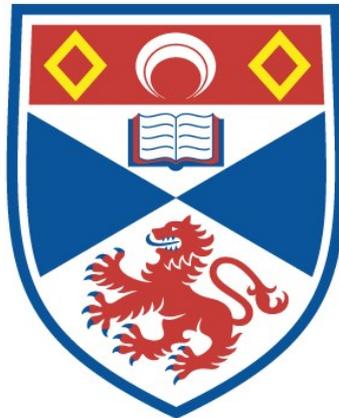


TYPES AND POLYMORPHISM IN PERSISTENT
PROGRAMMING SYSTEMS

Richard C. H. Connor

A Thesis Submitted for the Degree of PhD
at the
University of St Andrews



1991

Full metadata for this item is available in
St Andrews Research Repository
at:
<http://research-repository.st-andrews.ac.uk/>

Please use this identifier to cite or link to this item:
<http://hdl.handle.net/10023/13487>

This item is protected by original copyright

Types and Polymorphism in Persistent Programming Systems

Richard C. H. Connor

Department of Mathematical and Computational Sciences

University of St Andrews

St Andrews

Fife

KY16 9SS



ProQuest Number: 10167262

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



ProQuest 10167262

Published by ProQuest LLC (2017). Copyright of the Dissertation is held by the Author.

All rights reserved.

This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

TR A 1365

In submitting this thesis to the University of St Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and abstract will be published, and that a copy of the work may be made and supplied to any *bona fide* library or research worker.

Signed

Date 21/12/90

Declarations

I, Richard Charles Henry Connor, hereby certify that this thesis has been composed by myself, that it is a record of my own work, and that it has not been accepted in partial or complete fulfilment of any other degree or professional qualification.

Signed

Date 21st December 1990

I was admitted to the Faculty of Science of the University of St Andrews under Ordinance General No. 12 on 1st October 1985 and as a candidate for the degree of Ph.D. on 1st October 1985

Signed

Date 21st December 1990

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate to the Degree of Ph.D.

Signature of Supervisor

Date 21/12/90

Acknowledgements

Many people have contributed to the research presented in this thesis, and it is impossible to acknowledge them all. Working in a group, ideas are passed around and developed as a communal effort, and it is impossible to know from where, if anywhere, they have originated.

Fred Brown, Quintin Cutts, Al Dearle, Graham Kirby and Ron Morrison, in the PISA group at St Andrews, have all been involved with all of the research. From outside the group, discussions with Malcolm Atkinson, Mike Livesey, Dave McNally, Atsushi Ohori and John Rosenberg have all directly resulted in ideas presented here. To this I would like to add thanks to all other participants in PISA and FIDE project meetings, and anyone else I have discussed these topics with.

Finally Ron Morrison, my supervisor, must be thanked again for the very major part he has played not only in the construction of this thesis, but also in the development of its underlying ideals, and in my own development as a researcher.

- 1 Introduction1
- 1.1 Type systems4
 - 1.1.1 Modelling and protection4
 - 1.1.2 Safety and flexibility5
 - 1.1.3 Time of checking5
- 1.2 Persistence6
 - 1.2.1 Motivation6
 - 1.2.2 Definition of persistence7
 - 1.2.3 Orthogonal persistence.....8
 - 1.2.3.1 Removing complexity8
 - 1.2.4 Persistent languages.....9
 - 1.2.4.1 PS-algol and Napier8810
 - 1.2.5 Relation to database programming languages11
- 1.3 Polymorphism.....11
- 1.4 Persistent type systems.....14
- 1.5 Thesis structure.....17
 - 1.5.1 Software reuse in persistent systems.....17
 - 1.5.2 Models of protection18
 - 1.5.2.1 Protection18
 - 1.5.2.2 Viewing mechanisms19
 - 1.5.3 Typechecking across separately compiled modules19
 - 1.5.4 Implementation of polymorphism20
- 2 Software reuse23
 - 2.1 Introduction.....23
 - 2.1.1 Program sharing and binding23
 - 2.2 Binding mechanisms.....27
 - 2.2.1 Environments and binding.....27
 - 2.2.1.1 Static and dynamic environments27
 - 2.2.1.2 L-value and R-value binding.....29
 - 2.3 Napier88 environments.....32
 - 2.3.1 Creation and use of Napier88 environments32
 - 2.3.2 L-value binding34
 - 2.3.3 Constancy35
 - 2.3.4 Persistence.....36
 - 2.3.5 Programming system construction37
 - 2.4 Incremental construction and release.....44
 - 2.4.1 The example in Napier8846

2.4.2	The example in PS-algol.....	48
2.4.2.1	The data type pntnr.....	48
2.4.2.2	Persistence.....	48
2.4.2.3	System construction.....	50
2.4.3	Conclusions.....	52
2.5	Type abstraction.....	52
2.5.1	Universal quantification - specialised reuse.....	54
2.5.2	Existential quantification.....	57
2.6	Conclusions.....	64
3	Protection and viewing.....	66
3.1	Introduction.....	66
3.2	Protection of integrity.....	68
3.2.1	Integrity of data.....	68
3.2.2	Protection in persistent systems.....	70
3.3	Information hiding.....	72
3.3.1	Subtype Inheritance.....	73
3.3.2	1st-order information hiding.....	75
3.3.3	2nd-order information hiding.....	80
3.3.4	Conclusions.....	84
3.4	Viewing mechanisms.....	84
3.4.1	Database viewing mechanisms.....	85
3.4.2	View construction in Napier88.....	87
3.4.2.1	Creating a concrete representation.....	87
3.4.2.2	Placing concrete representations in the store.....	89
3.4.2.3	Creating abstract view types.....	90
3.4.2.4	Creating instances of views.....	92
3.4.2.5	Using views.....	93
3.4.2.6	Operations over more than one account.....	94
3.4.3	Privileged data access.....	95
3.5	Conclusions.....	99
4	Type equivalence in persistent systems.....	101
4.1	Introduction.....	101
4.2	Type equivalence models.....	102
4.2.1	Structural and Name Equivalence.....	102
4.2.1.1	Definitions.....	102
4.2.1.2	Example.....	103
4.2.2	Equivalence models within a program.....	104
4.2.3	Equivalence models within a programming system.....	106

4.2.3.1	Sharing a type between modules	108
4.2.3.2	Sharing a type between independent sub- systems.....	109
4.2.3.3	Abstracting over complex type definitions	110
4.2.3.4	Efficient execution	111
4.2.4	A Universal Equivalence Model	112
4.3	Implementation of structural type checking	113
4.3.1	Type Equivalence Checking.....	113
4.3.2	Representing Types	114
4.3.3	Efficient Structural Checking.....	115
4.3.4	Normalisation	116
4.3.5	Representing types by graphs.....	118
4.3.6	Representing Types by Strings	121
4.3.6.1	Normalising Graphs	122
4.3.6.2	Mapping to Strings.....	124
4.3.7	Comparison of Graphs and Strings	126
4.3.7.1	Speed	126
4.3.7.2	Space	126
4.3.7.3	Sharing.....	127
4.3.8	Measurements.....	128
4.3.8.1	Space	129
4.4	Constructing type representations	130
4.4.1	The Napier88 type system and algebra.....	130
4.4.2	Constructing type representations	133
4.4.2.1	Base types.....	134
4.4.2.2	Vectors	135
4.4.2.3	Structures	135
4.4.2.4	Variants.....	137
4.4.2.5	Procedures	138
4.4.3	Quantified types.....	139
4.4.4	Parameterised types	143
4.4.5	Specialisation.....	144
4.4.6	Recursive types	147
4.4.7	Parameterised recursive types.....	150
4.4.8	Recursive existentially quantified types.....	160
4.4.9	Free quantifiers	162
4.4.9.1	Static checking.....	162
4.4.9.2	Dynamic checking.....	165
4.4.10	Conclusions.....	168

4.5	Conclusions.....	169
5	On the implementation of polymorphism	171
5.1	Introduction.....	171
5.1.1	Compilation, architectures, and polymorphism.....	172
5.2	Parametric polymorphism.....	174
5.2.1	Machine-level type systems	174
5.2.2	Textual implementation	177
5.2.2.1	Multiple parameters	177
5.2.2.2	Calls within calls	178
5.2.2.3	First class procedures.....	179
5.2.2.4	Persistence	179
5.2.2.5	Conclusions.....	180
5.2.3	Uniform implementation.....	180
5.2.3.1	Fixed size value representations.....	181
5.2.3.2	Finding pointers.....	181
5.2.3.3	Conclusions.....	182
5.2.4	Semi-uniform, partly tagged implementation.....	183
5.2.4.1	Semi-uniform implementation	184
5.2.4.2	Conversion by procedural encapsulation.....	187
5.2.4.3	Cost of procedural encapsulation.....	193
5.2.4.4	Reverse encapsulation	194
5.2.4.5	Cost comparison of encapsulation and reverse encapsulation	196
5.2.4.6	Lazy conversion using tagging	197
5.2.4.7	Conclusions.....	201
5.2.5	Conclusions	201
5.3	Inclusion polymorphism	202
5.3.1	Subtype inheritance.....	203
5.3.2	Value representations and partial type abstraction	205
5.3.2.1	General formats	206
5.3.2.2	Specific formats	207
5.3.2.3	Bounded universal quantification	208
5.3.2.4	Implementations.....	209
5.3.3	Uniform field addressing	210
5.3.4	Tagged field addressing.....	212
5.3.4.1	Variants of string address maps	213
5.3.5	Partly tagged field addressing.....	215
5.3.5.1	Bounded universal quantification	216
5.3.5.2	Inclusion polymorphism	220

5.3.6 Comparisons	226
5.3.7 Generalisation of implementation techniques.....	228
5.3.7.1 Variants and subtyping	229
5.3.7.2 Uniform implementation	230
5.3.7.3 Tagged implementation.....	231
5.3.7.4 Partly tagged implementation	233
5.3.8 Conclusions	234
5.4 Other forms of polymorphism.....	235
5.4.1 Ad hoc polymorphism	236
5.4.1.1 Fully tagged ad hoc implementation	238
5.4.1.2 Partly tagged ad hoc implementation.....	239
5.4.1.3 Textual ad hoc implementation	240
5.4.2 Existential quantification.....	241
5.5 Conclusions.....	242
6 Conclusions	246
6.1 Related work.....	247
6.1.1 Past.....	247
6.1.2 Future	248
6.2 Applications building systems.....	249
6.2.1 Software reuse	249
6.2.1.1 Dynamic L-value binding	249
6.2.1.2 Quantification over dynamic bindings.....	250
6.2.2 Protection and viewing.....	251
6.2.2.1 Using types to hide information.....	251
6.2.2.2 Viewing with existentially quantified types	251
6.3 Programming system implementation	252
6.3.1 Typechecking across persistent systems.....	252
6.3.1.1 Type models	252
6.3.1.2 Structural checking across store.....	253
6.3.2 Implementation of polymorphism	253
6.3.2.1 General problems and classification of solutions.....	253
6.3.2.2 Parametric implementation	254
6.3.2.3 Inclusion implementation.....	254
6.4 Some final words	254
Appendix I.....	256
Napier88 Context Free Syntax	256
Session:.....	256
Type declarations:.....	256

Type descriptors:.....	256
Object declarations:	257
Clauses:	257
Expressions:.....	257
Value constructors:	258
Literals:.....	259
Miscellaneous and microsyntax:	259
Appendix II.....	261
Napier88 Type Rules.....	261
Meta-rules.....	261
Session :	261
Object Declarations :.....	261
Clauses :	261
Expressions :.....	262
Value constructors:	263
Literals :.....	264
Appendix III.....	265
The Napier88 Typechecker Interface	265
References	266

1 Introduction

Information systems frequently have to deal with longevity and scale in the data that they support. Data is required with lifetimes which match the real-world processes which they represent, which can range from microseconds to years. Data is also commonly required to support the work of large organisations, with time scales of up to hundreds of years.

One of the difficulties of dealing with such data is that it may not be possible to predict which data will be useful in the future, nor when data will be useful. For example, in a computer aided design system all of the design data must be available to the designers for as long as the design is in progress. However the whole design, including its history, calculations, tests and decisions may be required for a later investigation if the evolved design turns out to be flawed. The solution to such problems in non-computerised systems is traditionally to save as much data as possible - information is never deliberately thrown away.

The longevity of data in a computer system has a number of consequences. If data keeps on being collected, the size of the total body may become very large. As new uses are found for data, the number of programs within a system may also become large, and more people may become dependent upon the data. All these factors increase the potential complexity of a system.

The size of a body of data is always a relative problem to the storage technology available. Thus "large-scale" is interpreted as a volume large enough to challenge the current technology. The complexity of large systems, however, is technology independent. Problems of scale may be aggravated by other new technology, for example as a result of automatic data gathering. Scale may also be increased by the merging of human organisations, possibly at more than a linear rate.

A further problem in large systems is that of change. Users of a system should be given at least the appearance of stability, so a system does not change as they are using it. This may be achieved by the specification of constraints, which may themselves be subject to change. The critical changes to a system are those with global effect, which are changes in organisation, changes in representation and changes in meta-data. This last class is the most difficult to manage.

Many users may wish to invoke changes to a system, and many users may also have to modify their use of data as a consequence of other changes. One of the difficulties is for users to understand the semantics of changes, so that a change may be explained to them. It is important to limit the propagation of change, so that the system may appear to be stable to most users.

An ideal model for building applications in an integrated data-intensive system has been described by the FIDE project [FID90], and may be likened to the diagram in Figure 1.1.1. In such a model as much as possible is factored out of the application programs into a central repository. Using such a model it should be possible to accelerate software production, improve system reliability and gain economies from code re-use.

Currently, however, information systems are usually constructed within a set of loosely connected support systems, such as database systems, programming languages, programming environments and operating systems. Each of these components is often designed independently and built using a separate technology. The resulting inconsistencies between these technologies make programming of data-intensive applications difficult, expensive, and error-prone.

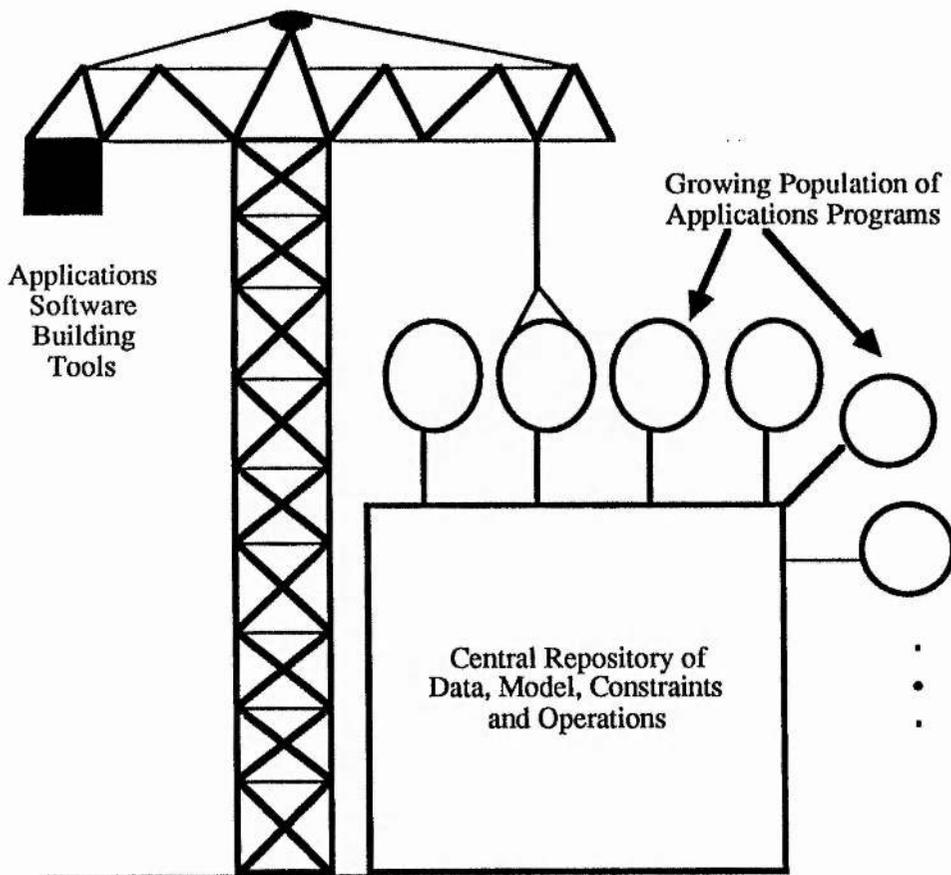


Figure 1.1.1 The FIDE model of applications building

The programming of systems which deal with large amounts of long-lived data is intrinsically difficult. However, the major failure of current technology as outlined above is that the lack of integration leads to avoidable complexity.

Persistent programming systems are one area in which to experiment with the removal of this avoidable complexity. By merging the distinction between long-term and short-term data, it becomes possible to model all of the activities required for a body of data within a single system. The consequences of this in terms of the type system are twofold. As data may never be manipulated outside its jurisdiction, the type system may be relied upon as a provider of system, rather than program, safety. As a consequence of this, however, the persistent type

system must be sufficiently flexible to allow the modelling of activities normally provided by untyped support systems.

1.1 Type systems

1.1.1 Modelling and protection

Cardelli and Wegner [CW85] give an account of how type systems arise naturally in programming languages, and indeed in human nature. "As soon as we start working in an untyped universe, we begin to organise it in different ways for different purposes." Thus, for example, the same bit pattern in a computer memory may be viewed in a number of different ways according to some properties of the entity it is intended to represent. The same bit patterns may be used to represent the character "a" and the number 97, but a programmer would normally wish to abstract over this fact.

This gives rise to the notion of a type system as an aid to modelling, by the provision of a set of regular interpretations of bit patterns. A programmer may think in terms of values modelled by a type system rather than in terms of a computer's memory state.

This modelling, however, may potentially be violated by the application of inappropriate operations. All values must be modelled by abstraction over bit patterns, but the operations available on bit patterns will not normally be appropriate over the models provided by a type system. Because of this, a type system usually also provides some mechanism by which the inappropriate application of operations is prohibited. Thus a type system is normally regarded as a provider of both modelling and protection within a programming system.

1.1.2 Safety and flexibility

As always, however, the extra safety provided by the prevention of operations in certain contexts causes a loss of flexibility. For example, it is reasonable to prohibit an operation such as bitwise *or* on two values which represent integers. On the other hand, it may be convenient for a program to model an integer as a bit pattern, or vice versa, in which case the prevention of this operation is not desirable.

The level at which a type system's protection is specified varies widely between programming languages. Languages such as C [KR78] provide only a modelling role, and if desired any operation may be applied to values of any type. Such languages are very flexible, but at the cost of some safety. Many languages allow at least some operations which violate the type system, for example it is not uncommon to be able to index past the bounds of an array. Again, this allows extra flexibility, but at the cost of safety.

A language whose type system prevents any operations which violate its modelling facilities is referred to as a strongly typed language. Before strong typing is enforced in a programming language, the designers must be confident that the modelling provided by the type system is adequate for the language's purpose. If type system modelling is inadequate, then programmers are forced to use inappropriate type models. In this case strong typing may be over-restrictive.

1.1.3 Time of checking

An operation which violates a model provided by a type system may be detected at any time before the operation is executed. In general, it is desirable to detect such errors as early as possible. There are two reasons for this:

- program safety

- program efficiency

If an error is detected before a program starts to execute then the semantics of failure need not be dealt with. It may not be acceptable to terminate the execution when a type error occurs. If the absence of type errors can be guaranteed before execution begins, then no dynamic resource is required to check for possible type errors, with a consequent increase in efficiency.

Languages where all type errors may be detected by static analysis of a program are known as statically typed languages. They have the desirable property that no operation which violates type system modelling can ever occur during execution. Once more, however, such systems must forfeit some flexibility to achieve this. For example, such a system can only allow manifest values to be used as array bounds and subscripts, as otherwise a dynamic error could occur. Most programming languages do rely upon at least some dynamic type checking.

Programming systems which deal with large amounts of heterogeneous data require some dynamic typechecking to allow necessary abstraction over the type of this data. A program which binds dynamically to its data should only have to describe the type of that subset of the data upon which it operates. To allow such partial specification in an evolving universe, an infinite union type is required [ABC83]. Before a value of such a type may be usefully manipulated, a dynamic type check is necessary.

1.2 Persistence

1.2.1 Motivation

The initial motivation for building persistent languages arose from the difficulties of storing and restoring data structures arising in CAD/CAM research [Atk78]. Complex mappings had to be constructed between complex program data

structures and database systems. These mappings were observed to have a number of costs:

- they were entirely extraneous to the required computation on the data, thus obscuring code and confusing programmers
- they were not subject to type checking, and consequently were a common source of error
- it was difficult to translate incrementally in either direction, and a consequent tendency was to load more data than necessary during a program's execution
- there were large computational overheads in performing the translation

The construction and maintenance of these mappings is a serious problem in a system which must deal with large amounts of long-lived data.

Persistent programming languages were designed initially to solve these problems. They eliminate discontinuities in the computational model, and economise on design and implementation effort by using the same concepts and constructs throughout the entire computational system.

1.2.2 Definition of persistence

Persistence is defined as the length of time for which data exists and is usable [ABC83]. A spectrum of persistence exists and is characterised by:

- transient results during expression evaluation
- local variables in procedure activations
- own variables, global variables and heap items
- data that exists between program executions
- data that exists between various versions of a program
- data that outlives a program

Traditionally, the first three categories are managed by a programming language and the others by a file system or database management system. A persistent system is one where all categories of longevity are catered for by a single programming language mechanism.

1.2.3 Orthogonal persistence

The following principles of persistent programming systems have been identified in [ABC83]:

- *Persistence Independence*
The manipulation of data in a program is independent of the persistence of that data. That has the consequence that a programmer can not specify the physical movement of data between long term and short term storage devices.
- *Persistent Data Type Orthogonality*
All data values are allowed the full range of persistence irrespective of their type.
- *Persistence Identification*
The identification of values which are to be made persistent is not related to the type system, computational model or control structures of a language

Programming languages which obey all three of these principles are classified as persistent programming languages, and are said to display orthogonal persistence.

1.2.3.1 Removing complexity

The understandability of a language is its most important property. The main difficulty with data intensive programming is understanding the mapping between the real world and the computational models. A non-persistent system may be represented by the diagram in Figure 1.2.1.

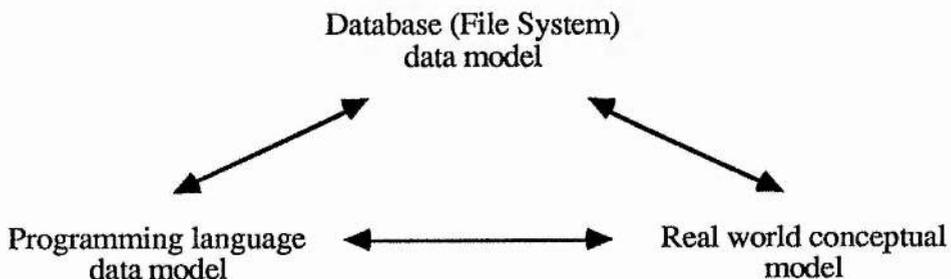


Figure 1.2.1 Conceptual mappings in a traditional system

This diagram shows three mappings between different models of the data. A programmer must understand all of these mappings, as well as the different models, to be confident about data manipulation. In a persistent system, as illustrated in Figure 1.2.2, only one such mapping exists between the two data models. Thus the task of modelling data within a system is greatly simplified.



