Env}) \in [\![\textbf{int}]\!]_\tau$

i.e. $[\![e_1 + e_2]\!]_{Env} \in [\![\textbf{int}]\!]_\tau$

Expression	$\tau, \pi \vdash e_1 - e_2 : \textbf{int}$	
To be proved	$[\![e_1 - e_2]\!]_{Env} \in [\![\textbf{int}]\!]_\tau$	
Inductive Hypotheses	$[\![e_1]\!]_{Env} \in [\![\textbf{int}]\!]_\tau$ and $[\![e_2]\!]_{Env} \in [\![\textbf{int}]\!]_\tau$	
	since $\tau, \pi \vdash e_1 : \textbf{int}$ and $\tau, \pi \vdash e_2 : \textbf{int}$	
		by type rule [intSub]

Inductive Step

$[\![e_1 - e_2]\!]_{Env} = subtract([\![e_1]\!]_{Env}, [\![e_2]\!]_{Env})$	by D7
$[\![\textbf{int}]\!]_\tau$ = Integer	from Table 3.1
$[\![e_1]\!]_{Env} \in$ Integer and $[\![e_2]\!]_{Env} \in$ Integer	by hypotheses

Since subtract is a function that takes two Integer values and returns an Integer, $subtract([\![e_1]\!]_{Env}, [\![e_2]\!]_{Env}) \in$ Integer

i.e. $subtract([\![e_1]\!]_{Env}, [\![e_2]\!]_{Env}) \in [\![\textbf{int}]\!]_\tau$

i.e. $[\![e_1 - e_2]\!]_{Env} \in [\![\textbf{int}]\!]_\tau$

Expression	$\tau, \pi \vdash e_1 = e_2 : \textbf{bool}$	
To be proved	$[\![e_1 = e_2]\!]_{Env} \in [\![\textbf{bool}]\!]_\tau$	
Inductive Hypotheses	$[\![e_1]\!]_{Env} \in [\![\textbf{int}]\!]_\tau$ and $[\![e_2]\!]_{Env} \in [\![\textbf{int}]\!]_\tau$	
	since $\tau, \pi \vdash e_1 : \textbf{int}$ and $\tau, \pi \vdash e_2 : \textbf{int}$	
		by type rule [intEq]

Inductive Step

$[\![e_1 = e_2]\!]_{Env} = intEquals([\![e_1]\!]_{Env}, [\![e_2]\!]_{Env})$ by D8

$[\![\textbf{int}]\!]_\tau$ = Integer from Table 3.1

$[\![e_1]\!]_{Env} \in$ Integer and $[\![e_2]\!]_{Env} \in$ Integer by hypotheses

Since intEquals is a function that takes two Integer values and returns a Boolean, $intEquals([\![e_1]\!]_{Env}, [\![e_2]\!]_{Env}) \in$ Boolean

i.e. $intEquals([\![e_1]\!]_{Env}, [\![e_2]\!]_{Env}) \in [\![\textbf{bool}]\!]_\tau$

i.e. $[\![e_1 = e_2]\!]_{Env} \in [\![\textbf{bool}]\!]_\tau$

4.4.2 Booleans

Expression	$\tau, \pi \vdash \sim e$: **bool**	
To be proved	$[\![\sim e]\!]_{Env} \in [\![\textbf{bool}]\!]_\tau$	
Inductive Hypothesis	$[\![e]\!]_{Env} \in [\![\textbf{bool}]\!]_\tau$	
	since $\tau, \pi \vdash e$: **bool**	by type rule [negation]

Inductive Step

$[\![\sim e]\!]_{Env} = notOp([\![e]\!]_{Env})$ by D10

$[\![\textbf{bool}]\!]_\tau$ = Boolean from Table 3.1

$[\![e]\!]_{Env} \in$ Boolean by hypothesis

Since notOp is a function that takes a Boolean value and returns a Boolean, $notOp([\![e]\!]_{Env}) \in$ Boolean

i.e. $notOp([\![e]\!]_{Env}) \in [\![\textbf{bool}]\!]_\tau$

i.e. $[\![\sim e]\!]_{Env} \in [\![\textbf{bool}]\!]_\tau$

Expression	$\tau, \pi \vdash e_1$ **or** e_2 : **bool**	
To be proved	$[\![e_1 \textbf{ or } e_2]\!]_{Env} \in [\![\textbf{bool}]\!]_\tau$	
Inductive Hypotheses	$[\![e_1]\!]_{Env} \in [\![\textbf{bool}]\!]_\tau$ and $[\![e_2]\!]_{Env} \in [\![\textbf{bool}]\!]_\tau$	
	since $\tau, \pi \vdash e_1$: **bool** and $\tau, \pi \vdash e_2$: **bool**	by type rule [or]

Inductive Step

$[\![e_1 \textbf{ or } e_2]\!]_{Env} = orOp([\![e_1]\!]_{Env}, [\![e_2]\!]_{Env})$ by D11

$[\![\textbf{bool}]\!]_\tau$ = Boolean from Table 3.1

$[\![e_1]\!]_{Env} \in$ Boolean and $[\![e_2]\!]_{Env} \in$ Boolean by hypotheses

Since orOp is a function that takes two Boolean values and returns a Boolean, orOp($[\![e_1]\!]_{Env}$, $[\![e_2]\!]_{Env}$) ∈ Boolean

i.e. orOp($[\![e_1]\!]_{Env}$, $[\![e_2]\!]_{Env}$) ∈ $[\![$ **bool** $]\!]_\tau$

i.e. $[\![e_1$ **or** $e_2]\!]_{Env}$ ∈ $[\![$ **bool** $]\!]_\tau$

Expression	$\tau, \pi \vdash e_1$ **and** e_2 : **bool**
To be proved	$[\![e_1$ **and** $e_2]\!]_{Env} \in [\![$ **bool** $]\!]_\tau$
Inductive Hypotheses	$[\![e_1]\!]_{Env} \in [\![$ **bool** $]\!]_\tau$ and $[\![e_2]\!]_{Env} \in [\![$ **bool** $]\!]_\tau$ since $\tau, \pi \vdash e_1$: **bool** and $\tau, \pi \vdash e_2$: **bool** by type rule [and]

Inductive Step

$[\![e_1$ **and** $e_2]\!]_{Env}$ = andOp($[\![e_1]\!]_{Env}$, $[\![e_2]\!]_{Env}$)	by D12
$[\![$ **bool** $]\!]_\tau$ = Boolean	from Table 3.1
$[\![e_1]\!]_{Env}$ ∈ Boolean and $[\![e_2]\!]_{Env}$ ∈ Boolean	by hypotheses

Since andOp is a function that takes two Boolean values and returns a Boolean, andOp($[\![e_1]\!]_{Env}$, $[\![e_2]\!]_{Env}$) ∈ Boolean

i.e. andOp($[\![e_1]\!]_{Env}$, $[\![e_2]\!]_{Env}$) ∈ $[\![$ **bool** $]\!]_\tau$

i.e. $[\![e_1$ **and** $e_2]\!]_{Env}$ ∈ $[\![$ **bool** $]\!]_\tau$

Expression	$\tau, \pi \vdash e_1 = e_2$: **bool**
To be proved	$[\![e_1 = e_2]\!]_{Env} \in [\![$ **bool** $]\!]_\tau$
Inductive Hypotheses	$[\![e_1]\!]_{Env} \in [\![$ **bool** $]\!]_\tau$ and $[\![e_2]\!]_{Env} \in [\![$ **bool** $]\!]_\tau$ since $\tau, \pi \vdash e_1$: **bool** and $\tau, \pi \vdash e_2$: **bool** by type rule [boolEq]

Inductive Step

$[\![e_1 = e_2]\!]_{Env}$ = boolEquals($[\![e_1]\!]_{Env}$, $[\![e_2]\!]_{Env}$)	by D13
$[\![$ **bool** $]\!]_\tau$ = Boolean	from Table 3.1
$[\![e_1]\!]_{Env}$ ∈ Boolean and $[\![e_2]\!]_{Env}$ ∈ Boolean	by hypotheses

Since boolEquals is a function that takes two Boolean values and returns a Boolean,
boolEquals($[\![e_1]\!]_{Env}$, $[\![e_2]\!]_{Env}$) ∈ Boolean

i.e. boolEquals($[\![e_1]\!]_{Env}$, $[\![e_2]\!]_{Env}$) ∈ $[\![$ **bool** $]\!]_\tau$

i.e. $[\![e_1 = e_2]\!]_{Env}$ ∈ $[\![$ **bool** $]\!]_\tau$

4.4.3 Records

Expression $\tau, \pi \vdash \{ l_i = e_i \}_{(i = 1, n)} : \{ l_i : T_i \}_{(i = 1, n)}$

To be proved $[\![\{ l_i = e_i \}_{(i = 1, n)}]\!]_{Env} \in [\![\{ l_i : T_i \}_{(i = 1, n)}]\!]_\tau$

Inductive Hypotheses $[\![e_1]\!]_{Env} \in [\![T_1]\!]_\tau , \ldots , [\![e_n]\!]_{Env} \in [\![T_n]\!]_\tau$

since $\tau, \pi \vdash e_1 : T_1, \ldots \tau, \pi \vdash e_n : T_n$

by type rule [recValue]

Inductive Step

$[\![\{ l_i = e_i \}_{(i = 1, n)}]\!]_{Env} =$

$\text{mkRec}(< l_1 , [\![e_1]\!]_{Env} > ++ \ldots ++ < l_n , [\![e_n]\!]_{Env} >)$ by D14

Since mkRec is an operation that takes a list of n pairs each of type < Label, T_i > and returns a record with fields corresponding to the labels,

$\text{mkRec}(< l_1 , [\![e_1]\!]_{Env} > ++ \ldots ++ < l_n , [\![e_n]\!]_{Env} >) \in$

$\text{Record} (l_k : T_k)_{(k = 1, n)}$

i.e. $\text{mkRec}(< l_1 , [\![e_1]\!]_{Env} > ++ \ldots ++ < l_n , [\![e_n]\!]_{Env} >) \in$

$[\![\{ l_i : T_i \}_{(i = 1, n)}]\!]_\tau$

i.e. $[\![\{ l_i = e_i \}_{(i = 1, n)}]\!]_{Env} \in [\![\{ l_i : T_i \}_{(i = 1, n)}]\!]_\tau$

Expression $\tau, \pi \vdash e.l : T$

To be proved $[\![e.l]\!]_{Env} \in [\![T]\!]_\tau$

Inductive Hypothesis $[\![e]\!]_{Env} \in \text{Record}(l : T)^+$

since $\tau, \pi \vdash e : \{ l : T \}^+$ by type rule [recDeref]

Inductive Step

$[\![e.l]\!]_{Env} = \text{getL}([\![e]\!]_{Env})$ by D15

$[\![e]\!]_{Env} \in \text{Record}(l : T)^+$ by hypothesis

Since getL takes a record and returns a value of the type of field l of that record,

$\text{getL}([\![e]\!]_{Env}) \in [\![T]\!]_\tau$

i.e. $[\![e.l]\!]_{Env} \in [\![T]\!]_\tau$

4.4.4 Variants

Expression $\tau, \pi \vdash [l_i = e] : [l_i : T_i]^+$

To be proved $[\![[l_i = e] : [l_i : T_i]^+]\!]_{Env} \in [\![[l_i : T_i]^+]\!]_\tau$

Inductive Hypothesis $[\![e]\!]_{Env} \in [\![T_i]\!]_\tau$

since $\tau, \pi \vdash e : T_i$ by type rule [varValue]

Inductive Step

$[\![[l_i = e] : [l_i : T_i]^+]\!]_{Env} = mkPair (l_i, [\![e]\!]_{Env})$ by D16

$[\![e]\!]_{Env} \in [\![T_i]\!]_\tau$ by hypothesis

Since mkPair is a function that takes a label and an expression and returns a pair representing the variant value with the expression associated with the label,

$mkPair(l_i, [\![e]\!]_{Env}) \in Pair(label, [\![T_i]\!]_\tau)$

But $[\![[l_i : T_i]^+]\!]_\tau = Pair(label, [\![T_i]\!]_\tau)$ from table 3.2

i.e. $[\![[l_i = e] : [l_i : T_i]^+]\!]_{Env} \in [\![[l_i : T_i]^+]\!]_\tau$

Expression $\tau, \pi \vdash$ **project** e **as** x **onto** $l : T_1$ **in** e_1 **else** $e_2 : T$

To be proved $[\![$ **project** e **as** x **onto** $l : T_1$ **in** e_1 **else** $e_2]\!]_{Env} \in [\![T]\!]_\tau$

Inductive Hyp. $[\![e]\!]_{Env} \in [\![[l : T_1]^+]\!]_\tau$ and

$[\![e_1]\!]_{Env_{x = snd([\![e]\!]_{Env}) \in [\![T_1]\!]}} \in [\![T]\!]_\tau$ and $[\![e_2]\!]_{Env} \in [\![T]\!]_\tau$

since $\tau, \pi \vdash e : [l : T_1]^+$ and $\tau, \pi_1 :: x : T_1 :: \pi_2 \vdash e_1 : T$

and $\tau, \pi \vdash e_2 : T$ by type rule [varProj]

Inductive Step

$[\![$ **project** e **as** x **onto** l **in** e_1 **else** $e_2]\!]_{Env} =$
$if(fst([\![e]\!]_{Env}) = l, [\![e_1]\!]_{Env_{x = snd([\![e]\!]_{Env}) \in [\![T_1]\!]}}, [\![e_2]\!]_{Env})$ by D17

But $[\![e]\!]_{Env} \in [\![[l : T_1]^+]\!]_\tau$ and $[\![e_1]\!]_{Env} \in [\![T]\!]_\tau$ and $[\![e_2]\!]_{Env} \in [\![T]\!]_\tau$ by hypotheses

But $[\![[l : T_1]^+]\!]_\tau = Pair(label, [\![T_1]\!]_\tau)$

Since if is a function that takes a Boolean and two expressions of the same type and returns an expression,

$if(fst([\![e]\!]_{Env}) = l, [\![e_1]\!]_{Env_{x = snd([\![e]\!]_{Env}) \in [\![T_1]\!]}}, [\![e_2]\!]_{Env}) \in [\![T]\!]_\tau$

i.e. $[\![$ **project** e **as** x **onto** $l : T_1$ **in** e_1 **else** $e_2]\!]_{Env} \in [\![T]\!]_\tau$

4.4.5 Functions

Expression $\tau, \pi \vdash$ **fun**($x : T_1 \rightarrow T_2$) e : **fun**($T_1 \rightarrow T_2$)

To be proved $[\![$ **fun**($x : T_1 \rightarrow T_2$) e $]\!]_{Env} \in [\![$ **fun**($T_1 \rightarrow T_2$) $]\!]_\tau$

Inductive Hypothesis $[\![e]\!]_{Env_{x = v : T_1}} \in [\![T_2]\!]_\tau$

since $\tau, \pi_1::x : T_1::\pi_2 \vdash e : T_2$ by type rule [funValue]

Inductive Step

$[\![\textbf{fun} (x : T_1 \rightarrow T_2) e]\!]_{Env}$ = mkFun(e, x, Env) D18

$[\![e]\!] Env_{x = v : T_1} \in [\![T_2]\!]_\tau$ by hypothesis

Since mkFun is a function which takes an expression, an identifier and an environment and returns a Function and from the hypothesis it can be seen that when x is assigned a value of type T_1 in Env then e will be of type T_2,

mkFun(e, x, Env) $\in$ Function($T_1 \rightarrow T_2$)

$[\![\textbf{fun}(T_1 \rightarrow T_2)]\!]_\tau$ = Function($T_1 \rightarrow T_2$) from table 3.1

i.e. $[\![\textbf{fun}(x : T_1 \rightarrow T_2) e]\!]_{Env} \in [\![\textbf{fun}(T_1 \rightarrow T_2)]\!]_\tau$

Expression $\tau, \pi \vdash e_1(e_2) : T_2$

To be proved $[\![e_1(e_2)]\!]_{Env} \in [\![T_2]\!]_\tau$

Inductive Hypotheses $[\![e_1]\!]_{Env} \in [\![\textbf{fun}(T_1 \rightarrow T_2)]\!]_\tau$ and $[\![e_2]\!]_{Env} \in [\![T_1]\!]_\tau$

since $e_2 : T_1$ and $e_1 : \textbf{fun}(T_1 \rightarrow T_2)$ by type rule [funApp]

Inductive Step

$[\![e_1(e_2)]\!]_{Env}$ = apply($[\![e_1]\!]_{Env}$, $[\![e_2]\!]_{Env}$) by D19

$[\![e_1]\!]_{Env} \in [\![\textbf{fun}(T_1 \rightarrow T_2)]\!]_\tau$ and $[\![e_2]\!]_{Env} \in [\![T_1]\!]_\tau$ by hypotheses

Since apply is a function which takes a Function and an expression and returns a value of the result type of the function,

apply($[\![e_1]\!]_{Env}$, $[\![e_2]\!]_{Env}$) $\in [\![T_2]\!]_\tau$

i.e. $[\![e_1(e_2)]\!]_{Env} \in [\![T_2]\!]_\tau$

4.4.6 Identifiers

Expression $\tau, \pi \vdash x : T$

To be proved $[\![x]\!]_{Env} \in [\![T]\!]_\tau$

Inductive Hypothesis < none >

Inductive Step

$[\![x]\!]_{Env_{x = V \in T}} = V$ by D20

Since Env contains a binding that indicates V belongs to T, $[\![x]\!]_{Env} \in [\![T]\!]_\tau$

4.4.7 Locations

Expression $\tau, \pi \vdash \wedge e : \mathbf{loc}(T)$

To be proved $[\![\wedge e]\!]_{Env} \in [\![\mathbf{loc}(T)]\!]_\tau$

Inductive Hypothesis $[\![e]\!]_{Env} \in [\![T]\!]_\tau$

since $\tau, \pi \vdash e : T$ by type rule [locValue]

Inductive Step

$[\![\wedge e]\!]_{Env} = mkLoc([\![e]\!]_{Env})$ by D21

$[\![e]\!]_{Env} \in [\![T]\!]_\tau$ from hypothesis

Since mkLoc takes an expression of type T and returns a Location(T),

$mkLoc([\![e]\!]_{Env}) \in$ Location(T)

i.e. $[\![\wedge e]\!]_{Env} \in [\![\mathbf{loc}(T)]\!]_\tau$

Expression $\tau, \pi \vdash e_1 := e_2 : \mathbf{unit}$

To be proved $[\![e_1 := e_2]\!]_{Env} \in [\![\mathbf{unit}]\!]_\tau$

Inductive Hypotheses $[\![e_1]\!]_{Env} \in [\![\mathbf{loc}(T)]\!]_\tau$ and $[\![e_2]\!]_{Env} \in [\![T]\!]_\tau$

since $\tau, \pi \vdash e_1 : \mathbf{loc}(T)$ and $\tau, \pi \vdash e_2 : T$

by type rule [assign]

Inductive Step

$[\![e_1 := e_2]\!]_{Env} = put([\![e_1]\!]_{Env} , [\![e_2]\!]_{Env})$ by D22

$[\![e_1]\!]_{Env} \in [\![\mathbf{loc}(T)]\!]_\tau$ and $[\![e_2]\!]_{Env} \in [\![T]\!]_\tau$ from hypotheses

$[\![\mathbf{loc}(T)]\!]_\tau$ = Location (T) and $[\![\mathbf{unit}]\!]$ = Unit from Table 3.1

Since put takes a Location (T) and T and returns Unit,

$put([\![e_1]\!]_{Env} , [\![e_2]\!]_{Env}) \in$ Unit

i.e. $[\![e_1 := e_2]\!]_{Env} \in [\![\mathbf{unit}]\!]_\tau$

Expression $\tau, \pi \vdash @ e : T$

To be proved $[\![@ e]\!]_{Env} \in [\![T]\!]_\tau$

Inductive Hypotheses $[\![e]\!]_{Env} \in [\![\mathbf{loc}(T)]\!]_\tau$

since $\tau, \pi \vdash e : \mathbf{loc}(T)$ by type rule [locDeref]

Inductive Step

$[\![@ e]\!]_{Env} = get([\![e]\!]_{Env})$ by D23

$[\![e]\!]_{Env} \in [\![\mathbf{loc}(T)]\!]_\tau$ from hypothesis

$[\![\textbf{loc}(T)]\!]_\tau$ = Location (T) from Table 3.1

Since get is a function that takes a Location (T) and returns a T,

get($[\![e]\!]_{Env}$) $\in [\![T]\!]_\tau$

i.e. $[\![@ e]\!]_{Env} \in [\![T]\!]_\tau$

4.4.8 Infinite Union

Expression	$\tau, \pi \vdash$ **inject**(e, T) : **any**	
To be proved	$[\![$ **inject**(e, T) $]\!]_{Env} \in [\![$ **any** $]\!]_\tau$	
Inductive Hypothesis	$[\![e]\!]_{Env} \in [\![T]\!]_\tau$	
	since $\tau, \pi \vdash e : T$	by type rule [anyInj]

Inductive Step

$[\![$ **inject**(e, T) $]\!]_{Env}$ = mkPair(getTypeRep(T), $[\![e]\!]_{Env}$) by D24

$[\![e]\!]_{Env} \in [\![T]\!]_\tau$ by hypothesis

$[\![$ **any** $]\!]$ = Pair(typeRep, $[\![T]\!]_\tau$) from Table 3.2

Since mkPair takes a typeRep and an expression and returns a Pair of the two,

mkPair(getTypeRep(T), $[\![e]\!]_{Env}$) $\in$ Pair(typeRep, $[\![T]\!]_\tau$)

i.e. $[\![$ **inject**(e, T) $]\!]_{Env} \in [\![$ **any** $]\!]_\tau$

Expression	$\tau, \pi \vdash$ **project** e **as** x **onto** T_1 **in** e_1 **else** $e_2 : T$	
To be proved	$[\![$ **project** e **as** x **onto** T_1 **in** e_1 **else** $e_2]\!]_{Env} \in [\![T]\!]_\tau$	
Inductive Hyp.	$[\![e]\!]_{Env} \in [\![$ **any** $]\!]_\tau$ and	
	$[\![e_1]\!]_{Env_{x = snd([\![e]\!]_{Env}) \in [\![T1]\!]}} \in [\![T]\!]_\tau$ and $[\![e_2]\!]_{Env} \in [\![T]\!]_\tau$	
	since $\tau, \pi \vdash e :$ **any** and $\tau, \pi_1 :: x : T_1 :: \pi_2 \vdash e_1 : T$	
	and $\tau, \pi \vdash e_2 : T$	by type rule [anyProj]

Inductive Step

$[\![$ **project** e **as** x **onto** T_1 **in** e_1 **else** $e_2]\!]_{Env}$ =

if(fst($[\![e]\!]_{Env}$) = getTypeRep(T), $[\![e_1]\!]_{Env_{x = snd([\![e]\!]_{Env}) \in [\![T]\!]}}$, $[\![e_2]\!]_{Env}$)

by D25

But $[\![e]\!]_{Env} \in [\![$ **any** $]\!]_\tau$ by hypotheses

But $[\![$ **any** $]\!]_\tau$ = Pair(typeRep, $[\![T_i]\!]_\tau$)

Since if is a function that takes a Boolean and two expressions of the same type and returns an expression,

if(fst(⟦ e ⟧$_{Env}$) = getTypeRep(T), ⟦ e_1 ⟧$_{Env_{x = snd(⟦ e ⟧_{Env})} ∈ ⟦ T ⟧}$, ⟦ e_2 ⟧$_{Env}$)

∈ ⟦ T ⟧$_\tau$

i.e. ⟦ **project** e **as** x **onto** l : T_1 **in** e_1 **else** e_2 ⟧$_{Env}$ ∈ ⟦ T ⟧$_\tau$

4.4.9 Sequence

Expression	$\tau, \pi \vdash e_1 ; e_2 : T$
To be proved	⟦ $e_1 ; e_2$ ⟧$_{Env}$ ∈ ⟦ T ⟧$_\tau$
Inductive Hypotheses	⟦ e_1 ⟧$_{Env}$ ∈ ⟦ **unit** ⟧$_\tau$ and ⟦ e_2 ⟧$_{Env}$ ∈ ⟦ T ⟧$_\tau$
	$\tau, \pi \vdash e_1$: **unit** and $\tau, \pi \vdash e_2 : T$ by type rule [seq]

Inductive Step

⟦ $e_1 ; e_2$ ⟧$_{Env}$ = ⟦ e_2 ⟧$_{Env_{e_1}}$ by D26

⟦ e_2 ⟧$_{Env}$ ∈ ⟦ T ⟧$_\tau$ by hypotheses

Therefore, ⟦ e_2 ⟧$_{Env_{e_1}}$ ∈ ⟦ T ⟧$_\tau$

i.e. ⟦ $e_1 ; e_2$ ⟧$_{Env}$ ∈ ⟦ T ⟧$_\tau$

4.4.10 Block

Expression	$\tau, \pi \vdash$ **begin** D ; e **end** : T
To be proved	⟦ **begin** D ; e **end** ⟧$_{Env}$ ∈ ⟦ T ⟧$_\tau$
Inductive Hypotheses	⟦ D ⟧$_{Env}$ ∈ ⟦ **unit** ⟧ and ⟦ e ⟧ Env_D ∈ ⟦ T ⟧$_\tau$
	since $\tau, \pi \vdash$ D : **unit** and $\tau, \pi \vdash$ e : T by type rule [block]

Inductive Step

⟦ **begin** D ; e **end** ⟧$_{Env}$ = ⟦ e ⟧$_{Env_D}$ by D27

But ⟦ e ⟧$_{Env_D}$ ∈ ⟦ T ⟧$_\tau$ by hypotheses

Therefore, ⟦ **begin** D ; e **end** ⟧$_{Env}$ ∈ ⟦ T ⟧$_\tau$

4.4.11 Conditional

Expression	$\tau, \pi \vdash$ **if** e **then** e_1 **else** e_2 : T
To be proved	⟦ **if** e **then** e_1 **else** e_2 ⟧$_{Env}$ ∈ ⟦ T ⟧$_\tau$

Inductive Hyp. $[\![e]\!]_{Env} \in [\![\textbf{bool}]\!]_\tau$, $[\![e_1]\!]_{Env} \in [\![T]\!]_\tau$ and $[\![e_2]\!]_{Env} \in [\![T]\!]_\tau$
since $\tau, \pi \vdash e : \textbf{bool}$, $\tau, \pi \vdash e_1 : T$ and $\tau, \pi \vdash e_2 : T$
by type rule [conditional]

Inductive Step

$[\![\textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2]\!]_{Env} = \text{if}([\![e]\!]_{Env}, [\![e_1]\!]_{Env}, [\![e_2]\!]_{Env})$ by D28

But $[\![e]\!]_{Env} \in [\![\textbf{bool}]\!]_\tau$, $[\![e_1]\!]_{Env} \in [\![T]\!]_\tau$ and $[\![e_2]\!]_{Env} \in [\![T]\!]_\tau$
by hypotheses

Since if is a function that takes a Boolean and two expressions of the same type and returns an expression of this type,

$\text{if}([\![e]\!]_{Env}, [\![e_1]\!]_{Env}, [\![e_2]\!]_{Env}) \in [\![T]\!]_\tau$

Therefore, $[\![\textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2]\!]_{Env} \in [\![T]\!]_\tau$

4.5 Summary

The definition of soundness states that if an expression e is of type T then the meaning of e belongs to the meaning of T. Structural induction has been used to prove this property for every construct in Base, thus proving the soundness of the whole type system. The proof strategy for any construct c of Base can be summarised as follows. The type rules of Base specify the expected type of c. They also specify the hypotheses which need to hold for this typing. The semantics (the meaning) of c is defined in terms of the meta-operations of Base. The type of the meaning of c is determined from the type of the meta-operations and the hypotheses. The meanings of various types in Base have been defined in section 3.1. If the meaning of the expected type of c is the same of the type of the meaning of c then the typing of c is sound. This strategy is again used in Chapter 9 to prove that the addition of extension polymorphism preserves the soundness of the type system of Base.