Figure 1.2.2 Conceptual mappings in a persistent system

The single mapping means not only that a system should be easier to understand, but also that no code needs to be written to maintain the mappings. In a data intensive application, maintaining the extra mappings is estimated to cost around 30% of the total code [AMP86].

1.2.4 Persistent languages

As the provision of persistence should be orthogonal to all other aspects of programming language design, it should be possible to have applicative, relational, logic, object, query and imperative persistent languages. Some known persistent languages are:

- DBPL [MS89] relational
- PS-algol [PS88] imperative
- Leibniz [Eve85] object
- E [RC89] almost object (C++)
- Galileo [ACO85] object
- Poly [Mat85] applicative
- Staple [DM90] applicative
- Amber [Car85] applicative
- Persistent Prolog [GMD85] logic
- χ [HS89] capability
- Napier88 [MBC89] imperative

1.2.4.1 PS-algol and Napier88

Two examples of orthogonally persistent languages are PS-algol [PS88] and Napier88 [MBC89]. In both languages, persistence is identified by reachability from distinguished roots in a persistent store. For a value to persist after the invocation of a program, it must be placed within the transitive closure of one of these distinguished roots. This model of persistence is highly suitable for imperative languages, but other models are required for different language paradigms.

Persistence in PS-algol is modelled by a set of standard procedures. A distinguished persistent root may be created by the standard procedure *createDatabase*, which takes a database name and password as parameters and returns a value which represents the new root. The same root may be accessed by another program invocation by use of the *openDatabase* procedure.

The root returned when creating or opening a database is a pointer to an instance of an associative lookup table, of type *Table*. Further standard procedures are defined to manipulate these values. *sEnter* takes as parameters a string key, a *Table*, and a value of an infinite union type. *sLookup* also takes a string key and a *Table*, and returns the most recent value which has been stored using *sEnter* with the same key and *Table*. Thus these tables may be used to store a value of arbitrary complexity in a manner which is orthogonal to any of its attributes.

The Napier88 model of persistence is similar but more flexible. Here, there is a single root of persistence, and the persistent store is calculated as the transitive closure of this root. The persistent root is of a data type *environment*, which is a dynamic model of the static environments normally found in block-structured programming languages. Using this data type, bindings may be incrementally added to the store, including bindings to other environments. This allows an

arbitrary store structure to be built from this root. The Napier88 system consists of a language and a populated persistent store, which contains amongst other things the standard functions and procedures.

1.2.5 Relation to database programming languages

A database programming language is one which is designed to deal with large amounts of long-term data, and a persistent language is one in which the longevity of data is orthogonal to its use. Although the motivation for these different kinds of language may be different, they converge rapidly to the same set of problems. A database programming language may start from a data model and endeavour to provide a general-purpose programming algebra over it, while a persistent language may start with a general-purpose programming algebra within which it must also provide a data model. Recent research and collaboration, particularly in the FIDE (Esprit II Basic Research Action 3070) project has shown that the two communities are on the point of merging, to tackle the outstanding problems caused by scale and longevity of data.

1.3 Polymorphism

Polymorphism was probably first identified by Strachey [Str67]. Although his discussion is mainly restricted to operators which may have several forms according to the type of their arguments, he also points out another more regular form of polymorphism, introduced by example of the function *map*. He introduces the terms "ad hoc" and "parametric" to describe these two forms of polymorphism, and mentions that both forms present a considerable challenge to the programming language designer.

The classifications made by Strachey are still in use today, although some other classes of polymorphism have also been identified. Polymorphism has been

defined in a more recent survey paper [CW85] as an attribute of a programming language which allows "some values and variables to have more than one type". This definition allows subtype inheritance, coercion and some models of abstract data types all to be classified as polymorphism. The term inclusion polymorphism has been added to those defined by Strachey to describe a general form of subtype inheritance. The following are working definitions of the different kinds of polymorphism:

- ad hoc: An operation is defined over a number of different types, and its semantics may depend upon the type of its operands. An example is the operator "+", which may be defined over integers and reals, and has a different interpretation for each type.
- parametric: Instances of the same type within a type description may be abstracted over by an implicit or explicit type parameter. An example is an identity function, where although the parameter and result may be of any type, they are statically known to be the same type.
- inclusion: The type of a value may be partially abstracted over, so that unnecessary type information need not be stated. An example is a function which is defined over any record value which has an *age* field of type integer.

All of these language concepts have evolved independently from each other. Languages such as early assemblers contained no polymorphism whatsoever. Ad hoc polymorphism was the first to appear, in languages such as Fortran [ANS78], which defined overloaded operators, such as "+", along with the ability to coerce values from integer to real according to the use of such operators. Parametric polymorphism appeared in ML [Mil83], and inclusion polymorphism in Simula [BDM73]. Existentially quantified types as described by Mitchell and Plotkin [MP88] is a model of abstract data types which introduces abstraction over a type description, and such types are included in the definition of parametric polymorphism given above.

The motivation for all of these diverse language models is the same: the need to abstract over type. As type systems include more static constraints, then flexibility is lost as well as safety gained. Some of this flexibility, however, may be regained without the loss of safety by the introduction of type abstraction. Polymorphism is viewed here not as some theoretical attribute of type systems but as a solution to a class of practical problems which require type abstraction.

Ad hoc polymorphism allows the meaning of an operation to depend upon the types of its operands. In early languages such as Fortran the motivation for this was to abstract over the different representations used for types such as single and double length floating point numbers, so that the same arithmetic operators could be applied to them. However, the use of an operator could still be statically determined from program analysis, and so no true type abstraction occurred. However, Keas [Kea88] and Wadler and Blott [WB89] have more recently shown how such polymorphism may be extended to allow abstraction over related sets of types.

Parametric polymorphism emerged in the design of ML as a necessity, rather than an invention. Milner states that for the kind of use for which ML was designed "it soon became intolerable to have to declare, for example, a new maplist function ... every time a new type of list is to be treated." [Mil83a] He stresses that the introduction of polymorphism was on purely practical grounds, to give the required type abstraction, rather than as an experiment in the introduction of such type schemes into a programming language. Indeed, the proper lineage for the style of type checking by inference was only discovered after its introduction.

Inclusion polymorphism first manifested in Simula and its object-oriented derivatives such as Smalltalk [GR83]. The major motivation here was the reuse of software. This is achieved by models of inheritance, where data types can be

derived by incremental enhancement of others. Then any operations which are defined over a type may also be applied to derived types. This allows, in retrospect, partial abstraction over data types. Once again, the need for this style of programming was apparent some time before its formal treatment, and even its classification as a form of polymorphism [Car84].

Existential data types were formally introduced by Mitchell and Plotkin as a general model for abstract data types. These types allow the hiding of information by abstracting over types, but maintain complete static type safety. The model described is not usually considered as a form of polymorphism, but involves abstraction over a given type signature in the manner of parametric polymorphism.

1.4 Persistent type systems

Type systems are historically viewed as mechanisms which impose static safety constraints upon a program. Within a persistent environment, however, the type system takes on a wider role.

Data manipulated by a programming language is governed by that language's type system. In non-persistent languages, however, data which persists for longer than the invocation of a program may only be achieved by the use of an operating system interface which is shared by all applications. As a consequence of this, such data passes beyond the jurisdiction of the type system of any one language.

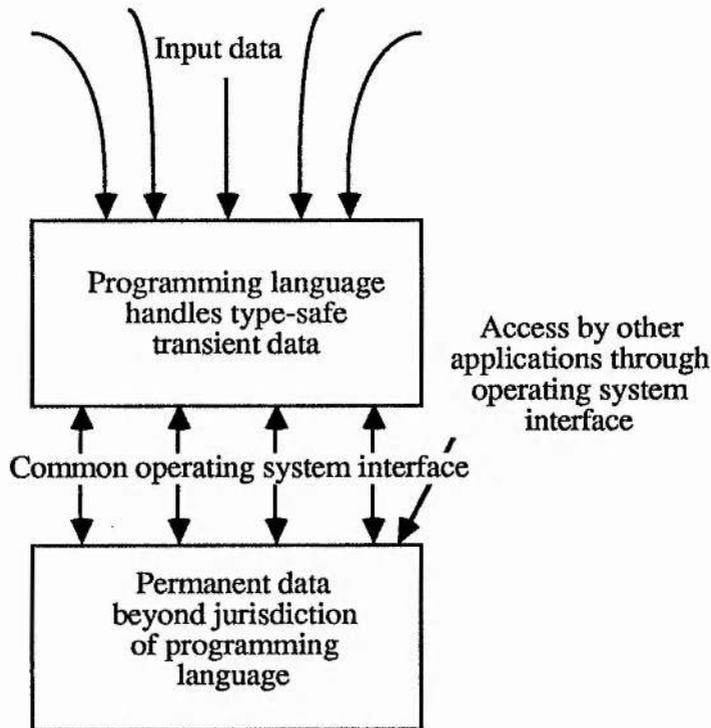


Figure 1.4.1 Traditional strategy for permanent and shared data

Mechanisms which govern long term data, such as protection and module binding, must be dealt with at the level of this interface. Historically this has the consequence that the type system may not be enforced, and knowledge of the typed structure of data may not be taken advantage of. This is shown diagrammatically in Figure 1.4.1.

In a persistent system, the storage of data beyond a single program invocation is handled by programming language mechanisms, and no common operating system interface is necessary. The only route by which data may be accessed is through the programming language, and so the type system of a single language may be used to enforce protection upon both transient and permanent data. High-level modelling may be relied upon for the entire lifetime of the data, as it never passes outside the language system. Figure 1.4.2 gives a diagrammatic view of such a system.

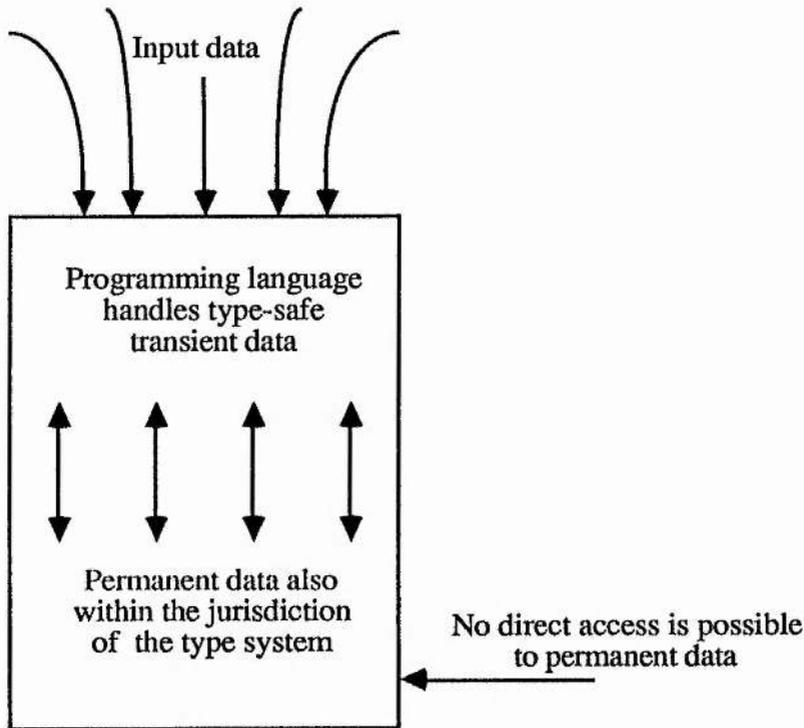


Figure 1.4.2 The persistent strategy

The universality of the persistent type system has consequences in terms of both the modelling and protection provided by the type system itself, and also presents some new challenges in terms of implementation.

With respect to modelling, the persistent type system must be sufficiently flexible to allow the modelling of activities normally provided by untyped support systems. Such activities include, for example, the linking of separately prepared program units, and file system access protection. Methods of achieving such flexibility whilst maximising safety include the controlled use of infinite unions, parametric and inclusion polymorphism.

With respect to protection, the increased role of a type system means that any protection mechanisms programmed at a high level may be fully relied upon to protect the data for its lifetime, as access from outside the constraints of the type

system is not possible. In particular, various high level information hiding techniques may be used to restrict data access, instead of relying upon the normally coarse grain and typeless control provided by outside technologies.

The implementation of such a system presents the challenges of combining the following features:

- a fine-grained type system to maximise safety, resulting in large and complex type descriptions
- infinite unions, parametric and inclusion polymorphism, to allow sufficient flexibility
- the efficient implementation of the above features in a persistent system

Efficiency is considered important as programmers are well known to sacrifice safety and long-term efficiency for short-term performance. Efficiency considerations within a persistent system are notably different from those in a non-persistent system.

1.5 Thesis structure

This thesis is divided into four main chapters. Two of these concentrate upon the issues of modelling and protection when a single type system is used to govern all data, and the other two on how such a system may be implemented. An outline of each of these chapters is given in the sections 1.5.1 to 1.5.4.

1.5.1 Software reuse in persistent systems

Persistent programming systems with first class procedure values [AM85] may be used to solve some of the problems associated with the incremental construction of large programming systems. In particular, software componentry may be shared, rather than copied, whilst maintaining the benefits of a strongly typed programming language. Apart from orthogonal persistence, two other language facilities are required to allow this.

A flexible, incremental binding mechanism is required to model a number of different methodologies for the binding of program components [Dea89]. In particular, carefully controlled dynamic and L-value binding in such a mechanism may allow the incremental construction of a complex software system.

Type abstraction is also necessary to maximise the sharing of software componentry, as it allows the specification of program segments that do not depend upon a full type description of the data over which they operate. Universal quantification allows abstraction over the type of a program segment, and existential quantification allows abstraction over the implementation of the data it uses [CW85, MBC87].

1.5.2 Models of protection

1.5.2.1 Protection

Large bodies of data are inherently valuable, and must be protected against accidental or deliberate misuse. This is true in both persistent and non-persistent programming systems. However, protection mechanisms in a persistent system may be modelled quite differently from a non-persistent system [MBC90, CDM90].

As all data within a persistent system are subject to type system constraints, no unlawful access to the data may occur except in the event of system failure. This gives the property of type safety to permanent data, which can not be achieved in a non-persistent system.

A further effect of the type safety of all data is that any protection which is programmed may not be revoked by a lower level of technology. This means that access protection and software constraints may be programmed as suitable for the individual data. A number of different language mechanisms may be used to

protect permanent and shared data, including subtype inheritance, procedural encapsulation, and abstract data types [MBC90].

1.5.2.2 Viewing mechanisms

The protection of data in database systems is normally achieved by integrity constraints and viewing mechanisms. In a persistent system, however, such mechanisms do not need to be specially provided as they may be programmed within a persistent programming language. Integrity constraints may be programmed within procedural encapsulation, and viewing mechanisms may be programmed with a combination of encapsulation and existentially quantified data types. The added advantage of existentially quantified types over traditional viewing mechanisms is that they are statically type checkable. The referential integrity provided by a persistent store ensures that views occur over the same instance of data, and the copying of data, with its associated integrity problems, is not necessary [CDM90].

1.5.3 Typechecking across separately compiled modules

Type equivalence checking is well understood within a program, but presents some extra problems in a persistent system which allows the independent compilation of co-operating modules [CBC90]. In a persistent system, data which is shared between modules does not move outside the language's type system, and so the type system must address the new issues. Commonly, languages which are not persistent can allow such co-operation only by using a store interface which is not type secure, such as a file system.

The traditional database schema can be regarded as a type. This leads to a requirement for the efficient manipulation of types in a persistent system to allow

provision of the facilities traditionally found in DBMS for schema editing, use and evolution.

Two models of type equivalence are in common use: name equivalence and structural equivalence. While name equivalence schemes are easier to implement and are generally more efficient they still have to use structural checks to provide important facilities such as schema merging. On the other hand structural equivalence, generally more flexible and less efficient, can often achieve the same performance as name equivalence [CBC90].

One important topic of research is how to improve the performance of structural equivalence checking. How such checking may be performed is fully investigated. The balance between constructing efficient representations of types in terms of store and the speed of the equivalence algorithm in comparing two representations is described.

Two main candidates for type representations are strings and graphs. The differences in practice of these representations highlight the difficulties in their construction and use. Some preliminary measurements are available and the main conclusion is that where the type schema is large and involves the sharing of types, the graph representation is much more efficient in terms of space. It may however be slower in terms of speed of checking depending on its use within a persistent store.

1.5.4 Implementation of polymorphism

Compilers traditionally use type information for two purposes: the first is to ensure static safety constraints within a program, and the second is to generate appropriate code where this depends on the type of operands.

Values of different types frequently have different machine-level representations, for reasons of efficiency. The difficulty of implementing polymorphic systems stems from the fact that, as the type may be abstracted over, the representation of a value is not always known statically. This causes problems with all kinds of polymorphism. With parametric polymorphism, operations which may manipulate a value of any type are defined over values whose type, and therefore representation, is not known. With inclusion polymorphism operations such as record dereference may be defined over values whose type is partially abstracted over. Representation dependent information, such as record offsets, may not always be calculated statically. With ad-hoc polymorphism there is a requirement for operations with different semantic meanings to be executed according to the type of the operands, which may be abstracted over in some systems.

A number of solutions to these problems have been proposed and successfully implemented [Mil83, Lis81, DD79, Mat85, BMS80, Fai82, Tur87, WB89]. All of these implementations however have been in non-persistent languages, and for various reasons are not well suited to the implementation of polymorphism in a persistent programming system.

Two new mechanisms for the implementation of polymorphism are described here. One of these solves the problems encountered when operations are to be performed on a value about which no type information is available. This is suitable for the implementation of parametric polymorphism and existential data types. This mechanism has been successfully used in the implementation of the persistent language Napier88 [MDC90].

The other proposal is for a mechanism which allows type-specific operations to be performed on a value whose type is partially abstracted over, and may therefore be used to implement inclusion polymorphism. This mechanism may also be useful

in object-oriented languages as a solution to object addressing with multiple inheritance [CDM89].

2 Software reuse

2.1 Introduction

Persistent programming systems may be used to solve some of the problems associated with the construction of large and complex programming systems. In particular, software componentry may be shared, rather than copied, whilst maintaining the benefits of a strongly typed programming language. Apart from orthogonal persistence, two other language facilities are required to allow this.

A flexible, incremental binding mechanism should be able to model a number of different methodologies for the binding of program components. In particular, such a mechanism should allow the incremental construction and evolution of a complex software system. So that a system constructed incrementally may be released, it should also be possible to bind the components together tightly at a later stage.

Type abstraction is also necessary to maximise the sharing of software componentry, as it allows the specification of program segments that do not depend upon a full type description of the data over which they operate. Universal quantification allows abstraction over the type of a program segment, and existential quantification allows abstraction over the implementation of the data it uses [CW85, MBC87].

2.1.1 Program sharing and binding

Type systems provide support for modelling and protection within programming languages. Traditionally, this support exists only within the confines of a single program. In a system which is orthogonally persistent, the domain of the type system is extended to data which is used by many invocations of the same or

different programs. A consequence of this is that a programmer need not be distracted by the writing of conversion routines to flatten and rebuild permanent data, nor the possibility that permanent data may be compromised by access from outside the control of the language's type system [ABC83].

A further ability of persistent languages is the type-safe storage of data of any type. This is not possible in a non-persistent language. For example, many languages include type systems which are able to model first-class procedures and abstract data types. Unless the internal form of such values is part of a language's semantic domain it is not possible for a programmer to specify flattening and restructuring of arbitrary data.

Within a persistent programming system, the ability to store procedures has been shown to be extremely powerful [AM85]. In particular, it allows sub-programs as well as data to be placed in the persistent store and flexibly bound as they are required during the execution of another program. This ability potentially allows all programming activity to be expressed within a single system, and goes some way to provide a technology suitable for the construction of an integrated project support environment [MBD85].

Programming languages normally define only a single mechanism for the exporting of data from beyond the scope of a single program, and this interface is used both for data which is to outlive a program and also for data which is to be shared by a number of programs. This is true of both persistent and non-persistent languages. In a persistent system, however, where procedures may be passed through this interface, a more flexible model of reusing software is made possible.

In a traditional operating system environment, an applications builder is limited in the way that a large application may be constructed. Separate modules may be

written to perform sub-tasks, but these must always be bound together from outside a language system. Usually they must be bound before a program starts to execute, using a mechanism such as a linker. In this case software reuse is limited to the provision of program libraries, which are linked into a program before its execution.

Programs which link to code libraries before execution usually do so either during compilation or just before execution. Linking at compilation time ensures that all the necessary code is available immediately execution is requested, but at the cost of multiple copies of the code throughout the permanent store. Recompile is necessary if a library routine is replaced with a new version. If linking is delayed until just before execution, then the newest versions of routines will always be incorporated, but it may be possible for a routine to be inaccessible to the linker. This introduces a new source of program failure. For a short program which is to be executed frequently, the cost of linking for each execution may be unacceptable.

It is possible to provide operating system utilities which allow more flexible linking strategies, such as the Unix *make* system [Fel79]. However, this introduces a source of complexity in the construction of a programming system by separating the specification of library use within a program from the binding strategy to be used for it. Furthermore, whether or not a particular library routine is executed depends upon the dynamic evaluation of a program, and may not be determined before execution begins. Systems which require linking before the start of execution may therefore waste effort in linking code which is not executed.

In a persistent system, program libraries may be placed as procedures within the persistent store itself. The time of binding to these procedures is specified within the programming language. A procedure in the store may be bound at any time before or during the execution of a program which uses it. Therefore only a single

version of the code is required, and programs have no obligation to bind to code that they never execute.

Some operating systems, such as Multics [Org72], have provided similar solutions to the problem of code and data sharing. Although such systems work well, they do not provide support at a level which allows type-safe sharing within the domain of a programming language.

There are further requirements for a persistent system to maximise its software reuse potential. It must have a flexible binding mechanism which makes it easy for a component builder to place procedures in the store, and for an application builder to bind to them. To allow for different kinds of change in systems, a number of different kinds of bindings are desirable, from fully static to fully dynamic [MAB90].

To minimise the number of components which are necessary, a persistent system should have the ability to abstract over type. Many general-purpose components only require partial knowledge of the type of the data they operate over, and if type abstraction mechanisms are not present then multiple copies of code, with different type descriptions, will appear even in a persistent system.

The language Napier88 [MBC89] includes an environment mechanism which allows for very flexible binding, and two kinds of type abstraction, namely universal and existential quantification. This language will be used to demonstrate how software reuse may be achieved in a suitable persistent system. To demonstrate that the binding methodology shown is made possible by persistent procedures, rather than the particular features of Napier88, it will also be sketched in PS-algol [PS88].