5 Background

The aim of this thesis is to address the problem of type evolution in persistent systems using polymorphism. Some of the concepts and issues that need to be understood in order to carry out this work are explained in this chapter. The salient features of polymorphism, type checking and object oriented programming are explained in sections 5.1 to 5.3. The relevance of each topic to the main work of this thesis is outlined at the end of each section.

5.1 Polymorphic Systems

Static typing in programming languages provides the ability of determining the type of every expression by static program analysis. While there are advantages to this mechanism, such as being able to detect type errors early and efficient program execution, it also demands that every variable and expression should be bound to a type at compile time. In a monomorphic type system this can sometimes be too restrictive and lead to loss of expressive power and flexibility. Polymorphism is one method by which a programming language can preserve static typing while easing some of its traditional restrictions.

A polymorphic type system is one in which values and variables can have more than one type. This can be contrasted with the more traditional monomorphic systems in which each value belongs to at most one type. Two of the most widely used kinds of polymorphism are parametric polymorphism and inclusion polymorphism. They are described in detail in later sections.

The correspondence between sets and types is first discussed in this section. A brief outline of the theory of partially ordered sets [Lew85, FS91] and lattices [Lew85] is then given as lattices are later used to illustrate type hierarchies in polymorphic type systems. The formal model, the concept and a programming language mechanism to implement the concept are described for both parametric and inclusion polymorphism. System F provides the formal basis for parametric polymorphism and system $F_{\leq}$ for inclusion polymorphism. While parametric polymorphism may be incorporated into a language by the use of universal quantification, subsumption and bounded quantification are mechanisms that support inclusion polymorphism.

5.1.1 Sets and Types

Polymorphism is described by Cardelli and Wegner [CW85] in terms of set theory. They assert that there is a universe of values, V, which includes integers,

cross products and functions. To a first approximation and for all programming languages, a type is a set of values of V. Values that belong to any type form a subset of V. If a value v has a type t then it is a member of the subset of V corresponding to t. Sets may overlap and values may have more than one type. This flexibility gives rise to polymorphic type systems. Parametric polymorphism can be modelled in terms of set intersection while inclusion polymorphism is modelled by set inclusion.

5.1.2 Partial Orders and Lattices

Consider a relation $p \in A \times A$ on a set A. p is said be a partial order on A if it has the following properties:

- reflexive

  if $(a, a) \in p$ for every $a \in A$

- anti-symmetric

  if $(a_1, a_2), (a_2, a_1) \in p$ implies $a_1 = a_2$ for $a_1, a_2 \in A$

- transitive

  if $(a_1, a_2), (a_2, a_3) \in p$ implies $(a_1, a_3) \in p$

The pair (A, p) is then referred to as a partially ordered set or a poset. A strict partial order $<$ on A can be defined as satisfying the following properties:

- irreflexive

  if $(a, a) \notin p$ for every $a \in A$

- asymmetric

  if $(a_1, a_2) \in p$ implies $(a_2, a_1) \notin p$

- transitive

  as defined before

In the context of a poset (A, $\leq$), and given that $<$ is the associated strict partial order of p, the following terms can be defined:

b is a successor of a in A if $a < b$ and is an immediate successor of a if there does not exist a c in A such that $a < c < b$ for $a, b \in A$. The inverse of these relations defines the predecessor and immediate predecessor relations. A member a_0 in A

is the least member of A if it is a predecessor of every other member of A. A member a_0 in A is a minimal of A if it has no predecessor. A poset always has one or more minimals but has a least member if and only if it is also the unique minimal. The inverse of these conditions define the greatest member and the maximals of a poset.

A member a_0 in A is a lower bound of a subset E of A iff $a_0 \leq x$ for every x in E. If the set of lower bounds of E has a greatest member then this member is called the infimum or the greatest lower bound of E. Similarly inverse conditions define the upper bounds and the supremum or the lowest upper bound. The bounds of E need not be members of E.

A poset (S, ≤) is a lattice if and only if every subset consisting of two members of S has an infimum and a supremum in S.

Consider set inclusion ($\subseteq$) over some universal set U. The subset relation is reflexive, anti-symmetric and transitive for any pair of sets A and B belonging to 2^U i.e. the set of all subsets of U [Lew85]. Therefore it is a partial order.

For the subset relation and any two members a and b in 2^U, the infimum is the intersection of a and b and the supremum is the union of a and b. Since both these sets belong to 2^U, the subset relation is a lattice. The diagrammatic notation shown below is used to represent lattices.

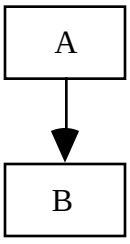

Figure 5.1.1 : Subset Lattice

The lattice in Figure 5.1.1 represents the fact that set B is a subset of set A.

5.1.3 System F

System F is an extension of the typed lambda calculus [Sch94] that provides the basis for polymorphic programming languages. It was introduced independently in the contexts of proof theory [Gir71, GTL89] and programming languages [Rey74]. In addition to the functionality provided by typed lambda calculus, System F permits the binding of type variables.

In the case of the typed lambda calculus, the type of every expression can be deduced from the types of the free variables it contains. Whenever a variable is bound, a type expression is also specified for it. For example, the identity function for values of type t can be written as λx : t. x. The meaning of such expressions depend on both the free variables and the free type variables.

System F provides the facility to specify bindings for type variables to create polymorphic functions. These are functions from types to values. For example, Λt. λx : t. x is the polymorphic identity function which takes a type t and returns the identity function for values of type t. The application of polymorphic functions to type expressions can be specified as (Λt. r)[w] where r is an expression and w is a type expression. This application can be reduced to an expression obtained from r by replacing every free occurrence of t by w, after any necessary alpha-conversion to avoid clashing of variable names. The type of a polymorphic function, which when applied to a type t produces a value of type w, is Λt. w. Therefore if expression r is of type w then the expression Λt. r is of type Λt. w. Parametric polymorphism is based on this model.

5.1.3.1 Parametric Polymorphism

In the case of parametric polymorphism the uniformity of type structure necessary for universal polymorphism is achieved by the use of type parameters. Implicit or explicit type parameters determine the type of argument for each application of a polymorphic function.

Consider defining an identity function each for integers, booleans and a record type called *Person.*

```
fun ( x : int -> int ) x

fun ( x : bool -> bool ) x

fun ( x : Person -> Person ) x
```

Figure 5.1.2 : Identity Functions

As can be seen from Figure 5.1.2, the signatures of all these functions have a similar structure and the function bodies are identical. It is easy to see that whatever the type used to define the identity function, these statements will hold. Therefore a polymorphic function, parameterised by the desired type, can be defined to replace all of these functions.

```
let genId = fun [ T ] ( x : T -> T ) x

intIdVal := genId [ int ] ( 100 )

let boolId = genId [ bool ]
```

Figure 5.1.3 : Polymorphic Identity Function

Figure 5.1.3 shows one model for defining and using a polymorphic identity function *genId*. This example uses universal quantification to implement parametric polymorphism. *intIdVal* is assigned the value returned by *genId* when it is parameterised by the type ***int*** and called with the value 100. In this case the type of *intIdVal* will be ***int***, as might be expected. In some type systems it is also possible to obtain a specialised function applicable to specific type by merely applying the polymorphic function to a type parameter. Napier88 [MCC+95, MBC+96] is an example of a language that provides this facility. *boolId* in Figure 5.1.3 is an instance of *genId* specialised to operate over boolean values. Its functionality will, in effect, be the same as that of the second function in Figure 5.1.2. Thus, polymorphic functions abstract over the argument types of functions.

In set theoretic terms, parametric polymorphism is modelled by set intersection. A polymorphic function can be considered as the intersection of all the monomorphic functions it can represent. Thus, using the examples in Figure 5.1.2, a type lattice can be drawn as shown in Figure 5.1.4 below.

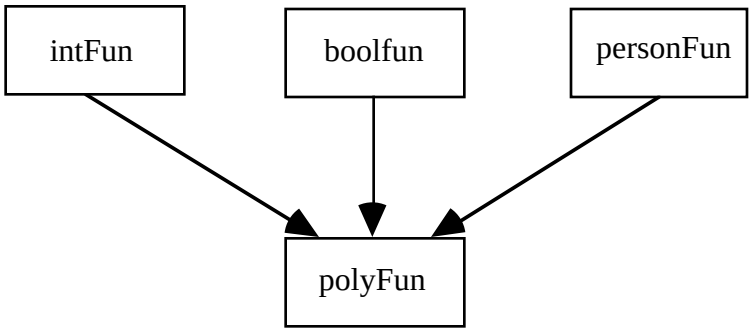

Figure 5.1.4 : Type Lattice for Parametric Polymorphism

In this lattice, *intFun*, *boolFun* and *personFun* stand for the types of the three functions defined in Figure 5.1.2 and *polyFun* is the type of the polymorphic identity function.

5.1.3.2 Quantification

Quantification [CW85, CM92] is a abstraction mechanism for polymorphism which allows the programmer to specify a type variable to range over the possible types which can be type parameters to a quantified function. There are two kinds of quantification mechanisms: universal and existential. Universal quantification, which is the model of interest to this thesis, has been used for the examples in Figure 5.1.3. It is a model of parametric polymorphism and allows the specification of generic code. This is usually achieved through parameter passing.

5.1.4 System $F_{\leq}$

System $F_{\leq}$ [Ghe90, Ghe93] is an extension of System F which provides the ability to express polymorphism over subtyping. In addition to lambda abstraction, System $F_{\leq}$ allows second order abstraction with respect to type variables and thus can model bounded quantification. A polymorphic function can have a bound type for the type variable and only the subtypes of this bound can be actual parameters to the polymorphic function. For example, $\forall t \leq T.\ r$ is such a function where T is the bound and r is an expression. The type of this function is $\Lambda t \leq T.\ w$ where w is the type of r and may be defined in terms of t. The type system of $F_{\leq}$ contains a type Top, whose canonical element is top, which is a supertype of all types. Thus, an unbounded lambda abstraction r may also be written as $\forall t \leq Top.\ r$. The application of a second order function $\forall t \leq T.\ r$ to a type A is specified as $\forall t \leq T.\ r\{ A \}$. Ghelli has extended this language further in System F-bounded [Ghe94] where it is possible to write bound types that contain the bounded variable.

It should be noted that there are differences in the notations used by System F and System $F_{\leq}$ to denote similar semantic entities. For example, System F uses Λ to stand for the construct that specifies type binding while System $F_{\leq}$ uses $\forall$. Thus, $\forall t.\ r$ in System $F_{\leq}$ will be written as $\Lambda t.\ r$ in System F.

5.1.4.1 Inclusion Polymorphism

Inclusion polymorphism is a combination of two concepts : a subtype relation and a programming language algebra that supports the subtyping relation. The subtyping relation between two types can be defined as follows : type A is a subtype of type B if all operations allowed on B are also allowed on A. This is written as $A \leq B$. A subtype has all the functionality of the supertype and possibly more. The programming algebra is the collection of constructs

provided by the language to implement the concepts it supports. One programming algebra rule that can be used to support subtyping is subsumption. The rule of subsumption can be written as :

$$\pi \vdash a : X \text{ and } X \leq Y \text{ implies } \pi \vdash a : Y$$

where π is the identifier - type environment introduced in Chapter 2, a is an expression and X and Y are types. The colon is used to denote 'is of type'. $\pi \vdash a : X$ means that from π a can be deduced to be of type X. Therefore the rule of subsumption can be stated as 'if a is of type X and X is a subtype of Y then a is also of type Y'. This allows a subtype value to be used wherever a supertype value is expected.

Since the subtyping relation is equivalent to the subset relation over the same universe of discourse, subtyping is also a partial order and a lattice. At the top of this lattice is the set representing the universe of values V and at the bottom, the empty set. Since types are sets, subtypes correspond to subsets and the notion that A is a subtype of B in the type space corresponds to the set theoretical condition that A is a subset of B in the value space. In the subsequent sections both lattices and Venn diagrams are used to illustrate subtype hierarchies.

This concept is discussed further in the next section using Cardelli's subtyping relation [Car84] as an example. Sections 5.1.4.3 and 5.1.4.5 present a discussion of subsumption and bounded universal quantification as programming language mechanisms to support the use of the subtyping relation.

5.1.4.2 Subsets, Subtypes and Type Constructors

In an operation based type system sets are formed over the value space defined by the operations that can be performed on the values. If an operation can manipulate values of a set, values of any its subsets can also be acted upon by that operation. Cardelli established a model of subtyping relation based on the notion of types as sets of values which assumes subsumption in the programming language algebra. Cardelli's subtyping relation is defined below.

The correspondence between subtype relation in type space and subset relation in value space is justified in this section by means of examples. The characterisation of type here is in terms of two criteria : applicability of operations and the set of possible results of these operations. Consider a type T with operations O_1 to O_n such that all possible results of these operations form

the sets R_1 to R_n respectively. If there is a value v such that O_1 to O_n can be applied to v and $O_i(v)$ is a member of R_i for all i from 1 to n, then v can be considered to be of type T. The subtyping relation is discussed for each of the common type constructors.

a) All types

$T \leq T$ for any type T

Any type is a subtype of itself. Similarly any set is a subset of itself.

b) Labelled cross products

A labelled cross product is a collection of values which have names or labels. The subtyping rule for labelled cross products is :

$t_1 \leq t_1', \ldots, t_n \leq t_n'$ implies $\{ a_1 : t_1, \ldots, a_{n+m} : t_{n+m}\} \leq \{ a_1 : t_1', \ldots, a_n : t_n'\}$
where $m \in \mathbb{N}$ and $n \in \mathbb{N}_1$

A labelled cross product S1 is a subtype of another labelled cross product S2 if S1 has all the fields of S2 and possibly more and the types of common fields in S1 are in turn subtypes of the those in S2. Figure 5.1.15 gives some examples of labelled cross product definitions illustrating the subtype relation.

```
type person is { name : format1 }

type teacher is { name : format2 , exp : int }

type student is { name : format2 , rollNo : int }

type tutor is { name : format2 , exp : int , rollNo : int }
```

Figure 5.1.5 : Examples of Labelled Cross Products

where *format2* ≤ *format1.* The relations that hold between the types in Figure 5.1.5 by virtue of the subtyping rule are specified along with explanation for each case in the table in Figure 5.1.6 below.

Relation	Explanation
teacher ≤ person	format2 ≤ format1 and teacher has an additional label exp
student ≤ person	format2 ≤ format1 and student has an additional label rollNo
tutor ≤ person	format2 ≤ format1 and tutor has two additional labels exp and rollNo
tutor ≤ teacher	tutor has an additional label rollNo
tutor ≤ student	tutor has an additional label exp

Figure 5.1.6 : Subtyping Relations for Examples

The type lattice for these types can be represented by the following diagram in Figure 5.1.7.

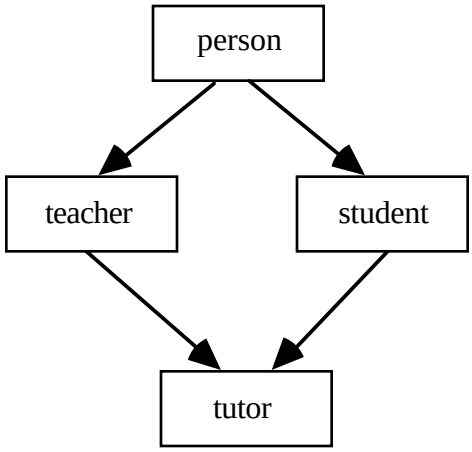

Figure 5.1.7 : Type Lattice for Example Types

The relations in terms of set theory can be illustrated by the Venn diagram below in Figure 5.1.8.

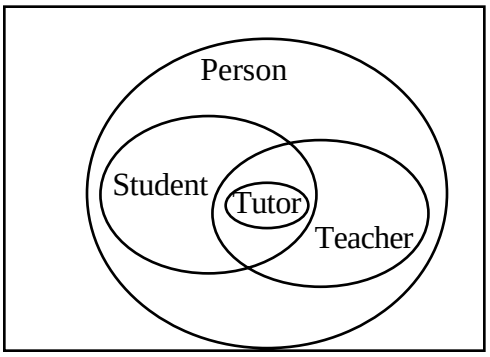

Figure 5.1.8 : Subset Relations

c) Labelled disjoint sums

A labelled disjoint sum represents a choice between a finite, named collection of types. The subtyping rule for labelled disjoint sums is :

$t_1 \leq t_1', \ldots, t_n \leq t_n'$ implies $[\ a_1 : t_1, \ldots, a_n : t_n\ ] \leq [\ a_1 : t_1', \ldots, a_{n+m} : t_{n+m}']$

A labelled disjoint sum V_1 is a subtype of another labelled disjoint sum V_2, if V_2 has all the branches of V_1 and possibly more and the types of the branches in V_1 are in turn subtypes of the corresponding branches in V_2. Given below in Figure 5.1.9 are examples of definitions of labelled disjoint sums where *higherIntermediate* ≤ *intermediate.*

type generalScale **is** [S : senior , I : intermediate ; J : junior]

type twoScale **is** [S : senior , I : intermediate]

type highScale **is** [S : senior , I : higherIntermediate , J : junior]

type seniorScale **is** [S : senior]

Figure 5.1.9 : Examples of Labelled Disjoint Sums

According to the subtyping rule given above, the following relations explained Figure 5.1.10 hold between these types :

Relation	Explanation
twoScale ≤ generalScale	generalScale has an additional branch J
highScale ≤ generalScale	higherIntermediate ≤ intermediate
seniorScale ≤ generalScale	generalScale has two additional branches I and J
seniorScale ≤ twoScale	twoScale has an additional branch I
seniorScale ≤ highScale	highScale has two additional branches I and J

Figure 5.1.10 : Relations between Labelled Disjoint Sums

The type lattice for these labelled disjoint sum definitions can be drawn as shown in Figure 5.1.11.

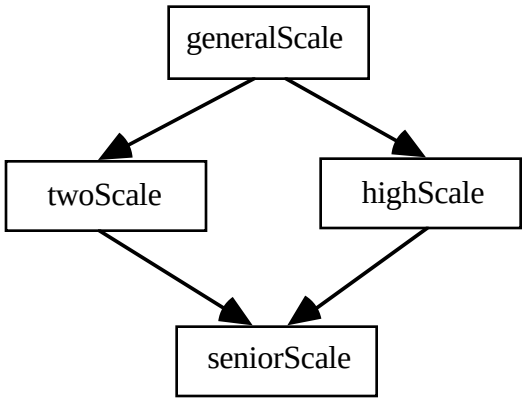

Figure 5.1.11 : Type Lattice for Disjoint Sum Examples

The subset relations between these types are illustrated by the Venn diagram in Figure 5.1.12 below where sS stands for seniorScale.

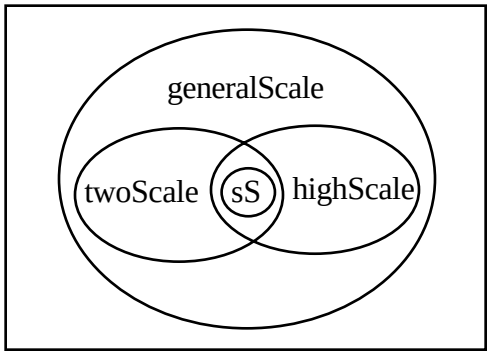

Figure 5.1.12 : Venn Diagram for Disjoint Sums

d) Functions

Functions are represented by an arrow operator requiring a domain and a range type. The subtyping rule for functions is :

$s' \leq s$ and $t \leq t'$ implies $s \rightarrow t \leq s' \rightarrow t'$

Thus this rule requires the argument type of the subtype to be greater and the result type of the subtype to be smaller than the corresponding types of the supertype. This condition is said to be contravariant on the argument type and covariant on the result type.

Assuming the definition of labelled cross product types employee, person, car and vehicle such that employee ≤ person and car ≤ vehicle, function types can be defined as follows in Figure 5.1.13.

type vehicleOfPerson **is fun** (person → vehicle)

type carOfPerson **is fun** (person → car)

type carOfEmployee **is fun** (employee → car)

type vehicleOfEmployee **is fun** (employee → vehicle)

Figure 5.1.13 : Examples of Functions

According to the subtyping rule given above, the following relations specified in Figure 5.1.14 hold :

Relation	Explanation
vehicleOfPerson ≤ vehicleOfEmployee	employee ≤ person
carOfPerson ≤ vehicleOfPerson	car ≤ vehicle
carOfPerson ≤ carOfEmployee	employee ≤ person
carOfPerson ≤ vehicleOfEmployee	employee ≤ person and car ≤ vehicle
carOfEmployee ≤ vehicleOfEmployee	car ≤ vehicle

Figure 5.1.14 : Relations between Functions

The type lattice for these function types can be drawn as shown in Figure 5.1.15 below.

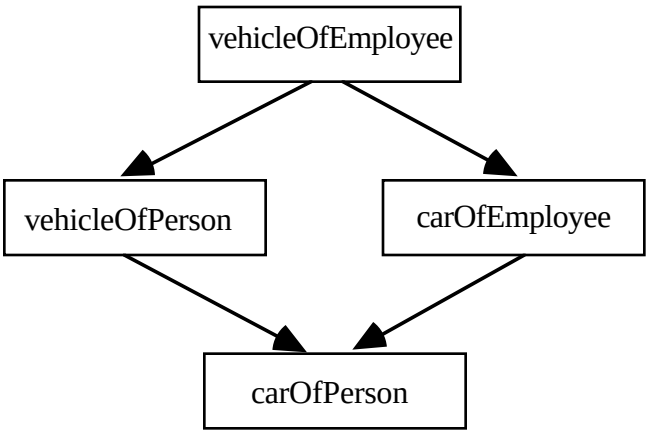

Figure 5.1.15 : Type Lattice for Function Examples

The subtype relations between these function types are illustrated by the Venn diagram given in Figure 5.1.16. In the diagram, F1, F2, F3 and F4 stand for vehicleOfPerson, carOfPerson, carOfEmployee and vehicleOfEmployee respectively.

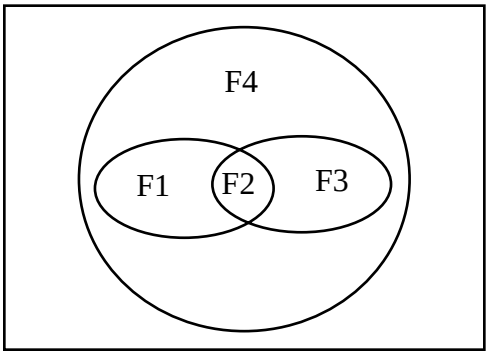

Figure 5.1.16 : Venn Diagram for Functions

e) Mutable values

The subtyping rule for mutable values is :

a) loc(A) ≤ loc(B) iff A ≤ B and B ≤ A

Since subtyping is a partial order satisfying the anti-symmetric condition, these conditions imply that A = B.

It is also possible to incorporate another rule for subtyping over mutable though it is not part of Cardelli's model and is not compatible with the corresponding set theory. However, by adopting suitable operations over mutable values, the soundness of the type system can be preserved despite the addition of the following rule:

b) loc(A) ≤ A

In Figure 5.1.17 given below, *locPer* and *locEmp* are examples of mutable types.

```
type person is { name : string }

type employee is { name : string ; eno : int }

type locPer is loc( person )

type locEmp is loc( employee )
```

Figure 5.1.17 : Mutable Types

According to the subtyping rules for locations, the subtyping relations between these types are given in Figure 5.1.18.

Relation	Explanation
locPer ≤ person	by rule b
locEmp ≤ employee	by rule b

Figure 5.1.18 : Subtyping Relation between Mutable Types

The type lattice for the location types in the example can be drawn as shown in Figure 5.1.19.

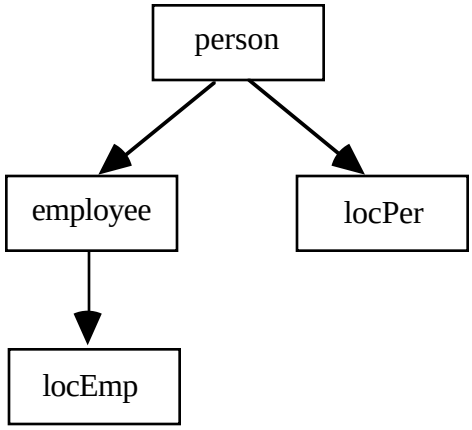

Figure 5.1.19 : Type Lattice for Mutable Type Examples

The subtype relations between the example mutable types can be illustrated by the following Venn diagram in Figure 5.1.20.

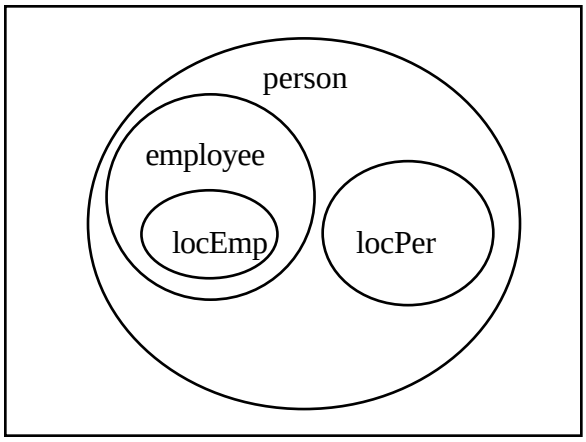

Figure 5.1.20 : Venn Diagram for Mutable Types

5.1.4.3 Subsumption

Subsumption is one programming language mechanism which enables the use of the subtyping relation in programs. It can be defined by the following rule.

$\pi \vdash a : t$ and $t \leq t'$ implies $\pi \vdash a : t'$

This means that if a value a is of type t and if t is a subtype (by the rules of the subtyping relation adopted) of another type t' then a is also of type t'.

If the subtyping relation described in section 5.1.3 were added to the type system of Base, described in Chapter 2, then subsumption can be used in the following contexts.

a) assignment

$\pi \vdash$ a : **loc**(X) , $\pi \vdash$ b : Y and Y ≤ X implies a := b is valid

If a belongs to the location type of type X, b is a value of type Y and Y is a subtype of X then it is type safe to assign b to a and b will be viewed as being of type X.

b) function application and result

$\pi \vdash$ f : s → t and $\pi \vdash$ a : s' where s' ≤ s implies
f(a) is meaningful and $\pi \vdash$ f(a) : t

If f is a function type from s to t and expression a is of type s' where s' is a subtype of s then f can be applied to a and the result of this application will be of type t.

c) projection from infinite union

$\pi \vdash$ s : X' , X' ≤ X and t = **inject**(s, X') then the projection of t onto X will be successful

If s is an expression of type X', X' is a subtype of X and if t is the result when s has been injected to form an **any** then the project operation on t, ***project*** *t* ***as*** *y* ***onto*** *X* ***in*** *E1* ***else*** *E2*, is valid and the expression corresponding to type X will be chosen.

5.1.4.4 Anomalies of Subsumption

While subsumption allows more flexibility of operations than is possible in a system without a similar mechanism for polymorphism, there is a conflict between the use of subsumption and type accuracy in a system [CM92]. This is due to the fact that using subsumption reduces the amount of static information available about the relationship between types. Consider the example in Figure 5.1.21 below.

```
let compare = fun( a, b : person -> bool ) . . .

let personId = fun( p : person -> person ) p
let s = personId( aStudent )
```

Figure 5.1.21 : Example of Subsumption

If the type system supports subsumption then it can not be assumed in the body of *compare* that *a* and *b* are of exactly the same type since either or both of them may belong to a type that is a subtype of *person*. A similar problem occurs with the definition of *personId* in the example. This function is defined to take a value of type *person* and return a result of type *person*. However subsumption allows the user to pass a value of type *student* (where *student* is a subtype of *person*) as the actual parameter to the function but the result returned will still be typed as *person*. Although these operations are type safe, some relevant semantic knowledge may be hidden from the user. A possible solution to this problem is described in section 5.1.5.

5.1.4.5 Bounded Quantification

One way to overcome the restrictions placed by the use of subsumption, explained in the previous section, is to replace subsumption with a mechanism that allows similar flexibility in the language but models the relationships between types explicitly. For example, given two types A and B, it should be possible to determine statically whether they are equivalent, in a subtype relation or whether one of them is a component of the other. Bounded quantification [CW85, Ghe90, CM92] is a mechanism that provides more type information with subtyping.