2.2 Binding mechanisms

2.2.1 Environments and binding

2.2.1.1 Static and dynamic environments

Environment in its general sense is defined as the set of identifiers which are in scope in a block-structured programming language. [Str67] Scope levels may be used to provide a contextual naming scheme within a single program. The rules for constructing an environment in block-structured languages are that an identifier is lexically in scope only between its declaration and the end of the block in which it is declared. This is shown in Figure 2.2.1.

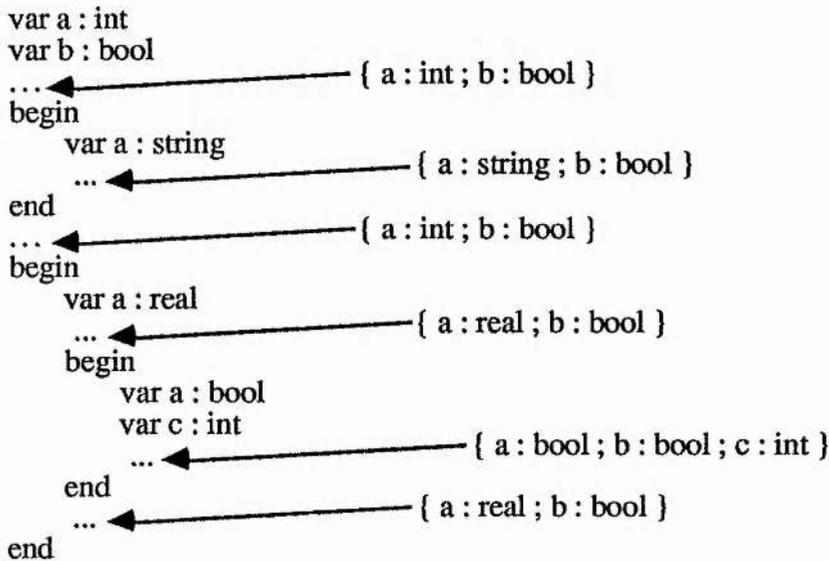


Figure 2.2.1 Static environments

The binding in such schemes is necessarily static; that is, static analysis of a program may always determine which instances of an identifier refer to which declaration. This has the consequence, of particular importance in a persistent programming system, that a declaration may not be referred to from outside the context of the program in which it is made.

In contrast, languages such as Lisp [McC62] use a dynamic binding scheme. An identifier binds to its most recent declaration during the evaluation of a program. Thus the declaration of an identifier enhances some dynamic environment with a new instance, and the use of an identifier causes a dynamic lookup to obtain the current binding. In general with this scheme binding may not be determined by static program analysis, as it depends on evaluation order. This binding mechanism is more flexible than that of block structure, and in a persistent system would potentially allow identifiers in separately prepared programs to refer to the same declaration. Its relative drawbacks are that it is less efficient, and fewer programming errors may be detected by static analysis.

In Napier88, the data type `env` is a dynamic model of static block-structured environments, with individual values of type `env` corresponding to lexical levels. Using values of this type, collections of bindings may be dynamically introduced at the start of a new scope level. The difference between Napier88's model of dynamic binding and that of a language such as Lisp is that Napier88's binding is mostly static, and dynamic binding is restricted to specified points of block entry. At these points, the static environment is enriched with those identifiers which will be bound from the dynamic environment during program execution. Thus each identifier used from a dynamic environment need only be bound dynamically once during the execution of a program. This model gives maximum safety and efficiency while still allowing the flexibility of sharing declarations between contexts.

Most of the work in the design and implementation of environments in Napier88 was carried out by Alan Dearle, and is reported in [Dea88, Dea89].

2.2.1.2 L-value and R-value binding

In a programming language with update, identifiers may bind to locations which contain values [Str67]. The identifier may be required, in different contexts, to be used to access either the location itself or its contained value.

In some languages, such as ML, the consistent use of an explicit dereference operator allows any expression regardless of context to denote either a location or a value. A new binding of an identifier to a location may be introduced, for example, by

```
val a = ref 7
```

The identifier a always denotes the location, and the contents of this location may only be accessed by an explicit dereference. For example, to increment the value within this location the identifier a on the left hand side denotes the location itself, and the expression $!a$ on the right hand side denotes the value contained therein:

```
a := !a + 1
```

Another identifier may be bound to the same location, for example

```
val b = a
```

after which an update to b will also affect the value stored in a . This is quite different from

```
val b = ref !a
```

which declares a new location initialised to the value contained in a .

Locations normally have a very restricted algebra defined over them: assignment and dereference are usually the only operations allowed. To reduce syntactic noise, many block-structured languages do not have an explicit dereference

operator, and whether an identifier denotes a location or its contained value is determined by its use. In Pascal, for example, in the statement

$$a := a + 1$$

the different instances of a bind to different entities: on the right hand side of the assignment operator to the value, and on the left hand side to the location. This may be thought of as an automatic insertion of the dereference operator in appropriate contexts. Most block-structured programming languages use the convention that an expression to the left of an assignment evaluates to a location, and anywhere else to the contained value. For this reason, bindings to a location are usually referred to as L-value bindings, and to a contained value as R-value bindings.

This convention makes it impossible for an identifier to bind to a location which exists before its declaration. For example, in the statement

$$a := b$$

an implicit dereference occurs when b is accessed. The effect is therefore to update the location bound to a with the value in the location bound to b , rather than updating the binding between a and its location with a binding between a and b 's location. Every declaration therefore introduces a new location to which the identifier always remains bound.

If it is wished to share a location between contexts, this may only be achieved by using a data type which itself contains locations, for example a record in a language with store semantics. If a and b are both record values, the statement

$$a := b$$

updates the location bound to a with the same instance of the record referred to in the location bound to b . If the record has a field x , then after this assignment the expressions

$$a(x)$$

and

$$b(x)$$

both specify the same L-value.

The programming language CPL [BBH63] introduces a new operator to allow binding of an identifier to a location which already exists. One of the operators in the language is a location assignment operator, which causes the right hand side of the assignment statement to evaluate to a location rather than its contained value.

The statement

$$a \equiv b$$

causes b to evaluate to the location bound to b , rather than the value stored in it. The effect of the statement is therefore to update the binding between a and the location it refers to. The purpose of this feature is to allow the manipulation of commonly accessed locations within data structures, so that, for example

$$\begin{aligned} &\text{let } a \equiv b[2, 2] \\ &a := \langle \text{expression} \rangle \end{aligned}$$

would have the same effect as

$$b[2, 2] := \langle \text{expression} \rangle$$

The general principle of evaluation to L-value according to convention saves introducing program noise, but assumes that the language designers may correctly determine all desirable uses of L-values.

In Napier88 evaluation to L-value occurs in two places, according to convention. The first is on the left hand side of an assignment operator. The second occurs on binding to a dynamic environment. At the point where an identifier is bound to a dynamic environment, the binding occurs to the L-value. This introduces the property of allowing identifiers within independently prepared programs to dynamically refer to the same L-value.

2.3 Napier88 environments

Dynamic binding may be modelled in Napier88 by the data type `env` [Dea89]. A value of this type represents a collection of bindings, each of which consists of an identifier, a type, and a value. The Napier88 model of mutability requires also a constancy indicator in association with each binding.

2.3.1 Creation and use of Napier88 environments

A new dynamic environment may be created in Napier88 with a procedure found in the standard environment, itself called *environment*. This is of type

```
proc( → env )
```

and returns an environment with no bindings. A new environment may be created and bound to an identifier by

```
let e := environment()
```

A binding may be added to an environment in the same way that a binding is normally added to the static environment, using a `let` declaration. This must now specify a destination environment, as well as an identifier, for the initialising expression. An integer value may be added to the new environment:

```
in e let a := 7
```

Notice that this does not enhance the static environment, only the environment denoted by *e*. Declarations enhance the static environment only if no dynamic environment value is stated. Thus

```
let a := 7
print( a )
```

will cause the the *print* procedure to be called with the value 7, but

```
in e let a := 7
print( a )
```

will fail during the static analysis of the program, as the identifier *a* is not available for use in the call of the *print* procedure.

Due to the dynamic nature of these environment values, one further operation is required which is not normally available in a static environment. This is to drop a binding from an environment, which may be written, for example, as

drop a from e

An important attribute of the **drop** operation is that it does not affect any program which has already formed a binding to the identifier which is dropped. It removes only the identifier from the environment, and not the location which is denoted by the identifier.

The final requirement is a method for a binding from a dynamic environment to be brought into a static one. In Napier88 this may be achieved by the **use** clause, for example:

```
use e with a : int in
begin
  print( a )
end
```

The **use** clause specifies the binding between the static and dynamic environments, performed once upon the block entry, and within the **use** clause the identifier *a* is

in the static environment. Notice that the type must be specified at the point of binding. All typechecking may therefore be performed statically, except for a single check at the time of binding to the dynamic environment.

The **use ... in** part of the **use** clause specifies a new lexical level, in which all of the bindings specified from the dynamic environment are introduced and bound to locations within the static environment. For example, using a language with the location assignment operator of CPL, the Napier88 code

```
use e with a , b : int ; c : string in  
begin  
    ...  
end
```

could be expressed as

```
begin  
    let a : int  $\equiv$  e( "a" )  
    let b : int  $\equiv$  e( "b" )  
    let c : string  $\equiv$  e( "c" )  
    begin  
        ...  
    end  
end
```

where the dynamic environment e is represented as a lookup procedure with varying result type.

One further point about the **use** clause is that it may specify a subset of the identifiers actually present within the environment. This partial specification means that a programmer need specify only the type of data which is of interest in each particular context.

2.3.2 L-value binding

The semantics of L-value and R-value bindings in block structure are preserved in the dynamic environments of Napier88. Thus the **use** clause brings the location into scope, and this location may be used as a normal L-value. The implication of

this is that, should an update be performed to a location bound from a dynamic environment, the effects of this update are apparent to any other context where the same binding is in scope. Thus side-effects are not restricted to the local scope. For example, the program in Figure 2.3.1 declares an identifier *a* within a new environment. Two dynamic bindings to this identifier will occur during the program's execution. The first of these, in the procedure *change*, is evaluated as the L-value associated with *a*, and the second as the R-value.

```
let new = environment()
in new let a := 7

let change = proc( e : env ; newValue : int )
  use e with a : int in a := newValue

change( new , 3 )
use new with a : int in print( a )
```

Figure 2.3.1 Binding to an L-value

The L-value binding in the procedure means that the update will show through, and the procedure *print* will be called with the value 3, rather than 7.

2.3.3 Constancy

In Napier88, constancy is an attribute of an L-value rather than of type [GM79]. This means that each L-value must have a specification of its constancy. With the names which denote locations in static environments, this is always statically determinable. In Napier88 constancy is specified by a single equality symbol in a declaration, for example

```
let a = 7
```

specifies *a* to be constant, whereas

```
let a := 7
```

allows *a* to be updated at some future time.

The same syntax is used to specify a constant L-value within a dynamic environment. The only difference is that violation of constancy may no longer always be detected statically, and so a dynamic constancy test is required. For example, the program in Figure 2.3.2 is the same as in Figure 2.3.1 except that the declaration of the identifier *a* is as a constant. This program will therefore fail during the execution of the procedure *change*.

```
let new = environment()
in new let a = 7

let change = proc( e : env ; newValue : int )
  use e with a : int in a := newValue

change( new , 3 )
use new with a : int in print( a )
```

Figure 2.3.2 Binding to a constant L-value

This failure could occur at different times. To catch the error as soon as possible, it would occur at the time of binding the updated identifier to the constant location. Alternatively, the failure may not be detected until the update is performed. This allows the successful completion of a program in which the update is present statically but is not executed. Napier88 chooses the latter of these options.

2.3.4 Persistence

The Napier88 system has a single predefined identifier, *PS*. This is a procedure which returns a value of type *env*. The environment returned is the root of the persistent store, and any value which is reachable from this root will persist. Any data value may be made to persist by arranging for it to be within this transitive closure.

As environments themselves are first-class data types, a network of environments may be used to model a naming convention within the persistent store. This could simulate, for example, a Unix directory structure. Figure 2.3.3 shows how this

could be used to create a new binding within such a scheme. Notice that this is an over-restrictive example of how environments may be used: no restrictions of either topology or type within a naming scheme are imposed by the language mechanism.

```
use PS() with user : env in
use user with staff : env in
use staff with richard : env in
begin
  let new = environment()
  in new let a := 7
  in richard let integers = new
end
```

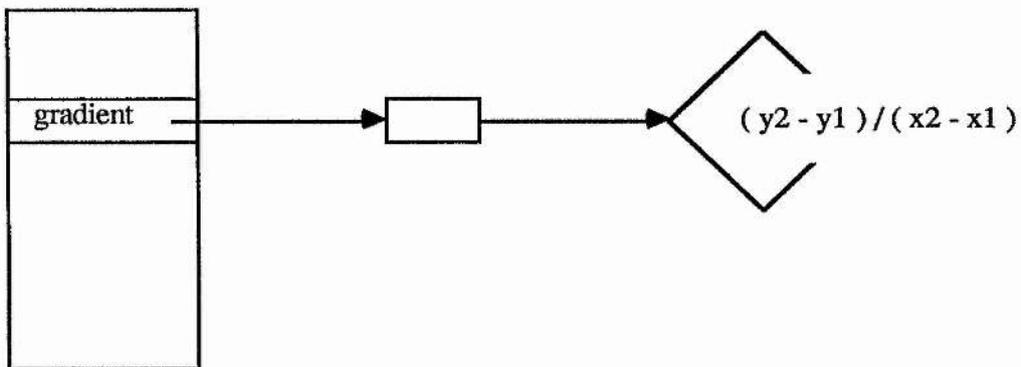
Figure 2.3.3 Binding to an example naming scheme

The program in Figure 2.3.3 causes a dynamic bind to an environment called *richard* which should already be in the persistent store. This environment becomes usable within the static environment of the main block. Another environment is created, and the binding of *a* to a location initialised with the value 7 is placed in it. The new environment is then placed in the environment *richard*. The new environment and its contained value are now within the transitive closure of the persistent store, and so will be made to persist after the program has finished executing.

2.3.5 Programming system construction

In Napier88 procedures are first-class data values, and so procedure values may be declared within static and dynamic environments in the same way as any other value. The basic binding mechanisms shown above may be used to model a number of different methodologies for the construction of software systems out of components. For the sake of clarity only the top level of the persistent store will be used in the following examples.

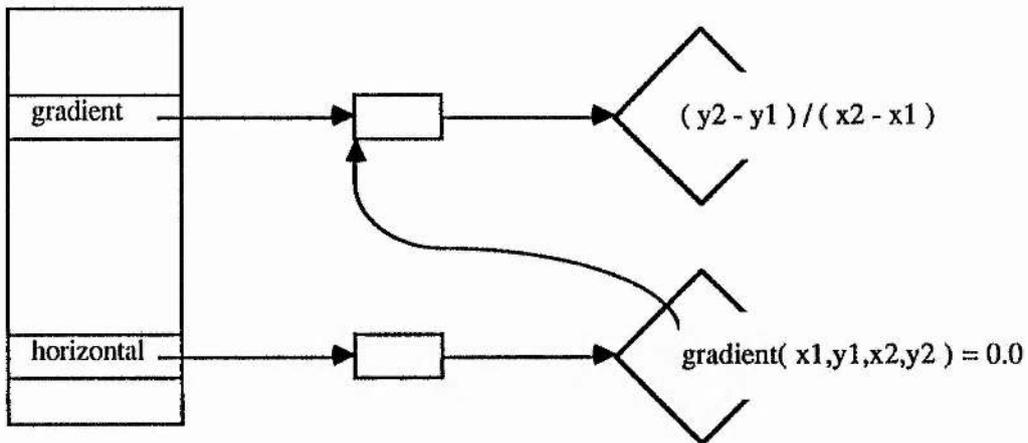
Figure 2.3.4 shows how a procedure which is to be used as a component may be declared and stored. This program contains the declaration of a procedure value which calculates the gradient of a line. When the program is executed, it places the new procedure within the persistent store. Above the program in Figure 2.3.4 is a diagrammatic representation of the persistent store after the execution of the program.



```
in PS() let gradient := proc( x1,y1,x2,y2 : real → real )
    ( y2 - y1 ) / ( x2 - x1 )
```

Figure 2.3.4 Adding *gradient* to the persistent store

Figure 2.3.5 shows the declaration of a procedure which uses the *gradient* component, and is itself placed in the persistent store. This program first looks up the *gradient* procedure from the persistent store. The binding here is to the L-value declared in the previous program. Another procedure is then declared which uses the binding to *gradient*, and this procedure is then itself placed in the persistent store. Notice that the new procedure is statically bound to the procedure *gradient*: no dynamic binding or typechecking is required for the execution of *horizontal*.



```

use PS() with gradient : proc( real,real,real,real → real ) in
begin
  let new = proc( x1,y1,x2,y2 : real → bool )
    gradient( x1,y1,x2,y2 ) = 0.0

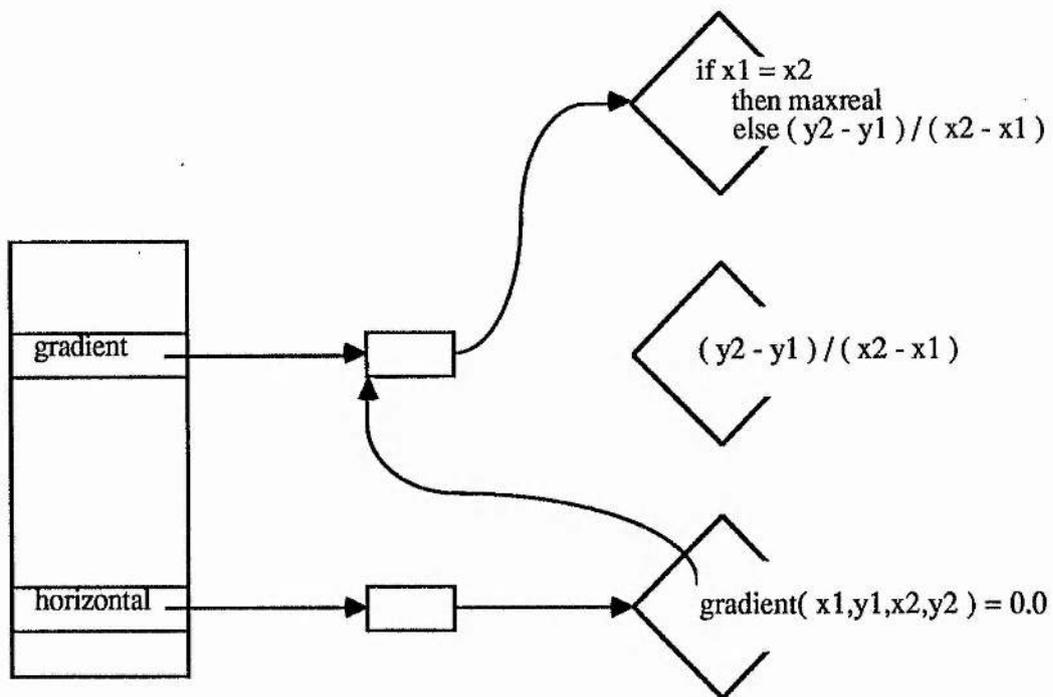
  in PS() let horizontal = new
end

```

Figure 2.3.5 Static binding within the persistent store

A component writer however is unlikely to provide a perfect definition at the first attempt, and so there may be a need to later update it. For example, after the procedure *horizontal* has been installed in a system, it may be noticed that *gradient* will fail if the two *x*-values happen to be the same. Figure 2.3.6 shows how a program which replaces the existing version of *gradient* with a new one may be written.

As both this program and *horizontal* are bound to the L-value denoted by *gradient* within the persistent store, the effect of this update is that future calls to *horizontal* will use the new version of *gradient*. Therefore the error has been corrected in place without any requirement to re-link or re-compile programs which use the component. This style of binding gives very effective support for the construction and development of new software systems in modules.



```

use PS() with gradient : proc( real,real,real,real → real ) in
  gradient := proc( x1,y1,x2,y2 : real → real )
    if x1 = x2 then maxreal else (y2 - y1) / (x2 - x1)
  end
end

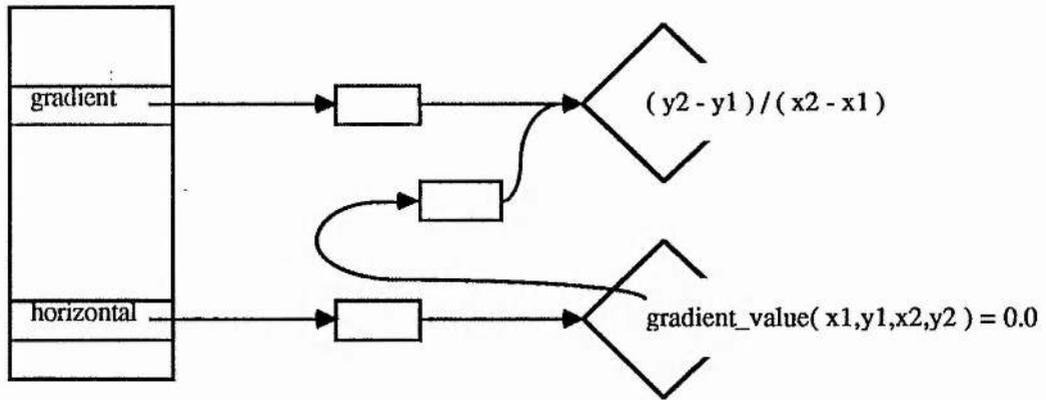
```

Figure 2.3.6 Updating the L-value binding

The above method of binding is unsuitable for some applications. For example, the writer of the horizontal procedure may know that it will work with the version of *gradient* in the store at the time, and therefore requires to bind to this version and remain unaffected if it is replaced. This may be achieved by binding to the R-value associated with *gradient*, as demonstrated in Figure 2.3.7.