In a system that supports bounded quantification without subsumption, subtyping is only permitted where explicitly specified; all other instances require an exact type match. Bounded quantification differs from the quantification mechanisms discussed in sections 5.1.2.3 to 5.1.2.6 in that a bound by subtyping is placed on the quantifier variable. Bounded universal quantification which is used later in the thesis is described below.

Bounded universal quantification provides a means to produce polymorphic code which can operate over all types which are subtypes of the bound type specified. The identity function *personId* from Figure 5.1.19 can now be

defined as shown in Figure 5.1.22. In this form, *t* ranges over all subtypes of *person*.

```
let personId = fun[ t ≤ person ]( p : t -> t ) p

let s = personId[ student ]( aStudent )

let compare = fun[ t ≤ person ](p1, p2 : t -> bool ) . . .
```

Figure 5.1.22 - Bounded Universal Quantification

This definition of the function, while still providing polymorphism, also allows the actual parameter and the result to be statically and exactly typed. Thus if *personId* was quantified by the type *student* and applied to a value of *student* as shown in Figure 5.1.20 the type of *s* can be statically deduced to be *student*. The use of bounded quantification also means that in the definition of function *compare*, parameters *p1* and *p2* can safely be assumed to be of the same type.

5.1.5 Summary

Sections 5.1.1 to 5.1.4 have presented the different kinds of polymorphism and their expressive power and uses have also been described. It can be seen that polymorphism is a means to provide various kinds of abstraction, software reuse and information at the type level. This idea is pursued further in Chapter 8 to use polymorphism to devise a way to capture type evolution in persistent systems.

5.2 Type Checking

The main function of a type checker is to check that a program is well-typed by comparing the expected type of an expression, deduced from its context, with the actual type. It determines, for example, whether a function has been supplied with the right type of actual parameter and whether a location has been assigned the right type of expression. For a complete piece of code to be correctly typed, every expression and statement in it must be correctly typed. In some systems such as Napier88 [Con88], the type checker also constructs the data structures that represent types in the persistent store. Structural type equivalence [CBC+90] checking is implemented by performing an equivalence test on such structures.

In the traditional compiler architecture, shown below in Figure 5.2.1, type checking is a separate phase that takes place between parsing and code generation.

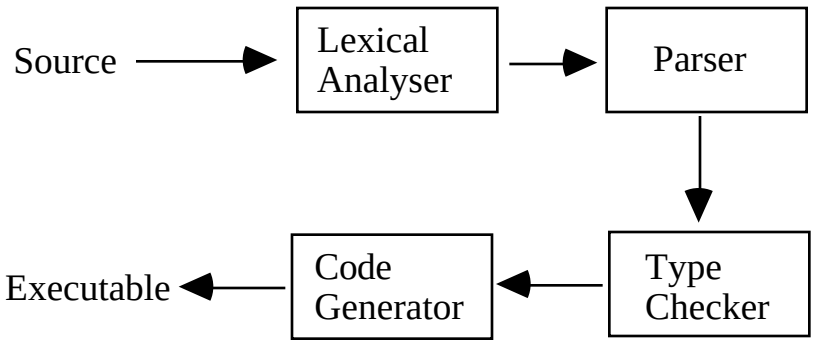

Figure 5.2.1 : Traditional Compiler Architecture

In modern architectures, this distinction between different phases of compiling may be blurred. For example, parsing, type checking and code generation for each language construct may be performed by a single unit of code.

In sections 5.2.2 and 5.2.3 a Napier88 like type checker is used to illustrate some of the key features of type checking. This type checker creates type representations as graphs and performs type checking by comparing these graphs for isomorphism. In this case, the interface of the type checker module provides three types of functions: constructors, selectors and operators. Constructors are used for the creation of type representations, selectors for dereferencing components of type representations and operators for performing type checking.

Section 5.2.2 describes the construction of type representations while section 5.2.3 presents some basic principles behind type checking monomorphic and polymorphic type systems. It is assumed that the systems use structural type equivalence.

5.2.1 Properties of Type Checking Algorithms

A type system is decidable if its type checking algorithm will always produce the correct answer to the question whether some expression is well typed. In a decidable system, the algorithm will exhibit all three of the following properties: soundness, completeness and convergence. An algorithm is sound if the answers it provides are always correct. A complete algorithm will always find the answer if there is one. An algorithm is said to be convergent if the

computation it performs is finite and therefore the algorithm will terminate; otherwise it is said to be divergent.

Based on these properties, sound algorithms may be classified into three categories:

- sound algorithms
- sound and complete algorithms
- sound, complete and convergent algorithms

These categories are further explained below using illustrations based on those presented in [Ghe93c]. In the descriptions given below, the following conventions are adopted:

- good terms are those correctly typed
- bad terms are those which are not correctly typed
- YES denotes the set of expressions for which the algorithm returns a positive answer while NO is the set for which a negative answer is returned
- the striped area represents those terms for which the algorithm is divergent

5.2.1.1 A Sound Algorithm

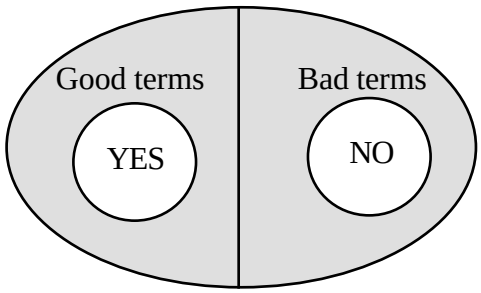

Figure 5.2.2 : Sound Algorithm

Figure 5.2.2 above illustrates a sound type checking algorithm. If a sound algorithm gives a YES answer then the term is correctly typed but it may fail to recognise some correctly typed terms as such. Similarly a NO answer from this algorithm will mean that the term is incorrectly typed but it may not recognise

all badly typed terms. A type system supported by such an algorithm is not decidable.

5.2.1.2 A Sound and Complete Algorithm

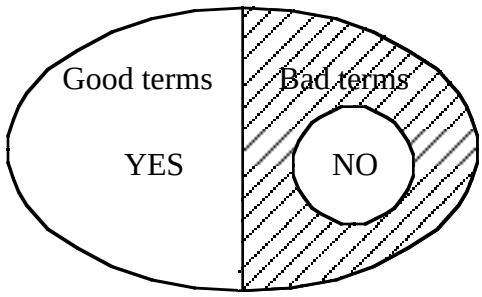

Figure 5.2.3 : Sound and Complete Algorithm

A complete algorithm is one which will always give a result provided there is a YES result. A sound and complete algorithm is illustrated in Figure 5.2.3. If a term is correctly typed then the algorithm will always give a YES answer but in the case of badly typed terms, its behaviour is similar to that of a sound algorithm. Thus this algorithm is convergent in the case of good terms but divergent otherwise. A type system supported by a sound and complete algorithm is said to be semi-decidable.

5.2.1.3 A Sound, Complete and Convergent Algorithm

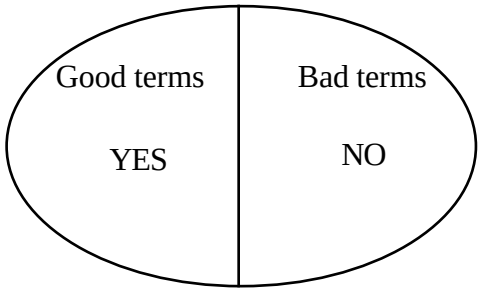

Figure 5.2.4 : Sound, Complete and Convergent Algorithm

A sound, complete and co-complete algorithm, as shown in Figure 5.2.4, will recognise all good terms and bad terms in the universe of discourse as such and is convergent in either case. A type system with such an algorithm is said to be decidable.

5.2.2 Type Representation

Figure 5.2.5 below illustrates a possible data structure for representing types taken from [Con90]. To simplify the example, it is assumed that the type operator *list* and the operations on it are predefined.

```
rec type TYPE is {  label : string ;
                    specificInfo : string ;
                    ref : list [ TYPE ]      }
```

Figure 5.2.5 : Definition of Type

TYPE is defined as a recursive record type which contains three fields. The *label* field is an indication of the type constructor (e.g. "base" for **int**, "rec" for records and "fun" for functions). Information specific to the type being represented, such as the name of the base type or field or branch names in the case of records and variants, is contained in the *specificInfo* field. *ref* contains the reference to a list of component types, such as field types for records and argument and result types for functions.

Using this definition, the data structures for base type ***int*** and a record type that has two fields *x* and *y* of types ***int*** and ***bool*** respectively can be diagrammatically shown in Figures 5.2.6 and 5.2.7 as follows.

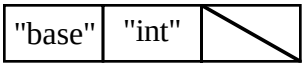

Figure 5.2.6 : Representation of *int*

The *label* field indicates that it is a base type, *specificInfo* indicates which base type it is and since it is a base type there are no references to any component types.

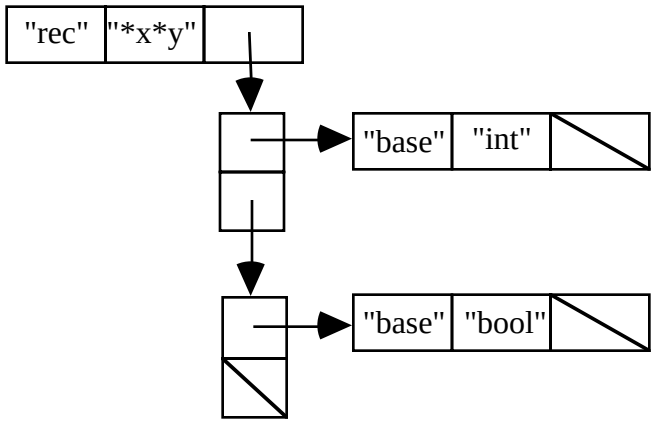

Figure 5.2.7 : Representation of a Record Type

For the record, *label* contains "rec", *specificInfo* contains the names of the two fields delimited by a * sign in order to distinguish the two names and *ref* contains a pointer to a list which contains the representations for the two field types.

5.2.3 Type Checking Algorithms

The type representation scheme presented in the last section constructs a graph for each type in the code being checked. These graphs can then be compared for isomorphism to check type equivalence. Since the type system is infinite, a recursive type checking algorithm is needed to test the isomorphism of the graphs.

5.2.3.1 Monomorphic Type Systems

In a monomorphic type system, type checking involves comparing the graph of the type expected and the graph of the actual type for structural equivalence. An outline of a recursive algorithm [Con90, CBC+90] that uses the type representation scheme of the previous section can be specified as follows:

```
rec let eqType = fun( t1, t2 : TYPE -> bool )
typeIdentity( t1, t2 ) or
( t1.label = t2.label and
  t1.specificInfo = t2.specificInfo and
  eqList( t1.ref, t2.ref ) )

& eqList = fun( l1, l2 : list [ TYPE ] -> bool )
( l1 = nil and l2 = nil ) or
( l1 ~= nil and l2 ~= nil and eqType( head( l1 ), head( l2 ) ) and
  eqList( tail( l1 ), tail( l2 ) ) )
```

Figure 5.2.8 : Basic Type Checking Algorithm

Thus the algorithm first checks for identity of the two graphs, in which case a full structural check is unnecessary. If they are not identical then the labels and specific information are checked for equality and a list equivalence test, which tests the equivalence of each pair of corresponding elements, is carried out on the reference fields of both types.

5.2.3.2 Polymorphic Type Systems

In polymorphic type systems, type checking is more complicated due to the fact that values can have more than one type. Hence a mere structural equivalence test will not suffice in these systems. Consider, for example, a type system that supports inclusion polymorphism. Then the type checker will not only need to be able to check for equivalence but also for the existence of a subtyping relation between two types. A simplified subtype checking algorithm for determining whether type t1 is a subtype of type t2 may be defined as follows:

```
rec let subtype = fun( t1, t2 : TYPE -> bool )
eqType( t1, t2 ) or
( t1.label = t2.label and
  subInfo( t1.specificInfo, t2.specificInfo)  and
  subList( t1.ref, t2.ref ) )
```

Figure 5.2.9 : Subtype Checking Algorithm

Thus the algorithm first performs an equivalence check since equivalence implies that the two types are in the subtyping relation. Otherwise the conditions necessary for subtyping are checked. For t1 to be a subtype of t2, the

labels of both types have to be the same, the specific information and reference fields of t1 and t2 have to satisfy the subtyping conditions necessary for the appropriate type constructor. Functions *subInfo* and *subList* perform these checks in the algorithm given above. It should be noted that this simplified algorithm does not support the subtyping rule which allows location types to be subtypes of their content types.

If the system supports bounded universal quantification based on inclusion then the functionality of the type checker becomes further complicated. Bounded quantification introduces type variables in the signature of the function. One way of type checking these type variables involves creating an environment in which inclusion bindings between type variables and their bound types are stored. It might also be necessary for the compiler to create new type variables and assign bounds to them as the code is compiled. This technique is discussed in detail in Chapter 10.

5.2.4 Summary

The main principles behind the process of type checking and the type checking algorithms have been discussed. The properties of type checking algorithms have also been described. These principles are made use of in Chapter 10 where the implementation of a type checker for a new form of polymorphism is described.

5.3 Object Oriented Programming

5.3.1 Introduction

Object orientation [DT88] has emerged as one of the most popular paradigms in recent years. Many of the ideas associated with it have their origin in the Simula language [DN66] and were later refined during the development of SmallTalk [GR83].

The basic concepts of object oriented programming are explained in this section. It should be noted however that this presents a general overview of the object oriented paradigm and that many object oriented languages deviate from this description in various details. Chapters 6 and 7 present some detailed account of the object oriented languages Eiffel, PolyTOIL and TooL and the object oriented system O_2.

5.3.2 Objects and Classes

In an object oriented system the real world is modelled by objects. Objects can be considered to be instances of abstract data types encapsulating both state and behaviour. State is represented by a collection of instance variables and behaviour is represented by operations or methods. Methods can be functions which return a value or procedures which do not. All computations are specified in terms of message sends. Objects have an identity which persists over time, independent of the changes to the state of the object. When a message is sent to an object, one of the methods available to the object is selected for execution depending on the message. Some possible responses from the object receiving the message involve changing its internal state, sending messages to other objects, replying with an answer, creating new objects or a combination of these. Binary methods, discussed further in section 5.4, are those which have a parameter whose type is intended to be the same as the receiver of the message.

There are two ways of creating an object:

- using prototypes

  new objects are created by using existing objects as prototypes. If this mechanism is used then the system must also permit the creation of objects by specifying a set of methods and instance variables.

- using classes

  a more usual approach is to specify the class of the object to be created. The class is then used as a template for creating object instances.

Classes may also be treated as first class values rather than types. Types provide interface information which determines the operations that can be applied, while classes contain implementation information including initial values for instance variables and bodies for methods. Classes themselves may have types, distinct from object types. Class types include types of instance variables and methods whereas object types only include types of methods. The example in Figure 5.3.1 below illustrates the above concepts.

```
class Person
   var
      name := "" : String;
      age := 0 : Integer;
      job := "" : String
   methods
      function getName() : String {return name}
      function getAge() : Integer {return age}
      procedure changeJob(newJob : String) {job := newJob}
end class
```

Figure 5.3.1 : A Class Definition

Person is defined as a class with three instance variables, *name, age* and *job* and three methods, *getName, getAge* and *changeJob, getName* and *getAge* are functions since they return a value and *changeJob* is a procedure since it does not. Class definitions include initial values for instance variables and complete definitions of the methods. The type of class *Person* can be specified as

```
classtype PersonClass is (    name : String ;
                              age : Integer ;
                              job : String ;
                              getName: fun(): String ;
                              getAge: fun(): Integer ;
                              changeJob: proc(String)  )
end classtype
```

Figure 5.3.2 : Class Types

An object belonging to class *Person* will have the following object type:

```
objecttype PersonType is (    getName: fun(): String ;
                              getAge: fun(): Integer ;
                              changeJob: proc(String)   )
end objecttype
```

Figure 5.3.3 : Object Types

As the only operations allowed on objects are method sends, the object type only specifies method types. Figure 5.3.4 shows how an object of a particular object type can be created.

```
var aPerson : PersonType;
. . .
aPerson := new (Person)
```

Figure 5.3.4 : Object Creation

The identifier for the intended object, *aPerson,* is first declared to be of type *PersonType* and then the ***new*** function is called with the appropriate class, in this case *Person*, which initialises the attributes of the object. ***new*** can be used with any class in the language to initialise its objects.

```
aName := aPerson.getName
```

Figure 5.3.5 : Method Invocation

Figure 5.3.5 above gives an example of method invocation. The dot notation is normally used for message sending. In this case, message *getName* is sent to the object *aPerson* and the result is stored in the variable *aName*.

Instance variables are only visible to the methods of that object. It is possible to change the values of instance variables of objects through the execution of their methods. However, methods associated with an object cannot be modified. In general, methods of an object can be mutually interdependent.

5.3.3 Inheritance

Inheritance is a mechanism which allows incremental definition of classes in object oriented systems. Thus it also provides software reuse. There are two ways of obtaining inheritance corresponding to the two ways in which objects can be created.

- default delegation of responsibility

  used by systems that make use of prototypes for object creation

- subclassing

  used by systems that make use of classes for object creation

There have been different views on the relation between inheritance and delegation. The one adopted here is the classification presented by [DT88]. It should also be noted that [Ste87] has proved that inheritance with subclassing and delegation can model one another. Subclassing, as the name suggests, deals with inheritance at the class level whereas delegation deals directly with objects. Both these categories are briefly discussed below.

5.3.4 Delegation

Delegation [Ste87, DT88] allows the incremental definition of objects. In a system that supports delegation, there is only one type of object. The real world entities are modelled by these objects and they are often referred to as instances without classes. Any object can be defined in terms of any other. Both methods and instance variables can be shared through delegation. If an object delegates an attribute to a prototype then any changes to this attribute will affect both objects. Therefore, objects in a delegation hierarchy may be dependent on one another. Figure 5.3.6 below gives a diagrammatic example of a delegation hierarchy.

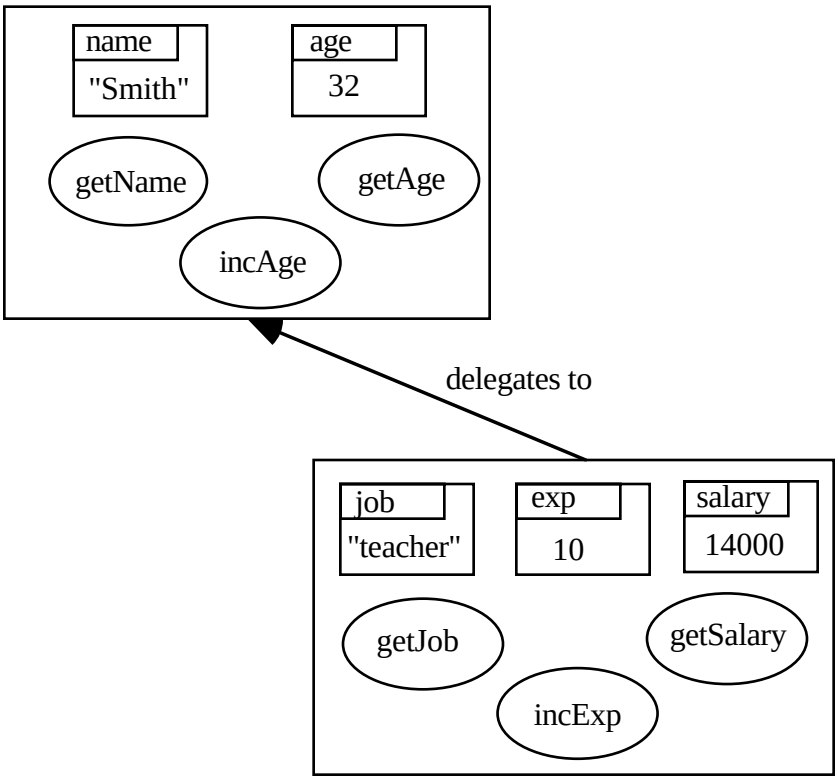

Figure 5.3.6 : A Delegation Hierarchy

In Figure 5.3.6, two objects which may be used to model a person and an employee are illustrated. Instance variables and the values associated with them

are shown in rectangles while methods are represented by ellipses. In this case the employee object delegates the instance variables *name* and *age* and the methods *getName*, *getAge* and *incAge* to the person object and declares only those attributes which are specific to the employee in its definition.

5.3.5 Subclassing

In class based languages, logically related attributes are grouped into classes. In this context classes can be considered as repositories of type and behaviour specifications that can be reused and modified by inheritance. Classes are related in a subclass hierarchy depending on the pattern of inheritance. If class A inherits from class B then A is a subclass of B and B is a superclass of A. This implies that the attributes available to an object of class A are not only those defined in class A but also those defined in any ancestor of class A in the subclass hierarchy. The hierarchy is specified using an inherit clause in the definition of classes. For example, in a class based language, the diagrammatic example of inheritance in Figure 5.3.6 can be specified with class definitions as shown in Figure 5.3.7 below.

```
class Person
   var
      name := "" : String;
      age := 0 : Integer;
   methods
      function getName() : String {return name}
      function getAge() : Integer {return age}
      procedure incAge {age := age + 1}
end class

class Employee
   inherits Person
   var
      job := "" : String;
      exp := 0 : Integer;
      salary := 0 : Integer;
   methods
      function getJob() : String {return job}
      function getSalary() : Integer {return salary}
      procedure incExp {exp := exp + 1}
end class
```

Figure 5.3.7 : Inheritance using Subclasses

Class *Person* is defined with instance variables *name* and *age* and methods *getName*, *getAge* and *incAge*. Any object belonging to this class will automatically have all these attributes associated with it. Class *Employee* is defined to inherit from class *Person* in addition to having its own attributes declared in the body. Therefore any object belonging to *Employee* will have the instance variables *name, age, job, exp* and *salary* and the methods *getName, getAge, incAge, getJob, getSalary* and *incExp* associated with it. Thus the ***inherits*** clause in *Employee* avoids the redefinition of all the attributes declared in *Person.*

An example of a subclass hierarchy incorporating the two classes in Figure 5.3.7 is shown below in Figure 5.3.8.

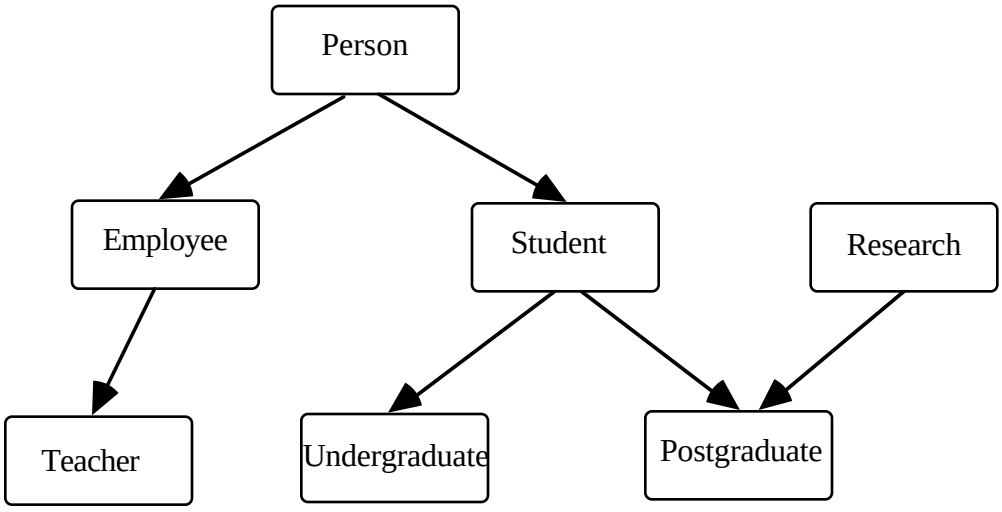

Figure 5.3.8 : A Subclass Hierarchy

Classes are represented by rectangular boxes in Figure 5.3.8 and an arrow from A to B indicates that class B inherits from class A. An interesting point in this hierarchy is the fact that *Postgraduate* inherits from two classes *Student* and *Research*. This is referred to as multiple inheritance.

5.3.6 self and MyType

The concept of object identity plays an important part in inheritance by subclassing. The keyword *self* (or *current* or *this*) is used in method bodies to refer to the receiver of the message and the keyword *MyType* is used to denote the type of self. If the method has not been inherited then *self* will always refer to an object of the class in which the method is defined and *MyType* will denote the object type of this object.

```
class Element
   var
      number = 0 : Integer;
      name = "" : String;
      nextElement = nil : MyType
   methods
      . . .
      function getNext() : MyType {return nextElement}
      procedure displayElement() {self.print}
      . . .
end class
```

Figure 5.3.9 : self and MyType

Figure 5.3.9 illustrates the use of *self* and *MyType*. Class *Element* contains an attribute *nextElement* which is typed as *MyType.* This implies that an object belonging to *Element* will contain a reference to another object of the same class. Since *nextElement* has not been inherited from another class, it can also be typed as *Element* without changing the semantics of the typing. The body of procedure *displayElement* contains a *print* message to the object denoted by *self*. Again, without inheritance, *self* will refer to an object of class *Element*.

However, during subclassing, the meaning of *MyType* changes automatically to refer to the object type associated with the subclass just as the meaning of *self* changes to correspond to an element of the subclass. This facility enables methods of the superclass to be used by any subclass without redefinition.

```
class Element2
   inherits Element
   var
      previousElement = nil : MyType
   methods
      . . .
      function getPrevious() : MyType {return previousElement}
      . . .
end class
```

Figure 5.3.10 : Inheritance of self and MyType

Given the definition of class *Element* in Figure 5.3.9, consider defining a subclass *Element2* that inherits *Element* as shown in Figure 5.3.10 above. Any object *e2* of *Element2* will have access to all the features of *Element*. Therefore a method of *Element* may be sent to *e2*. In this case the *MyType* instance of the method will refer to *Element2* and *self* will refer to *e2*.

The inheritance of binary methods in this context and the problems arising from it are discussed further in Chapter 6 and the solutions to these adopted by some object oriented languages are presented in Chapter 7.

5.3.7 Advantages of Object Oriented Programming

The following are some of the important advantages claimed for using an object oriented language for system construction [DT88]:

- it aids design, implementation and maintenance of complex systems by supporting modularity
- it aids code reuse and extensibility by supporting inheritance
- it may allow designs in which objects provide opportunities for variable grain parallelism and in which decisions relating to the implementation of objects (in software or hardware) are flexible

5.3.8 Summary

The salient features of the object oriented paradigm have been presented in this section. The topics that are of particular interest to this thesis are the inheritance of binary methods and the use of *self* and *MyType* to aid inheritance. Binary methods and their typing in the presence of inheritance are discussed in detail in the next chapter. The use of *self* and *MyType* allows methods of a superclass to be used in the subclass without redefinition. This concept plays an important role in addressing the problem of binary methods.

6 Schema Evolution

6.1 Introduction

Schema evolution in any system may require changes to types or schemata and changes to data (including programs) that conform to these types or schemata. Dealing with the effects of evolution at the two different levels requires different strategies. Programming languages provide mechanisms at the type level to capture evolution whereas database systems, in addition to any type level mechanisms, also require other tools to explicitly evolve data to keep it up to date with the changes to schemata. Eiffel, TooL and PolyTOIL are languages that provide mechanisms at the type level to capture evolution. O_2, Orion [BKK+87] and GemStone [PS87] are examples of systems that have developed mechanisms to deal with the changes caused by evolution at the data level.

A brief outline of the kinds of schema evolution and the effects it has on type and data is presented in this chapter. The O_2 mechanisms for dealing with evolution at the data level are also described since it is a typical object oriented database management system which is well established and contains advanced features that deal with evolution. Chapter 7 describes the strategies used by Eiffel, TooL and PolyTOIL at the type level.

6.2 The Effects of Schema Evolution

Databases contain data that has been logically grouped by schemata to model real world entities. The constantly changing needs of the applications that use databases require changes to data, programs and meta-data (schemata) in databases. The changes to the schemata of a database are referred to as schema evolution.

Schema evolution can generally be categorised into three types [MCC+93].

- additive evolution : the new schema models more semantic knowledge than the old schema
- subtractive evolution : the new schema models less semantic knowledge than the old schema
- descriptive evolution : the new schema models the same semantic knowledge as the old schema but in a different manner

In the case of additive evolution, previous programs will continue to execute in a type safe manner. However, their semantics may not match the updated schema

and hence they may need to be changed. Every change to the schema will also require a corresponding change to data belonging to that schema.

With subtractive evolution, programs which access the deleted parts of the data model have to be deleted themselves or modified. Corresponding changes to data belonging to the model are not necessary but will avoid wasting space.

After descriptive evolution, the semantics of the model will remain the same as before even though the database structure may change. Descriptive changes are often made for reasons of convenience or efficiency. This type of evolution is claimed to be the hardest to accommodate in traditional database systems.

In practice, any change to the schemata of a database is likely to involve a combination of all three kinds of evolution. Changing the schema of database while maintaining the consistency of data and programs belonging to that schema with the semantics of change has proved to be a difficult problem [Zdo86, SZ87, MCC+93, Odb94, Rod95]. There are two levels at which the problem of schema evolution may be addressed.

- type or class level
- data (instances of types or classes) level

At the type or class level, one mechanism for dealing with schema evolution is polymorphism. Since this provides data abstraction and software reuse, polymorphic code may accommodate evolution on a schema. However, any particular type of polymorphism supported by a system is unlikely to cover all possible changes to a schema.

Object oriented languages support inheritance as a means of abstraction and reuse and to capture evolution. In object oriented systems, refinement, often equated with inheritance, leads to the well-documented problem of the inheritance of binary methods [BCC+95, Cas95]. Binary operations, such as the = relation and the subset relation, take two arguments of the same type. In object oriented languages, operations are coded as methods and the receiver of the message is implicitly the first argument of these methods. The second parameter of the method is the only explicit argument. The term binary method is used to describe any method which has at least two arguments, with an implicit and an explicit argument of the same type.

The problem with binary methods is caused by their typing in the presence of inheritance. To ensure type safety, the type of an inherited method in the

subclass must be a subtype of its type in the superclass. Since methods are functions, this condition implies that their parameter types have to be contravariantly replaced by supertypes in the subclass.

The type of the implicit argument of the method automatically changes during subclassing to refer to the object type of the subclass. The explicit argument also changes in the same way as functions often evolve by specialising their parameter types. However, these type changes in subclasses may not produce subtypes. Consider the example in Figure 6.2.1 below.

```
class Coordinate
   var
      x := 0 : Integer;
      y := 0 : Integer;
   methods
      function equal( c : Coordinate ) : Boolean
               {return x = c.x and y = c.y}
end class

class ThreeDCoordinate
   inherits Coordinate
   var
      z := 0 : Integer;
   methods
      function equal( c : ThreeDCoordinate ) : Boolean
               {return x = c.x and y = c.y and z = c.z}
end class
```

Figure 6.2.1 : Inheritance of Binary Methods

Class *ThreeDCoordinate* inherits the features from class *Coordinate.* A new instance variable *z* is added in the subclass and the method *equal* is redefined to incorporate the new variable. Both the implicit and the explicit arguments to *equal* are now of type *ThreeDCoordinate*. Given this redefinition, the type of *equal* in the subclass is not a subtype of the original method type and hence the method in the subclass cannot be used where the original method is expected. Consider the use of the method *equal* in figure 6.2.2 below.

```
var newC : Coordinate

var aThreeDC : ThreeDCoordinate

procedure compare( x : Coordinate ) : Boolean
begin
    newC := new Coordinate( 50, 75 )
            ! a new instance of class Coordinate is assigned to newC
    x.equal( newC )
end

aThreeDC := new ThreeDCoordinate( 45, 60, 25 )

let b = compare( aThreeDC )
```

Figure 6.2.2 : An Unsafe Method Call

If procedure *compare* is called with an actual parameter of type *Coordinate* then the call is type safe. However, if the actual parameter is a *ThreeDCoordinate* , as shown in Figure 6.2.2, then a runtime error will occur when the method call is evaluated. Since the value of *x* is a *ThreeDCoordinate*, the code for *equal* in *ThreeDCoordinate* will be executed. But *newC* has no instance variable called *z* and hence the call will fail. Thus, contravariant subtyping rule is needed for type safety while the covariant substitution captures the evolutionary demands of method types.

At the data level, the main concerns regarding evolution in a database are how updates are carried out on the data and how the database is brought to a consistent state after schema modifications. To this end, the following issues must be addressed.

- Restructuring data to conform to the new schemata
- Moving instances of a class/type to another
- Determining when data is to be updated

Each of these issues is examined further in the context of the O_2 database system in the following section and the solutions adopted by O_2 in each case are presented.

6.3 Data Evolution in the O_2 Database System

6.3.1 Introduction

The O_2 model [Deu90, LRV90, Deu91] was developed by the Altair group in France. Although, as an object oriented database system, O_2 supports subtyping and inheritance, the topic of interest here is how it deals with evolution at the data level. The O_2 strategies [Zic89, Zic91] are used as an example of the issues involved in bringing the data up to date with the semantics of schema changes.

6.3.2 The O_2 System Structure

O_2 schemata are logical entities which group together data definitions. A schema can contain the following elements: classes, named objects and values, functions and applications. Each schema has one or more bases associated with it. An O_2 base groups together objects and values which conform to a schema.

Schemata and bases are physically stored on files grouped together into volumes. A volume is implemented as a Unix file. A named system is a collection of volumes.

6.3.3 An Overview of the O_2 Type System

Types in O_2 can be atomic or structured. A type is defined recursively from atomic types, named types, classes and constructors. Constructed types can be defined by applying tuple, list, set and unique set constructors to other types.

O_2 supports multiple inheritance. Any name collisions are resolved by local renaming. An inherited feature may be redefined in accordance with the covariant subtyping semantics.

Objects model real world entities. An object has an identity, a value and a behaviour defined by its methods. Objects are instances of classes just as values are instances of types.

Objects with the same value type and methods are grouped together in the same class. A class specification contains the following information: class name, class type, public and private properties, class methods and class inheritance. All O_2 classes are treated as subclasses of the system defined class *Object.* If a class does not explicitly specify any superclasses then it is an immediate descendant of *Object.*

Programs create values and objects during their execution which, by default, are discarded at the end of the execution. However an entity (object or value) will be persistent if it is directly or transitively reachable from a persistent root. O_2 provides a *name* declaration to create these roots. Persistence does not affect the manipulation of entities.

6.3.4 Schema Evolution in O_2

Schema modifications can be performed using special primitives, for example by adding or deleting attributes in a class, or by redefining the structure of a class. The special primitives available in O_2 for changing schemata are:

- creation of a new class
- modification of an existing class
- deletion of an existing class
- renaming of an existing class
- creation of an inheritance link between two classes
- deletion of an inheritance link between two classes
- creation of a new attribute
- modification of an existing attribute
- deletion of an existing attribute
- renaming of an existing attribute

6.3.5 Database Updates in O_2

Once the schemata have been changed to incorporate the requirements of evolution, the data that belong to them must be changed to be consistent with the schemata. The following sections describe how each of the data update issues stated earlier in section 6.2 are dealt with by the O_2 system.

6.3.5.1 Restructuring Data in O_2

O_2 makes use of conversion functions to restructure data after schema changes. Conversion implies that the structure of classes have been modified and that data belonging to these classes need to be changed to conform to the new structure.

It should be noted that data will still belong to the same class after restructuring. Conversion functions are attached to modified classes and contain specifications of how the data is to be changed.

Conversion functions may be user defined or default. The user has the option to explicitly define conversion functions and associate them with the modified classes. User defined conversion functions can be of two types: simple or complex. Simple functions perform transformations that only require local information of the object being accessed i.e. they do not need to access any other object. On the other hand, complex conversion functions perform the transformations using objects other than the one being accessed.

If the user has not specified any conversion functions, then the database system automatically transforms objects using default transformation rules. When a class is modified, each attribute of the class before and after modification are compared and the value of the attribute is transformed according to the default rules. These rules are outlined below:

- an attribute present in the class before but not after its modification (a deleted attribute) is ignored
- an attribute present in the class after but not before its modification (a new attribute) is initialised with default initial values
- an attribute present in the class both before and after its modification is transformed according to a set of rules which depend on the initial and final types of the attribute.

If user defined conversion functions are available then they take precedence over the default transformation rules.

6.3.5.2 Moving Data to Other Classes

Object migration in O_2 refers to the ability of an object to change its type by moving from one class to another. There are two ways in which objects may migrate in O_2.

- a single object may change its class
- an entire class extension or part of it may be migrated to another class

The root class *Object* has a system method *migrate()* associated with it. Since all classes are subclasses of *Object*, this method is available to every object in O_2. When invoked for a particular object, with the name of the target subclass as the input parameter, *migrate* enables the object to migrate from its class to the specified subclass. To avoid runtime type errors objects are only permitted to migrate to subclasses. Despite this limitation, migration is believed to be useful in the following cases:

- existing objects of superclasses need to be moved to a newly added subclass
- objects belonging to a class that is to be deleted need to be retained by moving them to subclasses

An entire class extension or part of it may also be moved to subclasses by associating migration functions with classes. These functions can specify selection conditions for migration based on the attributes of the class. Every object of the associated class that satisfies the selection conditions will be migrated to the required subclass using the system function *migrate*.

Conversion functions restructure data to keep them consistent with their modified classes. The classes and types to which data belong remain the same. Migration functions move data from one class to another, thus changing their types.

6.3.5.3 Time of Update in O_2

An important design decision for implementing database updates is when the database is brought to a consistent state with the updated schemata. This determines when conversion and migration functions will be executed over the data to be changed. There are two choices for the time of update:

- with an immediate strategy, objects in the database are updated as soon as the schema modification is performed.
- with a deferred strategy, objects are updated only when they are used after the schema modification.

O_2 provides support for both of these strategies and the user can choose one that is most appropriate to the application.

6.3.6 The Implementation of Database Updates

Both conversion and migration functions used for updating the database can be implemented using either the immediate or the deferred strategy. However the default implementation mechanism in O_2 for the execution of these functions is the deferred update. The basic principle which determines the correctness of any deferred transformation is that the end result of a transformation performed using deferred update must be the same as the result of the same transformation implemented using immediate update. Thus the implementation strategy should be orthogonal to the semantics of the transformation.

O_2 uses a technique known as screening to implement deferred update for both simple and complex conversion functions. With this technique, deleted or modified information in an object is not physically deleted but is retained in a screened part of the data structure representing the object. Only conversion functions are allowed access to screened information. Application programs can not see the screened part of an object. When an object is accessed by an application the system will execute the associated conversion function to perform the transformation.

Screening is particularly important for implementing complex conversion functions with deferred update as this combination can result in runtime errors or incorrect database information. Complex conversion functions require access to other objects which may already have been modified since the update is not immediate. However, screening ensures that the information that may have been modified or lost is stored in the screened part of the objects required.

The immediate database transformations in O_2 are implemented using the algorithm for deferred transformation. O_2 provides a schema command, *transform database*, which launches an internal tool that traces all the objects in the database that are not up to date. The conversion and migration functions associated with these objects are then executed using the deferred update algorithm. After the execution of this command, all objects in the database conform to the latest schema definition.

6.4 Summary

The effects of evolution on schema and data and the problems relating to them have been presented. As explained in the chapter, these problems can be addressed at two different levels. The mechanisms adopted by the O_2 object oriented database system for dealing with evolution at the data level have also

been discussed. These mechanisms provide an example of what will be needed if the system does not provide a means of capturing evolution at the type level. Chapter 7 describes existing type level strategies used by programming languages to deal with evolution.

7 Related Work

Chapter 6 examined the problems associated with schema evolution and gave an example of the way these problems are dealt with at the data level. Two different ways of addressing the problem of evolution at the type level in object oriented languages, especially the type safe inheritance of binary methods, are described in this chapter. The object oriented languages Eiffel, PolyTOIL and TooL are used as examples of systems supporting these techniques. A brief summary of how some other well-known languages such as SmallTalk [GR83, GGH+91], C++ [Str86, GGH+91] and Java [Fla97] deal with the inheritance of methods is also presented.

7.1 Eiffel

Eiffel [Mey92] is an object oriented programming language that uses covariant specialisation. The type system of Eiffel and its inheritance and type safety mechanisms are discussed in the following sections. Some comments on these mechanisms are presented in section 7.1.8.

7.1.1 An Overview of the Eiffel Type System

Eiffel supports the concepts of classes, objects, features and inheritance. The type system of Eiffel is based entirely on the notion of classes. Thus each object that exists during the execution of a system belongs to a class of that system. Such objects are called direct instances of the class. Deferred classes have no direct instances. They are incomplete abstractions which their descendants use as the basis for further refinement.

Classes introduce a set of features which may be of two kinds. Attributes represent fields of direct instances of the class while routines represent computations applicable to those instances. An attribute is either constant or variable. A routine is either a function (returns a result) or a procedure (does not return a result).

Every type in the language is based on a class known as the base class of the type. There are three possible kinds of types in Eiffel:

- reference types

  instances of reference types are references to objects which are created at run time through explicit creation operations

- expanded types

instances of expanded types are the objects themselves rather than references to objects and hence do not require creation operations

- formal generic names

  these correspond to type parameters to be provided during uses of a class by parents or proper descendants. Instances of these types may be references or objects

7.1.2 Genericity

Generic classes can be defined in Eiffel by parameterising class definitions.

```
class LINKED_LIST [T]
feature
. . .
end -- class LINKED_LIST
```

Figure 7.1.1 : A Generic Class

In Figure 7.1.1, LINKED_LIST is a generic class, parameterised by the formal generic parameter T. In order to obtain a type, a generic class must be supplied with a type as an actual generic parameter. A type thus obtained is referred to as a generic derivation of the base class, which in the example is LINKED_LIST. It is also possible to constrain the formal generic parameter by specifying a bound on it as shown in Figure 7.1.2 below.

```
class LINKED_LIST [T -> PERSON]
feature
. . .
end -- class LINKED_LIST
```

Figure 7.1.2 : A Constrained Generic Class

Any actual parameter corresponding to T must be a descendant of the class PERSON.

7.1.3 Inheritance

Eiffel supports the use of inheritance as a module extension (defining new classes from existing ones by adding or adapting features) and type refinement (defining new types as specialisations of existing ones) mechanism. Features obtained by a class C from its parents are called inherited features. If feature f

has been defined in C then there are two possibilities. If f is inherited then the definition is a redeclaration whereas if f is a new feature it is said to be immediate to C and is introduced in C. In the latter case, C is the class of origin for feature f.

In the graphical convention adopted by Eiffel designers for representing inheritance hierarchies, classes are represented by labelled ellipses and the inheritance relation is represented by directed arrows from the heir to the parent. Thus, Figure 7.1.3 denotes that B inherits from A.

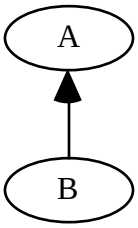

Figure 7.1.3 : Graphical Representation of Inheritance

Cycles are not permitted in the hierarchy. However, multiple, and as a result, repeated inheritance are permissible. Figures 7.1.4 and 7.1.5 below illustrate the inheritance structures for multiple and repeated inheritance.

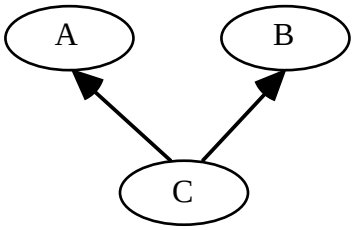

Figure 7.1.4 : Multiple Inheritance

Class C in Figure 7.1.4 inherits from both class A and class B.

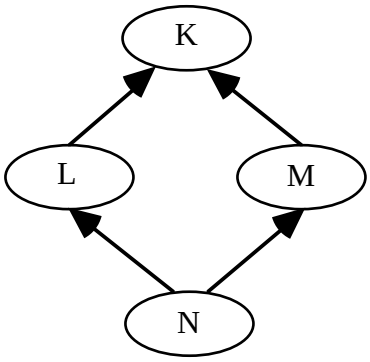

Figure 7.1.5 : Repeated Inheritance

Repeated inheritance occurs when an attribute is inherited by a class in more than one way. If two or more ancestors of a class have a common parent, then it may repeatedly inherit a feature from the common parent. In Figure 7.1.5 class N inherits from L and M which in turn inherit from class K. Thus N inherits from K in two different ways.

The simplest case of repeated inheritance, called direct repeated inheritance, is shown in Figure 7.1.6 below where *EMPLOYEE* is a repeated heir of *PERSON*. The class *PERSON* is specified twice in the inheritance part of the class definition of *EMPLOYEE*. This facility is useful if two copies of the same feature from *PERSON* are needed in *EMPLOYEE* or if a feature is required to be redefined in two different ways.

```
class EMPLOYEE inherit
  PERSON
    rename . . . redefine . . . end
  PERSON
    rename . . . redefine . . . end
 . . .
end -- class EMPLOYEE
```

Figure 7.1.6 : Direct Repeated Inheritance

Indirect repeated inheritance occurs when one parent of a class C is a proper descendant of a class A and one or more of the other parents are descendants of A. An example is given below in Figure 7.1.7.

```
class PERSON
  feature
   age : INTEGER;
   . . .
end -- class PERSON

class WORKER inherit
  PERSON
  . . .
end -- class WORKER

class EMPLOYEE inherit
  WORKER
  . . .
end -- class EMPLOYEE

class STUDENT inherit
  PERSON
  . . .
end -- class STUDENT

class DEMONSTRATOR inherit
  EMPLOYEE
  . . .
  STUDENT
  . . .
end -- class DEMONSTRATOR
```

Figure 7.1.7 : Indirect Repeated Inheritance

Class *DEMONSTRATOR* in Figure 7.1.7 repeatedly inherits the attribute *age* from *EMPLOYEE* and *STUDENT*. *EMPLOYEE* inherits it from *PERSON* through *WORKER* while *STUDENT* directly inherits it from *PERSON*.

Eiffel provides the programmer with some techniques to choose the result of repeated inheritance in any class. Sharing results in only one feature whereas replication permits several. Thus, the repeated inheritance rule can be stated as follows:

Let C be a class and $B_1, \ldots, B_n$ ($n \geq 2$) be parents of C with a common ancestor A. Let $f_1, \ldots, f_n$ be features of these respective parents all having as origin the same feature f of A. Then any subset of these features inherited by C under the same final name yields a single feature for C while any two features inherited under different names yield two different features for C. Features are said to be shared in the first case and replicated in the second. Renaming inherited features is one way to replicate them.

7.1.4 Feature Calls

In the Eiffel model, as is common to object oriented languages, the fundamental means of computation is to apply an operation to an object. Given the model, this operation has to be a feature of the class to which the object belongs. A call is an application of a feature to an object possibly with arguments. Thus, with the common dot notation, the structure of a call is

```
object.feature(parameter_list)
```

Object and parameter list are optional to a feature call. Figure 7.1.8 below gives examples.

```
aPerson.incAge

aStudent.enterGrade(stGrade)

fixSalary(increment)
```

Figure 7.1.8 : Feature Calls

The first call is the application of routine *incAge*, which takes no parameters, to an object *aPerson* of class *PERSON*. The second call contains all the components of a feature call. In the third call, the target object *fixSalary* operates over is the predefined entity *Current* which represents the current object of system execution and may be omitted in a feature call.

If the feature is an attribute or a function then the call is syntactically an expression. If it is a procedure then the call is an instruction.

7.1.5 Conformance

Conformance is a relation between types which determines when a type may be used in place of another. The conformance relation is based on inheritance. A type V will conform to a type T where the following conditions hold.

- the base class of V is a descendant of the base class of T
- If V is a generically derived type then its actual generic parameters conform to the corresponding ones in T
- If T is expanded then no inheritance is allowed; V can only be identical to T or its base type

Conformance validates the use of many operations. Any of the following will be valid if V conforms to T according to the definition given above, with x of type T and y of type V.

- the assignment x := y
- the routine call r (. . , y, . .) where x is the formal parameter declared in r at the position of y
- the creation ! V ! x, . . . which creates an instance of V and attaches x to it
- the redeclaration of x as being of type V in a proper descendant where x is an attribute, a function or a routine argument
- any use of C [. . , V, . .] with V as the actual parameter where the corresponding formal parameter of C is constrained by T

The conformance relation can also be applied to signatures of features in classes. Using the definition of type conformance specified earlier, the concept of signature conformance can be defined as follows.

A signature t = (< B_1, . . . , B_n >, < S >) conforms to a signature s = (< A_1, . . . , A_m >, < R >) if and only if the following conditions are satisfied.

- each of the two sequence components of t has the same number of elements as the corresponding components in s
- every type T_i in each of the two sequence components of t conforms to the corresponding type S_i in the corresponding components of s

7.1.6 Reattachment of Entities

At any point during execution, every entity of the Eiffel system has an attachment status: it is either attached to an object or it is void. There are four reattachment operations which may change the attachment status of an entity

- association of an actual argument to a formal argument during a routine call
- the assignment instruction which may attach an entity to a new object or remove an existing attachment
- the assignment attempt instruction which conditionally performs the same function as the assignment instruction in cases where the assignment may be statically considered unsafe
- the creation instruction which attaches an entity to a newly created object

Figure 7.1.9 below gives examples of all four operations in the order given above.

```
r ( . . ., y,  . . . ) with routine declaration r ( . . . , x : T, . . . ) is . . .

x := y

x ?= y

!! x
```

Figure 7.1.9 : Reattachment Operations

In the first two cases, the condition required for reattachment is that the type of the source conforms to the type of the target. These are called direct reattachment. They also have the same semantics: for reference types, the target is made to refer to the object attached to the source, otherwise it is made void.

However, the assignment attempt applies only to reference types and is free from any conformance constraints. This is referred to as reverse reattachment since it is possible to perform assignments which go against the normal conformance rules if it is known that the assignment will be type safe. Assignment attempt behaves as an assignment instruction if the dynamic type of the source object conforms to the type of the target, otherwise the target is made void.

7.1.7 Type Checking Feature Calls

Since feature calls perform most of the computations in Eiffel, the validity of feature calls essentially ensures the type safety of the system. There are two levels of type validity in Eiffel.

- class level validity
- system level validity

In the absence of inheritance and genericity, checking class level validity is straight forward and is sufficient to guarantee type safety. For the routine call *target.fname(y)* to be valid, the type of *target* must have a feature of final name *fname*; this feature must be available to the class from the which the call is made; and the feature must have the requested signature i.e. it must be a routine with a single formal argument which *y* conforms to. Calls which satisfy these conditions are said to be class-valid.

Class validity does not imply that the call is valid since the dynamic type of an object may be different from its static type. This difference is caused by the inheritance mechanism being used in the following instances.

- a class may override the export policies of its parents
- a routine redefinition may replace the type of a formal argument by a type conforming to the original (covariant argument typing)

To overcome this problem, a system level validity check was proposed. The possible dynamic types of an object are said to form its dynamic type set. For a feature call to be system-valid, the conditions for class validity must be enforced for each class in the dynamic type set of the object concerned. A call is unconditionally valid if it is both class-valid and system-valid.

However, this policy is not implemented as it requires access to the entire system [Mey97] and could not be performed incrementally [Mey97]. Therefore, instead of this check, the notions of polymorphic entity and catcall were introduced and a new type rule based on these notions was formulated to ensure validity of feature calls.

An entity x is polymorphic if it can be attached to objects of more than one type. The following are the possible circumstances for x to be polymorphic:

- the assignment x := y where y is of a different type from x or is polymorphic

- x is a formal routine argument
- x is an external function

A routine call is polymorphic if its target is polymorphic. A routine is a cat (changing availability or type) if a redefinition changes its export status or the type of any of its arguments. A routine call is a catcall if a redefinition makes it invalid because of a change to its export status or argument type. The type rule necessary to ensure validity of a call is that polymorphic catcalls are invalid i.e. a routine call cannot be both polymorphic and a catcall.

7.1.8 Comments

The inheritance relation, covariant specialisation of features and generic classes provide the facilities for dealing with evolution in Eiffel. However the use of these mechanisms requires a means to guarantee the type safety of operations such as routine calls and assignments.

Eiffel does not support subtyping. It defines a conformance relation to check for compatibility between target and source objects. Thus, conformance is the only relation that controls the validity of operations. Although conformance models the inheritance patterns for features, it is essentially covariant and hence it is possible to attach an object of a more general type to one which is a subtype.

Eiffel provides two mechanisms for dealing with this problem. The assignment attempt is a construct that tests for the dynamic type of a source object at runtime and performs the assignment only if it is type safe. The designers of Eiffel claim that just as programming without covariance and descendant hiding (ability of a descendant to override the export policies of its parents) will be too restrictive, static typing will be too restrictive without assignment attempts. The other mechanism for ensuring validity of operations is making polymorphic catcalls invalid, as described in the previous section.

7.2 Type Matching

In object oriented languages, it is often necessary to define subclasses that inherit binary methods from their superclasses. Since subtyping fails to capture this form of evolution, type safe inheritance of binary methods during subclassing is not possible. The concept of matching [Bru95, BCC+95, AC96, Bru96] has been proposed to address this problem. Two languages that have incorporated matching in their type systems are introduced here.

PolyTOIL [BSG95, Bru96] was developed by Bruce et al at Williams College, Massachusetts. The Tycoon object-oriented Language (TooL) [GM95, GM96] evolved from the language Tycoon developed at Hamburg University. Various features relevant to inheritance, polymorphism and evolution are described for each language in the following sections.

Matching is a relation between two object types. It eases some of the restrictions placed by the subtyping relation on inheritance while preserving type safety and static typing. Since matching coincides with the specialisation of method types in subclasses, generic code bounded by matching can capture the evolution of types in object oriented languages.

7.2.1 Motivation

The aim of the PolyTOIL language is to achieve the following:

- support for incremental modification of code
- capturing accurate type information especially in the case of methods inherited or redefined in subclasses
- type safe inheritance of binary methods

The motivation for designing TooL as a new language based on Tycoon was to verify the following hypotheses:

- that a purely object oriented language leads to more uniform and easier to understand program libraries
- that type matching increases code reuse in complex libraries

7.2.2 Syntax

The syntax of PolyTOIL is very similar to the one presented in section 5.3 to illustrate object oriented programming.

A TooL program is a set of (mutually recursive) named class definitions. A typical class definition in TooL is shown below in Figure 7.2.1.

```
class aClass
  private
    x : Int
    . . .
  public
    m1 (x : Self) : Bool
    . . .
```

Figure 7.2.1 : Class Definition in TooL

The keyword *Self* indicates the self reference of classes. Figure 7.2.2 shows the object type of *aClass*.

```
Interface aClass (Self)
 { m1(x : Self) : Bool, . . . }
```

Figure 7.2.2 : Object Types in TooL

Object types contain only the method types. A class definition implicitly defines its object type. TooL allows parameterisation of class definitions by an element type. These classes are referred to as generic classes. It is also possible to specify a bound on the element type.

```
class WriteStream (E <: Object)
  put (e : E) : Void
```

Figure 7.2.3 : Parameterisation of Classes in TooL

In the example in Figure 7.2.3, *WriteStream* is a generic class, *E* is its element type and *E* is subtype-bounded by *Object.* This bound enables the programmer to specify the basic methods that any element type will be required to support. TooL also permits the parameterisation of individual method signatures.

It should be noted here that both languages use structural equivalence and hence there is an implicit hierarchy of types and an explicit hierarchy of classes.