Here, a new location *gradient_value* is declared and initialised with the R-value of the binding *gradient*. This location is declared as constant, and may never be updated. Therefore the new procedure is bound statically to the value of the procedure body of *gradient* which is in the store at the time this program is executed. The diagram in Figure 2.3.7 shows the state of the store after this program has been executed but before *gradient* is updated by the program in Figure

2.3.6, and Figure 2.3.8 shows the state of the store after *gradient* is updated by the program in Figure 2.3.6.



```

use PS() with gradient : proc( real,real,real,real → real ) in
begin
  let gradient_value = gradient
  let new = proc( x1,y1,x2,y2 : real → bool )
    gradient_value( x1,y1,x2,y2 ) = 0.0
  in PS() let horizontal := new
end

```

Figure 2.3.7 Creating a binding to an R-value

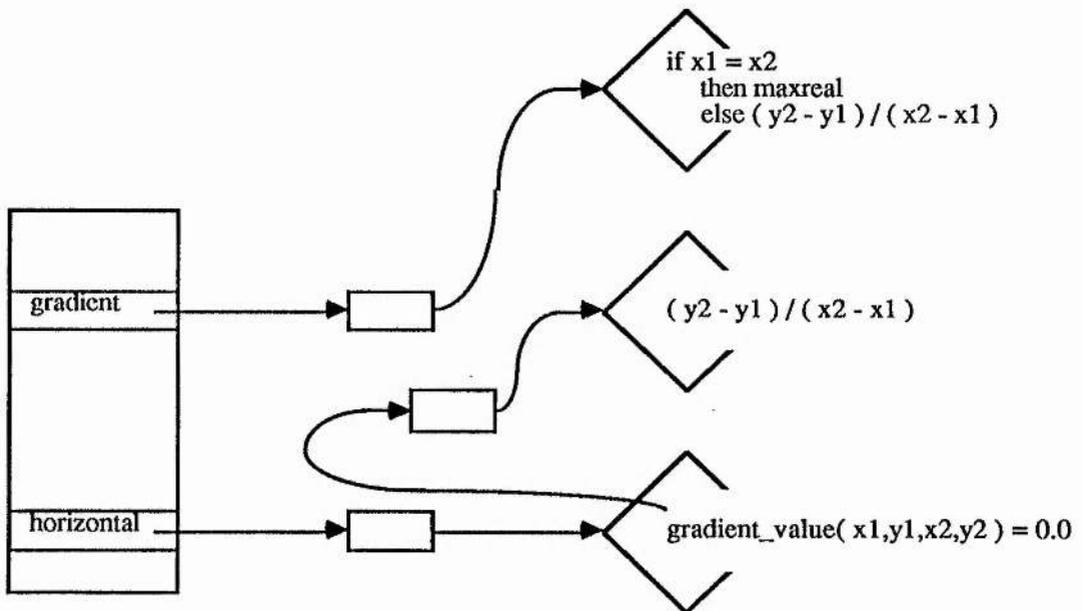
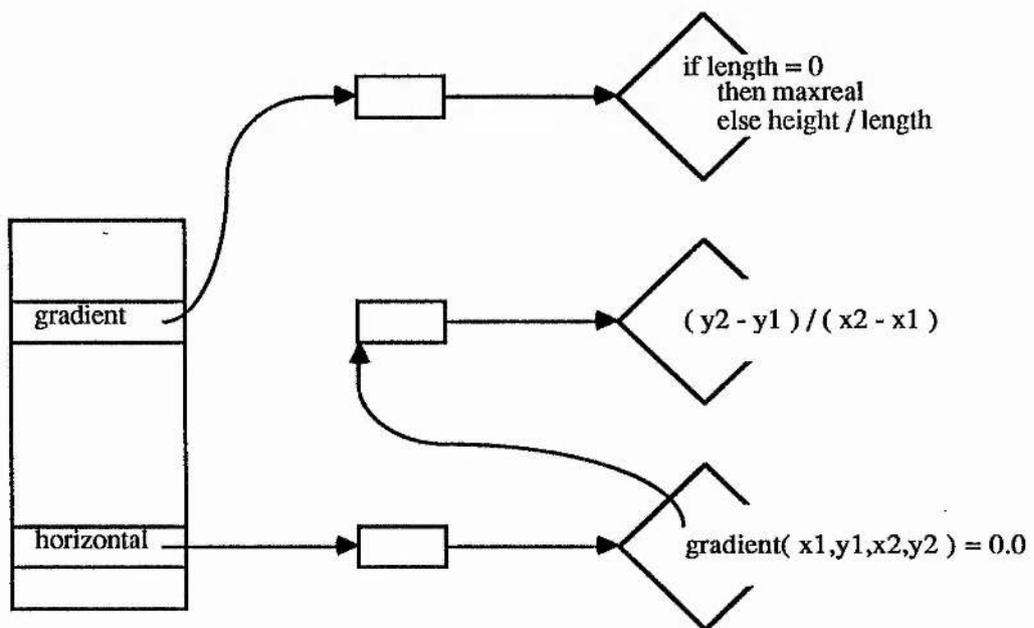


Figure 2.3.8 After the update with the R-value binding

There is a further possibility that the type of the procedure may be required to change, and in this case an update would not be possible. Figure 2.3.9 shows how replacement with a procedure of a new type may be achieved by dropping the binding and replacing it with a new one. The diagram shows the state of the store if the program was executed immediately after the programs which first defined *gradient* and *horizontal*, in Figure 2.3.4 and Figure 2.3.5.

Notice that the dropping of the identifier from the environment does not affect any program which has already bound to its associated L-value. Only the access route via the environment is removed, and the associated location and value may still exist.



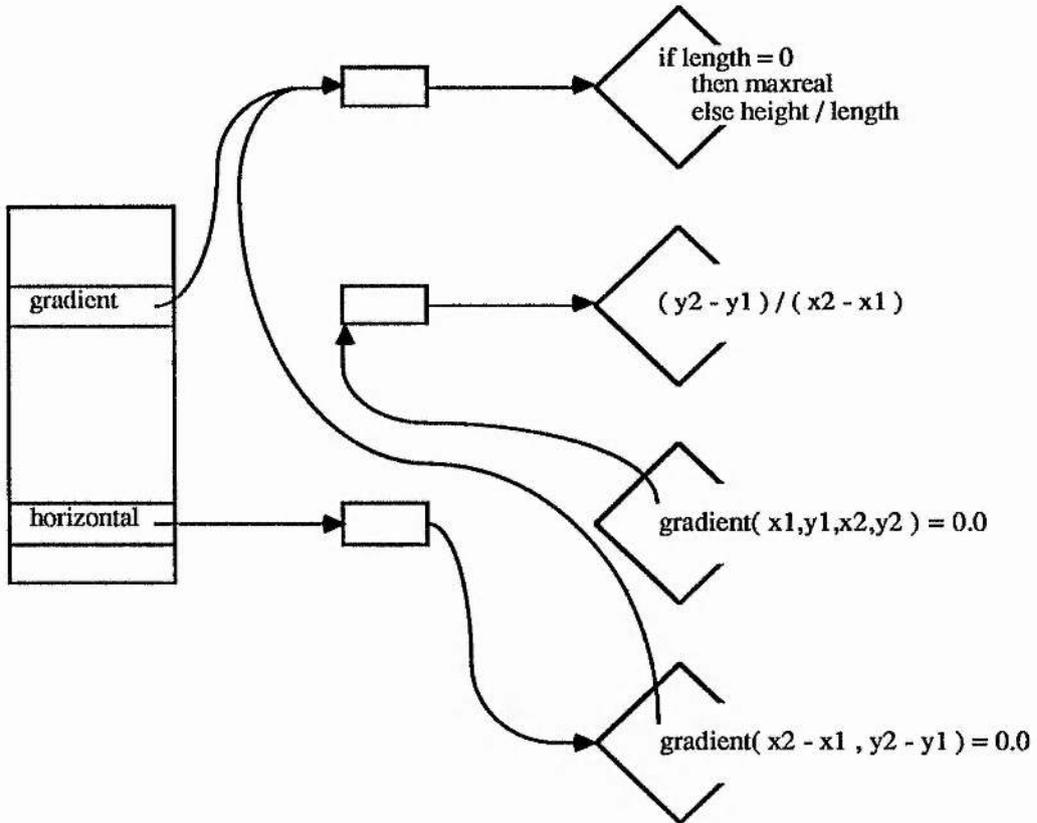
```

drop gradient from PS()
in PS() let gradient := proc( length,height : real → real )
  if length = 0 then maxreal else height / length
  
```

Figure 2.3.9 Making a new location for an identifier

The procedure *horizontal* would be unaffected by the execution of the above program, as they were both statically bound to the procedure below the level of

this access route. If *horizontal* is required to make use of the new *gradient* procedure, it must be re-written for the new type. This is demonstrated in Figure 2.3.10.



```

use PS() with  gradient  : proc( real,real → real ) ,
              horizontal : proc( real,real,real,real → bool )
in
  horizontal := proc( x1,y1,x2,y2 : real → bool )
                gradient( x2 - x1 , y2 - y1 ) = 0.0

```

Figure 2.3.10 Binding to the new location

The execution of this program causes *horizontal* to be overwritten with a new procedure which is statically bound to the L-value of the most up-to-date version of *gradient*. (Figure 2.3.6) Notice in this example that the effects of the change in type of the procedure have been limited to those components which access it directly. A program which is statically bound to the L-value of *horizontal* may be unaware of its internal change and continue to execute correctly.

Notice that at this point no access exists to the earlier versions of both procedures, and the locations, closures and code objects associated with them may now be garbage collected.

One further style of binding is possible. A procedure may dynamically bind to a location each time it is executed, for example *dynamicHorizontal* may be defined as in Figure 2.3.11.

```
in PS() let dynamicHorizontal =  
  proc( x1,y1,x2,y2 : real → bool )  
  use PS() with  
    gradient : proc( real,real,real,real → real ) in  
      gradient( x1,y1,x2,y2 ) = 0.0
```

Figure 2.3.11 Fully dynamic binding

Here the dynamic binding to the stored value of *gradient* is not performed until *horizontal* is executed, but must be re-performed on each execution. This means that if the identifier is dropped and replaced with another of the same type then the replacement value will be bound to. If it is dropped without replacement, or replaced with a different type, then the execution of *dynamicHorizontal* will fail. However, this binding ensures that the program will access the latest value with this identifier within the environment. This is the style of binding provided by the Multics operating system.

2.4 Incremental construction and release

The combination of L-value bindings and procedures in the persistent store gives a useful model for the construction of complex software systems. What would normally be constructed as a complex program, perhaps with many mutually recursive procedures, may instead be written with each procedure in a separate module. A stub for each procedure is first placed in a suitable location. Each procedure may now be written as a program which first statically binds the L-

values of any other procedures it requires, then declares the required procedure, and finally updates its own L-value in the store. This gives a model where any procedure within the whole system may be individually edited and recompiled without any change to the rest of the system.

Napier88 is particularly suited to modelling such a methodology because of its ability to share L-values between identifiers in different contexts: in fact, this was a major reason for some of the design decisions made in the Napier88 environments. The advantages of this binding mechanism will be highlighted by showing the same example first in Napier88, and then in PS-algol, a persistent language with first-class procedures and L-value bindings, but without the ability to share a location between identifiers. This example is due to Cutts and Dearle [CD90].

A good example of a complex piece of software with many mutually recursive procedures is a programming language compiler designed by the recursive descent method [DM81]. To illustrate the construction methodology, the construction of a parser for the following simple grammar will be illustrated:

```
<clause> ::= <expr> | <if clause>
<if clause> ::= if <boolean> then <clause> else <clause>
<expr> ::= <arith> | <boolean> | <clause>
<arith> ::= 1 | 2 | 3 | ... | <clause>
<boolean> ::= true | false | <clause>
```

The interrelationships of the corresponding parsing procedures may be described diagrammatically as in figure 2.4.1.

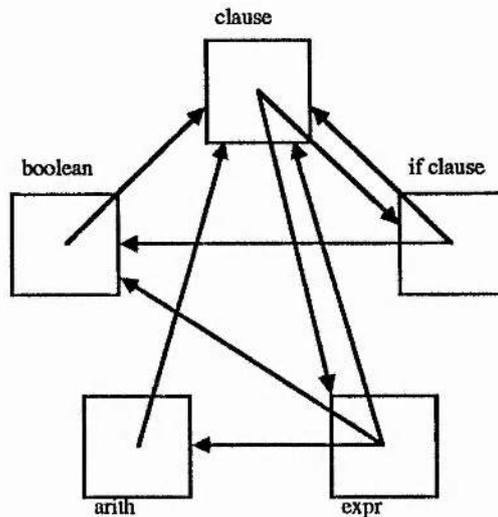


Figure 2.4.1 Mutual recursion

2.4.1 The example in Napier88

To set up such a system in Napier88, first procedure stubs must be set up for each procedure in an environment or a collection of environments. For the sake of clarity, each of the parsing procedures is assumed to be without parameters or result, and will be placed in the same environment. Stubs may be set up as in Figure 2.4.2.

```

use PS() with user : env ; environment : proc( → env ) in
begin
  let new = environment()

  in new let clause := proc() ; {}
  in new let if_clause := proc() ; {}
  in new let expr := proc() ; {}
  in new let arith := proc() ; {}
  in new let boolean := proc() ; {}

  in user let parser = new
end

```

Figure 2.4.2 Creating the procedure stubs

A template is then constructed for each of the procedures. This template must bind to the L-value of the other procedures which it uses, declare the procedure, and

update its own L-value within the persistent store. This then allows each procedure to be independently edited, compiled and loaded into the parser without any change to the rest of the system. For example, the template for the procedure *clause* would be as in Figure 2.4.3. In this program, of the three identifiers bound to in the *parser* environment only *clause* is used as an L-value, with *if_clause* and *expr* being used as R-values within the new body of the procedure *clause*.

```

use PS() with user : env in
use user with parser : env in
use parser with clause , if_clause , expr : proc()
in
    clause := proc()      !* this is the part that may be edited later
        if nextSymbol = "if"
            then if_clause()
            else expr()

```

Figure 2.4.3 Updating the procedure *clause*

Once all of these templates have been satisfactorily completed, the system is ready for release. At this point, it may no longer be desirable for any user to have the ability to update the value of a procedure. All unnecessary access may at this point be dropped, with only a single procedure left to access the entire software system. In this example, this could be arranged as in Figure 2.4.4.

```

use PS() with user : env in
use user with parser : env in
use parser with clause : proc() in
begin
    let call_parser = proc() ; clause()
    drop parser from user
    in user let compiler = call_parser
end

```

Figure 2.4.4 Releasing the completed system

After the execution of this program, all access to the individual procedures has been lost except for the ability to call the top-level one. This has effectively sealed the system, and access to its components is no longer possible.

2.4.2 The example in PS-algol

2.4.2.1 The data type `pntr`

The type `pntr` in PS-algol denotes the infinite union of all structure types. A structure type may be declared as, for example,

```
structure student( name : string ; matricNo : int )
```

This has the effect of introducing the identifier *student* as a constructor function. All values created using it are of type `pntr`, rather than type *student*, and are type compatible with any other `pntr`.

Type safety is maintained by a dynamic type check. Although all structure values are type compatible, the field names in a structure declaration are introduced to the static environment and are unique within any scope level. When a dereference is encountered, it is possible to determine the expected type of the `pntr` value by means of the fieldname. This assertion is checked before the dereference is executed. Thus in the example

```
let deref = proc( x : pntr → string ) ; x( name )
```

the identifier *name* asserts statically that the value *x* should be of type *student*. If it is not, a failure will be generated at execution time.

The reserved identifier `nil` denotes a reserved value of type `pntr`. It stands for a unique structure value which has no fields.

2.4.2.2 Persistence

Persistence in PS-algol is modelled by a set of standard procedures. The model of persistence is reachability from specified persistent roots. A new root may be created by the standard procedure *createDatabase*, which takes a database name and

password as parameters and returns a value of type **pntr**. The root may be accessed by another program invocation by use of the *openDatabase* procedure.

The root returned by the creating or opening a database is a pointer to an instance of an associative lookup table, of type *Table*. Further standard procedures are defined to manipulate these values. *sEnter* takes as parameters a string key, a *Table*, and a value of type **pntr**. *sLookup* also takes a string key and a *Table*, and returns the most recent value which has been stored using *sEnter* with the same key and *Table*. Thus these tables may be used to store a value of arbitrary complexity in a manner which is orthogonal to any of its attributes.

In PS-algol, the persistent store is only checkpointed when an explicit *commit* procedure is called by the user. If a program terminates without executing a *commit*, then no change is made to the persistent store. *commit* is a procedure of type

```
proc( → pntr )
```

which returns **nil** if successful and a record of the error otherwise.

As an example, the following program places a new *student* in the persistent store:

```
let newDB = createDatabase( "students" , "friend" )
structure student( name : string ; matricNo : int )

sEnter( "new student" , newDB , student( "Harold" , 12345 ) )

write if commit() = nil then "ok" else "failed"
```

The student may be accessed by another program:

```
let newDB = openDatabase( "students" , "friend" , "read" )
structure student( name : string ; matricNo : int )

let harold = sLookup( "new student" , newDB )
```

These two programs refer to the same instance, and so the L-values contained within the structure are sharable between separately compiled programs.

2.4.2.3 System construction

As with the Napier88 example, procedure stubs are first created for the different parsing procedures. This is achieved by the program in Figure 2.4.5. This program first of all creates a new database, which will act as a root of persistence. This has the name "parser" and the password "friend", and its initial form is an empty lookup table. The next line represents the type of a structure which will be used to hold all the parsing procedures. A data type such as a structure is necessary to model the sharing of L-values which is required. An instance of this type, initialised with procedure stubs, is created, and entered into the root table of the database with the string "parser procs". The *commit* is then required to make the effects of this program persist.

```
let theDb = createDatabase( "parser" , "friend" )
structure procs( proc() clause , ifClause , expr , arith , boolean )

let stub = proc() ; {}
let new = procs( stub ,stub ,stub ,stub ,stub )

sEnter( "parser procs" , theDb , new )

write if commit() = nil then "Parser db created" else "failed"
```

Figure 2.4.5 Creating the procedure stubs

Figure 2.4.6 shows the program which contains the template of the procedure *clause*. This program first opens the database "parser" to access the data structure which contains all the parsing procedures, then declares a new procedure body which may use these procedures, and finally updates the location which contains the procedure *clause*. Again, an explicit *commit* is necessary to make the effects permanent. Notice that, as sharing L-values between identifiers is not possible in PS-algol, dereferences must be made to access each of the other procedures.

```

structure procs( proc() clause , ifClause , expr , arith , boolean )
let theDb = openDatabase( "parser" , "friend" , "write" )
let theProcs = sLookup( "parser procs" , theDb )

theProcs( clause ) := proc()
begin
    let IfClause = theProcs( ifClause )
    let Expr = theProcs( expr )

    !* this is the part which may be edited later
    if nextSymbol = "if"
        then IfClause()
        else Expr()
end

write if commit() = nil then "clause installed" else "failed"

```

Figure 2.4.6 Updating the procedure *clause*

Once the system has been satisfactorily completed, it may be released by arranging for a different access to the top-level procedure and removing the other access routes from the system. This could be achieved by the program in Figure 2.4.7.

This program first declares two structures, the first to access the existing procedures and the second to store only the top-level one. The top-level procedure *Clause* is then looked up from the old database. A new database is created, and a structure containing the top-level procedure is placed in it. The next action is to remove access to the old procedure structures, which may be achieved by overwriting the containing structure with a **nil** pointer. Once again *commit* must be executed to make these changes permanent.

```

structure procs( proc() clause , ifClause , expr , arith , boolean )
structure Parser( proc() theParser )

let oldDb = openDatabase( "parser" , "friend" , "write" )
let theProcs = sLookup( "parser procs" , oldDb )
let Clause = theProcs( clause )

let release = createDatabase( "Release Parser" , "password" )
sEnter( "Parser" , release , Parser( Clause ) )

sEnter( "parser procs" , oldDb , nil )

write if commit() = nil then "System installed" else "failed"

```

Figure 2.4.7 Releasing the completed system

2.4.3 Conclusions

A model of system construction and release in persistent systems has been shown. This relies upon persistence, first-class procedures, and the sharing of L-values between programs. The methodology has been sketched in two persistent languages with these features, namely Napier88 and PS-algol.

This model of system construction may be likened to the use of scaffolding during the construction of a civil engineering project. Scaffolding allows the builders to access those parts of the construction which would not be accessible by normal users. Once the construction reaches completion, the scaffolding is then removed to prevent this style of access for everyday users.

2.5 Type abstraction

Given that a flexible model of software component sharing may be achieved, a further desire is to maximise the amount of sharing possible within a system. To achieve this, a further requirement for persistent languages is the ability to abstract over type. Type abstraction allows a single procedure to be used over data of various types, in cases where the procedure requires only a partial knowledge of

the type of its operands. The absence of type abstraction may lead to multiple copies of code which perform the same logical actions.

For example, consider the procedure specifications in Figure 2.5.1. The square brackets in these procedure types denote type parameterisation. This is merely a shorthand method which allows abstraction over the syntactic definition of types, and so saves writing down a full description of a type each time it is used.

```
cardinality_array_student : proc( array[ student ] → int )  
    ! returns the number of students in the array  
cardinality_list_student : proc( list[ student ] → int )  
    ! returns the number of students in the list  
cardinality_array_office : proc( array[ office ] → int )  
    ! returns the number of offices in the array  
cardinality_list_office : proc( list[ office ] → int )  
    ! returns the number of offices in the list
```

Figure 2.5.1 Different type specifications for the same task

In this example a potentially geometric explosion of system components can be seen. Even for a task as simple as this, the number of procedures required is the number of different implementations multiplied by the number of different uses. This problem becomes even worse with procedures which operate over a number of different parameters.

In a typeless language, or in a dynamically typed language, only a single procedure would be required:

```
cardinality :  
    ! procedure which returns the number of items in a collection
```

Although the logical definition is now precisely what is required, and is no longer over-specified, all the benefits associated with static typing have been lost. For example, no error would necessarily be detected by program analysis if the procedure were applied to a data object which did not represent any kind of

collection, and the program would fail during execution. The difference between untyped and dynamically typed languages is that the latter would fail in a well-specified manner.

The ideal situation would be to assert as much static detail as possible, but without losing the required generality. The required procedure type would ideally have the following information contained in it

```
cardinality : proc( collection of something → int )  
!           returns the number of items in the collection
```

With this type specification the resulting system may still be fully statically type checked, with the attendant benefits of safety and efficiency.

Such types may be achieved using the mechanisms of universal and existential quantification, which are particular models of parametric polymorphism and abstract data types. Using parametric polymorphism, the type of the collected values may be abstracted over, and using abstract types the type of the implementation of the collection may be abstracted over. In this way the full safety of static checking is maintained, but no over-specification is necessary.

2.5.1 Universal quantification - specialised reuse

Universal quantification is the mechanism by which one or more of the parameter types of a procedure may be abstracted over. For example, the identity function is one which may be applied to a parameter of any type. Using the notation introduced by Cardelli and Wegner [CW85], the type of this function may be denoted by

$$\forall t. (t \rightarrow t)$$

which should be read as

"the type of a function which, for all t in the set of types, takes an argument of type t and returns a result of the same type t "

In the Cardelli and Wegner set model of types, this type represents the set which is the infinite intersection of all procedure types which take a single parameter and return a result of the same type. In Napier88, the syntax of this type is denoted by

proc[t]($t \rightarrow t$)

which should be read in the same way. To construct an instance of such a procedure identifiers for the formal parameters and a procedure body are necessary:

proc[t]($x : t \rightarrow t$) ; x

which may be bound to an identifier by a declaration:

let *identity* = **proc**[t]($x : t \rightarrow t$) ; x

To call the procedure, it must be applied to an appropriate type parameter before being applied to a value:

let *seven* = *identity*[**int**](7)

or

let *hello* = *identity*[**string**]("hello")

or even

let *identity2* = *identity*[**proc**[t]($t \rightarrow t$)](*identity*)

Using this type abstraction mechanism it is possible to write a smaller set of *cardinality* functions. A procedure which returns the number of items in a list is independent of the type of these items, and may be abstracted over in this manner. The above set of four procedures may be replaced by two, shown in Figure 2.5.2.

```
cardinality_array : proc[ t ]( array[ t ] → int )  
    !     returns the number of items in the array  
  
cardinality_list : proc[ t ]( list[ t ] → int )  
    !     returns the number of items in the list
```

Figure 2.5.2 Reducing the type specifications

To use a procedure which calculates the number of students in a list, the programmer may now write

```
let number = cardinality_list[ student ]( myList )
```

instead of

```
let number = cardinality_list_student( myList )
```

It can be seen that the programming task is identical in both cases: the programmer must somehow obtain an appropriate procedure and apply it to the list of students. The difference is in the way that this procedure is obtained: in the first case, the type of the particular list may be simply used as a parameter to specialise a more general procedure, whereas in the second case the type must be used in some way to aid the finding of the correct procedure.