7.2.3 Subtyping

Subtyping, along with subsumption, provides a mechanism for using objects of a type where objects of a different but related type are expected. The definition of subtyping in PolyTOIL can be stated as :

ObjectType{m_j : T'_j} $1 \le j \le k$ <: ObjectType{m_i : T_i} $1 \le i \le l$

where m_j and m_i are methods and T'_j and T_i are method types

if 1) $l \le k$ and for each $i \le l$, T'_i <: T_i

2) no method type has a contravariant occurrence of **MyType**

Thus, in Figure 7.2.4 *Student* is a subtype of *Person.*

```
Person = class
    var
        name = "": string
    methods
        function setName(newName: string) {name := newName}
        function getName(): string {return name}
end class

Student = class
    inherits Person
    var
        id = 0: Integer
    methods
        function setId(newId: Integer) {id := newId}
        function getId(): Integer {return id}
end class
```

Figure 7.2.4 : Subtyping in PolyTOIL

In TooL, the relation that A is a subtype of B is written as A <: B. The subtype relation is defined by structural induction on object types. An object type called *Object* is the top of the subtyping lattice and the bottom is *Nil*.. The important uses of subtyping are subsumption and bounded quantification with bounds specified by subtyping.

```
class WriteStream (E <: Object)
  put (e : E) : Void

class File
  get : Char
  put (ch : Char) : Void
  close : Void
```

Figure 7.2.5 : Subtyping in TooL

In Figure 7.2.5, *File* is a subtype of *WriteStream (Char)*. This means that if class *Object* is defined as shown in Figure 7.2.6

```
class Object
  printOn (aStream : WriteStream (Char))
```

Figure 7.2.6 : Definition of Class *Object* in TooL

and *stdout* is an instance of the class *File* then the message send to a string literal object in Figure 7.2.7 is valid.

```
"a string".printOn (stdout)
```

Figure 7.2.7 : Inherited Method Call in TooL

7.2.4 Subclasses

The concept of subclasses is introduced to support the reuse of instance variables and methods of an existing class (referred to as the superclass) in defining a new class (the subclass). The common members are said to be inherited by the subclass. In PolyTOIL, the keyword ***super*** is used in a subclass to refer to the superclass from which it inherits. A subclass can modify a method it inherited from the superclass. Thus subclasses support incremental modification. For example, a subclass of a class *Point* can be specified as shown in Figure 7.2.8 below.

```
ColourPoint = class
    inherits Point modifying move
    var
        colour := green: ColourType
    methods
        function getColour(): ColourType {return colour}
        . . .
        procedure move(dx, dy: integer) {super.move; colour :=
                                                        red}
end class
```

Figure 7.2.8 : Subclasses in PolyTOIL

In an object, ***MyType*** is simply the type of that object since no inheritance is possible at that level. In classes, it will have a flexible meaning. The following example in Figure 7.2.9 illustrates the use of ***self*** and ***MyType***.

```
Node = class
    var
        value = 0: Integer;
        next = nil: MyType
    methods
        . . .
        function getNext(): MyType {return next}
        procedure setNext(newNext: MyType) {next := newNext}
        procedure attachRight(newNext: MyType)
                                                {setNext(newNext)}
    end class


DbleNode = class
    inherits Node modifying attachRight
    var
        prev = nil: MyType
    methods
        . . .
        procedure setPrev(newPrev: MyType) {prev := newPrev}
        procedure attachRight(newNext: MyType)
                    {setNext(newNext); newNext.setPrev(self)}
end class
```

Figure 7.2.9 : self and MyType in PolyTOIL

In *Node*, the instance variable *next* is declared to be of type ***MyType***. The methods shown also make use of ***MyType*** for parameter and result types. For an object of class *Node*, ***MyType*** merely denotes *Node*. However, when *Node* is inherited by *DbleNode*, the meaning of ***MyType*** in the inherited features automatically changes to refer to *DbleNode* even though the features are defined in *Node*.

There are distinct hierarchies for subclasses and subtypes. It is possible to have subclasses which are not subtypes of their superclasses and subtypes which are not subclasses. Both concepts support reuse in different ways. If type B is a subtype of type A then any operation available to A can also be performed over B. This depends only on the interface information of objects. Subclasses allow reuse of code inside classes i.e. in their definition.

7.2.5 Type Quantification

Classes can be made generic by type parameterisation. Both PolyTOIL and TooL support bounded parametric polymorphism. Figure 7.2.5 contains an example of a class parameterised by a bounded type.

7.2.6 Inheritance

The use of inheritance in PolyTOIL has been illustrated in detail in section 7.2.4 which describes the use of subclasses in the language.

Figure 7.2.10 gives an example of inheritance in TooL.

```
class Indexed (K <: Object, E <: Equality)
super Bounded (E), Sortable (E), Keyed (K, E)
  . . .
```

Figure 7.2.10 : Inheritance in TooL

All objects of class *Equality* have the equality operation defined over them. Class *Indexed* inherits from three other classes *Bounded*, *Sortable* and *Keyed.* In parameterised class definitions different element types can inherit from different superclasses. Possible conflicts due to multiple inheritance are solved by linearisation of the inheritance tree. In addition to methods, type parameters of superclasses may also be refined during inheritance.

```
class Point
super Equality
  x : Int
  y : Int
  . . .

class Set (E <: Object)
  add (e : E) : Void
  includes (e : E) : Bool
  iterate (F <: Object, unit : F, f : fun (: F, : E) : F) : F
  . . .

class PointSet (E <: Point)
super Set (E)
  averageX () : Int {self.iterate (0, fun (total : Int, e : E)
            total + e.x / size)}
```

Figure 7.2.11 : Refining Type Parameters in Subclasses in TooL

In Figure 7.2.11, class *PointSet* inherits from class *Set* but refines the bound of the type parameter from *Object* to *Point.*

7.2.7 Matching

The matching relation between two object types in PolyTOIL is defined by

$$\text{ObjectType}\{m_j : T'_j\}\ 1 \le j \le k \ <\#\ \text{ObjectType}\{m_i : T_i\}\ 1 \le i \le n$$

if $n \le k$ and for each $i \le n$, $T'_i <: T_i$

Object type A matches object type B if for every method $m_i : T_i$ in B there is a corresponding method $m_j : T'_j$ in A such that T'_i is a subtype of T_i. This is a weaker relation than subtyping. No assumption is made on the meaning of ***MyType*** while determining whether two method types are in subtype relation. Therefore the method type of *attachRight* in *DbleNode* will be a subtype of the method type in *Node* and the object type of *DbleNode* will match that of *Node* . It should be noted these object types are not in subtyping relation since *attachRight* has a contravariant occurrence of ***MyType***.

In TooL, A matches B is denoted by A <∗: B. An object type A matches an object type B if they are subtypes under the assumption that the corresponding

Self types are equal. Matching does not support subsumption in general. It supports inheritance and specialisation of methods with contravariant occurrences of the recursion variable. As in the case of subtyping, matching is also defined by structural induction on object types.

```
class Equality
  "=" (x : Self) : Bool

class Int
  "=" (x : Self) : Bool
  "+" (x : Self) : Self
  "-" (x : Self) : Self
```

Figure 7.2.12 : Matching Classes in TooL

Class *Int* matches *Equality* in the example in Figure 7.2.12 above. A subtype relationship is not possible between the two classes due to the contravariant occurrence of *Self* in the method signature of "=". Matching allows the following generic method, which tests the equality of two objects and returns the negated result, to be defined as shown in Figure 7.2.13 below.

```
"!=" (T <*: Equality, x : T, y : T)
  { ! (x = y) }
```

Figure 7.2.13 : Method Definition using Matching in TooL

The only necessary condition is that both arguments are of some type T that matches *Equality*. A generic method defined using subtyping instead of matching will not accept objects of class *Int* as arguments even though it is type safe to do so.

7.2.8 Type Checking self

In PolyTOIL, any subclass will always match its superclass but is not necessarily a subtype. Subtypes always match. If the only occurrences of ***MyType*** in method types are covariant then the two types match iff they are subtypes. Methods of an object are type checked under the assumption that ***MyType*** matches that object type. Though more method bodies can be type checked if it is assumed that ***MyType*** is the same as the object type, they will not be type correct when inherited. If object o of type S has a method m of type T then o.m has type T[S/***MyType***]. The type of the message send in Figure 7.2.14

below will be *proc(DbleNodeType)* where *DbleNodeType* is the object type corresponding to *DbleNode*.

```
var aDbleNode, anotherDbleNode: DbleNodeType;
. . .
aDbleNode := new (DbleNode);
. . .
aDbleNode.setNext(anotherDbleNode)
```

Figure 7.2.14 : Typing Method Calls in PolyTOIL

During type checking, method bodies will have to be checked with some assumption about the type *Self.* TooL allows the programmer explicit control over this assumption. *Self* can be match-bounded, subtype-bounded or equivalent. These conditions can be specified as demonstrated in Figure 7.2.15.

```
class Equality              ! match bound
Self <*: class Equality     ! match bound
Self <: class Equality      ! subtype bound
Self = class Equality       ! equivalent
```

Figure 7.2.15 : Binding Self in TooL

The default is taken to be match-bound. If *Self* is explicitly subtype-bound then the subclasses of the class will always be subtypes. If *Self* is match-bound then subclasses will match the superclass. If *Self* is specified to be equivalent then any subclasses will have to have exactly the same type as the superclass.

Self constraints assumed during modular type checking are enforced when actual subclassing takes place. Consider the possible cases when *Self* is bound to a class C. If *Self* is bound by matching then method signatures of C are copied into the subclass. If it is bound by subtyping then method signatures of C are copied into the subclass and all inherited occurrences of *Self* are replaced by C. If *Self* is equivalent to C then method signatures of C are copied into the subclass and a check is performed to make sure no additional methods are defined or refined in the subclass.

7.2.9 Adding Bounded Polymorphism

Polymorphism is essential for cleanly expressing data structures such as container classes and operations over them. Container classes represent

collections of objects, usually of the same or related types such as a list of elements. The similarity in different implementations of a list for different types of elements can be captured by polymorphic code.

Some container classes require the support for a minimum set of operations from the types of elements they can represent. A binary search tree will need elements whose types will support comparisons. These operations can be contained in a PolyTOIL object type as specified in Figure 7.2.16.

```
Comparable = ObjectType
    equal       : Func(MyType): Boolean;
    greaterThan: Func(MyType): Boolean;
    lessThan    : Func(MyType): Boolean
end
```

Figure 7.2.16 : Operations for Comparisons in PolyTOIL

The language provides a construct to express the dependency that any type which is kept in the binary search tree should support at least these operations. It will not suffice to say that the element type has to be a subtype of *Comparable* since there are contravariant occurrences of ***MyType*** in method types and hence no non-trivial subtype can be found. On the other hand, matching can be used to express this dependency precisely.

If some type T <# *Comparable* then T will have at least these 3 methods each with a function type which takes an argument of the same type as the receiver and returns a Boolean. This mechanism for restricting type parameters using matching is called bounded matching. The binary search tree example can then be completed as shown below in Figure 7.2.17

```
BTreeNode = class(T <# Comparable; v: T)
    var
        value = v: T;
        left  = nil: MyType;
        right = nil: MyType
    methods
        function getValue(): T {return value}
        . . .
end class


BinSearchTree = class(T <# Comparable)
    var
        root = nil: BTreeNodeType(T)
    methods
        function find(elmt: T): Boolean { . . . }
        procedure insert(newElmt: T) {. . . }
        function isEmpty(): Boolean {return (root = nil)}
end class
```

Figure 7.2.17 : Binary Search Tree using Matching in PolyTOIL

Bounded polymorphism in TooL is similar to form it takes in PolyTOIL. An example of a polymorphic function bounded by matching is given in Figure 7.2.13.

7.2.10 Reconciling Subtyping, Matching and Quantification

Since a given piece of TooL code can refer to many types and type variables each of which can be bound by either matching or subtyping, it is deemed important to have type rules that refer to both matching and subtyping lattices.

The following rule states that 'within a static context S, a type variable X is a subtype of a given type T, if within the same context, X matches an object type with method suite M (written Object Type (Self) M) and it can be proved that this object type is a subtype of T whereby all occurrences of Self within M have been replaced by X'.

$$\frac{S \vdash X \ <*: \text{ObjectType (Self) M} \quad S, X \ <: T \vdash \text{ObjectType (Self) M}\,[X / \text{Self}] \ <: T}{S \vdash X \ <: T}$$

This rule provides a safe conservative approximation of the proof steps, taken by the compiler, if the exact type structure of X is known. This rule is necessary to prove, for example, that X is a subtype of *Void* in Figure 7.2.18 and to ensure type safety.

```
class EqualitySet (E <*: Equality)
super Set (E)
  includes (x : E) : Bool {elements.some (fun (e : E) {e = x})}
```

Figure 7.2.18 : An Example

7.2.11 Replacing Subtyping by Matching

The systems described so far are static typing systems that support replacing of methods by subtypes in subclasses, automatic updating of parameter types in special cases and safe uses of covariance in parameter types. However, this flexibility also results in increased complexity. Bounded polymorphism is necessary to achieve expressibility but the bounds represented by matching are more useful in this paradigm than those represented by subtyping. Matching is claimed to be simpler and more natural. Moreover, the difference between the two relations is quite subtle, depending only on the absence of contravariant occurrences of ***MyType***.

Therefore, PolyTOIL seeks to replace subtyping with a generalised form of matching to reduce complexity in the type system. In order to achieve this, the matching relation has to be made primitive i.e. independent of the subtyping relation. Its definition can then be refined to

$$\text{ObjectType}\{m_i : T_i\}\ 1 \le i \le k\ \ <\#\ \text{ObjectType}\{m_i : T_i\}\ 1 \le i \le n \quad \text{iff } n \le k$$

This is a more restrictive definition that does not allow method types to be explicitly changed in subclasses. Thus, object type A matches object type B if A can be obtained by adding more methods to B. The object types of *Person* and *Student1* in Figure 7.2.19 will match while those of *Person* and *Student2* will not.

```
Person = class
    var
        name = aName: NameType1
    methods
        function setName(newName: NameType1)
                                        {name := newName}
        function getName(): NameType1 {return name}
end class

Student1 = class
    inherits Person
    var
        id = 0: Integer
    methods
        function setId(newId: Integer) {id := newId}
        function getId(): Integer {return id}
end class

Student2 = class
    inherits Student modifying getName
    methods
        function getName(): NameType2 {. . .}
                        ! NameType2 <: NameType1
end class
```

Figure 7.2.19 : Using the New Definition of Matching

All uses of subtyping in parameters which are of object types are replaced by polymorphic functions using bounded matching. A function *setName* to be used with objects of any type that matches *NameType1* can be written

```
function setName(N <# NameType1, newName: N) {. . .}
```

Figure 7.2.20 : A Match-bound Function

A new constructor # is introduced to simplify the notation. If T is a type then #T is also a type. An object will be of type #T if it has any type that matches T, that is

$$\text{if } a : S \text{ and } S <\# T \text{ then } a : \#T$$

This also allows a form of subsumption:

$$\text{if } S <\# T \text{ and } a : \#S \text{ then } a : \#T$$

The previous example can then be written as

```
function setName(newName: #NameType1) {. . .}
```

Figure 7.2.21 : A Refined Definition of setName

The # constructor now gives at least the flexibility of subtyping. Only object types can be annotated with #. Record types could be interpreted as degenerate object types or a notion of matching could be defined for them which simply corresponds to record extension.

The other important use of subtyping is in assignments to variables. To deal with this case, #T is treated as the existential type $\exists t <\# T.t$ (some type t that matches T) to yield the rule:

$$\text{if } x : \#T \text{ is a variable declaration and } e : S \text{ for } S <\# T$$
$$\text{then } x := e \text{ is type correct}$$

It should be noted that #-types are not used as an abbreviation for bounded matching here. They also provide greater flexibility than was possible with subtyping. The following example in Figure 7.2.22 illustrates a possible use of #-types.

```
var myName: #NameType1
. . .
Person = class(NameType <# NameType1, aName: NameType)
    var
        name = aName: NameType
            . . .
    methods
        function setName(newName: NameType)
                                                {name := newName}
        function getName(): NameType {return name}
        . . .
end class
. . .
myName := aPerson.getName
```

Figure 7.2.22 : Using # Types

This extra flexibility does cause a problem in that it is no longer possible to statically determine the exact type of a message send if it involves a binary method. As explained before, if o : S and S <# ObjectType{m : T} then the type of o.m is T[S / **MyType**]. However, if o : #S and T is a binary method then the type of o.m cannot be determined since o's type is only known up to matching. Therefore where binary methods are needed, explicit bounded matching has to be used.

This system does not allow method types to be changed in subclasses. It is possible to deal with this problem by further generalising the matching relation. This would need a matching relation to be defined on function types. It is not incorporated into the language as it is believed that the resulting complexity outweighs the benefits. However, a brief outline of how matching may be extended to function types is given in the next section.

7.2.12 Extending Matching to Function Types

The previous definition of matching does not allow method types to be explicitly changed in subclasses. To remove this restriction, it is necessary to generalise matching by defining a matching relation on function types. The intended definition can be expressed by the following rules:

Func(t <# a) : r <# Func(t <# a') : r iff a' <# a

Func(#a) : #r <# Func(#a') : #r' iff a' <# a and r <# r'

Func(#a) : r <# Func(a') : r iff a' <# a

The definition can then be refined to

$ObjectType\{m_j : T'_j\}_{1 \le j \le m} <\#\ ObjectType\{m_i : T_i\}_{1 \le i \le n}$

if $n \le m$ and for each $i \le n$, $T'_i <\# T_i$

7.3 Other Languages

The subsections below examine how some other well known languages deal with the issues of inheritance and binary methods.

7.3.1 Simula

Simula [DN66, GGH+91] was the first language to support inheritance. It provides single inheritance. The current object can be referred as *this* followed by the qualifying class. Simula provides boolean operators to check whether an object belongs to a class or is an instance of one of its subclasses. Virtual declarations allow objects to access the innermost redefinition of instance variables and methods to be accessed. All subclasses are treated as subtypes of the superclass. If methods are over-ridden in subclasses then the changed methods must be of the same type as those in the superclass. This restriction avoids the problem caused by binary methods.

7.3.2 SmallTalk

SmallTalk-80 [GR83, GGH+91] supports multiple inheritance whilst SmallTalk/V only provides single inheritance. The current object may be referred to as *self.* Methods may be over-ridden in subclasses. However, SmallTalk does not have a static type system. All type errors are detected and dealt with at run time.

7.3.3 Ada

Ada [You84] provides encapsulation through the package mechanism and polymorphism through genericity of program units. However, inheritance and subtyping are not supported by the Ada type system and hence a corresponding problem with binary methods does not arise.

7.3.4 C++

C++ [Str86, GGH+91] supports multiple inheritance. It uses the keyword *this*, without qualification, to refer to the current object. If methods are declared to be virtual in the base class then the redefined methods in derived classes are accessed when they are called from an object of a derived class. C++ does not permit the types of over-ridden methods to be changed in derived classes. This restriction prevents the problem of binary methods. C++ also provides run time type identification which makes use of dynamic type cast to determine the actual type of a class instance.

7.3.5 Java

Java [Fla97] supports single inheritance. However if a class is declared to be final then it can not be extended to form a subclass. The key word *super* is used in subclasses to denote the superclass and the keyword *this* in the body of a method refers to the object through which the method is invoked. Inherited methods may be over-ridden in the subclass by redefinition. An object belonging to a subclass can be used in place of one of the superclass. A dynamic method lookup is used to choose the correct method for a method call.

7.4 Summary

The object oriented languages Eiffel, PolyTOIL and TooL offer type level solutions to the problems caused by type evolution. Eiffel uses a conformance relation to determine when two types are compatible. Although this relation captures the inheritance requirements, it does not guarantee type safety. Therefore Eiffel uses assignment attempts and avoids polymorphic catcalls to ensure that operations are type safe.

TooL and PolyTOIL make use of the matching relation to determine the compatibility of method types. Both these languages use the same definition of the matching relation. Methods of a class type checked under the assumption that any occurrences of *MyType* only match the object type of the class, rather than that *MyType* is the same as the object type of the class. This assumption ensures that inherited methods are type safe in the subclass since the meaning of MyType in any subclass will always match the object type of the superclass.

The other languages considered in this chapter either deal with the problem dynamically or restrict the redefinition of methods in subclasses in order to

avoid the problem. Another type level solution to the problem of type evolution in a more general context is presented in the next chapter.

8 Extension Polymorphism

The problem of schema evolution in databases and how various systems deal with this problem are discussed in chapters 6 and 7. The related problem with covariant inheritance of methods in object oriented programming languages and the different ways in which it has been dealt with in some languages were also described.

Polymorphism offers abstraction over types in programming languages and is therefore an important choice for addressing type evolution. Inclusion polymorphism has often been quoted as a solution. This is mainly due to the fact that one of the most common type changes, the addition of new fields to a record type, corresponds exactly to the subtype relation for records. However, inclusion does not always match the common evolution patterns of other type constructors including functions. The terms subtyping and inheritance are often confused and used interchangeably. This often leads to unsound or at best dynamically checked languages.

A new mechanism for dealing with type evolution in persistent systems is described in this chapter. The aim of this work is to provide explicit language level support for type evolution. Instead of attempting to adapt a subtyping mechanism to correspond to evolution, the changes to type definitions in evolving persistent systems are investigated from first principles. The most common evolution patterns for the different type constructors are then used to formulate a new form of evolutionary paradigm called extension polymorphism. The underlying hypothesis of this exercise is that by utilising this facility programmers will be able to write code which will continue to be typed correctly (soundly and statically) as the types of data it operates over evolve.

Extension closely models the process through which type definitions evolve by adding more components or refining existing components. This form of evolution is commonly referred to as additive evolution. The notion of extension is formalised by defining an extension relation between types. Appropriate language mechanisms are then derived to implement polymorphism over this relation. The experiment is structured in three main phases:

- defining a formal model of evolutionary patterns
- deriving language mechanisms to support this model
- implementing these mechanisms and using them for applications

The formal model and the language mechanisms are discussed in detail in this chapter, a language supporting extension polymorphism is defined and its soundness proved in Chapter 9 and issues of implementation are discussed in Chapter 10.

Although extension is defined independently of any existing strategies for capturing type evolution, it does not necessarily disallow any other polymorphism mechanism from co-existing in the same system. For instance, there is some interesting interaction between extension and inclusion polymorphism which is further explored later in this chapter.

8.1 Examples of Extension in Persistent Systems

The patterns of additive evolution for different type constructors in evolving applications in a persistent programming environment [MBC+87, MBC+93] are given in this section. In each case the extent to which inclusion polymorphism can model the changes mentioned is also described. The core language Base, defined in chapter 2, is used for examples of code. In order to make it easy to write down more interesting examples, a base type called **string**, which is a string of characters, is added to Base and functions are allowed to have more than one argument but it remains unchanged otherwise. It is important to remember here that Base does not support any form of polymorphism.

The definition of refinement in this context is recursive. Whenever a type is said to evolve by refining a component type, it can be assumed that the component type evolves in a manner consistent with the general pattern of evolution described in sections 8.2.1 to 8.2.4.

8.1.1 Records

Record types typically evolve by the addition of more fields or by the refinement of existing fields.

```
type employee is { name : string ; addr : address }

type employee2a is { name : string ; addr : address ;
                     scale : loc( int )                }

type employee2b is { name : string ; addr : address2 }
```

Figure 8.1 : Record Evolution

In the examples in Figure 8.1 above, type *employee* might evolve by addition of the field *scale* to *employee2a* or by refinement of the field *addr* to *employee2b*.

The evolution from *employee* to *employee2a* can always be modelled by inclusion. If *address2* is a subtype of *address* then the evolution to *employee2b* is also captured by inclusion.

8.1.2 Variants

Variants evolve in two possible ways : the addition of a branch and the refinement of the type of an existing branch.

```
type grade is [ S : senior ; I : intermediate ; J : junior ]

type grade2a is [   S : senior ; I : intermediate ; J : junior ;
                    B : beginner                                ]

type grade2b is [   S : senior ; I : intermediate2 ; J : junior ]
```

Figure 8.2 : Variant Evolution

In Figure 8.2, type *grade* may evolve to *grade2a* by the addition of branch B or to type *grade2b* by the refinement to the type of branch I.

Since *grade2a* contains an additional branch, it is not a subtype of *grade* and hence this evolution can not be modelled by inclusion. The only case where the inclusion relation captures additive evolution in variants is when a variant type evolves by simply refining a branch type and the new branch type is a subtype of the previous one. For example, if *intermediate2* is a subtype of *intermediate* then the evolution from *grade* to *grade2b* can be captured by inclusion. In all other cases, inclusion does not match additive evolution.

8.1.3 Functions

Function types evolve by refining their argument and result types. Thus, in Figure 8.3, the function type *getAddr* can evolve to *getAddr2*, *getAddr3* and *getAddr4* by the refinement of the argument type, the result type and both types respectively.

type getAddr **is fun** (employee → address)

type getAddr2 **is fun** (employee2 → address)

type getAddr3 **is fun** (employee → address2)

type getAddr4 **is fun** (employee2 → address2)

Figure 8.3 : Function Evolution

The evolution of function types in *getAddr2* and *getAddr4* coincide with the notion of covariant subtyping. The parameter type becomes more specialised as the function type evolves. Inclusion polymorphism cannot capture these changes since the subtyping rule for functions requires argument types to become more general in a subtype.

Another interesting possibility is the evolution of a function type by the addition of one or more argument types. Consider the function types in Figure 8.4 below.

type averageFun **is fun** (**int**, **int** → **real**)

type averageFun2 **is fun** (**int**, **int**, **int** → **real**)

Figure 8.4 : Evolution of Functions by Addition of Arguments

averageFun can represent the type of a function which takes two integers and returns their average as a real number. If the user now wishes to refine this function so that it calculates the average of three integers then the type of the new function can be represented by *averageFun2.* Thus, the function type *averageFun* has evolved by adding another argument.

Functions which can support a varying number of arguments are called variadic functions. If more arguments than necessary are supplied to such a function then it ignores the additional ones. If fewer arguments than necessary are given, then default values are assumed for the arguments without actual parameters. For example, assume that both the function types in Figure 8.4 represent variadic functions. A function of type *averageFun2* can then be used in place of a function of type *averageFun* with 0 as the default value for the third argument. Conversely a function of type *averageFun* may be used in place of one of type *averageFun2* by ignoring the third actual parameter.

This facility provides a mechanism for dealing with the evolution of function types by the addition of arguments. However, in order to simplify the resulting type system, this form of evolution is not supported by the extension relation defined later in this chapter.

8.1.4 Locations

Mutability is explicitly modelled in Base by the ***loc*** type constructor. The evolution of location types is the result of the evolution of the type of the value contained in the location.

type dept **is** { manager : **loc** (employee) ; workers : **int** }

type dept2 **is** { manager : **loc** (employee2) ; workers : **int** }

type dept3 **is** { manager : **loc** (employee) ; workers : **loc** (**int**) }

Figure 8.5 : Location Evolution

In Figure 8.5, *dept* evolves to *dept2* when the type of the *manager* field is refined from location of *employee* to location of *employee2*. Another commonly observed change is when a component type which is not originally not defined to be mutable to become so. The refinement from *dept* to *dept3* illustrates this case.

Inclusion polymorphism allows only trivial subtyping over location types (that is **loc** (A) ≤ **loc** (A)) in a language which supports an explicit location dereference operation. If dereferencing locations is performed implicitly then the additional subtyping rule **loc** (A) ≤ A may also be allowed.

8.2 The Extension Relation

Based on the patterns in the previous section, the extension relation can now be formally defined. The notation A ← B is used to denote that type A is extended from type B. This means that A may be formed by extension from B.

8.2.1 Reflection

T ← T for any type T R1

Any type is an extension of itself.

8.2.2 Base Types

$T \leftarrow B$ iff $T = B$ for any base type B R2

Any type extended from a base type is the same as that base type, that is only trivial extension, shown in the previous section, is permitted over base types.

8.2.3 Records

$\{ l_i : T_i \}_{(i = 1, n)} \leftarrow \{ l'_j : T'_j \}_{(j = 1, m)}$

iff $m \leq n$ and for k in 1 . . m, $l_k = l'_k$ and $T_k \leftarrow T'_k$ R3

A record type A is an extension of another record type B if and only if A has at least the fields of B and, for the common fields in both types, the labels are the same and the corresponding field types are in extension relation.

8.2.4 Variants

$[l_i : T_i]_{(i = 1, n)} \leftarrow [l'_j : T'_j]_{(j = 1, m)}$

iff $m \leq n$ and for k in 1 . . m, $l_k = l'_k$ and $T_k \leftarrow T'_k$ R4

A variant type A is an extension of another variant type B if and only if A has at least the branches of B and, for the common branches in both types, the labels are the same and the corresponding branch types are in the extension relation.

8.2.5 Functions

fun ($p \rightarrow q$) $\leftarrow$ **fun** ($p' \rightarrow q'$) iff $p \leftarrow p'$ and $q \leftarrow q'$ R5

A function type A is extended from another function type B if and only if the argument type of A is extended from the argument type of B and the result type of A is extended from the result type of B.

8.2.6 Locations

loc (p) $\leftarrow$ **loc** (p') iff $p \leftarrow p'$ R6

loc (p) $\leftarrow$ p R7

A location type A is an extension of another location type B if the type of values A contains is an extension of the type of values contained by B. A location type is an extension of the type of value it contains.

8.3 Adding Polymorphism

Given the formal definition of the extension relation specified in the previous section, polymorphism mechanisms over the relation which will allow the static typing of extended types may now be defined.

8.3.1 Typing Extension Variables

The first step towards obtaining polymorphism is to derive a typing for extension variables. This requires a mechanism that will allow expressions to be typed as 'some type that is extended from T' where T is any legal type in the language.

One way to achieve this typing is to introduce ← as an explicit type modifier, allowing the syntactic notation ←T to signify extension from some type T. Consider defining a *getEmpName* function, which takes an *employee* and returns the value of its *name* field. If this function is required to operate over not only *employee* but any type extended from it then it can be specified as shown in Figure 8.5.

```
let getEmpName = fun ( e : ← employee → string ) e.name
```

Figure 8.5 : Typing Extension

However, this notation causes problems with structural type equivalence as the example in Figure 8.6 below shows.

```
let someFun = fun ( e1 : ← employee ; e2 : ← employee )
```

Figure 8.6 : Equivalence of Two Types

In Figure 8.6, even though *e1* and *e2* are both typed as ←*employee*, they can not be assumed to share the same type. The only condition required by this typing is that *e1* is of type which is extended from *employee* and *e2* is of a type which is also extended from *employee*. However this does not imply that they are the same type. In order to specify the relationship that two parameters share the same type in this context, a precisely typed algebra is needed.

To overcome this problem, a new typing based on the notion of extension type variables is introduced. In addition to the type and value environments τ and π introduced in section 2.4, a third environment called ε is created for these

variables. ε contains judgements of the form t ← T where t is any type identifier and T is any type in the language, including the ones composed from other extension type variables. Extension type judgements in the language are restricted to those of the form e : t where t is a member of ε. Two expressions which are typed by extension type variables are deemed to be type equivalent if and only if they share the same type variable.

There are two ways of introducing extension type variables: explicit introduction through the use of a form of bounded universal quantification and implicit introduction through the application of operations to expressions already typed with extension variables.

8.3.2 Explicit Extension Variables

A form of bounded universal quantification is used to introduce extension type variables explicitly. However it is based on extension rather than inclusion. An example of the use of this construct is given below in Figure 8.6.

```
let changeManager = fun [ t ← dept ] ( d : t , e : employee → unit )
                                . . .
```

Figure 8.6 : Quantification using Extension

The term quantifier variable is used to refer to type identifiers introduced by bounded quantification, such as *t* in Figure 8.6. The meaning of *t*, as before, is some type that is extended from *dept*.

To illustrate the point made about the typing of quantifier variables in the previous section, consider the function *sameAddr* in Figure 8.7 below.

```
let sameAddr = fun [ emp ← employee ] ( e1, e2 : emp → bool )
                                . . .
```

Figure 8.7 : Typing Quantifier Variables

Even though the structure of *emp* may not be known exactly, the values denoted by *e1* and *e2* can safely be assumed to share the same type.

An important difference between this mechanism and bounded universal quantification is that this type abstraction does not imply inclusion. With universal quantification bounded by inclusion the type quantifier always stands for a type which is a subtype of the bound type. Therefore any operation defined

over the bound type can be safely applied to values of the quantifier variable. However, with extension polymorphism, the types denoted by the quantifier variable are not intended to be in the subtyping relation with the bound type. As a result, it is not always type safe to apply operations defined for the bound type on the values of the quantifier variable. Figure 8.8 gives an example of this case.

```
! per1 : loc ( person )

let assignPer = fun ( p1 : person ) per1 := p1

let polyAssign = fun [ per ← person ] ( p1 : per ) per1 := p1
```

Figure 8.8 : Operations on Bound and Quantifier Types

In Figure 8.8, the identifier *per1* is of type ***loc****(person)* where *person* is not a base type. The function *assignPer* takes a value of type *person* and assigns it to *per1*. According to the typing rule for assignment this is type correct. Now consider defining a polymorphic function *polyAssign* which performed the assignment for values of any type that is extended from *person*. If it is defined as shown in Figure 8.8 then the body of this function will not be type correct. This is because an exact type match is required for assignment in order to maintain type safety. It should be noted that in a system that supports subtyping and subsumption if the bound specified is for inclusion as shown below in Figure 8.9 then the body of *polyAssign* will type check.

```
let polyAssign = fun [ per ≤ person ] ( p1 : per ) per1 := p1
```

Figure 8.9 : Bounded Universal Quantification with Inclusion

As in the case of the example in Figure 8.8, values belonging to the quantifier variable in a universally quantified function can not claim all the operations permitted for the bound type. The exact operations that are available to quantifier variables are examined in section 8.4.4.

Thus the typing of these variables is more restrictive than for those introduced by bounded universal quantification based on inclusion. This is to ensure the soundness of the type system and to provide exact type information wherever possible.

The ability to relate quantifiers increases the expressiveness of the system. Consider the example in Figure 8.10 below.

```
let aFun = fun [ e ← employee, d ← { manager : loc( e ) } ]
                    ( p : e, t : d → e )
                t.manager := p
```

Figure 8.10 : Related Quantifier Variables

The definition of the quantifier variable *d* depends on the other quantifier variable *e*. *e* stands for any type extended from *employee* and *d* stands for any type extended from a record type that has a field *manager* of type ***loc(e)***. The semantics of extension over quantifier variables should be clarified to provide a clear meaning for *d* in this context. In order to achieve a useful abstraction only trivial extension is allowed over quantifier variables. This means that for a call to *aFun*, defined in Figure 8.10, to be correctly typed, the two specialising instances of *e* must be the same, rather than being in extension relation. Figure 8.11 gives some examples.

```
type dept is { manager : loc ( employee ) ; workers : int }

type dept2 is { manager : loc ( employee2 ) ; workers : int }

!   e1 : employee ; e2 : employee2  where employee2 ← employee
!   d1 : dept ;  d2 : dept2

let a = aFun [ employee, dept ] ( e1, d1 )

let b = aFun [ employee2, dept2 ] ( e2, d2 )

let c = aFun [ employee, dept2 ] ( e1, d2 )
```

Figure 8.11 : Using Related Quantifier Variables

The first two function calls in Figure 8.11 are correctly typed since the actual parameter supplied for *e* is the same in both instances of the quantifier variable. But the third call to *aFun* is incorrect as *dept2* has the component type *employee2* but *employee* has been supplied as the actual parameter for the first type variable.

8.3.3 Implicit Extension Variables

Unlike explicit extension variables which are introduced by the programmer, implicit extension variables are internally introduced by the type checker. Given the definition of type *employee* in previous sections, consider the following example in Figure 8.12.

```
let derefAddr = fun [ emp ← employee ] ( e : emp )
begin
   let x = e.addr
end
```

Figure 8.12 : Implicit Extension Variables

The quantifier variable *emp* stands for any type that is extended from *employee*. Therefore the type *e.addr* is not *address* but some type that is extended from *address*. When determining the type of variable *x*, the type checker will create a new identifier for this extension variable and add the extension binding to ε.

The restrictions on extension, for example allowing only trivial extension over base types and extension variables, sometimes permit the type checker to coerce new implicit extension variables to known types.

```
let changeManager = fun [ emp ← employee,
                                dep ← { manager : loc ( emp ) } ]
                            ( e: emp, d : dep )
                        d.manager := e
```

Figure 8.13 : Location Evolution

In Figure 8.13, the type checker first derives the type of *d.manager* as a new extension variable bounded by ***loc (emp)***. The type of the content of this location is derived as another new extension variable bounded by *emp* and this type can then be deduced to be *emp* itself since only trivial extension is allowed, thus typing the two sides of the assignment as being equivalent.

8.3.4 Polymorphism over Type Constructors

We now present a case by case analysis of the use of extension polymorphism with different type constructors. In each case, the operations allowed upon the abstracted form are also discussed.

8.3.4.1 Base Types

As stated earlier, only trivial extension is allowed over base types. Therefore the abstracted form can be coerced to the bound type and all operations defined over the bound type can also be used for the extended type.

```
type dept3 is { manager : loc ( employee ) ; workers : loc( int ) }

let incWorkers = fun [ dep ← dept3 ] ( d : dep )
                        d.workers := @( d.workers ) + 1
```

Figure 8.14 : Operations on Extended Base Types

For example, the definition of the function *incWorkers,* which increments the number of workers by one, in Figure 8.14 above is correctly typed since the type of *@d.workers* can be guaranteed to be exactly ***int***.

8.3.4.2 Records

Since extension and inclusion relations are the same for records, in their case too all the operations defined for the bound type are available for the abstraction.

```
type dept is { manager : loc ( employee ) ; workers : int }

let getWorkers = fun [ dep ← dept ] ( d : dep → int ) d.workers
```

Figure 8.15 : Operations on Extended Records

In Figure 8.15 above, the operation to dereference the *workers* field is available to any type that is extended from *dept*. In general, any dereference operation that is applicable to a record type is also applicable to any type that is extended from it.

8.3.4.3 Variants

An extended variant type may belong to any of the branches of the bound or some new branch of the extended type which is not known statically. A straight forward solution to this problem is to restrict the use of the abstracted variant values to within a multi-branch case project statement terminated by a default clause.

```
type grade is [  S : senior ;
                 I : intermediate ;
                 J : junior          ]

let projGrade = fun [ newGrade ← grade ] ( g : newGrade )
begin
   let ans = project g as X onto
      S :  X ;
      I :   X ;
      J :  X ;
   default : X
end
```

Figure 8.16 : Operations on Extended Variants

In Figure 8.16, function *projGrade* operates over any type extended from the variant type *grade*. If the value of extended type provided as actual parameter to a call to *projGrade* belongs to an additional branch of the extended type then it will be captured by the default clause.

8.3.4.4 Functions

Consider the example in Figure 8.17 below.

```
let polyFun = fun [ aFun ← fun ( X → Y) ] ( f : aFun ) ;
                  . . .
```

Figure 8.17 : Operations on Extended Functions

In general, function *f*, supplied as parameter to *polyFun* can not be applied. The only information on the parameter type is that it is extended from *X* and therefore *f* could only be safely applied when there are no extensions on *X*, for example when *X* is a base type. But in these cases, the quantified function is not particularly useful.

However, extended function types can be used in a meaningful manner when related quantifiers are used.

```
let empPoly = fun [ empFunRec ← { empFun : fun (emp ) },
                    emp ← employee ] ( f : empFunRec, e : emp ) ;
                    f.empFun ( e )
```

Figure 8.18 : Operations on Extended Functions - 2

In Figure 8.18 above, the function call in the body of *empPoly* is valid since only trivial extension is permitted over quantifier variables. More generally, if the argument type of an extended function can be statically determined to be a known type then the function can be applied in a type safe manner to a value of this type.

8.3.4.5 Locations

Updates to locations are only allowed when the left hand side of the assignment statement may be statically deduced to be the location type of the right hand side.

```
let changeManager = fun [ emp ← employee,
                          dep ← { manager : loc ( emp ) } ]
                        ( e: emp, d : dep ) ; d.manager := e
```

Figure 8.19 : Operations on Extended Locations

In Figure 8.19, for reasons explained in section 8.4.3, the type of the expression *d.manager* can be statically determined to the location type of the type of *e*. Therefore this update is valid.

Dereferencing a value of an extended type is always allowed though care must be taken when specifying the type that is returned.

```
type dept is { manager : loc ( employee ) ; workers : int }

let DerefManager = fun [ dep ← dept ] ( d : dep ) ;
                       let m = @( d.manager )
```

Figure 8.20 : Dereferencing Extended Locations

In the example in Figure 8.20, the type of *m* will be *t*, where *t* is some type extended from *employee*.

8.3.5 Quantified Functions

Since quantified functions have been introduced as a new type constructor, the definition of the extension relation needs to be modified to include the following rule:

fun [t ← X] (p → q) ← **fun** [t ← Y] (p' → q') iff X ← Y, p ← p' and q ← q'

A quantified function type A is extended from another quantified function type B if and only if the quantifier bound of A is extended from the quantifier bound of B, the argument type of A is extended from the argument type of B and result type of A is extended from the result type of B.

The work done on these functions so far indicates that useful abstractions by extension over quantified functions can only be obtained by using related quantifiers. However, the importance of allowing only trivial extension over quantifier variables should be stressed here.

8.4 Interaction with Other Kinds of Polymorphism

Incorporating extension polymorphism into a language does not preclude the use of other kinds of polymorphism in that language. At the risk of complicating the type system of the language and obscuring the syntax for various abstractions, extension polymorphism can co-exist, for example, with universal and inclusion polymorphism.

There are some interesting interactions between extension polymorphism and inclusion polymorphism with subsumption described in chapter 5. In order to explain the possible interactions, the concepts of *positive* and *negative* types are first introduced. These describe subsets of the type space which have the following properties:

positive = {A ∈ Type | X ← A implies X ≤ A}

negative = {A ∈ Type | X ← A implies A ≤ X}

For example, *aRec*, defined in Figure 8.21, is positive as any type extended from it is also a subtype of it.

```
type aRec is { a : int ; b : { c : bool ; d : real } }
```

Figure 8.21 : Positive Interaction

On the other hand *aVar* in Figure 8.22 is negative.

```
type aVar is [ a : int ; b : [ c : bool ; d : real ] ]
```

Figure 8.22 : Negative Interaction

Notice that these sets, which are not disjoint, both include many non-trivial members. It is relatively straightforward to characterise the notions of positive and negative by examination of the combination of the structures of the inclusion and extension relations over each type constructor. The result is that some statically known extension relationships may be used to imply inclusion relationships, and therefore the application of the subsumption rule thus allowing type safe substitution.

8.5 Summary

A new mechanism for dealing with type evolution in persistent systems at the type level has been described. The concept of extension seeks to capture additive evolution which is one of the most common forms of evolution in persistent systems. An extension relation which formalises this concept is defined for all the type constructors introduced in Base. A programming language mechanism which provides polymorphism over this relation is presented. Finally possible patterns of interaction between inclusion and extension polymorphism are examined. Chapter 9 presents a language extended from Base that supports extension polymorphism and provides a proof of soundness for its type system.

9 A Language with Extension Polymorphism

The core language Base, defined in Chapter 2, is used as a starting point to incorporate extension polymorphism. The resulting language is called Ext. The additions and changes to Base which are needed for implementing extension polymorphism are described in sections 9.1 to 9.6 and a proof of soundness for the new type system is given in section 9.7.

9.1 Types

The only change required to the definition of types is the addition of quantified functions.

T ::= . . . | **fun** [t ← T] (T → T) | . . .

These functions are quantified by specifying a quantifier variable and the extension bound type both within square brackets and separated by a left arrow. The argument and result types as usual are given within round brackets separated by a right arrow.

9.2 Expressions

The new expressions in the language can be defined by the following syntax :

E ::= . . . | **fun** [t ← T] (x : T → T) E | E [T] (E) | . . .

where

Syntax	Interpretation
fun [t ← T] (x : T_1 → T_2) E	quantified function value
E_1[T] (E_2)	quantified function application

Figure 9.1 : New Expressions in Ext

A function value is created by specifying extension type variable and the extension type if the function is quantified and the formal parameter, argument and result types and the expression for function body. A function can be applied by supplying it with the extension type within square brackets and an actual parameter for the argument within round brackets.

9.3 Typing Rules

In addition to the environments τ and π introduced in Chapter 2 for holding type and identifier bindings, a new environment ε is added here to contain extension bindings. The pair < T_1, T_2 > in ε indicates that type T_1 is extended from type T_2. A new declaration function *extDecl* is defined to take a list of bindings as its argument and update ε with the new bindings. In sections 9.3 through to 9.7, the convention adopted for naming extension variables that will be introduced by the type checker is that a variable that stands for a type extended from type T will be denoted by t. It should be noted that only the new rules that are required to incorporate the effects of adding extension polymorphism are presented in this section. The type rules for the other constructs are the same as those described in Chapter 2 for the type system of Base.

9.3.1 Base Types

$$\frac{\varepsilon_1 :: t \leftarrow \textbf{int} :: \varepsilon_2,\ \tau,\ \pi \vdash e : t}{\varepsilon,\ \tau,\ \pi \vdash e : \textbf{int}} \qquad \text{[intExt]}$$

If ε with a binding <t, **int**>, τ and π and imply that an expression e is of type t then e can be deduced to be of type **int**.

$$\frac{\varepsilon_1 :: t \leftarrow \textbf{bool} :: \varepsilon_2,\ \tau,\ \pi \vdash e : t}{\varepsilon,\ \tau,\ \pi \vdash e : \textbf{bool}} \qquad \text{[boolExt]}$$

If ε with a binding <t, **bool**>, τ and π and imply that an expression e is of type t then e can be deduced to be of type **bool**.

$$\frac{\varepsilon_1 :: t \leftarrow \textbf{unit} :: \varepsilon_2,\ \tau,\ \pi \vdash e : t}{\varepsilon,\ \tau,\ \pi \vdash e : \textbf{unit}} \qquad \text{[unitExt]}$$

If ε with a binding <t, **unit**>, τ and π and imply that an expression e is of type t then e can be deduced to be of type **unit**.

9.3.2 Records

$$\frac{\varepsilon_1 :: s \leftarrow \{ l : T \}^+ :: \varepsilon_2,\ \ \tau,\ \ \pi \vdash e : s}{\varepsilon_1 :: s \leftarrow \{ l : T \}^+ :: \varepsilon_2 :: t \leftarrow T :: \varepsilon_3,\ \tau,\ \pi \vdash e.l : t} \qquad \text{[recExtDeref]}$$

If, from ε with a binding that implies that s is extended from a record type with at least the field l of type T, τ and π, the expression e can be deduced to be of

type s then the result of the dereference operation on e, `e.l`, is of type t where t is extended from T.

9.3.3 Variants

$$\frac{\varepsilon,\ \tau,\ \pi \vdash e_1 : T \quad \varepsilon,\ \tau,\ \pi \vdash e_2 : T \quad \varepsilon_1 :: v \leftarrow [\ l_i : T_i\ ]^+ :: \varepsilon_2,\ \ \tau,\ \pi \vdash e : v}{\varepsilon_1 :: v \leftarrow [\ l_i : T_i\ ]^+ :: \varepsilon_2 :: t \leftarrow T :: \varepsilon_3,\ \tau,\ \pi \vdash \mathsf{exp}\ : t}$$

[varExtProj]

where exp stands for the project operation on the variant type, **`project`** `e` **`as`** `x` **`onto`** l_i `:` T_i `in` e_1 **`else`** e_2

If expressions e_1 and e_2 are both of type T and if from ε with a binding that implies that v is extended from a variant type with at least a label l_i which is of type T_i, τ and π, the type of e can be deduced to be v then the project operation on e has type t where t is some type that is extended from T.

9.3.4 Quantified Functions

$$\frac{\varepsilon_1 :: t \leftarrow T :: \varepsilon_2,\ \tau,\ \pi_1 :: x : T_1 :: \pi_2 \vdash e : T_2}{\varepsilon,\ \tau,\ \pi \vdash \mathbf{fun}[\ t \leftarrow T\ ](\ x\ :\ T_1 \rightarrow T_2\ )\ e : \mathbf{fun}[\ t \leftarrow T\ ](\ T_1 \rightarrow T_2\ )}$$

[qFunValue]

If ε with a new binding < t, T >, τ and π with a new binding < x, T_1 > imply that expression e is of type T_2 then the quantified function expression **`fun`**`[t ← T] ( x : T1 → T2 ) e` is of type **fun** [t ← T]($T_1 \rightarrow T_2$).

$$\frac{\varepsilon,\ \tau,\ \pi \vdash e : \mathbf{fun}[\ t \leftarrow T\ ](\ T_1 \rightarrow T_2) \quad \varepsilon \vdash T_3 \leftarrow T \quad \varepsilon,\ \tau,\ \pi \vdash e_1 : T_1[\ T_3\ /\ t\ ]}{\varepsilon,\ \tau,\ \pi \vdash e\ [\ T_3\ ]\ (\ e_1\ ) : T_2[\ T_3\ /\ t\ ]}$$

[qFunApp]

If expression e is of a quantified function type **fun** [t ← T]($T_1 \rightarrow T_2$), type T_3 is extended from type T and expression e_1 is of type T_1 with any occurrences of t replaced by T_3 then the expression `e[ T3 ]( e1 )` is of type T_2 with any occurrences of t in it replaced by T_3.

9.3.5 Locations

$$\frac{\varepsilon_1 :: s \leftarrow \textbf{loc}(\ T\ ) :: \varepsilon_2,\ \tau,\ \pi \vdash e : s}{\varepsilon_1 :: s \leftarrow \textbf{loc}(\ T\ ) :: \varepsilon_2 :: t \leftarrow T :: \varepsilon_3,\ \tau,\ \pi \vdash @e : t} \quad \text{[locExtDeref]}$$

If type s is extended from type **loc**(T) and the type of the expression e can be deduced to be s then the dereference expression @e is of type t where t is some type extended from T.

9.3.6 Infinite Union

$$\frac{\varepsilon_1 :: t \leftarrow \textbf{any} :: \varepsilon_2,\ \tau,\ \pi \vdash e : t}{\varepsilon,\ \tau,\ \pi \vdash e : \textbf{any}} \quad \text{[anyExt]}$$

If ε with a binding <t, **any**>, τ and π imply that e is of type t then e can be deduced to be of type **any**.

9.3.7 Extension Quantifier Variables

$$\frac{\varepsilon_1 :: s \leftarrow t :: \varepsilon_2 :: t \leftarrow T :: \varepsilon_3,\ \tau,\ \pi \vdash e : s}{\varepsilon_1 :: t \leftarrow T :: \varepsilon_2,\ \tau,\ \pi \vdash e : t} \quad \text{[quantExt]}$$

If ε with the two bindings < s, t > and < t, T >, τ and π imply that expression e is of type s then e can be deduced to be of type t.

9.4 Semantic Context

The semantic context for quantified function types is specified in Figure 9.2 given below.

Type	Context	Denoted by
fun[t ← T] ($T_1 \rightarrow T_2$)	the set of quantified functions from ⟦ T_1 ⟧ to ⟦ T_2 ⟧ with ⟦ t ⟧ in both bounded by ⟦ T ⟧	QFunction (⟦ T ⟧, ⟦ T_1 ⟧, ⟦ T_2 ⟧)

Figure 9.2 : Semantic Context for Quantified Functions

9.5 Meta-operations

The meta-operations defined over quantified function types are described in Figure 9.3 below.

Semantic Type	Meta-operations
QFunction (T, T_1, T_2)	mkQFun : (variable, Type, Expression, variable, environment $\rightarrow$ QFunction (T, T_1, T_2)) qApply : (QFunction (T, T_1, T_2), Type, $T_1 \rightarrow T_2$)

Figure 9.3 : Meta-operations on Quantified Functions

Quantified functions have a *make quantified function* meta-operation which takes the quantifier variable, the bound type, the expression for the function body, the parameter variable and the environment in which the expression is to be evaluated and returns a quantified function. The *apply* operation takes a quantified function, a type and a parameter value and returns a value of the result type.

9.6 Semantics

The semantics for quantified functions can now be defined in terms of the meta-operations.

9.6.1 Quantified Functions

$[\![\textbf{fun} [t \leftarrow T] (x : T_1 \rightarrow T_2) e]\!]_{Env} = mkQFun(t, T, e, x, Env)$ D29

$[\![e_1[T] (e_2)]\!]_{Env} = qApply ([\![e_1]\!]_{Env}, T, [\![e_2]\!]_{Env})$ D30

9.7 Proof of Soundness

A proof of soundness of the type system of Ext is given in this section. As before, soundness of typing is proved by structural induction. However, the definition of soundness is extended to incorporate the new extension environment as follows:

$\varepsilon, \tau, \pi \vdash e : T \Rightarrow [\![e]\!]_{Env} \in [\![T]\!]_{\varepsilon, \tau}$

where $\forall i \in Env, \exists i \in \pi . [\![Env.i]\!] \in [\![\pi.i]\!]$

Only those expressions which are affected by extension polymorphism and the new expressions are considered here. The proof for the rest remains the same as

that given in Chapter 4 for Base. The different type constructors with extension are dealt with in Sections 9.7.1 to 9.7.6.

The notation used in the proof is similar to that used in Chapter 4 for Base. Env_b is used to denote Env containing a binding b. The same notation is used for environments ε, τ and π.