The first case, however, greatly reduces the number of names which are necessary within the system, and therefore facilitates the task of finding the correct one. As only a single procedure exists, the correctness criteria of the system need only be applied in a single case, and the chances of error in the system are reduced. In the second case, there is a much smaller possibility that the required procedure even exists. In terms of software reuse, it is much more likely that the more general procedure may be suitable for use as a component in a later system. This also applies to the construction of a single system, where the types used are prone to evolution between the software engineering cycles. Therefore the difficulty of changing a system between cycles may be lessened.

2.5.2 Existential quantification

Existential quantification [MP88] is another mechanism which allows types to be abstracted over, but in the signatures of record types rather than procedure types. Again using the terminology introduced by Cardelli and Wegner, a simple example of an existential type may be denoted

$$\exists i . [\text{value} : i]$$

The square brackets here are the notation used by Cardelli and Wegner to denote a record type. The above type description may be read as

"the type of a record with a single field *value* whose type has only the restriction that it is a member of the set of all types"

The Napier88 syntax which denotes this type is

$$\text{abstype}[i](\text{value} : i)$$

which again may be read the same way.

There are, however, no useful operations over this type. A more interesting example is

$$\text{abstype}[i](\begin{array}{ll} \text{value} & : i ; \\ \text{change} & : \text{proc}(i \rightarrow i) ; \\ \text{print} & : \text{proc}(i \rightarrow \text{string}) \end{array})$$

As the existential quantifier *i* quantifies the whole record type, it is possible to include as part of the definition the types of procedures which manipulate values of the abstracted type. However, the concrete type with which the value in the *value* field is initialised is not accessible. Such types therefore have the power to abstract over the implementation of complex values by keeping their representation types hidden. This is the other facility that was required to achieve the goal of only a single *cardinality* function.

To create a value of an existential type, a language must contain a type widening operator which in some way accredits a structure type with its more general existential form. In Napier88, the identifiers given to existential types in the type algebra are overloaded with this type widening capability. Thus to create a value of the above type a name must first be assigned for it:

```
type example is abstype[ i ](   value   : i ;
                               change  : proc( i → i ) ;
                               print   : proc( i → string ) )
```

The name *example* may now be used as a constructor function whose result is a value of the abstract type *example*, effectively losing some of the type information of its arguments. For example, if the following procedures are already in scope

```
successor : proc( int → int )
iformat   : proc( int → string )
```

then

```
example[ int ]( 7 , successor , iformat )
```

creates a value of type *example*. The explicit type parameterisation could always be inferred from the existential signature, but in Napier88 is included for syntactic completeness. After this value has been created, its abstracted types may never again be discovered.

Before an existentially quantified type can be used a special binding operation must be provided to preserve static typechecking properties. Different instances of an existentially quantified type may be constructed by abstracting over different concrete types. As the values and procedures may abstract over different implementations, it is essential that they may not be mixed. Once a type has been abstracted over within the context of a single record value, components of that type should only be compatible with each other.

Napier88 provides a **use** clause to introduce a temporary static binding, an example of which is shown in Figure 2.5.3.

```
use <abstract value> as X in  
begin  
    X( change )( X( value ) )  
    ...  
end
```

Figure 2.5.3 The existential use clause

This static binding gives a framework in which it is possible to restrict mixing of components to those cases where it may be statically proved that they are from the same existential value. Without the **use** clause this may not be possible, as in general the identity of the abstract value may not be determined statically. If a temporary static binding is not insisted upon, then dynamic typechecking must be introduced [CM88, OTC90].

The specific problem of abstracting over the implementation of a collection type is now addressed. For the sake of clarity the operations over a collection type will be restricted to the addition of a new element and scanning the collection. When a new collection is created it is empty. Scanning will be modelled by a procedure whose parameter is a collection and whose result is another procedure. On successive calls, the result procedure will return a new value from the collection. When the collection is exhausted, a special fail value will from then on be returned.

An existential type which models a collection of values of type t may be denoted as in Figure 2.5.4. As well as the two procedures which operate over collections, the package contains an identifier *theCollection* which denotes the collection itself, and *failValue*, which is the result of the result of the scan procedure after it has returned every element in the collection.

```

type collection[ t ] is abstype[ i ]
(
  theCollection      : i ;
  failValue         : t ;
  add               : proc( i, t → i ) ;
  scan              : proc( i → proc( → t ) )
)

```

Figure 2.5.4 An abstract collection type

For each different implementation type, a procedure is required to create a value of the abstract type using the appropriate implementation. As this procedure must be able to create a collection of any type, it is itself universally quantified, where its type parameter abstracts over the type of values within the collection. The procedure has a single parameter, which must be supplied a value of the collected type to be used by convention as a fail value. For example, Figure 2.5.5 shows how a collection using a list implementation may be defined.

```

let listCollection = proc[ t ]( failValue : t → collection[ t ] )
begin
  !* specify representation type (list )
  rec type list is variant( cons : node ; tip : null )
  & node is structure( hd : t ; tl : list )

  !* procedure to create empty list
  let createList = proc( → list ) ; list( tip : nil )

  !* procedure to add an element to a list
  let addToList = proc( l : list ; x : t → list )
    list( cons : node( x, l ) )

  !* procedure which generates another procedure, each call
  !* returns the next node from the original list parameter
  let scanList = proc( listPointer : list → proc( → t ) )
  begin
    proc( → t )
    project listPointer as temp onto
      cons : begin
        listPointer := temp( tl )
        temp( hd )
      end
    default : failValue
  end

  !* specialise the collection type to collection[ t ], and then
  !* specialise this existential type to abstract over list,
  !* Then use this to construct a new abstract package
  collection[ t ][ list ]( createList(), failValue, addToList, scanList )
end

```

Figure 2.5.5 Creating a collection for a particular implementation

Some points of Napier88 need to be explained for this example. The representation type, defined recursively as *list*, is a variant labelled by *cons*, a list node, or *tip*, a trivial value representing the end of the list. The type **null** has only a single value, pre-defined as **nil**. A value of a variant type may be created by using a named variant type as a constructor function, and the branch must be specified as overlapping types are allowed. The **project** clause may be used to test a variant value for its branch. This also introduces an identifier, in this case *temp*, of the branch type after a successful projection. One further point to notice is that the procedure returned by *scanList* uses the block retention semantics of Napier88; the update performed to *listPointer* will affect successive calls of the procedure.

As a consequence of orthogonal persistence, values of universally and existentially quantified types may be placed in the persistent store in the same way as any other value. For the rest of the examples, it is assumed that a store with the structure shown in Figure 2.5.6 already exists. Each of the rectangles represent environments. The collection environment contains two others, one of which contains the operations defined over collections, and the other of which contains instances of collections.

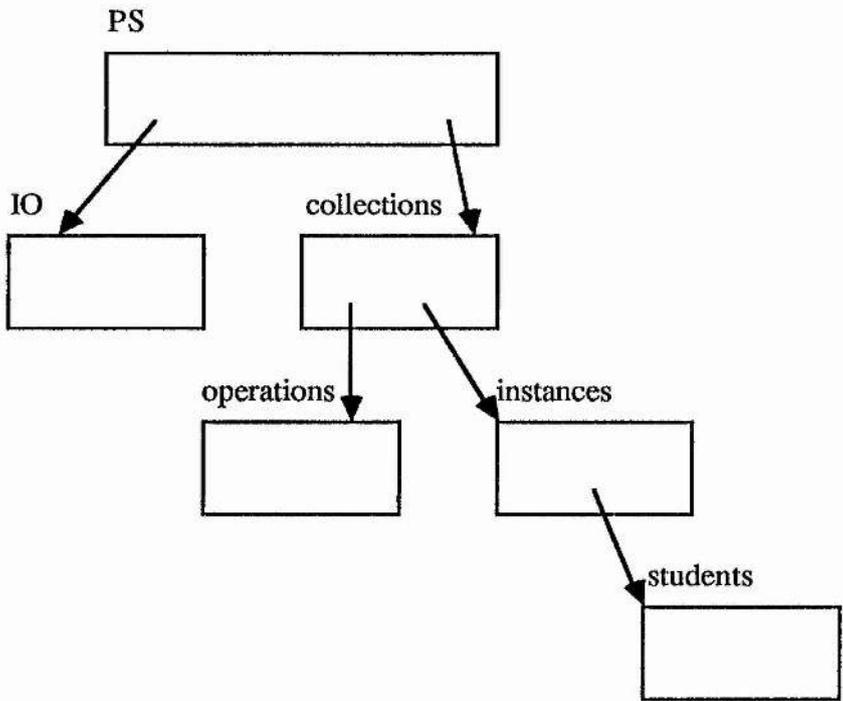


Figure 2.5.6 The structure of a persistent store

Figure 2.5.7 shows how the generic procedure shown in Figure 2.5.5 may be placed in the persistent store in a suitable location, probably in an environment which contains a number of other procedures which generate different collection types.

```

let listcollection = ...

use PS() with collections : env in
use collections with operations : env in
in operations let listRep = listCollection

```

Figure 2.5.7 Placing the generator in the store

Another program may now be used to create a new collection of, for example, students and to place this also in the persistent store. This is shown in Figure 2.5.8.

```

use PS() with collections : env in
use collections with operations, instances : env in
use instances with students : env in
use operations with listRep : proc[ t ]( t → collection[ t ] ) in
begin
  type student is
    structure( age , matricNo : int ; name : string )

  let dummyStudent = student( 0,0,"" )

  in students let classOf90 = listRep[ student ]( dummyStudent )
end

```

Figure 2.5.8 Generating and storing a collection of students

Figure 2.5.9 shows how the fully general cardinality procedure may be written and placed in the persistent store at a suitable location.

```

use PS() with collections : env in
use collections with operations : env in
in operations let cardinality = proc[ t ]( C : collection[ t ] → int )
begin
  let res := 0
  use C as thisOne in
  begin
    let getNext = thisOne( scan )( thisOne( theCollection ) )
    while getNext() ~= thisOne( failValue ) do res := res + 1
  end
  res
end

```

Figure 2.5.9 Storing a generic cardinality procedure

Finally, Figure 2.5.10 shows how a query to discover the number of student in the Class of '90 may be written by binding to the appropriate components within the store and applying them.

```
type student is structure( age , matricNo : int ; name : string )  
  
use PS() with IO , collections : env in  
use IO with writeString : proc( string ) ; writeInt : proc( int ) in  
use collections with operations , instances : env in  
use operations with cardinality : proc[ t ]( collection[ t ] → int ) in  
use instances with students : env in  
use students with classOf90 : collection[ student ] in  
begin  
    let noOfStudents = cardinality[ student ]( classOf90 )  
    writeString( "The number of students in the Class of '90 is " )  
    writeInt( noOfStudents )  
end
```

Figure 2.5.10 A query using the generic cardinality procedure

Two kinds of type abstraction have been used here to avoid the necessity of overspecifying the type of general components. A single procedure has been specified which calculates the cardinality of any collection. Without using type abstraction a potentially large number of procedures would have been necessary instead. This single procedure loses none of the safety of the static typechecking, and does not introduce any extra necessity for dynamic type checking.

2.6 Conclusions

It has been shown how a persistent programming system may be used to solve some of the problems associated with the incremental construction of large programming systems. In particular, it has been shown how code may be shared, rather than copied, whilst preserving the benefits of a strongly typed programming language. Apart from orthogonal persistence, two other language facilities are required to allow this. These are a flexible binding mechanism and type abstraction.

A flexible binding mechanism has been shown which is able to model a number of different methodologies for the binding of program components. In particular, the mechanism allows the incremental construction of a complex software system, which may later be tightly bound together as a single unit when the system is to be released.

Type abstraction is also necessary to maximise the sharing of software componentry, as it allows the specification of procedures whose specification does not depend upon a full type description of their parameters. Universal quantification allows abstraction over the type of a procedure itself, and existential quantification over the type of a parameter's implementation. Using these features a single *cardinality* procedure was constructed and placed in the persistent store, from where it may be shared by many different applications.

Writing component software using these paradigms is only slightly more difficult than writing software specific for a particular task. Using such software, however, is no more difficult than using conventional library routines.

3 Protection and viewing

3.1 Introduction

Large bodies of data are inherently valuable, and must be protected against accidental or deliberate misuse. This is true in both persistent and non-persistent programming systems. However, protection mechanisms in a persistent system may be modelled quite differently from a non-persistent system [MBC90, CDM90].

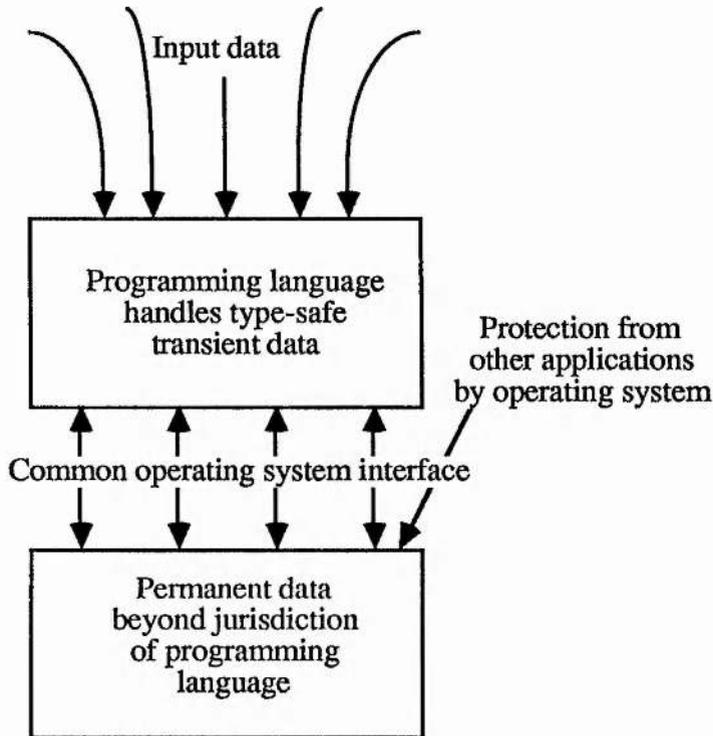


Figure 3.1.1 Traditional protection strategy

Traditionally, protection is provided by a type system when data is manipulated by a programming language, and by the operating system between program invocations. The operating system protection mechanisms are necessary because of the common operating system interface, available to any other user of the same operating system. This is shown graphically in Figure 3.1.1. Protection of

permanent data may not be programmed within a high-level language as the type system may be broken using the operating system interface to the data. It must therefore be programmed below the level of this interface, which increases the difficulty of providing flexible protection mechanisms suitable to the individual needs of data.

In a persistent system, the storage of data beyond a single program invocation is handled by the programming language mechanism, and no common operating system interface is necessary. This is depicted in Figure 3.1.2. The only route by which data may be accessed is through the programming language, and so a single type system may be used to enforce protection upon both transient and permanent data. High-level constructs, such as procedural encapsulation, may be relied upon for the entire lifetime of the data, as it never passes outside the language system.

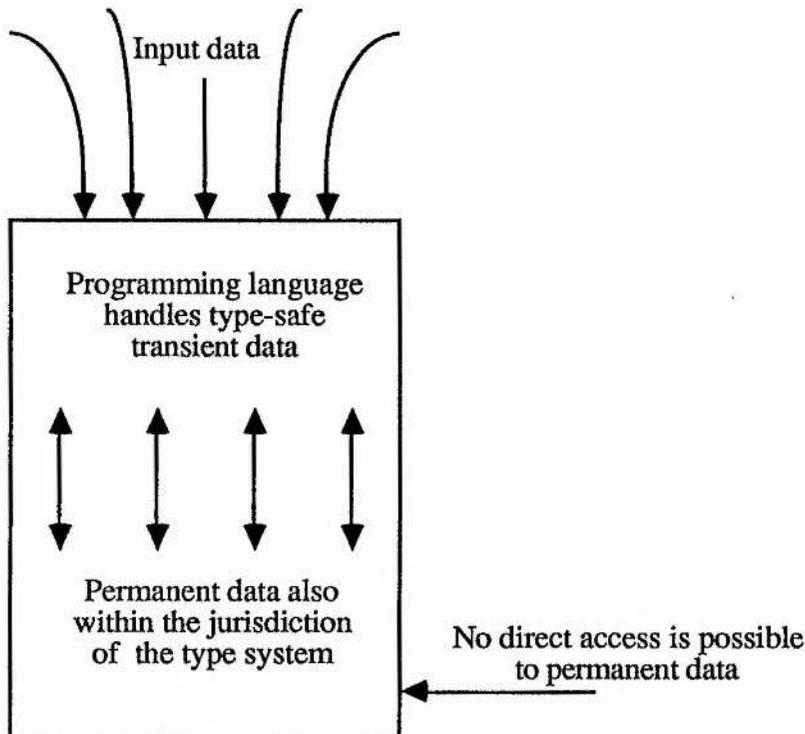


Figure 3.1.2 Persistent protection strategy

There are some systems which lie between these two extremes, such as programming languages which provide a strongly-typed file system interface [Ich83, Mil83]. These retain programmed protection from within a single language system, but if a common interface is used for permanent data then the type system may be compromised by an application written in a different language.

This chapter is an examination of the style of protection which is possible in a persistent system. After a general discussion of data integrity and mechanisms for its protection, various methodologies suitable for persistent systems are discussed. It is then shown how a database viewing mechanism may be constructed in a high-level language using procedural encapsulation and existentially quantified data types.

3.2 Protection of integrity

3.2.1 Integrity of data

Persistent object systems support large collections of data that have often been constructed incrementally by a community of users [ABC83]. Such data is inherently valuable and requires protection from deliberate or accidental misuse. If data is misused in such a way that the accuracy or consistency of the data model is compromised, it is said to have lost its integrity. If a body of data loses its integrity, its value is greatly diminished.

Any computer system must have means of limiting the possible ways in which the integrity of data may be lost. The common factor of all systems which preserve integrity is that they limit the operations which may be performed on data by a user. These limitations are seen throughout most computer systems, ranging for example from hardware addressing protection schemes to complex software integrity constraints.

There are two categories of failure by which loss of integrity may occur. The first is by the failure of a system so that it does not maintain its specified protection protocols. An extreme example of this is a disk head crash, which may violate a file protection scheme defined by the operating system. Another example would be an error in the operating system software which allowed users to access data for which they did not have sufficient privilege. In general, the extent of the damage which may occur as the result of such failure may not be determined in advance, and the best that can be done is the incremental saving of the system state so that it may be restored at a later date. This class of failure will not be further addressed here.

The second class of failure is caused by protection mechanisms which are insufficient to accurately specify the integrity of the data. The data may then lose its integrity by the application of legal operations to it. A number of mechanisms, such as capability systems, encryption methods, type systems and database integrity constraints have been used to provide protection against data misuse in computer systems. All of these mechanisms add security, but at varying degrees of cost.

Once a program gains access to data it may often alter it in a legal but undesirable manner. This can happen because the constraints placed on the user by the capability system, type system or integrity constraint system provide rather coarse grain control over the data. The principle of minimum necessary privilege may cause the protection mechanism to be too fine grained to be either enforced efficiently or expressed succinctly.

Thus the second problem with integrity is that the checking system may cause the specified integrity of the data to be preserved, but it may still contain information that is no longer of real use or is even erroneous. That is, even when enforced, the

protection mechanism will not guarantee that programs produce desirable transformations on the data. For example, the age recorded for a particular person may be incorrect. The challenge is to find an acceptable level of security that can be obtained for a reasonable price in terms of efficient checking and succinct expression.

3.2.2 Protection in persistent systems

Traditionally, computer systems rely upon the type systems of individual programming languages as the only protection mechanism for transient data, and provide a number of different mechanisms, such as file system protection or database constraints, for permanent data. This is because transient and persistent data exist in two separate universes, with different operations available upon them. In persistent programming systems, however, there is no clear difference between transient and permanent data, and this two-level protection scheme is not an appropriate model. A different protection strategy is required.

Another reason for the traditional two-level protection strategy is that data manipulated within a program is not normally accessible by another program, until it has been written to the permanent store through the operating system interface. The access protection schemes provided for permanent data are therefore not relevant to data manipulated within a program. Again, this is not true in a persistent programming system, where there is not necessarily a distinction between shared and non-shared data.

A major difference between persistent and traditional programming systems is that the persistent language's type system governs all data within the system. Traditionally, data is not type-safe after it has been exported from the context of an executing program. The provision of type-safe permanent data provides one kind of control that is not available in a non-persistent model. If a programming

language system does not have direct and unshared physical control over data storage, then it may not guarantee the type safety of stored data. This is because there would be no way of preventing an application programmed in a different language from accessing the data.

Type systems by themselves, however, lack two important categories of integrity protection. These are access protection, to prevent data from being accessed by programs which do not require it, and integrity constraints, which are normally finer-grained than those which may be modelled by type systems.

Both of these categories of protection may be explicitly programmed in many programming languages. However, in a non-persistent system, this protection may not be extended to data exported from a program. This is because the protection specified can not be guaranteed as exported data is not type-safe.

In a persistent system, however, data storage is performed entirely within a programming language, and there is no problem with the storage and sharing of any kind of data. Access protection and integrity constraints may therefore be flexibly programmed according to the needs of the individual data.

Access protection may be programmed in a persistent system by the use of constructs which allow information hiding. These include inclusion polymorphism, abstract data types, and higher-order procedures. What these have in common is that they may hide part of the state of a program by introducing state which is not directly denotable. This makes it impossible to specify another program which will access the hidden state.

3.3 Information hiding

Information hiding is defined here as any programming method enforced by the type system of a language which limits the computation allowed upon data. This may be achieved by restricting either the access or type interface to the data. Some languages allow abstractions over the basic operations defined by the type system, including complex and dynamically evaluated constructs. Where such abstractions have type system support they are usually referred to as abstract data types. Information hiding may be relied upon as a protection mechanism within the context of a strong type system, and in particular within a persistent programming system.

Programming language type systems themselves operate by the use of information hiding over the operating system and hardware operations available to them. This abstraction is at a lower level, and is not necessarily available to the programmer. For example the integer type is defined in most type systems not as a mathematical integer but instead as a restricted interface over its hardware implementation. In strongly-typed languages the user is prevented from using operations such as bitwise rotate and xor on these values, and extra functionality such as conversion to and from equivalent character strings is also provided. Similarly, in most persistent systems, it is the use of information hiding which prevents the use of arbitrary arithmetic on address values.