9.7.1 Base Types with Extension

Expression	$\varepsilon, \tau, \pi \vdash e : \mathbf{int}$
To be proved	$[\![e]\!]_{Env} \in [\![\mathbf{int}]\!]_{\varepsilon, \tau}$
Inductive Hypothesis	$[\![e]\!]_{Env} \in [\![t]\!]_{\varepsilon_{t \leftarrow \mathbf{int}}, \tau}$
	since $\varepsilon_{t \leftarrow \mathbf{int}}, \tau, \pi \vdash e : t$ by type rule [intExt]

Inductive Step

$[\![e]\!]_{Env} \in [\![t]\!]_{\varepsilon_{t \leftarrow \mathbf{int}}, \tau}$ from hypothesis

Since any type that is extended from a base type is identical to that base type (extension relation definition rule R1),

t = **int**

Therefore, $[\![e]\!]_{Env} \in [\![\mathbf{int}]\!]_{\varepsilon, \tau}$

Expression	$\varepsilon, \tau, \pi \vdash e : \mathbf{bool}$
To be proved	$[\![e]\!]_{Env} \in [\![\mathbf{bool}]\!]_{\varepsilon, \tau}$
Inductive Hypothesis	$[\![e]\!]_{Env} \in [\![t]\!]_{\varepsilon_{t \leftarrow \mathbf{bool}}, \tau}$
	since $\varepsilon_{t \leftarrow \mathbf{bool}}, \tau, \pi \vdash e : t$ by type rule [boolExt]

Inductive Step

$[\![e]\!]_{Env} \in [\![t]\!]_{\varepsilon_{t \leftarrow \mathbf{bool}}, \tau}$ from hypothesis

Since any type that is extended from a base type is identical to that base type (extension relation definition rule R1),

t = **bool**

Therefore, $[\![e]\!]_{Env} \in [\![\mathbf{bool}]\!]_{\varepsilon, \tau}$

9.7.2 Records with Extension

Expression	$\varepsilon, \tau, \pi \vdash e.l : t$ where $t \leftarrow T$
To be proved	$[\![e.l]\!]_{Env} \in [\![t]\!]_{\varepsilon_{S \leftarrow \{ l : T \}^+, t \leftarrow T}, \tau}$

Inductive Hypothesis $[\![e]\!]_{Env} \in [\![s]\!]_{\varepsilon_{s \leftarrow \{ l : T \}^+}, \tau}$

since $\varepsilon_{s \leftarrow \{ l : T \}^+}, \tau, \pi \vdash e : s$ by type rule [recExtDeref]

Inductive Step

$[\![e.l]\!]_{Env} = getL([\![e]\!]_{Env})$ by D15

$[\![e]\!]_{Env} \in [\![s]\!]_{\varepsilon_{s \leftarrow \{ l : T \}^+}, \tau}$ by hypothesis

Since getL takes a Record and returns a value of the type of field l of that record and s is extended from a record type with a field l, getL is applicable to a value of type s and will return a type that is extended from the type of l. Thus,

$getL([\![e]\!]_{Env}) \in [\![t]\!]_{\varepsilon_{s \leftarrow \{ l : T \}^+, t \leftarrow T}, \tau}$

i.e. $[\![e.l]\!]_{Env} \in [\![t]\!]_{\varepsilon_{s \leftarrow \{ l : T \}^+, t \leftarrow T}, \tau}$

9.7.3 Variants with Extension

Expression $\varepsilon, \tau, \pi \vdash$ **project** e **as** x **onto** $l : T_i$ **in** e_1 **else** $e_2 : T$

To be proved $[\![$ **project** e **as** x **onto** $l : T_i$ **in** e_1 **else** $e_2]\!]_{Env} \in$
$[\![T]\!]_{\varepsilon_{v \leftarrow [l : Ti]^+}, \tau}$

Ind. Hypotheses $[\![e]\!]_{Env} \in [\![v]\!]_{\varepsilon_{v \leftarrow [l : Ti]^+}, \tau}$ and

$[\![e_1]\!]_{Env_{x = snd([\![e]\!]Env) \in [\![Ti]\!]}} \in [\![T]\!]_{\varepsilon, \tau}$ and

$[\![e_2]\!]_{Env} \in [\![T]\!]_{\varepsilon, \tau}$

since $\varepsilon_{v \leftarrow [l : Ti]^+}, \tau, \pi \vdash e : v$,

$\varepsilon, \tau, \pi_{x : Ti} \vdash e_1 : T$

and $\varepsilon, \tau, \pi \vdash e_2 : T$ by type rule [varExtProj]

Inductive Step

$[\![$ **project** e **as** x **onto** $l : T_i$ **in** e_1 **else** $e_2]\!]_{Env}$ =
$if(fst([\![e]\!]_{Env}) = l, [\![e_1]\!]_{Env_{x = snd([\![e]\!]Env) \in [\![Ti]\!]}}, [\![e_2]\!]_{Env})$ by D17

But $[\![e]\!]_{Env} \in [\![v]\!]_{\varepsilon_{v \leftarrow [l : Ti]^+}, \tau}$ and $[\![e_1]\!]_{Env\ x = snd([\![e]\!]Env) \in [\![Ti]\!]} \in [\![T]\!]_{\varepsilon, \tau}$
and $[\![e_2]\!]_{Env} \in [\![T]\!]_{\varepsilon, \tau}$ by hypotheses

But $[\![[l : T_i]^+]\!]_{\varepsilon, \tau} = Pair(label, [\![T_i]\!]_{\varepsilon, \tau})$

Since if is a function that takes a Boolean and two expressions of the same type and returns an expression,

$if(fst([\![e]\!]_{Env}) = l, [\![e_1]\!]_{Env_{x = snd([\![e]\!]Env) \in [\![Ti]\!]}}, [\![e_2]\!]_{Env}) \in [\![T]\!]_{\varepsilon, \tau}$

$[\![$ **project** e **as** x **onto** $l : T_i$ **in** e_1 **else** $e_2]\!]$ Env $_{V \leftarrow [l : T_i]^+} \in [\![T]\!]_{\varepsilon, \tau}$

9.7.4 Quantified Functions

Expression $\varepsilon, \tau, \pi \vdash \textbf{fun}[\ t \leftarrow T\ ](\ x : T_1 \rightarrow\ T_2\ )\ e : \textbf{fun}[\ t \leftarrow T\ ](\ T_1 \rightarrow\ T_2\ )$

To be proved $[\![\ \textbf{fun}[\ t \leftarrow T\ ](\ x : T_1 \rightarrow\ T_2\ )\ e\ ]\!]_{Env} \in$
$[\![\ \textbf{fun}[\ t \leftarrow T\ ](\ T_1 \rightarrow\ T_2\ )\ ]\!]_{\varepsilon, \tau}$

Inductive Hypothesis $[\![\ e\ ]\!]_{Env_{x = v : T_1}} \in [\![\ T_2\ ]\!]_{\varepsilon_{t \leftarrow T}, \tau}$

since $\varepsilon_{t \leftarrow T}, \tau, \pi_{x : T_1} \vdash e : T_2$

by type rule [qFunValue]

Inductive Step

$[\![\ \textbf{fun}[\ t \leftarrow T\ ](\ x : T_1 \rightarrow\ T_2\ )\ e\ ]\!]_{Env}$ = mkQFun(t, T, e, x, Env) by D29

$[\![\ e\ ]\!]_{Env_{x = v : T_1}} \in [\![\ T_2\ ]\!]_{\varepsilon_{t \leftarrow T}, \tau}$ by hypothesis

Since mkQFun is a function which takes a variable, a type, an expression, an identifier and an environment and returns a QFunction and from the hypothesis it can be seen that when x is assigned a value of type T_1 and t is extended from T then e will be of type T_2,

mkQFun(t, T, e, x, Env) $\in$ QFunction (T, T_1, T_2)

$[\![\ \textbf{fun}[\ t \leftarrow T\ ](\ T_1 \rightarrow\ T_2\ )\ ]\!]_{\varepsilon, \tau}$ = QFunction (T, T_1, T_2) from table 8.24

i.e. $[\![\ \textbf{fun}[\ t \leftarrow T\ ](\ x : T_1 \rightarrow\ T_2\ )\ e\ ]\!]_{Env} \in [\![\ \textbf{fun}[\ t \leftarrow T\ ](\ T_1 \rightarrow\ T_2\ )]\!]_{\varepsilon, \tau}$

Expression $\varepsilon, \tau, \pi \vdash e\ [\ T_3\ ]\ (\ e_1\ ) : T_2$

To be proved $[\![\ e\ [\ T_3\ ]\ (\ e_1\ )\ ]\!]_{Env} \in [\![\ T_2\ [\ t = T_3\ ]\ ]\!]_{\varepsilon_{T3 \leftarrow T}, \tau}$

Inductive Hypotheses $[\![\ e\ ]\!]_{Env} \in [\![\ \textbf{fun}[\ t \leftarrow T\ ](\ T_1 \rightarrow\ T_2\ )\ ]\!]_{\varepsilon, \tau}$ and
$[\![\ e_1\ ]\!]_{Env} \in [\![\ T_1[\ t = T_3\ ]\ ]\!]_{\varepsilon_{T3 \leftarrow T}, \tau}$

since $\varepsilon_{T3 \leftarrow T}, \tau, \pi \vdash e_1 : T_1\ [\ t = T_3\ ]$ and

$\varepsilon, \tau, \pi \vdash e : \textbf{fun}[\ t \leftarrow T\ ](\ T_1 \rightarrow\ T_2\ )$

by type rule [qFunApp]

Inductive Step

$[\![\ e\ [\ T_3\ ]\ (\ e_1\ )\ ]\!]_{Env}$ = qApply($[\![\ e\ ]\!]_{Env}$, T_3, $[\![\ e_1\ ]\!]_{Env}$) by D30

$[\![\ e\ ]\!]_{Env} \in [\![\ \textbf{fun}[\ t \leftarrow T\ ](\ T_1 \rightarrow\ T_2\ )\ ]\!]_{\varepsilon, \tau}$ and
$[\![\ e_1\ ]\!]_{Env} \in [\![\ T_1[\ t = T_3\ ]\ ]\!]_{\varepsilon_{T3 \leftarrow T}, \tau}$ by hypotheses

Since qApply is a function which takes a QFunction, a type and an Expression of the argument type of the function and returns a value of the result type of the function with any occurrences of the quantifier variable replaced by the type parameter,

$qApply(\llbracket e \rrbracket_{Env}, T_3, \llbracket e_1 \rrbracket_{Env}) \in \llbracket T_2[t = T_3] \rrbracket_{\varepsilon_{T3} \leftarrow T, \tau}$

i.e. $\llbracket e\,[T_3]\,(e_1) \rrbracket_{Env} \in \llbracket T_2\,[t = T_3] \rrbracket_{\varepsilon_{T3} \leftarrow T, \tau}$

9.7.5 Locations with Extension

Expression	$\varepsilon, \tau, \pi \vdash @\ e : t$	
To be proved	$\llbracket @\ e \rrbracket_{Env} \in \llbracket t \rrbracket_{\varepsilon_S \leftarrow \mathbf{loc}(T),\ t \leftarrow T, \tau}$	
Inductive Hypothesis	$\llbracket e \rrbracket_{Env} \in \llbracket s \rrbracket_{\varepsilon_S \leftarrow \mathbf{loc}(T), \tau}$	
	since $\varepsilon_{S \leftarrow \mathbf{loc}(T)}, \tau, \pi \vdash e : s$	
		by type rule [locExtDeref]

Inductive Step

$\llbracket @\ e \rrbracket_{Env} = get(\llbracket e \rrbracket_{Env})$	by D23
$\llbracket e \rrbracket_{Env} \in \llbracket s \rrbracket_{\varepsilon_S \leftarrow \mathbf{loc}(T), \tau}$	by hypothesis

But s is a **loc** type as it is extended from one.

Since get is a function that takes a Location and returns a value of the content type,

$get(\llbracket e \rrbracket_{Env}) \in \llbracket t \rrbracket_{\varepsilon_S \leftarrow \mathbf{loc}(T),\ t \leftarrow T, \tau}$

i.e. $\llbracket @\ e \rrbracket_{Env} \in \llbracket t \rrbracket_{\varepsilon_S \leftarrow \mathbf{loc}(T),\ t \leftarrow T, \tau}$

9.7.6 Infinite Union with Extension

Expression	$\varepsilon, \tau, \pi \vdash e :$ **any**	
To be proved	$\llbracket e \rrbracket_{Env} \in \llbracket$ **any** $\rrbracket_{\varepsilon, \tau}$	
Inductive Hypothesis	$\llbracket e \rrbracket_{Env} \in \llbracket t \rrbracket_{\varepsilon_t \leftarrow \mathbf{any}, \tau}$	
	since $\varepsilon_{t \leftarrow \mathbf{any}}, \tau, \pi \vdash e : t$	by type rule [anyExt]

Inductive Step

$\llbracket e \rrbracket_{Env} \in \llbracket t \rrbracket_{\varepsilon_t \leftarrow \mathbf{any}, \tau}$	from hypothesis

Since any type that is extended from **any** is identical to **any** (extension relation definition rule R1),

t = **any**

Therefore, $\llbracket e \rrbracket_{Env} \in \llbracket$ **any** $\rrbracket_{\varepsilon, \tau}$

Since the soundness of typing has been proved for all expressions affected by the addition of extension polymorphism to a type system that was already proved sound, it has been shown that the resulting type system is also sound.

9.8 Summary

The experimental language Base is extended to incorporate extension polymorphism. A formal definition of the resulting language Ext has been given and the soundness of its type system proved.

10 Type Checking of Extension Polymorphism

This chapter describes a type checker implemented for extension polymorphism. The design decisions made for implementation of the type checker, the type representations chosen and the process of type checking itself are explained.

10.1 Implementation Strategy

10.1.1 Functionality

The main focus here is to implement a type checking algorithm for extension polymorphism. Thus, while lexical analysis, name and scope checking and syntax analysis are needed in order to type check the code in this context, it is not necessary to generate executable code for these programs. For this reason, code generation is omitted from this experiment.

The type representations chosen are similar to those presented in [Con88] for the Napier88 type checking module. The type equivalence algorithm is again based on the algorithm presented in [Con88] and [Con90].

10.1.2 Implementation Procedure

The type checker for extension polymorphism is written in S-algol [Mor79, CM82]. The first step is to implement a type checker for the core language Base. Since Base does not support any form of polymorphism, implementing a type checker for it is a relatively straight forward task. The type checker is then extended to incorporate support for the extension relation and the polymorphism mechanism over it.

The following are the important parts of the implementation:

- type representations
- type equivalence checking
- type extension checking
- dealing with extension quantifiers

Each of the above is discussed in detail in the following sections. Since one of the main aims of the implementation is to check the validity and the possibility of supporting such a polymorphism mechanism, efficiency, in terms of space or time, is not an important consideration during implementation. Thus, for

example, no attempt is made to normalise the graph representation or optimise the type checking algorithms.

10.2 Type Representations

As a first step towards implementing the type checker, type representations are defined for the different type constructors in the language. Since Base contains type constructors of various structures and as space efficiency is not a major goal, a graph representation is used to model types. Therefore types here are represented as directed graphs with nodes representing the various type constructs in the language and edges representing the links between types. The general format for any node is shown in Figure 10.1 below.

Figure 10.1 : General Type Representation

The *label* part of the representation identifies the type construct associated with the type. The table in Figure 10.2 below lists the labels corresponding to the various type constructs in the language.

Type Construct	Type Label
base type	base
record	rec
record field	field
variant	var
variant branch	branch
function	fun
location	loc
any	any
quantifier variable	quant
quantified function	qfun

Figure 10.2 : Type Labels

The *specific-info* part contains any information that is specific to the type being represented, such as the names of base types. Figure 10.3 shows a table listing

the specific information required for each type construct introduced in Figure 10.2.

Type Construct	Specific Information
base type	name of the base type
record	none
record field	field name
variant	none
variant branch	branch name
function	none
location	none
any	none
quantifier variable	a unique identifier
quantified function	none

Figure 10.3 : Specific Information

A quantifier variable requires a unique identifier as its specific information in order to distinguish different instances of the same quantifier.

The *references* part of the type representation contains links to any associated or component types. Whenever there is a list of types being referred to by a node, there is an implicit ordering of the elements in the list which may be used as part of the type information. The table in Figure 10.4 lists the references for various type constructs.

Type Construct	References to
base type	none
record	list of field types
record field	field type
variant	list of branch types
variant branch	branch type
function	list containing parameter and result types
location	type contained in location
any	none
quantifier variable	bound type
quantified function	list containing quantifier, parameter and result types

Figure 10.4 : References

Sections 10.2.1 to 10.2.8 give examples of type representations for every type constructor in the language.

10.2.1 Base Types

Each base type is represented by a single node. For example, the type in Figure 10.5 below is represented by the node shown in Figure 10.6.

type I **is int**

Figure 10.5 : Integer Example

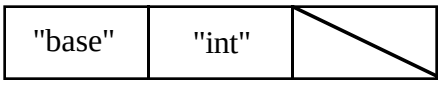

Figure 10.6 : Representation for Integers

10.2.2 Records

Records are represented by a node whose *references* part points to a list of other nodes representing the fields of the records. Each field node has the corresponding field name as its *specificInfo* and its *references* points to the node representing the type of the field. Thus, *person* in Figure 10.7 will be represented by the type graph in Figure 10.8.

```
type Person is { age : int ; graduate : bool }
```

Figure 10.7 : Record Example

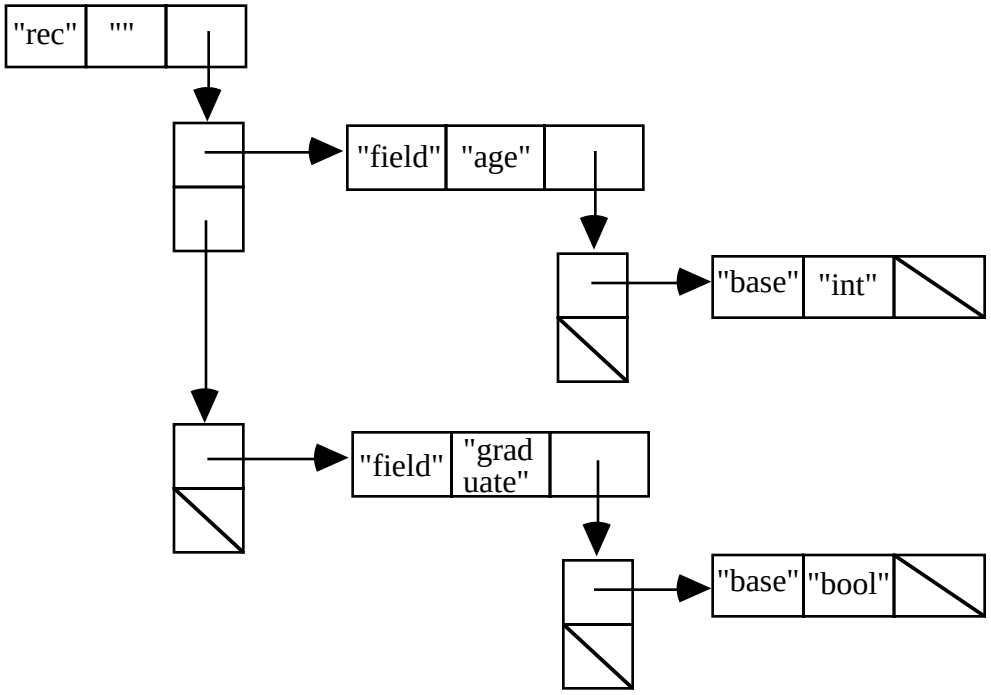

Figure 10.8 : Representation for Records

10.2.3 Variants

The type representation structure of variants is similar to that of records. Their *references* field points to a list of nodes which contain the names of the branches and links to their types. Therefore, the variant type *Base* in Figure 10.9 will be represented as shown in Figure 10.10.

```
type Base is [ I : int ; B : bool ]
```

Figure 10.9 : Variant Example

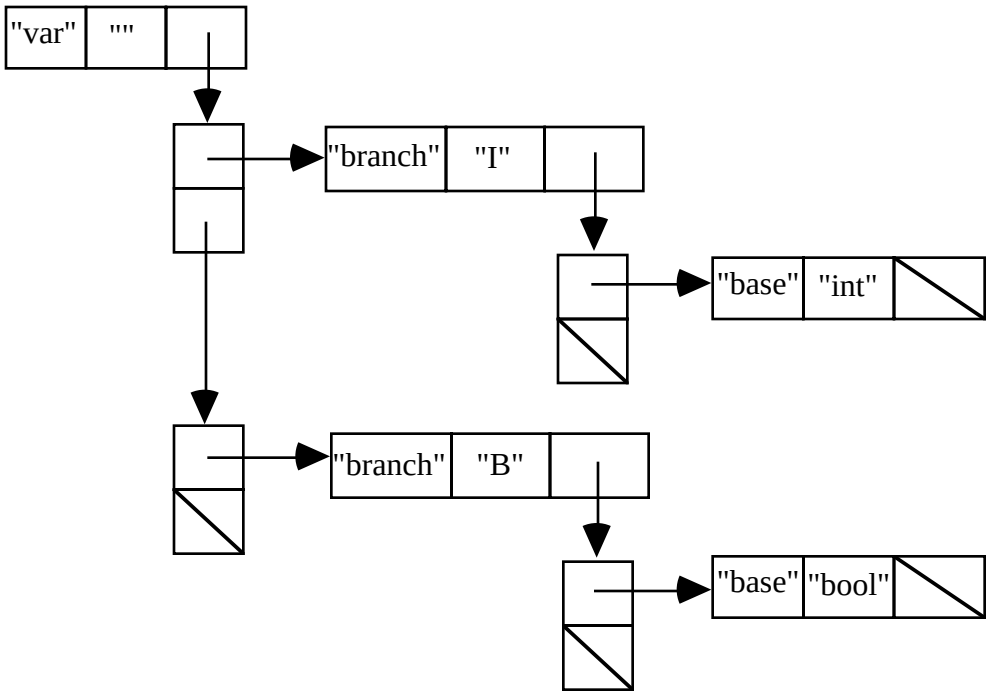

Figure 10.10 : Representation for Variants

10.2.4 Functions

The references part of a function type representation points to a list of two elements. The first element of the list points to the argument type node and the second to the result type node. Thus the function type in Figure 10.11 is represented by the graph in Figure 10.12. If the function does not have either of these components then the corresponding list element points to nil.

```
type square is fun( int → int )
```

Figure 10.11 - Function Example

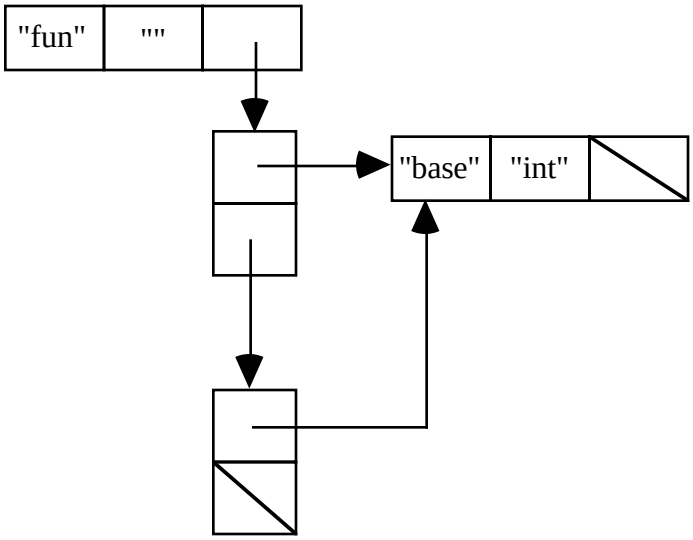

Figure 10.12 : Representation for Functions

10.2.5 Locations

A location type is represented by a node whose *references* part points to the node representing the type of the content. For example, *intLoc* in Figure 10.13 has the representation shown in Figure 10.14.

```
type intLoc is loc( int )
```

Figure 10.13 : Location Example

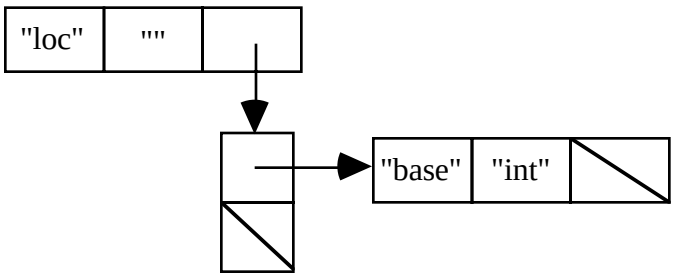

Figure 10.14 : Representation for Locations

10.2.6 Any

The infinite union type is represented by a single node with label *any*. Figure 10.16 illustrates this.

```
type anyType is any
```

Figure 10.15 : A Type Alias for any

Figure 10.16 : Representation for any

10.2.7 Quantifier Variables

Quantifier variables are represented by a node whose *references* part points to the representation of the bound type of the function. For example, the quantifier variable *t* from Figure 10.17 is represented as shown in Figure 10.18.

```
type getAgeFun is fun[ t ← Person ] ( t → int )
```

Figure 10.17 : Quantifier Variable Example

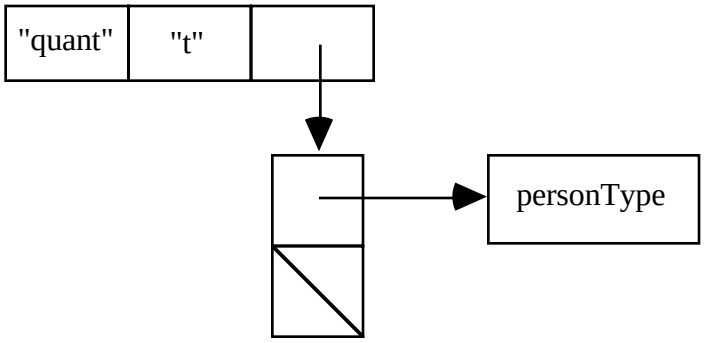

Figure 10.18 : Representation for Quantifier Variables

personType in Figure 10.18 stands for the representation of the record type *Person* shown in Figure 10.8.

10.2.8 Quantified Functions

The representation of quantified function types is based on that of monomorphic functions. The references part of the node contains a list of three elements in this case. In addition to the argument and result type representations described in section 10.3.4, there is also a link to the quantifier type representation described in section 10.3.7. Therefore quantified function type *qfun* in Figure 10.19 will be represented as shown in Figure 10.20.

type qfun **is fun** [t ← **int**] (t → t)

Figure 10.19 : Quantified Function

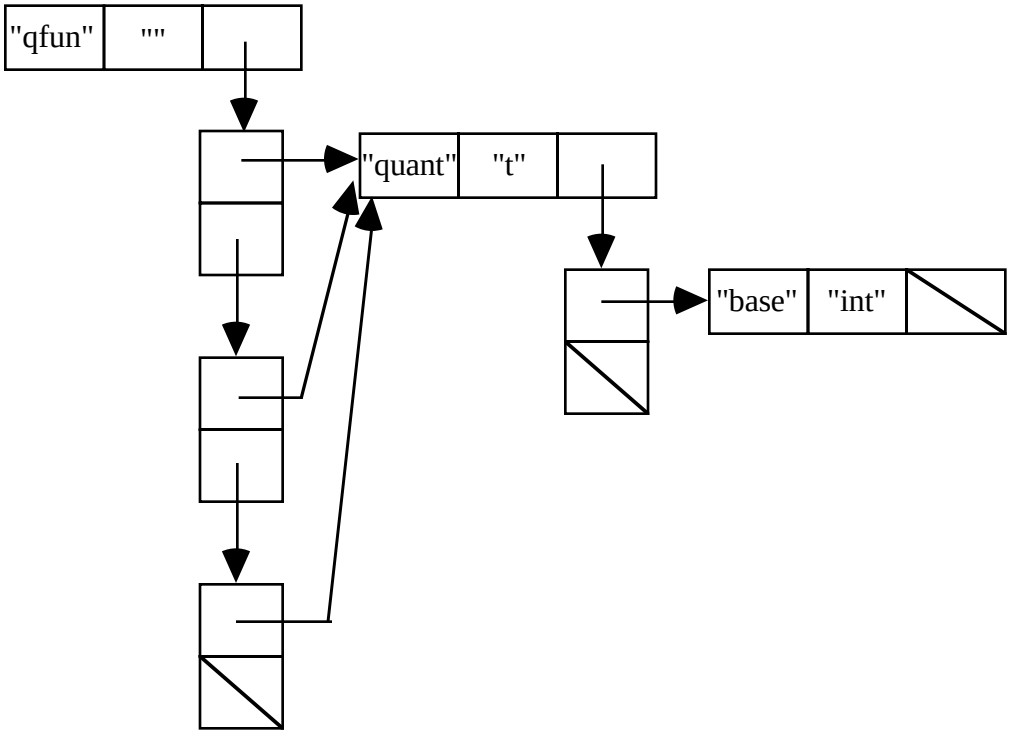

Figure 10.20 : Representation for Quantified Functions

10.3 Type Checking

10.3.1 Type Equivalence Checking

The type system described supports structural type equivalence. Given the basic representation for types, shown in Figure 10.1, a structural type equivalence checking algorithm can be defined recursively over it. In essence an algorithm to determine the equivalence of two type representations should check for the following:

- equality of the two labels
- equality of the two specific information parts
- recursive equivalence of types being referred to

If the general type representation can be written as shown in Figure 10.21 below

```
type typeRep is { label : string ; specificInfo : string ;
                  references : list [ typeRep ]          }
```

Figure 10.21 : Code for Type Representation

then a basic recursive equivalence checking algorithm can be defined as

```
rec let eqType = fun ( a, b : typeRep → bool )
typeIdentity( a, b ) or
( a.label = b.label ) and ( a.specificInfo = b.specificInfo ) and
eqList( a.references, b.references )

&

eqList = fun ( p, q : list [ typeRep ] → bool )
( p is nil and q is nil ) or
( ~( p is nil  ) and ~( q is nil ) and eqType( head( p ), head( q ) ) and
  eqList( tail( p ), tail( q ) ) )
```

Figure 10.22 : Basic Type Equivalence Checking Algorithm

These definitions are based on the ones presented in [Con90]. If the two types being checked for equivalence are identical then their equivalence is decided immediately. Otherwise a recursive scan of the type graph is performed. This algorithm suffices for type equivalence checking in Ext.

10.3.2 Type Extension Checking

The algorithm for determining whether two types are in the extension relation is structurally similar to the type equivalence checking algorithm. It is based on the definition of the extension relation given in Chapter 8. An outline of this algorithm is given in Figure 10.23 below.

```
let extType = fun( t1, t2 : typeRep → bool )
typeIdentity( t1, t2 ) or
case t1.label of
  "quant"          : t1.label = t2.label and
                     t1.specificInfo = t2.specificInfo and
                     eqList( t1.references, t2.references )

  "loc"            : if t2.label = "loc" then
                       extList( t1.references, t2.references )
                     else eqType( head( t1.references ), t2 )

  default          : t1.label = t2.label and
                     t1.specificInfo = t2.specificInfo and
                     extList( t1.references, t2.references )
```

Figure 10.23 : Checking for Extension

extType checks whether the type represented by t1 is extended from the type represented by t2. Since any type is extended from itself, if two types are identical then they are in the extension relation. As with type equivalence, this property is detected immediately. In all other cases, a recursive scan of the type graph is performed to determine whether the two types conform to the relation.

For all type constructors except quantifier variables and location types, equality tests on the *label* and *specificInfo* fields and a recursive pairwise check of the list of references will complete the extension test. Quantifier variables can only be in the extension relation if they are equivalent hence the same definition as before. If one of the types is a location then there are two possibilities for extension relation. If the second type is also a location then the two component types will have to be in extension relation. Otherwise the component type of the first type and the second type have to be equivalent according to the rules in Figure 10.22.