There are three well-known mechanisms which allow the programming of information hiding within a strong type system. These are subtype inheritance, procedural encapsulation (1st-order information hiding) and existential data types (2nd-order information hiding). Subtype inheritance achieves protection by removing type information, causing the static failure of programs which may try to perform undesirable accesses. 1st-order information hiding prevents the

protected data from being named by an untrusted program, allowing access only through a procedural interface. 2nd-order hiding is somewhere between these two, allowing access mainly through procedures, but also allowing the protected data to be named. This data is, however, viewed through a mechanism which causes type information loss, and therefore allows only a limited set of operations to be performed on it. These three mechanisms may also be used in combination.

3.3.1 Subtype Inheritance

In general, systems which allow subtype inheritance allow any data value to be used in place of one with less functionality. One type is a subtype of another if all operations allowed on the second type are also allowed on the first. In the most general form of subtyping, often referred to as inclusion polymorphism, it is type correct for the use of any value to be replaced by the use of any of its subtypes.

A number of different semantics are possible for the definition of a subtype rule. Here the semantics of Cardelli [Car84] are adopted, using structural type equivalence and an implicit subtyping rule. It should also be made clear that the examples in this section do not use Napier88, which has no subtyping.

Subtype inheritance is usually regarded as a general modelling technique. In particular it allows the declaration of procedures which operate over any type with at least a set of required properties. However, using an object as one of its supertypes is also equivalent to hiding some of the functionality which the object possesses. For example, the following introduces the names *employee* and *person* as record types:

```
type employee is structure( name , address : string ; salary : int )  
type person is structure( name , address : string )
```

In the above, two structure types are declared. The first, called *employee*, has three fields called *name*, *address* and *salary* with types string, string and integer

respectively. The second, called *person*, has two fields called *name* and *address* both of type string. Type *employee* is a subtype of type *person*, and an object of type *employee* may be substituted in any context where an object of type *person* is expected. This would have the effect of hiding the salary field by the loss of type information. If another user is only to be allowed this restricted access to employee objects, this view of the object may be exported, for example by use of an explicit type coercion:

```
let joe = employee( "Joe Doe" , "1 Assignment Boulevard" , 100000 )  
let exportJoe : person = joe
```

In this *joe* is declared to be an object of type *employee* with the given field values. *exportJoe* is of type *person*, denoted by the type after the ":" symbol, but has the value *joe*. This means that a user of the value *exportJoe* will now have the value of the original record. However, it is not possible to express an operation to access the salary field of this value due to the restrictions of the static type system. That is, the *salary* field cannot be used with the object *exportJoe* since such a program would fail during static analysis.

This mechanism allows only simple information hiding compared with the other methods of 1st and 2nd order information hiding. Its advantages are that it is simple and elegant to use, and is easy to understand.

There are a number of problems which arise with inclusion polymorphism, particularly in conjunction with languages with update. Various solutions have been proposed to these, which are beyond the scope of this thesis [Car89, WZ89]. The problems are not believed to be insurmountable, and persistent languages with update and inclusion polymorphism, such as Galileo [ACO85], have been successfully implemented.

3.3.2 1st-order information hiding

Access to data can also be restricted by only allowing access to procedures which are defined over the data, and not allowing the data itself to be visible. This is a common model for abstract data types, and is known as 1st-order information hiding [CW85]. It may be achieved in a number of ways but it will be described here in terms of a language which has first-class procedure values and block-style scoping. Access to the original data may then be removed simply by its identifier becoming unavailable. For example, the following type defines a *Person* as a record containing procedures which define three operations:

```
type Person is structure
(
  getName,
  getAddress      : proc( → string );
  putAddress      : proc( string )
)
```

This allows a finer grain of restriction than that achieved by subtype inheritance, in that the name and address may be read, but only the address may be changed. Access to the data may be removed by placing its declaration in a block so that its identifier is lost from scope after the *Person* object has been constructed. This is shown in Figure 3.3.1. The exported procedures, which have the data encapsulated within their closures, are then the only way in which the original value may be accessed. Again, this relies upon the static properties of the system to prevent the access since a program which attempts direct access to *joe* will fail statically by the scoping rules of the language.

```
let exportJoe =
begin
  let joe = employee( "Joe Doe", "1 Assignment Boulevard", 100000 )
  Person(   proc( → string ) ; joe( name ),
            proc( → string ) ; joe( address ),
            proc( new : string ) ; joe( address ) := new )
end
```

Figure 3.3.1 Hiding the Data Representation

In Figure 3.3.1, *exportJoe* is declared to have the value obtained by executing the block. This is a structure of type *Person* with three procedure fields. Each procedure uses the object *joe* which is inaccessible by any other means after exit from the block.

Further flexibility is possible using encapsulation in that dynamic properties may be specified, and access may be denied dynamically if required. For example, perhaps there exists an integrity constraint that an address may not be more than 100 characters long. This can be programmed in the procedural encapsulation, as shown in Figure 3.3.2. The only difference here is that the *putAddress* procedure checks the dynamic constraint, and raises an exception if it is not met.

```
let exportJoe =
begin
  let joe = employee( "Joe Doe" , "1 Assignment Boulevard" , 100000 )
  Person(   proc( → string ) ; joe( name ) ,
           proc( → string ) ; joe( address ) ,
           proc( new : string ) ;
             if length( new ) ≤ 100
             then joe( address ) := new
             else raise longAddress( new. ) )
end
```

Figure 3.3.2 Refining the Interface

A particular example of a dynamic constraint allows access to the original data to be protected by password. A procedure can be provided in the interface which will return direct access to a user with sufficient privilege. Figure 3.3.3 shows the extended definition required, with an extra procedure in type *extraPerson* which returns the representation of the data only if it is supplied with a string equal to the password used to create it. In this situation, the programmer responsible for constructing the view of the data will have enough information to extract the representation. Alternatively, it would be possible to arrange system-wide passwords which would decide whether access is allowed or not.

```

type extraPerson is structure
(
  getEmployee    : proc( string → employee ) ;
  getName,
  getAddress     : proc( → string ) ;
  putAddress     : proc( string )
)

let exportJoe = proc( password : string → extraPerson )
begin
  let joe = employee( "Joe Doe" , "1 Assignment Boulevard" , 100000 )
  extraPerson(   proc( attempt : string → employee )
                if attempt = password
                  then joe
                  else failValue,
                proc( → string ) ; joe( name ) ,
                proc( → string ) ; joe( address ) ,
                proc( new : string ) ; joe( address ) := new )
end

```

Figure 3.3.3 Protection by Password

This technique gives a result not dissimilar from the kind of module provided by Pebble [BL84] and a high level language analogy of capabilities [DvH66, NW74].

Instead of a string password, an unforgeable "software capability" may be used. An example of this, otherwise identical to that of Figure 3.3.3, is shown in Figure 3.3.4. In place of the string password is one of type *capability*, a void structure type. The equality operation in "*attempt = password*" means a test of identity on the void structure type. A void structure value is passed to the *exportJoe* procedure which creates the package. After this, the *getEmployee* procedure may never be called successfully without access to the same instance of this void structure. If the capability is adequately protected, then access protection for the procedure is of no consequence. In this way a single capability may be used to protect a number of procedure closures.

```

type capability is structure()

type extraPerson is structure
(
  getEmployee      : proc( capability → employee );
  getName,
  getAddress       : proc( → string );
  putAddress       : proc( string )
)

let exportJoe = proc( password : capability → extraPerson )
begin
  let joe = employee( "Joe Doe" , "1 Assignment Boulevard" , 100000 )
  extraPerson(      proc( attempt : capability → employee )
                if attempt = password
                then joe
                else failValue,
  proc( → string ) ; joe( name ),
  proc( → string ) ; joe( address ),
  proc( new : string ) ; joe( address ) := new )
end

```

Figure 3.3.4 Protection by Software Capability

This technique relies upon the referential integrity of the persistent store. It is a consequence of this that for any type over which identity is defined, the identity of any value is unique for the lifetime of the system. Therefore the identity of such a value is unforgeable. If such a value is bound to the closure of a procedure as its password, then to use the procedure a programmer must somehow have access to the same value.

In Chapter 2 it was shown how the data type environment in Napier88 could be used to impose a flexible naming scheme on the persistent store. As environments are first-class values, procedural encapsulation may be used to protect such a naming scheme. Figure 3.3.5, for example, shows how a procedure which models a password-protected environment may be placed in the persistent store.

```

use PS() with user, IO, failValue : env ; environment : proc( → env ) in
use IO with writeString : proc( string ) ; readString : proc( → string ) in
begin
    let encapsulated = environment()

    writeString( "Please enter password for environment richard: " )
    let password = readString()

    in user let richard = proc( attempt : string → env )
        if attempt = password then encapsulated else failValue
end

```

Figure 3.3.5 A password-protected environment

The first **use** clause here asserts the presence within the persistent store of environments with the identifiers *user*, *IO*, and *failValue*, as well as the procedure *environment* which may be used to create an empty environment. *IO* is then asserted to contain procedures *readString* and *writeString*, for communicating with the user. Inside a new block, a new environment *encapsulated* is first created. A password is then read from the user. This password and the new environment are both encapsulated within the closure of a new procedure, which returns the new environment on presentation of the same password. This procedure is placed in the *user* environment.

A further consequence of referential integrity within the persistent store is that a number of different interfaces may be programmed and exported from the original data. This allows the construction of multiple views on the same data. Figure 3.3.6 shows a program which places two interfaces to the same item of data in the persistent store; one interface may read but not update the salary field, the other may update but not read it. These interfaces are placed in password-protected environments, respectively called *readers* and *writers*.

```

use PS() with IO , user : env in
use IO with writeString : proc( string ) ; readString : proc( → string ) in
use user with readers , writers : proc( string → env ) in
begin
  let joe = employee( "Joe Doe" , "1 Assignment Boulevard" , 100000 )

  writeString( "Please enter password for environment readers: " )
  in readers( readString() ) let readJoeSalary =
    proc( → int ) ; joe( salary )

  writeString( "Please enter password for environment writers: " )
  in writers( readString() ) let writeJoeSalary =
    proc( new : int ) ; joe( salary ) := new
end

```

Figure 3.3.6 Different interfaces on the same data instance

This program again makes some static assertions about the data value it expects to find in the store when it is executed. These include two procedures, *readers* and *writers*, which model password-protected environments. In the main block, a new encapsulated value is first declared. Two new procedures are declared which contain this value in their closures, and respectively placed in the protected environments, assuming that the correct password is presented for each by the user of this program.

This combination of type system protection and referential integrity may be used to provide a flexible viewing mechanism over the persistent store.

3.3.3 2nd-order information hiding

2nd-order information hiding does not restrict access to the protected values, but instead abstracts over the type of the protected value to restrict operations allowed on it. Thus the protected values may be manipulated for some basic operations, such as assignment and perhaps equality, but their normal operations are not allowed due to the type view. This allows the representation objects themselves to be safely placed in the interface along with the procedures which manipulate them.

This power can almost be achieved using a combination of subtyping and 1st-order hiding. For example, Figure 3.3.7 shows how a reference to the representation may be safely placed in the interface by effectively removing all type information from it. The representation may be accessed as a value, but only a highly restrictive access is possible as there is very little type information available. It may be assigned and tested for equality, but none of its fields may be accessed due to the static typing restrictions.

Within the block expression *joe* is used to initialise one of the fields of type `structure()`. *joe* is of type *employee*, which is a subtype of `structure()` and the initialisation is legal. However, the fields of *joe* may not be accessed via this route.

```

type Person is structure
(
  absPerson      : structure( );
  getName,
  getAddress     : proc( → string );
  putAddress     : proc( string )
)

let exportJoe =
begin
  let joe = employee( "Joe Doe" , "1 Assignment Boulevard" , 100000 )
  Person(   joe,
           proc( → string ) ; joe( name ) ,
           proc( → string ) ; joe( address ) ,
           proc( new : string ) ; joe( address ) := new )
end

```

Figure 3.3.7 Removing the Type Information

Using this technique however it is not possible to know that two such abstracted values are the same type. This may be desirable for some applications, for example, a *Person* may also have a father and mother in the interface, along with a field for a favourite parent which changes between them. Subtyping alone does not provide enough information to allow this. For example, if the definition is:

```

type Person is structure
(
  absPerson, mum , dad , favourite : structure( ) ;
  getName , getAddress             : proc( → string ) ;
  putAddress                        : proc( string )
)

```

then it is not in general allowable to write

```

exportJoe( favourite ) := exportJoe( mum )

```

as there is no way of ensuring statically that *mum* and *favourite* are the same type.

One mechanism which allows 2nd-order information hiding is the existential data type as described in Chapter 2. This allows the definition of interface types which are abstracted over. As names for these types are declared before the existential type definition, different parts of the definition may be bound to the same type. As before, only the basic operations defined on all types are allowed over the abstracted types, but values which are abstracted by the same name are statically known to be compatible. *Person* as above may be redefined as:

```

type Person is absType[ absPersonType ]
(
  absPerson,mum,dad,favourite      : absPersonType ;
  getName,getAddress               : proc( → string ) ;
  putAddress                       : proc( string )
)

```

The identifier in square brackets before the body of the type declaration declares a name for a type which is abstracted over. This allows a tighter definition of such types, as it can now be seen where the same type appears in the interface. Components of the same instance of an interface may be bound together by a **use** clause. For example,

```

use exportJoe as joe in
  joe( favourite ) := joe( mum )

```

may be statically determined to be type correct, as the *favourite* and *mum* fields must be type compatible to allow the object to be created.

This static binding of equivalent types may also be used to allow the interface procedures to be defined over the type of the hidden representation. A more flexible definition which allows the name and address operations to be performed on any of the people in the interface would be:

```
type Person is abstype[ absPersonType ]
(
  absPerson,mum,
  dad,favourite      : absPersonType ;
  getName,getAddress : proc( absPersonType → string ) ;
  putAddress         : proc( absPersonType,string )
)
```

This allows the definition of n-ary operations over the hidden representation type. For example, a procedure may be placed in the interface which tests if two people have the same address:

```
type Person is absType[ absPersonType ]
(
  absPerson,mum,
  dad,favourite      : absPersonType ;
  getName,
  getAddress         : proc( absPersonType → string ) ;
  putAddress         : proc( absPersonType,string ) ;
  sameAddress        : proc( absPersonType,absPersonType → bool )
)
```

This example illustrates a major difference in power between 1st-order and 2nd-order information hiding. With 2nd-order, a type is abstracted over, and procedures may be defined over this type. With 1st-order hiding, it is the object itself which is hidden within its procedural interface. Procedures which operate over more than one such object may not be defined sensibly within this interface. Therefore any operations defined over two instances must be written at a higher level, using the interface. At best this creates syntactic noise and is inefficient at execution time. It also means that such operations are defined in the module which uses the abstract objects, rather than the module which creates them. Some examples are not possible to write without changing the original interface.

3.3.4 Conclusions

The mechanisms of subtype inheritance, procedural encapsulation, and existential data types have been discussed with relation to the programming of protection within a persistent system. These mechanisms may be used to program protection only in a persistent system; in a non-persistent system the protected data is always susceptible to abuse by the common operating system interface through which its storage must be arranged.

Protection in database systems is normally provided by viewing mechanisms. These allow database programmers to set up different interfaces over the same data, so that users with different privileges may perform different operations. It is now shown how the information hiding techniques presented above may be used to program very flexible viewing mechanisms within a persistent programming system.

3.4 Viewing mechanisms

Viewing mechanisms have been traditionally used in database systems both to provide security and as a conceptual support mechanism for the user. Views are an abstraction mechanism that allow the user to concentrate on a subset of types and values within the database schema whilst ignoring the details of the rest of the database. By concentrating the user on the view of current interest both security and conceptual support are achieved.

Persistent programming languages integrate both the technology and methodology of programming languages and database management systems. One particular area of difficulty in this integration has been the friction caused by the type mechanisms of programming languages and databases being incompatible. Programming languages tend to provide strong, often static type systems with little to aid the

expression or modelling of large uniform structure. Databases, on the other hand, are much more concerned with capturing this notion of bulk expression than with static or even strong typing. Both, however, are concerned with the integrity of the data.

Another area of difficulty in the integration of programming languages with database systems has been to demonstrate that the facilities found necessary in both systems are not lost or compromised by the integration. Viewing mechanisms which have been used so successfully in the database community have not found widespread acceptance in the programming language world. They therefore constitute a source of irritation in the above integration.

3.4.1 Database viewing mechanisms

Viewing mechanisms are traditionally used to provide security and information hiding. Indeed, in some relational database systems, such as INGRES [SWK76], a relational viewing mechanism is the only security mechanism available. A view relation is one which is defined over others to provide a subset of the database, usually at the same level of abstraction. A slightly higher level may be achieved by allowing relations to include derived data, for example, an age field in a view might abstract over a date of birth field in the schema.

Security provided by view relations is often restricted to simple information hiding by means of projection and selection. For example, if a clerk is not permitted to access a date of birth field, then the projected data may contain all the attributes with the exception of this one. If the clerk may not access any data about people under the age of twenty-one, then the view relation will be that formed by the appropriate selection.

Read-only security may be obtained in some database systems by restricting updates on view relations. Although this restriction is normally due to conceptual and implementation problems, rather than being a deliberate feature of a database system design, it may be used to some effect for this purpose. Some systems, for example IMS [IBM78], go further than this, and the database programmer can allow or disallow insertions, deletions, or modifications of data in view relations. This allows a fine grain of control for data protection purposes.

There are a number of problems associated with view relations. Often a new relation is created, with copied data. This means that updates (where permitted) to the view relation will not be reflected in the underlying relations, with the obvious loss of integrity. This problem may be partly solved by the use of delayed evaluation, where names are not evaluated until used in a query. This allows the database programmer to define views over the database schema before all of the data is in place.

Those systems where data copying does not occur seem to incur major integrity problems. For this reason, System R [ABC76] disallows updates on views which are constructed using a join operation. There is a more serious example in IMS, where deletion of a segment, if permitted, also deletes all descendant segments, including cases where they are not even a part of the view in question!

A much higher-level concept of a viewing mechanism is provided by the UMIST Abstract Data Store [Pow85]. This is a software tool which supports abstract data objects together with mechanisms for providing different views over them. The system provides a consistent graphical user interface which allows the user to directly manipulate objects via multiple views. Changes in objects are automatically reflected in other views since they contain no information about the state of objects, only the manner in which they are displayed. This provides a

powerful tool for a user to build a store of objects which may be maintained and incremented interactively, and it seems possible that it may be sufficient for a surprisingly large class of problem. However, the inability to write general-purpose programs over the store must be seen as a major drawback for many applications.

3.4.2 View construction in Napier88

For a programming example, we will use a banking system in which customers have access to their accounts through autoteller machines. This poses the classic database problems of having a very large bulk of updateable data with different users requiring different operations on it. Here we will concentrate on views in the system. We shall restrict ourselves at first to the autoteller machines, which have different styles of access to accounts according to which bank the machine belongs. A customer's own bank may have full access to an account whereas another bank may not access the customer's account balance, but must know if a withdrawal may be made.

The Napier88 features of existential data types and environments are described in detail in Chapter 2.

3.4.2.1 Creating a concrete representation

We will begin by defining a concrete representation of a user account and the procedures that operate on that type. Here we are not interested in how the account data is stored (i.e. in a relation, a β -tree, etc.), and we will assume that we have previously declared a lookup function indexed by account number which returns the account associated with that account number.

We will implement an account using a record-like data structure which in Napier88 is called a structure. The structure type, called *account*, has three fields: *balance*

which holds the account balance; *oLimit* which contains the overdraft limit and *pin* which contains the password necessary to access the account. There is one instance of this type in the database for each user account.

One special instance of this type called *failAc* is created to use as a fail value. This is used in the *getAc* procedure if an illegal password is supplied when attempting to access an account.

Five procedures operate on the account directly, they are:

- withdraw* This procedure checks to see that sufficient funds are in the specified account to make a withdrawal. If there are, the amount specified is debited from the account.
- balance* This returns the balance of an account.
- oLimit* This returns the overdraft limit of an account, this is a negative number.
- sufficient* This indicates whether sufficient funds are in an account to make a withdrawal of a specified amount.
- getAc* This procedure interfaces with the persistent store, it looks up an account in an associative storage structure and checks the supplied password. If the password matches the *pin* field in the account, the account is returned, otherwise the fail value *failAc* is returned.

The Napier88 code necessary to implement the concrete type representations and the code to operate on the concrete type representation of an account is shown in Figure 3.4.1.

As mentioned earlier, we have made use of a predefined *lookup* procedure, as its implementation is of no interest in this context.

```

type account is structure( balance,oLimit : int ; pin : string )
let failAc = account( 0,0,"" )      ! This is a fail value

let withdraw = proc( debit : int ; ac : account )
begin
    let result = ac( balance ) - debit
    if result > ac( oLimit ) and debit > 0 do ac( balance ) := result
end

let balance = proc( ac : account → int ) ; ac( balance )

let oLimit = proc( ac : account → int ) ; ac( oLimit )

let sufficient = proc( debit : int ; ac : account → bool )
    ac( balance ) - debit > ac( oLimit )

let getAc = proc( accountNumber : int ; passwd : string → account )
begin
    let new = lookup( accountNumber )
    if new( pin ) = passwd then new else failAc
end

```

Figure 3.4.1 The concrete representation and procedures

3.4.2.2 Placing concrete representations in the store

Of course, in a real bank there would be a requirement for the procedures and data defined above to persist over a long period of time. In order to achieve this in practice it is necessary to place them in the persistent store. We will assume that an environment called *bank* already exists in the root environment. The above example may be rewritten as shown in Figure 3.4.2.

```

use PS() with bank : env in
begin
    let failAc = account( 0,0,"" )      ! Must be declared both in local
    in bank let failAc = failAc          ! scope and in bank environment

    in bank let withdraw = proc( debit : int ; ac : account ) ; ...
    in bank let balance = proc( ac : account → int ) ; ...
    in bank let oLimit = proc( ac : account → int ) ; ...
    in bank let sufficient = proc( debit : int ; ac : account → bool ) ; ...
    in bank let getAc = proc( accountNumber : int ;
        passwd : string → account ) ; ...
end

```

Figure 3.4.2 Using the persistent store

Since the environment called *bank* is reachable from the persistent root the procedures will persist when the program terminates. However, notice that if we allow programmers access to the concrete representations, the database will be vulnerable to misuse. For example, the unscrupulous programmer could write,

```
let myAc = getAc( 34589001,"3478" )
myAc( balance ) := myAc( balance ) + 1000000
```

yielding a net profit of one million pounds (a very high programmer productivity ratio). For this reason, it is necessary to protect the concrete representation.

3.4.2.3 Creating abstract view types

For the purpose of this example we will define two abstract data types which provide interfaces for the procedures shown in the last section. The first is to be used by an account holder's own bank. The abstract type shown below, called *localTeller* has the following five fields:

- failAc*: is returned by *getAc* if a password check fails,
- getAc*: which will take as input an account number, and a secret password typed into the machine by the customer, and provided that the secret number is correct, return the representation of that account, otherwise it will return the fail value *failAc*,
- withdraw*: which will remove the amount specified from the account, unless there are insufficient funds, in which case it will do nothing,
- balance*: which returns the balance in the account, and,
- oLimit*: which returns the account overdraft limit.

The type of *localTeller* is defined in Figure 3.4.3.

```

type amount is int    ! an amount of money
type number is int   ! an account number
type passwd is int   ! a secret number

```

```

type localTeller is abstype[ ac ](
  failAc      : ac
  getAc       : proc( number,passwd → ac )
  withdraw    : proc( amount,ac )
  balance     : proc( ac → amount )
  oLimit      : proc( ac → amount ) )

```

Figure 3.4.3 The type of a local autoteller machine

The concrete type named as *ac*, may never be discovered, therefore the programmer is forced to access accounts only using the procedures provided in the interface of the abstract data type. Notice that the fail value *failAc* appears in the interface of the abstract type. This permits the user of an instance of this abstract type to check for failure in the application of the *getAc* procedure. This is only possible because in Napier88 equality is defined over all values of all types, including values whose types are abstracted over. Note that it is not possible to write such an abstract type in standard ML [Mil83] since equality is defined only over the so called 'eq' types – a subset of the type domain. In Napier88, equality is defined as identity on all non-scalar values.

Similarly, we can define another abstract type for use by another bank. Other banks are not allowed to discover a customer's balance or limit so a slightly different interface is required. In this type a procedure called *sufficient* is provided so that the bank may ensure that sufficient funds are available in the account. We may define the type *remoteTeller* as in Figure 3.4.4.

```

type remoteTeller is abstype[ ac ]
(
  failAc      : ac
  getAc       : proc( number,passwd → ac )
  withdraw    : proc( amount,ac )
  sufficient   : proc( amount,ac → bool )
)

```

Figure 3.4.4 The type of a remote autoteller machine

3.4.2.4 Creating instances of views

Here, an instance of the type *localTeller* and an instance of the type *remoteTeller* are required to act as views over the single implementation shown above. This may be achieved by creating instances of the two types as shown in Figure 3.4.5.

```
use PS() with tellerEnv, bank : env in ! Assuming these environments
use bank with                          ! have been properly constructed
    failAc : account ;                 ! and initialised
    withdraw : proc( int,account ) ;
    balance,
    oLimit : proc( account → int ) ;
    sufficient : proc( int,account → bool ) ;
    getAc : proc( int,string → account ) in
begin
    in tellerEnv let local = localTeller[ account ]
        ( failAc,
          getAc,
          withdraw,
          balance,
          oLimit )

    in tellerEnv let remote = remoteTeller[ account ]
        ( failAc,
          getAc,
          withdraw,
          sufficient )
end
```

Figure 3.4.5 Placing the interfaces in the store

The program extracts the procedures placed in the environment denoted by *bank* and creates an instance of the types *localTeller* and *remoteTeller*. These are initialised using the procedures from the *bank* environment. Using this technique, the level of procedural indirection normally found in viewing mechanisms is not required. Consequently there are both space and time advantages of this technique.

When this program terminates the two abstract objects denoted by *local* and *remote* in the environment denoted by *tellerEnv* will be committed to the persistent store, since they are within the transitive closure of the persistent root.

Although we have chosen to create the two abstract types used in this example in the same code segment this was not strictly necessary. Any programmer with access to the bank environment is free to create new abstract types which interface with the concrete types at any time in the future in a similar manner.

3.4.2.5 Using views

Two views of the database now exist in the environment called *tellerEnv* reachable from the root of persistence. In order to use the instance of *localTeller* in the *tellerEnv* the auto-teller programmer has only to write a main loop which uses the procedural interface correctly. This would look something like Figure 3.4.6.

```
use PS() with tellerEnv : env in
use tellerEnv with local : localTeller in

use local as package in
begin
  let getAccount = package( getAc )
  let withdraw = package( withdraw )
  let findBalance = package( balance )
  let findLimit = package( oLimit )

  ! code to use above procedures ...
end
```

Figure 3.4.6 Programming a local teller machine

Similarly, the instance of *remoteTeller* could be accessed as in Figure 3.4.7.

```
use PS() with tellerEnv : env in
use tellerEnv with remote : remoteTeller in

use remote as package in
begin
  let getAccount = package( getAc )
  let sufficient = package( sufficient )
  let withdraw = package( withdraw )

  ...
end
```

Figure 3.4.7 Programming a remote teller machine

The interesting point here is that these procedures manipulate the same objects as those in the previous interface, and so provide a different view of them. If the account information is kept in a relational data structure, this is equivalent to a relational viewing mechanism: *getAc* is a relational select on a primary key. Notice that although the views update the same data, no integrity problems arise.

3.4.2.6 Operations over more than one account

Suppose another function had been required from the auto-teller, so that a customer with two accounts may use the machine to transfer money from one to another. We can allow this by redefining the interface of *localTeller* as in Figure 3.4.8.

```
type localTeller is abstype[ ac ]
(
  failAc      : ac
  getAc       : proc( number,passwd → ac )
  withdraw    : proc( amount,ac )
  transfer    : proc( amount,ac,ac )
  balance     : proc( ac → amount )
  oLimit      : proc( ac → amount )
)
```

Figure 3.4.8 The interface with transfer

The important difference between this new procedure, *transfer*, and those already discussed is that it is defined over more than one value of the witness type. This illustrates a major difference in descriptive power between first-order and second-order information hiding. With second-order, a type is abstracted over, and procedures may be defined over this type. With first-order hiding, it is the object itself which is hidden within its procedural interface. Procedures which operate over more than one such object may not be sensibly defined within this interface. Therefore any operations defined over two instances must be written at a higher level, using the interface. At best this creates syntactic noise and is inefficient at execution time. It also means that such operations are defined in the module which

uses the abstract objects, rather than the module which creates them. Some examples, such as this one, may not be written without changing the original interface.

This example highlights another difference between this style of abstract data type and that used in the language ML. Although the use of the type is similar to an ML-style type, the definition of structural type equivalence allows objects of the type to be passed between different compilation units. This is not the case with ML abstract types, which are only compatible if they refer to the same instance of the type definition: construction and use by independently-prepared modules is not possible by this mechanism and must be achieved otherwise [Har85].

3.4.3 Privileged data access

It has been shown how a number of different abstract interfaces may be created to operate over a single collection of data. For the above example, this may be represented diagrammatically as in Figure 3.4.9. The shaded boxes represent views, i.e. existential data types, and the clear boxes represent nodes in a protected naming scheme. As this all occurs within the type system of a persistent programming language, no other access is possible to the raw data of the account representations.

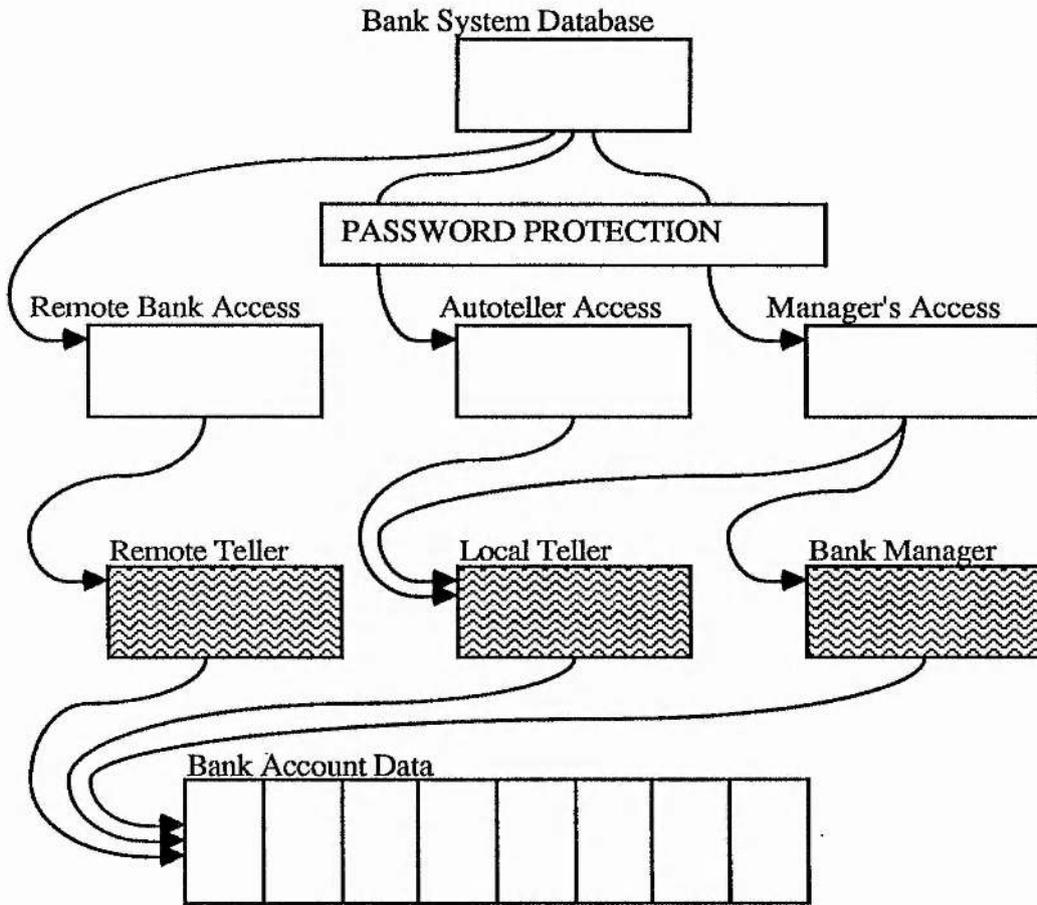


Figure 3.4.9 The abstract interfaces to the accounts

A problem exists with abstract data types in long-lived persistent systems if, when the abstract type is formed, access to the original representation of the data is lost. This is a requirement of these types for the purposes of safety and abstraction, but there is an underlying assumption that no access to the data will ever be required apart from that specified in the abstract interfaces. For a large body of persistent data this is unrealistic.

A serious problem occurs if an error leaves the database in an inconsistent state. In the banking example, this could happen if one of the auto-tellers develops a mechanical failure which prevents it from dispensing the requested money after the

withdraw operation has been executed. The easiest way for such an error to be rectified is for a privileged user, such as the database administrator, to be allowed access to the concrete representation of the account and adjust the balance. If such access is not allowed, then an error may occur which leaves the database in an inconsistent state from which it is not possible to recover.

This kind of access may be treated as another view over the same data, with no abstraction in the interface. To prevent unauthorised users from gaining access to the information contained in this environment, a password protection scheme may be used. As previously shown, an environment can be hidden inside a procedure which requires a password. The restricted definitions and data can be placed safely in this environment without fear of access by unprivileged users. The extended database for the bank account example is shown in Figure 3.4.10.

This level of data access is also desirable for database programmers in other circumstances. It is not uncommon for a new operation to be required on data which is abstracted over: an example of this is if the procedure *transfer* described above became a requirement after the system had been installed. Although it may be possible to write such operations in a new user module, such use is often contrived and will always be inefficient. Using this technique, the database administrator may construct a new interface over the representation level of the data, with no associated loss of efficiency.

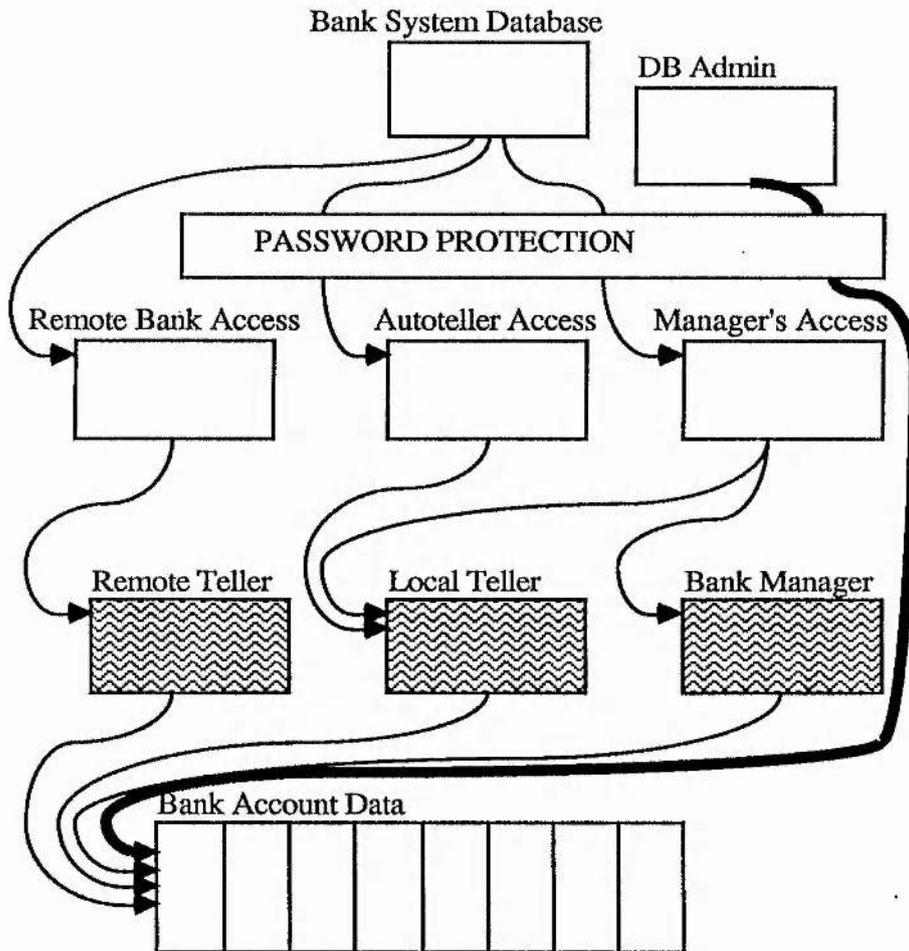


Figure 3.4.10 Privileged data access

The above holds whether the new interface required is confined to adding an extra operation to one of the existing views, or whether an entire new view is necessary. The autoteller example given above could thus be added to an existing system, rather than being part of the design of the original system. Furthermore, none of the code would be any different from that shown, and programs which used the original interface would continue to work. This demonstrates the flexibility of this paradigm, and shows a way forward for the incremental construction of complex software systems.

3.5 Conclusions

Persistent programming systems provide support for the incremental gathering of large amounts of data. The value of such data is greatly reduced if its integrity is lost, and this must therefore be protected as far as possible. Traditionally protection uses a two-level model, with different schemes in operation for data within programs and permanent data. In a persistent system this difference does not exist, and so a single protection mechanism must be used for all data within the system.

A prerequisite for a persistent programming system is that all data are subject to the constraints of the persistent language's type system. This means that no unlawful access to the data may ever occur except in the event of system failure. This gives the aspect of type safety to permanent data, which may not be achieved in a non-persistent system.

A further effect of the type safety of all data is that protection which is programmed may not be revoked by a lower level of technology. This means that access protection and software constraints may be programmed as suitable for the individual data. A number of different language mechanisms may be used to protect permanent and shared data, including subtype inheritance, procedural encapsulation, and abstract data types.

The protection of data in database systems is normally achieved by integrity constraints and viewing mechanisms. In a persistent system, however, such mechanisms do not need to be specially provided as they may be programmed within a persistent programming language. Integrity constraints may be programmed within procedural encapsulation, and viewing mechanisms may be programmed with a combination of encapsulation and existential data types. The

added advantage of existential types over traditional viewing mechanisms is that they are statically type checkable. This use of the persistent store also ensures that views occur over the same instance of data, and the copying of data, with its associated further integrity problems, is not necessary.

4 Type equivalence in persistent systems

4.1 Introduction

Type equivalence checking is a mechanism used to ensure that operations are applied only to values of appropriate types. In statically typed languages, type equivalence need only be checked by the compiler. In languages with some dynamic type checking, equivalence checks must also be carried out by the run-time system.

Equivalence checking is well understood within a program, but presents some new problems in a persistent system. These problems stem from the independent compilation of co-operating modules. In a persistent system, data which is shared between modules stays within the boundaries of the language's type system. Languages which are not persistent can allow such co-operation only by using an external store interface, which is often not type secure, such as a file system.

There are three main points of discussion within this topic. First is a review of the common models of type equivalence, structural and name equivalence. The models are compared in their strengths and weaknesses for modelling and protection within a program. It is then shown that in a programming system capable of separate compilation, both models are inadequate without the additional ability to export and import type definitions between compilation units. Structural type checking turns out to be a requirement in both models, but may be limited to certain well-defined places within a name equivalence system.

After this there is a discussion of implementation techniques to achieve maximum efficiency for the activity of structural typechecking. Two possible representations

are shown, graphs and strings, and their tradeoffs are exposed. Some tentative measurements, taken from the implementation of the Napier88 system, are given.

The final section retrospectively shows how the type system of a programming language may be mapped onto the representations previously discussed. Napier88 is used as a detailed example of such a mapping, and the algorithms used for the construction of type representations are explained in full.

4.2 Type equivalence models

4.2.1 Structural and Name Equivalence

4.2.1.1 Definitions

Type equivalence may be defined by one of two common models: structural or name equivalence. In both of these models, the types consist of a set structure which partitions the value space of a language. Membership of these sets is decided by a different mechanism in the two models.

In name equivalence, membership of a type is an attribute designated at the time a value is created. New types may be declared which, at the time of definition, have no values belonging to them. Every type declaration must include a description, either implicit or explicit, of the set of legal operations on values of this type. The creation of a value is associated in some way with a type declaration, and the value is attributed with this type. Two values are type equivalent if and only if they have been created in association with the same type declaration.

In structural equivalence, membership of a type is defined by some properties of the values themselves. Values may be of the same type only if they have the same set of operations defined over them. Type equivalence is therefore an implicit property of values, and values do not need to be constructed with reference to a

type definition. To ensure that type equivalence is well-defined, a language definition must include a set of type rules. These define the universe of discourse of the language and allow the type of any value to be deduced.

4.2.1.2 Example

The programming language Ada [Ich83] is a good example of a system that primarily uses name equivalence checking. An example is the definitions of the types *ANIMAL* and *VEHICLE* given below:

```
type ANIMAL is record  
  Age : INTEGER ;  
  Weight : REAL ;  
end record ;
```

```
type VEHICLE is record  
  Age : INTEGER ;  
  Weight : REAL ;  
end record ;
```

ANIMAL and *VEHICLE* define sets of values over which the same operations are legal: they both represent a labelled cross product *Age* : INTEGER x *Weight* : REAL. However, values of type *ANIMAL* are not type equivalent to values of type *VEHICLE*.

It should be stressed that name equivalence is not defined according to the textual form of the name used to define the type. Thus if the above definition of *ANIMAL* were repeated in a different scope, data created from the different definitions would not be compatible. This also raises the question of equivalence when type definitions come from different dynamic invocations of the same code.

The language Napier88 will be used to describe examples of structural equivalence. The type definitions of *ANIMAL* and *VEHICLE* given in Ada earlier would be in Napier88:

```
type animal is structure( Age : int ; Weight : real )  
type vehicle is structure( Age : int ; Weight : real )
```

These types are equivalent in Napier88 since the type declaration is only regarded as providing a syntactic shorthand for the set described in the type expression. The set of values in this case is the labelled cross product

Age : int x *Weight* : real.

4.2.2 Equivalence models within a program

The main advantage of name equivalence is that it eliminates a class of error which structural equivalence does not. That is "accidental" type equivalence, when two types that are being used to model different entities happen by chance to have the same structure. In name equivalence systems, an attempt to mix these types will be detected as an error, whereas in structural equivalence it will not. Thus name equivalence gives a finer grain of static program safety but, as always, at the cost of a loss of flexibility. The loss of flexibility here is manifested as an inability to describe anonymous types.

The programming language Ada introduces an *ad hoc* mechanism to achieve the effect of anonymous types. For example, it is not possible with strict name equivalence to write a procedure that will accept a parameter of a particular shape, such as a one dimensional array of a fixed size. The declaration

```
procedure Add-elements( A : array ( 1 .. 6 ) of INTEGER ) ;
```

is achieved in Ada by having an anonymous type mechanism for structure matching on arrays, rather than name equivalence. Such a disadvantage is only a minor drawback within a module since it only takes a type definition to resolve the problem. For example

```
type INT_ARRAY_ONE_SIX is array( 1 .. 6 ) of INTEGER ;  
procedure Add-elements( A : INT_ARRAY_ONE_SIX ) ;
```

At worst strict name equivalence is verbose. However, variable size arrays cause more problems since a different type name and procedure declaration are required

for each size. Ada uses another *ad hoc* mechanism, that of unconstrained arrays, to overcome this.

The extra flexibility of structural equivalence is that it allows type names to be used anywhere a type expression is valid and vice versa. This means that anonymous types, i.e. type expressions, may be used. In Napier88 a procedure may be defined that returns the *Age* field of any record of type *Age : int x Weight : real* :

```
let age = proc( x : structure( Age : int ; Weight : real ) → int ) ; x( Age )
```

In the above the identifier *age* is declared to be a procedure which takes a parameter *x* of type *structure(Age : int ; Weight : real)* and returns an integer result. The body of the procedure is the expression *x(Age)* which is the Napier88 syntax for selecting the *Age* field of the structure *x*. The result is the value of the *Age* field. This procedure may be used for values of type *vehicle* or *animal*.

Another example of the use of structural equivalence is given in the passing of procedures as parameters. The procedure *integral* is given below.

```
let integral =  
  proc( f : proc( real → real ) ; a , b : real ; no_of_steps : int → real )  
  begin  
    let h = ( b - a ) / float( no_of_steps )  
    let sum := 0.5 * ( f( a ) + f( b ) )  
    for i = 1 to no_of_steps do  
      begin  
        a := a + h  
        sum := sum + f( a )  
      end  
    h * sum  
  end
```

In this code fragment the identifier *integral* is declared to be a procedure that takes four parameters and returns a *real* result. It calculates the integral of a function between two limits, *a* and *b*, using the trapezoidal rule with *no_of_steps* intervals. These variables are given as parameters. The first parameter is the function to be integrated. Its name is *f* and its type is specified by the type expression

proc(real → real).

The type of f is thus a procedure that takes a **real** parameter and returns a **real** result, and any procedure of this structural form may be used. If a name equivalence system were to achieve this, every procedure of this form would have to be created with reference to the same type declaration, which may lead to excessive program noise.

There is a further problem here with standard procedures. As these have been declared before the compilation of any other programs, they must have been created using types which are incompatible with those declared in a new module. Therefore they may not be used as parameters to this kind of procedure without another new mechanism.