10.3.3 Dealing with Extension Quantifier Variables

Since extension quantifier variables play a major role in the polymorphism mechanism implemented here, a detailed account of how they are dealt with by the type checker is presented in this section.

10.3.3.1 Creating Extension Quantifier Variables

Chapter 8 presents a description of the two kinds of quantifier variables. Explicit quantifier variables are created explicitly in the program through the definition of quantified functions, while implicit variables are created internally by the type checker as a result of the application of operations on values belonging to other quantifier variables.

The type checker provides a function *quantifier* with the following interface to create a type representation for quantifier variables.

```
let quantifier = fun( qId : string; bType : typeRep → typeRep )
```

Figure 10.24 : Signature of *quantifier* Function

Thus it takes the unique identifier for the quantifier variable and its bound type and produces a quantifier type representation with the identifier as specific information and the *references* pointing to *bType*.

Quantifier variables can only be explicitly created by defining quantified functions. Consider the example in Figure 10.25 below.

```
let getAgeFun = fun[ t ← Person ] ( p : t → int )
```

Figure 10.25 : A Quantified Function

In this case, the type checker will perform the following actions. It declares the extension relation between *t* and *Person* by entering the identifier and the type representation of *Person* into the current scope of the ε environment. It checks whether the bound type is one over which only trivial extension is allowed. If this is the case then the <identifier, type rep> pair is entered into the current scope of the τ environment since the identifier has to stand for the bound type. Finally a type representation for the quantifier variable is created, using the *quantifier* function specified in Figure 10.24, to be used as part of the type graph for the function.

Implicit quantifier variables may be introduced as a result of two operations: dereferencing a field of an extended record or dereferencing an extended location. In these instances the type checker produces a unique identifier to stand for the result type of the operation. Consider the record dereference in Figure 10.26.

```
type Address is { town : string }
type Person is { name : string ; addr : Address }

let assignAddr = fun[ t ← Person ] ( p : t )
begin
   let a := p.addr
end
```

Figure 10.26 : Dereferencing an Extended Record

When it gets to the expression *p.addr*, the type checker will generate the string **t_addr* as the identifier for the new quantifier variable. If an extended location type t had been dereferenced then the string **t_@* is generated. Once the unique identifier is known, the type checker carries out the functions specified earlier for explicit quantifier variables i.e. the necessary entries are made in the appropriate environments and a type representation is constructed for the quantifier variable using the identifier generated.

10.3.3.2 Using Extension Quantifier Variables

The naming and type representation scheme described above allows the type compatibility of extension quantifier variables to be detected.

Whenever the function that constructs type representations in the type checker encounters an identifier it checks the current scope of the τ environment and then the ε environment to check whether the identifier has already been declared. Checking them in that order also means that the most specific type can be associated with the identifier. For example, in Figure 10.25, once the type representation for the explicit quantifier variable *t* has been constructed and added to the necessary environments, it can then be referred to as the type of the argument.

Generating strings during dereferencing in a standard manner to stand for implicit quantifier variables also means that other static dereferences in the same scope can be detected to be equivalent. For example, consider the *changeAddr* function defined in Figure 10.27 below.

```
type Address is { town : string }
type Person is { name : string ; addr : Address }

let changeAddr = fun[ t ← Person ] ( p, q : t )
begin
   let a := p.addr
   let b := q.addr
   a := b
end
```

Figure 10.27 : Type Compatibility

The same identifier is generated to denote the implicit quantifier variable for both dereferences. This validates the third assignment operation.

10.4 Summary

An implementation of the extension relation and polymorphism mechanism defined over it described in Chapter 8 is described. The type representations for different type structures are illustrated. The algorithms used for type equivalence checking and checking for the extension relation are described. The treatment of extension quantifier variables in the type checker is presented.

11 Conclusions

11.1 A Polymorphic Type System for Evolution

11.1.1 Aim and Motivation

The aim of the work in this thesis is to develop a polymorphism mechanism that deals with additive evolution in persistent programming systems. In databases and persistent systems, evolution is unavoidable. Given this fact, it is necessary to devise means of ensuring the soundness and consistency of programs and data with the changes that are constantly made. The concepts of subtyping, inheritance and evolution have often been confused and used interchangeably, with the end result that subtyping is used for situations and purposes which it does not naturally capture.

An example of such use is the inheritance of binary methods during subclassing in object oriented languages. If evolution demands that a subclass is defined using an existing class, it is common for the subclass to inherit some features of its superclass and possibly redefine them in its body. If these features are in the subtyping relation with the original features of the superclass then they may be used wherever the original features are used. However a problem occurs in the case of binary methods. The subtyping rule and the general evolution pattern for these function types do not coincide. Using subtyping to capture this evolution can lead to unsound systems, or in the best case, dynamic checks to ensure type safety.

The motivation for this work is the lack any mechanism that is particularly designed for dealing with additive evolution.

11.1.2 Related Work

Programming languages and database systems have used various strategies to cope with the problems caused by evolution. The object oriented programming language Eiffel uses covariant subtyping and dynamic type checks to overcome the problem of the inheritance of binary methods. In this instance, two function types are treated as being in the subtype relation even though the argument of the subtype is covariantly overridden as opposed to the contravariance condition demanded by Cardelli's subtyping rules. The dynamic checks are needed to ensure that the use of covariant subtyping is type safe.

The strategies used by the O_2 object oriented database system are interesting examples of the way schema evolution is dealt with at the data level. O_2 uses special primitives for performing schema modifications such as creation of a new class or modification, deletion or renaming of an existing class. Once modifications to the schema have taken place, data conversion and migration functions are used to ensure that data is kept consistent with the changes.

Matching has recently been proposed as a solution to the problem of type safe inheritance of binary methods in object oriented programming. Matching, like subtyping, is a relation between two types but is weaker than the latter. It is usually defined in terms of subtyping using the following rule: an object type A matches an object type B if A is a subtype of B under the assumption that any corresponding Self types (used to denote the object type of the receiver of a method) are equal. Matching does not support subsumption in general but allows methods of a subclass that matches its superclass to be safely used in place of those of the superclass.

In these examples, only Eiffel and languages supporting matching provide a solution at the type system level. Eiffel's use of subtyping where a covariant solution is needed requires dynamic checking to guarantee type safety. Matching provides an effective solution to the problem of binary methods but it only deals with object types in object oriented programming and is not applicable to non object oriented systems.

11.1.3 Extension Polymorphism

Unlike traditional programming languages, the presence of persistence also introduces the problem of keeping a potentially large volume of data consistent with any type changes that may occur. The existing solutions above do not meet all these requirements. Subtyping and inheritance, though they are useful in other circumstances, do not provide the answer to this problem since there is a mismatch between subtyping and inheritance in object oriented languages and existing solutions do not scale to other paradigms.

Since existing solutions do not completely capture evolution, a new mechanism, called extension polymorphism, is presented here. One of the main requirements in persistent systems is that in the face of type evolution, code should continue to work safely over data.

The proposed solution is a polymorphism mechanism over a relation that captures the patterns of type evolution in persistent systems. Since additive

evolution is the most common in these systems, the extension relation is defined to model how the most common type constructors used in programming languages evolve by refinement. Finally a bounded quantification mechanism, with the bounds enforcing extension rather than subtyping relation and where the quantifier variables could be related, is introduced.

11.1.4 Type Checking Extension Polymorphism

A type checker for extension polymorphism, implemented in S-algol, is presented. It builds type representations similar to the ones presented in [Con88, Con90, CBC+90] and performs a structural check on these representations to determine type equivalence and extension. The type variables introduced by bounded quantification are type checked using an environment which is a list of bindings between type variables and their extension bounds. It is also necessary for the type checker to create new type variables during compilation in order to type some expressions.

11.1.5 Properties of a Type System with Extension Polymorphism

A type system with extension polymorphism has been formally defined in sections 9.1 to 9.6 of this thesis and its soundness has been proved in section 9.7. A type checking algorithm based on the type rules of this system is also believed to be complete and convergent. However, a proof of these properties is beyond the scope of the thesis.

11.2 Advantages of Extension Polymorphism

Some of the benefits of incorporating extension polymorphism into a type system are listed below.

- it models the most common patterns of type evolution in persistent systems
- it deals with the most common type constructors
- it ensures that programs will continue to work safely over data in the case of additive evolution i.e. the integrity constraints on data up to the limit of the type system are guaranteed by soundness
- it is possible to use it in conjunction with other kinds of polymorphism to increase expressive power

- the extension rules together with the use of bounded quantification and related quantifiers provide more type information than is traditionally available with polymorphism, thus permitting more accurate modelling

11.3 Disadvantages of Extension Polymorphism

Some of the difficulties that may be encountered in using extension polymorphism as it is described in this thesis are the following:

- complicated syntax for bounded quantification with extension, especially when using related quantifiers

- function applications and location updates require the use of related quantifiers except in trivial cases

11.4 Future Work

The following aspects of extension polymorphism are possibilities for future work in this area:

- incorporating and studying the effects of recursion in a system supporting extension polymorphism

- implementing extension polymorphism in a full persistent programming language

- incorporating extension polymorphism into a system that also supports parametric and inclusion polymorphism and verifying the patterns of interaction

- simplifying the syntax used for extension polymorphism to make it easier for programmers to write code

- investigating the definition of a new relation that models subtractive evolution and polymorphism mechanisms over this relation

Recursion has not been included in the version of extension polymorphism presented in this thesis. [Ghe93b] has shown that the addition of recursion to System $F_{\leq}$, which is the basis for bounded quantification with subtyping, is not conservative. Therefore, recursion is omitted from the initial type system with extension polymorphism. However, it will be of interest to examine the following issues: extension of recursive types, recursive specification of

quantifier variables, the effects of recursion on the type checking algorithm and the proof of soundness.

Extension polymorphism described in the thesis is incorporated into Base which is a non-persistent language designed for experiments on polymorphism but without the complexity of a full programming language. It will be a valuable assessment to add extension polymorphism to a persistent programming language, such as Napier88, to study the effects on its implementation and to evaluate its usefulness.

The language with extension polymorphism does not support any other forms of polymorphism. Some possible interactions between extension and inclusion polymorphism are predicted in section 8.5. It will be an interesting exercise to include extension in a language that also supports parametric and inclusion polymorphism and to study the effects on its expressive power and to verify the patterns of interaction.

As stated in section 11.3, one of the difficulties in using extension polymorphism may be the complicated syntax for specifying polymorphic functions. It will be a useful modification to the system to simplify the syntax for extension polymorphism. A form of type inference mechanism may even be considered.

A contraction relation that is designed to model subtractive evolution will be an interesting avenue of research. The definition of such a relation will be the reverse of extension in most cases. Since the contraction relation will coincide with the subtyping relation for some type constructors, polymorphism over this relation may be easier to understand.

Appendix A

A Context-free Definition of the Language Base

Programs

program ::= sequence ?

sequence ::= [decl-seq] exp-seq

Type Constructors

type-id ::= **int** | **bool** | **any** | **unit** | **decl** | identifier | type-constructor

type-constructor ::= record-type | variant-type | loc-type | fun-type

record-type ::= { field-list }

field-list ::= identifier-list : type-id [, field-list]

identifier-list ::= identifier [, identifier]*

variant-type ::= ⟦ field-list ⟧

loc-type ::= **loc** (type-id)

fun-type ::= **fun** ([type-id] [→ type-id])

Declarations

decl-seq ::= type-decl | obj-decl [; decl-seq]

type-decl ::= **type** identifier **is** type-id

obj-decl ::= **let** identifier = expression

Expressions

exp-seq ::= expression [; expression]

expression ::= exp1 [**or** expression]*

exp1 ::= exp2 [**and** exp2]*

exp2 ::= [~] exp3 [= exp3]

exp3 ::= exp4 [add-op exp4]*

exp4 ::= [add-op] exp5

add-op ::= + | -

exp5 ::= literal | value-constructor |

expression.identifier |

project expression **as** identifier **onto** case |

expression (expression) | identifier |

expression := expression | @ expression |

begin sequence **end** |

if expression **then** expression **else** expression

case ::= var-case | any-case

var-case ::= identifier : type-id **in** expression **else** expression

any-case ::= type-id **in** expression **else** expression

Literals

literal ::= int-literal | bool-literal

int-literal ::= [add-op] digit [digit]*

bool-literal ::= **true** | **false**

Value Constructors

value-constructor ::= record-cons | variant-cons | fun-cons | loc-cons |

any-cons

record-cons ::= { record-init-list }

record-init-list ::= identifier = expression [, record-init-list]

variant-cons ::= [[identifier : expression]] : type-id

fun-cons ::= **fun** ([named-parameter] [→ type-id]) expression

named-parameter ::= identifier : type-id

loc-cons ::= ^ expression

any-cons ::= **inject**(expression, type-id)

In Appendices A and B, the microsyntax of the languages such as the definitions of *identifier* and *digit* are omitted.

Appendix B

A Context-free Definition of the Language Ext

Programs

program ::= sequence ?

sequence ::= [decl-seq] exp-seq

Type Constructors

type-id ::= **int** | **bool** | **any** | **unit** | **decl** | identifier | type-constructor

type-constructor ::= record-type | variant-type | loc-type | fun-type

record-type ::= ｛ field-list ｝

field-list ::= identifier-list ： type-id [， field-list]

identifier-list ::= identifier [， identifier]*

variant-type ::= ⟦ field-list ⟧

loc-type ::= **loc** （ type-id ）

fun-type ::= **fun** [⟦ quantification ⟧] （ [type-id] [→ type-id] ）

Declarations

decl-seq ::= type-decl | obj-decl [； decl-seq]

type-decl ::= **type** identifier **is** type-id

obj-decl ::= **let** identifier ＝ expression

Expressions

exp-seq ::= expression [； expression]

expression ::= exp1 [**or** expression]*

exp1 ::= exp2 [**and** exp2]*

exp2 ::= [～] exp3 [＝ exp3]

exp3 ::= exp4 [add-op exp4]*

exp4 ::= [add-op] exp5

add-op ::= + | -

exp5 ::= literal | value-constructor |

expression.identifier |

project expression **as** identifier **onto** case |

expression[⟦ type-id ⟧](expression) | identifier |

expression := expression | @ expression |

begin sequence **end** |

if expression **then** expression **else** expression

case ::= var-case | any-case

var-case ::= identifier : type-id **in** expression **else** expression

any-case ::= type-id **in** expression **else** expression

Literals

literal ::= int-literal | bool-literal

int-literal ::= [add-op] digit [digit]*

bool-literal ::= **true** | **false**

Value Constructors

value-constructor ::= record-cons | variant-cons | fun-cons | loc-cons |

any-cons

record-cons ::= { record-init-list }

record-init-list ::= identifier = expression [, record-init-list]

variant-cons ::= ⟦ identifier : expression ⟧ : type-id

fun-cons	::=	**fun** [⟦ quantification ⟧] ⦇ [named-parameter] [→ type-id] ⦈ expression
quantification	::=	identifier ← type-id
named-parameter	::=	identifier : type-id
loc-cons	::=	^ expression
any-cons	::=	**inject**⦇ expression, type-id ⦈

Glossary

The following terms are defined in the context in which they are used in this thesis.

Covariance	Consider two function types $F : A \rightarrow B$ and $F' : A' \rightarrow B'$. For F' to be a subtype of F, B' must be a subtype of B. This condition which requires the result types of functions to vary in the same direction as the function types is said to be covariant.
Contravariance	In the example above, for F' to be a subtype of F, A must be a subtype of A'. This condition which requires the argument types of functions to vary in the opposite direction as the function types is said to be contravariant.
Conservativity	A system S is conservative over a system S' (which is a subset of S) if for properties P in S', the following condition holds: if P can be derived from A then P can be derived from S'
Soundness	An algorithm is sound if the answers it provides are always correct.
Completeness	An algorithm is complete if it will always find the answer if there is one.
Convergence	An algorithm is convergent if the computation it performs is finite and will terminate.
Subtyping	Subtyping is a relation between two types which can be used for type safe substitutability. A type A is a subtype of a type B if all operations of B can also be applied to values of A.
Conformance	Conformance is a relation defined by languages such as Eiffel to check compatibility between two types.
Extension	Extension is a new relation between types presented in this thesis to model additive evolution of types.
Matching	Matching is a relation between two object types used by some object oriented languages to determine when binary methods can be safely inherited and used. It is less restrictive than the subtyping relation. An object type A matches an object type B if A is a subtype of B under the assumption that any corresponding Self types are equal.
Subsumption	Subsumption is a substitution mechanism that allows a value of a subtype to be used wherever a value of its supertype is expected.

References

[ABC+83] Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R. “An Approach to Persistent Programming”. Computer Journal 26, 4 (1983) pp 360-365.

[AC96] Abadi, M. & Cardelli, L. “On Subtyping and Matching”. ACM Transactions on Programming Languages and Systems 18, 4 (1996).

[AM86] Atkinson, M.P. & Morrison, R. “Integrated Persistent Programming Systems”. In Proc. 19th International Conference on Systems Sciences, Hawaii (1986) pp 842-854.

[AM95] Atkinson, M.P. & Morrison, R. “Orthogonally Persistent Object Systems”. VLDB Journal 4, 3 (1995) pp 319-401.

[AMP86] Atkinson, M.P., Morrison, R. & Pratten, G.D. “Designing a Persistent Information Space Architecture”. In Proc. 10th IFIP World Congress, Dublin (1986) pp 115-120.

[BCC+95] Bruce, K.B., Cardelli, L., Castagna, G., The Hopkins Object Group, Leavens, G.T. & Pierce, B. “On Binary Methods”. Theory and Practice of Object Systems 1 (1995).

[BKK+87] Banerjee, J., Kim, W., Kim, H.-J. & Korth, H.F. “Semantics and Implementation of Schema Evolution in Object-Oriented Databases”. ACM SIGMOD Record 16, 3 (1987) pp 311-322.

[Bru95] Bruce, K.B. “Subtyping is not a good match for OOL's” (1995).

[Bru96] Bruce, K.B. “Typing in object-oriented languages: Achieving expressiveness and safety”. Williams College (1996).

[BSG95] Bruce, K.B., Schuett, A. & van Gent, R. “PolyTOIL: A Type-Safe Polymorphic Object-Oriented Language”. Williams College (1995).

[Car84] Cardelli, L. “A Semantics of Multiple Inheritance”. In **Lecture Notes in Computer Science 173**, Kahn, G., MacQueen, D.B. & Plotkin, G. (ed), Springer-Verlag, Proc. International Symposium

on the Semantics of Data Types, Sophia-Antipolis, France, In Series: , Goos, G. & Hartmanis, J. (series ed) (1984) pp 51-67.

[Cas95] Castagna, G. “Covariance and Contravariance: Conflict without a Cause”. ACM Transactions on Programming Languages and Systems 17, 3 (1995).

[CBC+90] Connor, R.C.H., Brown, A.B., Cutts, Q.I., Dearle, A., Morrison, R. & Rosenberg, J. “Type Equivalence Checking in Persistent Object Systems”. In **Implementing Persistent Object Bases, Principles and Practice**, Dearle, A., Shaw, G.M. & Zdonik, S.B. (ed), Morgan Kaufmann, Proc. 4th International Workshop on Persistent Object Systems, Martha’s Vineyard, USA, In Series: (1990) pp 151-164.

[CL90] Cardelli, L. & Longo, G. “A Semantic Basis for Quest”. DEC SRC Technical Report 55 (1990).

[Cla92] Clamen, S.M. “Type Evolution and Instance Adaptation”. Carnegie Mellon University Technical Report CMU-CS-92-133 (1992).

[Cla94] Clamen, S.M. “Schema Evolution and Integration”. Distributed and Parallel Databases 2, 1 (1994) pp 101-126.

[CM82] Cole, A.J. & Morrison, R. **An Introduction to Programming with S-algol**. Cambridge University Press, Cambridge, England (1982).

[CM92] Connor, R.C.H. & Morrison, R. “Subtyping Without Tears”. In Proc. 15th Australian Computer Science Conference, Hobart, Tasmania (1992) pp 209-225, Technical Report ESPRIT BRA Project 3070 FIDE FIDE/92/34.

[Coc83] Cockshott, W.P. “Orthogonal Persistence”. Ph.D. Thesis, University of Edinburgh (1983).

[Con88] Connor, R.C.H. “The Napier Type-Checking Module”. Universities of Glasgow and St Andrews Technical Report PPRR-58-88 (1988).

[Con90] Connor, R.C.H. “Types and Polymorphism in Persistent Programming Systems”. Ph.D. Thesis, University of St Andrews (1990).

[CW85] Cardelli, L. & Wegner, P. “On Understanding Types, Data Abstraction and Polymorphism”. ACM Computing Surveys 17, 4 (1985) pp 471-523.

[Deu90] Deux, O. “The Story of O_2”. IEEE Transactions on Knowledge and Data Engineering 2, 1 (1990).

[Deu91] Deux, O. et al. “The O_2 System”. Communications of the ACM 34, 10 (1991) pp 34-48.

[DN66] Dahl, O. & Nygaard, K. “Simula, an Algol-Based Simulation Language”. Communications of the ACM 9, 9 (1966) pp 671-678.

[DT88] Danforth, S. & Tomlinson, C. “Type Theories and Object-Oriented Programming”. ACM Computing Surveys 20, 1 (1988) pp 29-72.

[Fla97] Flanagan, D. **Java in a Nutshell**. O'Reilly & Associates, Inc (1997).

[FS91] Fejer, P.A. & Simovici, D.A. **Mathematical Foundations of Computer Science**. Springer-Verlag (1991).

[GGH+91] Blair, G., Gallagher, J., Hutchison, D. & Shepherd, D. (ed) **Object-oriented Languages, Systems and Applications**. Pitman Publishing (1991).

[Ghe90] Ghelli, G. “Decidability of Type Assignment for the System F≤”. ESPRIT BRA Project 3070 FIDE Technical Report FIDE/90/2 (1990).

[Ghe93] Ghelli, G. “Divergence of F≤ Type Checking”. ESPRIT BRA Project 6309 $FIDE_2$ Technical Report FIDE/93/66 (1993).

[Ghe93b] Ghelli, G. “Recursive Types Are Not Conservative Over F≤”. ESPRIT BRA Project 6309 $FIDE_2$ Technical Report FIDE/93/68 (1993).

[Ghe93c] Ghelli, G. “Decidability of Type Checking for Type Systems with Polymorphism, Subtyping and Recursive Types” (1993).

[Ghe94] Ghelli, G. “Termination of system F-bounded”. ESPRIT BRA Project 6309 $FIDE_2$ Technical Report FIDE/94/101 (1994).

[Gir71] Girard, J.-Y. “Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types”. In Proc. 2nd Scandinavian Logic Symposium (1971).

[GM95] Gawecki, A. & Matthes, F. “TooL: A Persistent Language Integrating Subtyping, Matching and Type Quantification”. ESPRIT BRA Project 6309 $FIDE_2$ Technical Report FIDE/95/135 (1995).

[GM96] Gawecki, A. & Matthes, F. “Integrating Subtyping, Matching and Type Quantification: A Practical Perspective”. In Proc. Proceedings of the 10th European Conference on Object-Oriented Programming, ECOOP'96, Linz, Austria (1996).

[GR83] Goldberg, A. & Robson, D. **Smalltalk-80: The Language and its Implementation**. Addison Wesley, Reading, Massachusetts (1983).

[GTL89] Girard, J.-Y., Taylor, P. & Lafont, Y. **Proofs and Types**. Cambridge University Press (1989).

[Lew85] Lew, A. **Computer Science: A Mathematical Introduction**. Prentice-Hall International (1985).

[LRV90] Lécluse, C., Richard, P. & Velez, F. “O_2, an Object-Oriented Data Model”. In **Readings in Object-Oriented Database Systems**, Zdonik, S.B. & Maier, D. (ed), Morgan Kaufman, In Series: (1990) pp 227-236.

[MBC+87] Morrison, R., Brown, A.L., Carrick, R., Connor, R.C.H., Dearle, A. & Atkinson, M.P. “Polymorphism, Persistence and Software Reuse in a Strongly Typed Object Oriented Environment”. Software Engineering Journal, December (1987) pp 199-204.

[MBC+93] Morrison, R., Baker, C., Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C. & Munro, D. “Delivering the Benefits of Persistence to System Construction and Execution” (1993).

[MBC+96] Morrison, R., Brown, A.L., Connor, R.C.H., Cutts, Q.I., Dearle, A., Kirby, G.N.C. & Munro, D.S. “Napier88 Reference Manual (Release 2.2.1)”. University of St Andrews (1996).

[MBC+96b] Morrison, R., Brown, A.L., Connor, R.C.H., Cutts, Q.I., Dearle, A., Kirby, G.N.C. & Munro, D.S. “Napier88 Release 2.2.1”. University of St Andrews (1996).

[MCC+93] Morrison, R., Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C. & Stemple, D. “Mechanisms for Controlling Evolution in Persistent Object Systems”. Journal of Microprocessors and Microprogramming 17, 3 (1993) pp 173-181.

[MCC+95] Morrison, R., Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C., Munro, D.S. & Atkinson, M.P. “The Napier88 Persistent Programming Language and Environment”. In **FIDE Book**, Springer-Verlag, In Series: (1995).

[Mey92] Meyer, B. **Eiffel: The Language**. Prentice-Hall (1992).

[Mey97] Meyer, B. “Static typing and other mysteries of life”. (1997).

[Mil78] Milner, R. “A Theory of Type Polymorphism in Programming”. Journal of Computer and System Sciences 17, 3 (1978) pp 348-375.

[MM96] Malhotra, A. & Munroe, S.J. “Schema Evolution in Persistent Object Systems”. In Proc. 7th International Workshop on Persistent Object Systems, Cape May, NJ, USA (1996).

[Mor79] Morrison, R. “S-algol Language Reference Manual”. University of St Andrews Technical Report CS/79/1 (1979).

[Odb94] Odberg, E. “A Global Perspective of Schema Modification Management for Object-Oriented Databases”. In **Persistent Object Systems, Tarascon 1994**, Atkinson, M.P., Maier, D. & Benzaken, V. (ed), Springer-Verlag, Proc. 6th International Workshop on Persistent Object Systems, Tarascon, France, In

Series: Workshops in Computing, van Rijsbergen, C.J. (series ed) (1994).

[PS87] Penney, D.J. & Stein, J. “Class Modification in the GemStone Object-Oriented DBMS”. ACM SIGPLAN Notices 22, 12 (1987) pp 111-117.

[Rey74] Reynolds, J.C. “Towards a Theory of Type Structure”. Lecture Notes in Computer Science 19 (1974).

[Rod92] Roddick, J.F. “Schema Evolution in Database Systems - An Annotated Bibliography”. University of South Australia (1992).

[Rod95] Roddick, J.F. “A Survey of Schema Versioning Issues for Database Systems”. Information and Software Technology 37, 7 (1995) pp 383-393.

[Sch94] Schmidt, D.A. **The Structure of Typed Programming Languages**. The MIT Press (1994).

[Ste87] Stein, L.A. “Delegation is Inheritance”. In Proc. OOPSLA '87 (1987).

[Str67] Strachey, C. **Fundamental Concepts in Programming Languages**. Oxford University Press, Oxford (1967).

[Str86] Stroustrup, B. **The C++ Programming Language**. Addison-Wesley (1986).

[SZ87] Skarra, A.H. & Zdonik, S.B. “Type Evolution in an Object-Oriented Database”. In **Research Directions in Object-Oriented Programming**, Shriver, B. & Wegner, P. (ed), MIT Press, In Series: Computer Systems, Schwetman (series ed) (1987) pp 393-415.

[You84] Young, S.J. **An Introduction to ADA**. Ellis Horwood (1984).

[Zdo86] Zdonik, S.B. “Maintaining Consistency in a Database with Changing Types”. ACM SIGPLAN Notices 21, 10 (1986) pp 120-127.

[Zic89] Zicari, R. “Schema Updates in the O2 Object-Oriented Database System”. Politecnico di Milano (1989).

[Zic91] Zicari, R. “A Framework for Schema Updates in an Object-Oriented Database System”. In Proc. 7th IEEE International Conference on Data Engineering, Kobe, Japan (1991) pp 2-13.