4.2.3 Equivalence models within a programming system

The above problem with standard procedures is a special case of a general problem for programming systems which are constructed from a number of program units. For data to be safely shared between programs a type system must provide modelling and protection facilities over more than a single compilation unit. For example, one program may be responsible for the creation of data which represent animals, and for the placing of these data into a persistent store. Another program may operate over the data which have been placed in the store by the first program. Both of these programs must therefore have the ability to specify the same type for the data to be safely manipulated.

A naïve solution would be to immediately eliminate name equivalence, as structural equivalence does not present a problem due to its ability to specify anonymous types. However, the needs of separate compilation also imply the following requirements of a type system:

- to share the type of data between modules

- to share the type of data between independently prepared sub-systems
- to abstract over complex type definitions within a sub-system
- to execute typechecking efficiently during program execution

Neither name nor structural equivalence is sufficient to answer all of these. Name equivalence does not allow equivalent types to be declared in different compilation units, and so does not address the first two requirements. Structural equivalence however is adequate for both of these. Neither system is sufficient for the third requirement, as with a structural equivalence system types must be re-declared wherever they are used in different modules. For the final point, name equivalence presents no problems, but a test for structural equivalence of two complex types may be very expensive.

The same solution, however, may be applied to both models of equivalence to provide adequate solutions to all of these requirements. If the extra ability to export type definitions from the context of their declaration is allowed, both structural and name equivalence models may adequately support all of the above requirements.

A minimal model for the sharing of type definitions among compilation units is a dictionary of type definitions, with operations to store and retrieve them. **typeExport** takes a string and a type and enters the type in the dictionary using the string as a key. **typeImport** takes only a string key, and results in the retrieval of the associated type.

In a programming language system based on the above type import and export model there would be a number of further requirements and problems which are not addressed here. These include problems with the time of evaluation, multiple and structured dictionaries, removing types, name clashes, and the semantics of failure. These problems are all orthogonal to the discussion of name and structural type equivalence.

4.2.3.1 Sharing a type between modules

Two programs, one of which places a datum of type *animal* into the persistent store, and the other of which retrieves it and prints its weight, will be used as an example. The environment mechanism of Napier88 will be used as a store interface. This mechanism is independent of the type equivalence semantics.

In a structural equivalence system, the ability to describe anonymous types implicitly allows the sharing of data types between modules without the exporting and importing of type definitions. For example, in Napier88 the example may be written as follows:

```
!** Program 1 - create and store an animal **
type animal is structure( Age : int ; Weight : real )
let newElephant = animal( 3 , 7.62 )      ! the type name is also a constructor
in PS() let Dumbo = newElephant          ! put the new value into the store

!** Program 2 - print Dumbo's weight **
type animal is structure( Age : int ; Weight : real )
use PS() with Dumbo : animal in         ! find the animal in the store
print( Dumbo( Weight ) )                ! dereference the Weight and print it
```

This example will successfully retrieve the value from the store, as the two different instances of the type definition are equivalent. If Napier88 had name equivalence semantics, this example would fail, as the two types have different declarations. To share data across programs in a name equivalence system the type definition must be exported.

The above programs may alternatively be written using type export and import as follows:

```

!** Program 1 - create and store an animal **
type animal is structure( Age : int ; Weight : real )
typeExport( "animal" , animal )      ! store type animal in the dictionary
let newElephant = animal( 3 , 7.62 )
in PS() let Dumbo = newElephant

!** Program 2 - print Dumbo's weight **
type animal is typeImport( "animal" ) ! look up type animal
use PS() with Dumbo : animal in
print( Dumbo( Weight ) )

```

Now both programs share the same definition of the type *animal*. They will execute successfully in either a name or structural equivalence system.

Ada does not provide mechanisms which allow these programs to be constructed in a type secure manner. Similar programs could be written, one of which would store a value in a file and the other of which would retrieve it. However, this may be achieved only by having separate type declarations in the two programs, and coercing the type of the file contents when the value was retrieved. There is no test for any kind of type equivalence, and the coercion is not a type-safe operation.

4.2.3.2 Sharing a type between independent sub-systems

In the above example program, a type definition is shared to force type equivalence between modules of a programming system. However, a further requirement often stated for such systems is the ability to merge program and data which have been independently prepared, and which as such may not have access to a common type dictionary.

If such a merge is to succeed, then both systems must have a foreknowledge of the structure of the types which are to be compatible, otherwise the system could not work at all after the merge. The essence is that this knowledge may be passed by a medium other than the computing system which is being constructed.

With a structural equivalence system, there is no further difficulty, except perhaps one of performance. With name equivalence, if there is no way for the sub-

systems to share at least some of the system's address space, then there is no possible way two types may be compatible without some further addition to the model. This may take the form of an assertion by one or both of the sub-systems' programmers that two types, one from each sub-system, are intended to be compatible. If, for example, each sub-system had its own type dictionary, then the dictionaries could be merged, with assertions being made about which pairs of types represented within the dictionaries were to be compatible.

At the time of the merge, a structural check is required to make sure that each pair of asserted equivalent types do indeed have the same structure, otherwise strong typing could be compromised. If such merging is to be performed during the execution of programs, then the structural equivalence testing poses some important questions of performance.

4.2.3.3 Abstracting over complex type definitions

One effect of using a type system with fine grain control over data and maximum static checking is that type definitions may become very large. Even for relatively straightforward applications, mutually recursive types consisting of hundreds of individual definitions are common. In a persistent language with flexible binding control, it becomes desirable to write systems in many small units, each of which can be understood by a single programmer. Typically these units are at most a few tens of lines of code.

These small units may however manipulate values of complex types. It is not acceptable to force these small units each to include their own copy of a huge type definition, as this takes them once more out of the intellectual grasp of a human being. In a database system, for example, every use may be forced to specify the entire database schema. The export and import of type definitions may therefore

be seen as a solution to another requirement, that of abstracting over complex type definitions which are used in a number of different modules.

Notice that this abstraction is never an extra problem for a name equivalence system, as it is already a requirement for a single definition of a type to be accessible for type equivalence to succeed. However, it is possible to design a structural equivalence system which does not include an ability to abstract over shared type definitions. Such a system would allow arbitrary type equivalence across compilation boundaries, but would also force the problems of large repeated type definitions onto its programmers.

4.2.3.4 Efficient execution

Name equivalence checking may always be implemented efficiently by the use of a single type name server within a sub-system. As only a finite number of type definitions may be made within a sub-system, the type equivalence check may always be coerced into a check of integer equality.

Structural equivalence, on the other hand, has a danger of being highly inefficient. As already stated, type definitions may become very large. The cost of a full structural check over large representations is intrinsically high, although ways of optimising it are shown in the next section.

In practice, the cost in efficiency of the two extended systems described should be approximately the same. In a structural equivalence system, it is still possible for equivalent types to share the same declaration. When this is the case, a full structural check is not necessary.

Within a program unit, structural checks are very uncommon as anonymous type definitions are rarely used. As shown in the discussion of name equivalence,

anonymous type definitions need never be used. In the cases where they are used to eliminate program noise, the definitions would be simple and therefore inexpensive to check. Complex types used in different units of a sub-system could be shared, and the same efficiency as that of name equivalence testing may be achieved. The full cost of structural typechecking need only be paid for the merging of independently prepared sub-systems, and in this case the full structural check is necessary for both name and structural equivalence systems.

4.2.4 A Universal Equivalence Model

It can therefore be seen that both structural and name equivalence semantics are suitable for use in a persistent programming language. The tradeoffs within a program unit are clear, being a slightly finer grain of control versus the ability to specify anonymous types.

Perhaps surprisingly, when the typechecking models are suitably extended and examined in further detail for typechecking within a programming system, no further difference may easily be distinguished between the two systems. This is because any complex type which is shared between units of a sub-system is expected to have only a single definition in both systems, which therefore forces type equivalence whether the semantics are name or structural. In the case of merging sub-systems, the same structural testing must be performed for both systems, the only difference being that it is implicit for structural equivalence systems and must be made explicit for name equivalence systems.

Given that this structural test is necessary for all systems, and may be made during program execution, it is desirable to make it as efficient as possible.

4.3 Implementation of structural type checking

4.3.1 Type Equivalence Checking

Structural type equivalence was first introduced in the programming language Algol-68 [vWMP69]. In a structural equivalence type system, the types consist of sets defined over the value space of a language. Membership of these is defined by some properties of the values themselves. Values may be of the same type only if they have the same set of operations defined over them. Type equivalence is therefore an implicit property of a value, and values do not need to be constructed with reference to a type definition. To decide type equivalence, a language definition must include a set of type rules. These define the universe of discourse of the language and allow the type of any value to be deduced.

The universe of discourse of a type system may be represented by the set of base types and the set of type constructors. Type constructors allow the derivation of new types from other types and perhaps some other information. Where the language is data type complete, the universe of discourse is infinite, consisting of the closure of the recursive application of the type constructors over the base types.

The structural type equivalence relation may be described with a similar set of rules. For two types to be equivalent, they must be created with the same type constructor and in an equivalent manner, using types which are themselves equivalent. An equivalence rule must be defined for each different type constructor.

To perform structural type equivalence checking, it is necessary to build representations of types which contain sufficient information to establish the defined equivalence for each constructed type. An equivalence function which compares two instances of such representations must also be defined. The

essential feature of any representation type is that there exists a well-defined mapping from the value space of the representations to the type space of the language. It may be desirable in some systems for different values to represent the same type, as long as the equivalence algorithm used implements an equivalence relation which respects the semantics of structural type equivalence.

4.3.2 Representing Types

Any type is either a base type or a constructed type. Constructed types are a composition of other types, along with some information specific to the particular construction. This information could consist of, for example, field names in a record type or the ordering of parameters for a procedure type. In general, therefore, a type representation consists of three parts:

- a label, to determine which base type or constructor it represents
- the information specific to the construction of this type, if any
- a set of references to other type representations

The equivalence algorithm for a representation must check that, for any two representations, that the labels are the same, the specific information is compatible, and that the other types referred to are recursively equivalent.

For some type systems there is a requirement that the chain of references may be circular. This is the case in a type system with recursion, where circular references are used to achieve a finite representation. For example, the type of an integer list may be

```
rec type IntList is structure( head : int ; tail : IntList )
```

Also, to represent a type system which includes values of either universally or existentially quantified types, it is essential for any quantifier type to contain a reference to the type to which it is bound, to allow either inference or explicit

specialisation to deduce the correct type equivalence rules of values with these types.

These circular references, although not increasing the conceptual complexity of type representations, are the source of serious problems with the efficient implementation of a structural equivalence algorithm.

4.3.3 Efficient Structural Checking

There are two factors which can cause serious problems with the efficient implementation of structural checking. A trend in modern programming languages, and particularly database programming languages, is to provide more and more sophisticated type systems which allow more program errors to be detected statically. This is currently pushing knowledge of static type checking to its limits, and there are even systems which need to employ theorem provers within the type checking system [SSS88]. Programmers are encouraged to provide the most detailed type specification possible, as this increases the chance of a programming error being detected before execution. As a consequence of this, type specifications may become extremely large and complex. It may be imagined that the size of a database schema specified statically as a type is considerable. In a system which performs structural equivalence checking dynamically, it must be possible to check types of this complexity without incurring an unacceptable overhead.

The problem of large representations is compounded by the fact that they may contain cycles. In general, an algorithm which traverses a potentially cyclic structure must check at each stage that its area of current interest has not been previously traversed. If this check is not made, then the algorithm cannot be guaranteed to terminate.

The check for cycles must be made on an attribute which is unique to each component of the representation, rather than to the type constructor it represents. This may be, for example, the identity of a node in a graph representation or the starting position within a string representation. The difficulty here is that there is only a small amount of information specific to a particular constructor, most of the important information of the type representation being resident in its topology. It may not be possible to define a useful ordering over the node instances for the purpose of a fast lookup. This depends on the chosen representation and the implementation language. If there is no good ordering, the major cost of the equivalence algorithm becomes a check for equivalent cycles, and its complexity is $O(n^2)$ where n is the number of nodes. This is because during the traversal of the graph, itself of $O(n)$, the cost of checking whether a node has been previously visited is itself $O(n)$ [CBC90].

The performance of algorithms to check type equivalence is crucial in a persistent system, as checking may frequently be required during the execution of a program. After some more general discussion of efficiency considerations, two different implementations of structural equivalence checking are described.

4.3.4 Normalisation

It may be seen that there is a major tradeoff between the cost of constructing type representations and the cost of executing the equivalence algorithm. For example, strings which consist of definitions within a language's type algebra contain sufficient information to perform equivalence checking, but the checking algorithm is complex. As the construction of representations is a task performed during the static checking of the program, and equivalence checking may be performed during execution, it is clearly desirable to put as much of the burden as possible into the building of the representations.

For example, consider a type system which includes a structure type which is a labelled cross product. Two such type constructions are considered equivalent if they are constructed over equivalent types using the same labels, but the order of the labels is not significant. Therefore,

structure(a : int ; b : bool)

and

structure(b : bool ; a : int)

are equivalent. In general, as the ordering of the fields is unimportant, representations may be constructed with the fields in any order. In this case, the equivalence algorithm must allow for this during its execution. The fields may however be rearranged by placing them in alphabetical order according to the labels. If this is the case, the equivalence algorithm may then assume that the ordering of the fields is significant. Thus the task of equivalence checking may be simplified at the cost of complicating the task of building representations.

In general, it is possible for many differently "shaped" representations to be constructed for equivalent types. For example, consider the equivalent types:

structure(a,b : structure(c : int))

and

structure(a : structure(c : int) ; b : structure(c : int))

If the algorithm which constructs type representations is written naïvely, then the first of these definitions may result in what is, in some sense, a minimal representation of this type, whereas the second may contain duplicate components.

A normal form is one in which no two component representations are equivalent to each other. The construction of a normalised representation may be highly expensive computationally, as it involves checking every component representation

for equivalence with every other one. Balanced against this, for some classes of representation the equivalence algorithm for normalised representations may be substantially faster. This will be discussed in more detail in Section 4.3.6.

4.3.5 Representing types by graphs

In an implementation language which has a constructor type such as a record or structure, a graph representation of types is straightforward and elegant. It is highly suitable because of the recursive nature of type definitions and the requirement to have circular references between constructor nodes. This makes such representations simple to build and to decompose, and as such they are ideally suited for static type checking purposes. In one implementation of the Napier88 system the following representation type is used, where for simplicity the type *list* and its usual operators are assumed to be pre-defined.

```
rec type TYPE is structure
(
    label : int ;
    specificInfo : string ;
    references : list[ TYPE ]
)
```

This is sufficient to uniquely represent any type which may be described by the Napier88 type system using some straightforward mapping rules. The *label* field distinguishes the base type or constructor each node represents. The *specificInfo* field contains information such as structure field names, concatenated with markers to form a single string. The *references* field represents all references to other types from this type constructor. This has an implicit ordering which may be used as part of the type information where required.

Type equivalence is a recursively defined algorithm over this structure, and must check only for equality of the *label* and *specificInfo* fields, before recursively

checking any representation in the *references* field. The following algorithm would work for type systems where cycles do not occur:

```

rec let eqType = proc( a,b : TYPE → bool )
    a = b or      !** this means pointer equality (identity)
    (
        a( label ) = b( label ) and
        a( specificInfo ) = b( specificInfo ) and
        eqList( a( references ) , b( references ) )
    )

& eqList = proc( a,b : list[ TYPE ] → bool )
    ( a is tip and b is tip ) or
    (
        a isnt tip and b isnt tip and
        eqType( head( a ),head( b ) ) and eqList( tail( a ),tail( b ) )
    )

```

As described previously, it may often be the case that the types being checked have the same identity. In this case the equivalence is detected immediately, otherwise the full structural check is necessary. Notice that the test for identity is also performed recursively, which may optimise cases where two different representations share components with the same identity.

When the possibility of cyclic structures is introduced, it is necessary to take further steps to ensure the termination of the algorithm for equivalent types. This may be achieved by keeping a note of all pairs of nodes that are traversed in a "loop table", as depicted in Figure 4.3.1. Before any pair of nodes is traversed, a check is made to see whether the same pair has already been encountered. If they have, then either the full recursive check over these nodes has already been performed, or else is in the process of being performed. If the check has already been performed, then the nodes must be equivalent, otherwise the algorithm would have already been terminated with failure. In the case where the test is still in progress, these nodes may safely be assumed to be equivalent. If they turn out to be equivalent then the assumption is correct and re-traversal of the loop has been avoided. If they turn out to be non-equivalent then the algorithm will in any case end with failure from another branch of the recursion.

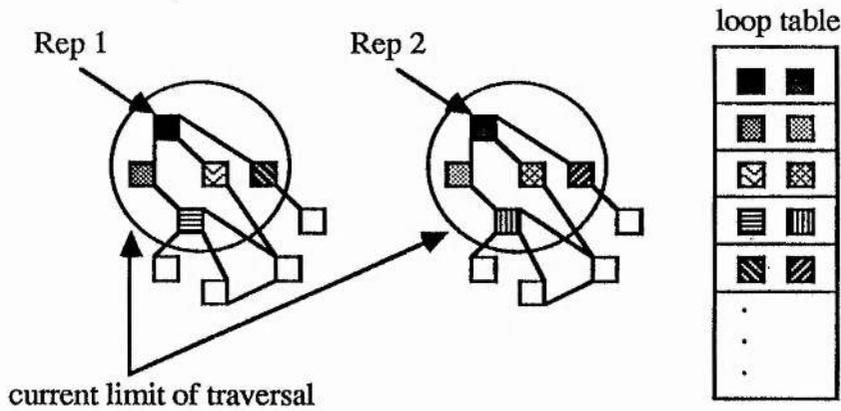


Figure 4.3.1 Placing traversed nodes in a loop table

The new algorithm looks like this:

```

rec let eqType = proc( a,b : TYPE → bool )
  a = b or
  in_loop_table( a,b ) or
begin
  add_to_loop_table( a,b )
  a( label ) = b( label ) and
  ...

```

The use of the loop table not only ensures termination in the case of a cycle in the graph, but also prevents multiple traversals of a shared component within the graph. This enhances the potential efficiency of the algorithm.

With this representation, recording and looking up pairs of nodes in the loop table may cause a performance problem, as no suitable key is readily available to use for indexing. This can be simply solved by introducing an extra key field into the type representations. When each type representation is created, this field is initialised with a value which may be used as a key for the node. Each pair of nodes, when encountered on the first traversal, may be tabulated using one of these keys. This technique minimises the cost of checking for cycles.

Another possibility is to use a pseudo-random number instead of a key, and to use a hashing algorithm based on this. The reason for using pseudo-random keys is

that the hash table may be efficiently preserved between executions of the equivalence function, and will then act as a memo table for all pointer pairs which are compared more than once. The pseudo-random keys reduce the possibility of hash clusters forming.

This persistent hash table has the interesting feature that it is not required to preserve the correctness of the algorithm, and so may be re-initialised at any time. It is important also to note that should the equivalence algorithm fail, all nodes added during the comparison must be removed. For this reason, a "shadow-copied" table is used, which may be either preserved or restored depending on the outcome of the equivalence test.

The use of these techniques allows the check for cycles to be performed in constant time, and so the checking algorithm may achieve complexity of $O(n)$ where n is the number of nodes.

4.3.6 Representing Types by Strings

There are many possible ways of representing types by strings. One possibility is to use strings which consist of type definitions within the type algebra of the programming language, which would normally be sufficiently powerful to provide a representation for any type in the language's universe of discourse. However, for a sophisticated type system, any equivalence algorithm over such representations would be highly inefficient.

A unique string representation is more useful. Such a transformation from types to strings allows the type equivalence relation to be modelled by string equality. This may be implemented in a computer by a block comparison, an extremely fast operation on most machines.

A constructive proof that a unique string form exists for a particular type system is beyond the aims of this thesis. A method of construction will instead be outlined. It should be pointed out that the outlined method is by no means the most efficient, but is described for reasons of clarity.

The method relies upon the assumption that the graph representation and equivalence algorithm outlined above are sufficient to model type equivalence in the system in question. Firstly an algorithm will be described which produces a normal form of any such graph. Another algorithm will then be described, which maps graphs to strings. This mapping has the desired effect of producing a unique string representation for any type.

4.3.6.1 Normalising Graphs

The condition for a graph to be normal is that no two nodes within a type representation represent equivalent types. Therefore the type equivalence algorithm may be applied to any two nodes within such a graph, and will always fail unless the nodes have the same identity. The algorithm we will describe to map a graph to its normal form operates by copying the graph, but mapping any equivalent nodes in the original graph to a single new node.

This algorithm relies upon a data structure similar to the loop table of the previous algorithm. In this context we will describe it as a "memo table". The table still contains pairs of nodes, but now each node traversed is paired with its new copy, as shown in Figure 4.3.2. Each node traversed is placed in the table as before, and the table is used to memoise the result which has already been, or is currently being, calculated for the normal form which represents the node. The algorithm to produce the normal form of a node checks in this table to see whether it has previously produced, or is in the process of producing, a normal representation for

an equivalent node. The check is thus based upon the equivalence algorithm previously described, rather than simple identity.

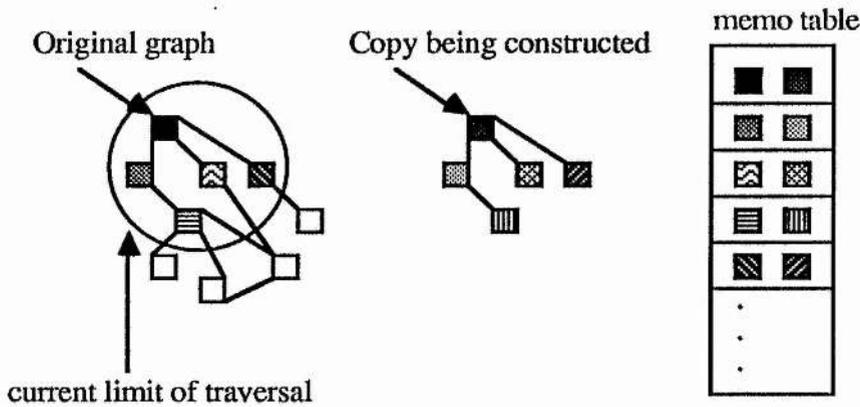


Figure 4.3.2 Using a memo table to normalise graphs

The algorithm is as follows:

1. Construct a new memo table.
2. To copy a node, first check in the memo table to see if a node with an equivalent type has already been copied. If it has, return the corresponding node stored with it. Otherwise,
 - a. Create a new node, without filling in the fields.
 - b. Add to the memo table a pair consisting of the node being copied and the new "dummy" node.
 - c. Fill in the fields appropriately, including recursive use of this algorithm to fill in the component types.

Notice the way that each new node must be created as a "dummy" so that each pair of identities may be added to the memo table before any recursive calls are made. Notice also that because the test applied to the memo table is equivalence, rather than identity, any nodes in the original graph which are equivalent will be mapped to the copy of the first of these nodes which is traversed during the copy algorithm. The resultant graph is therefore normal. For any type system which may be correctly represented using this graph representation, there exists only a single normal form which represents each type. Therefore there exists a reversible mapping between normal representations and types, and so this representation is

unique. This is achieved at the cost of executing the equivalence algorithm over every pair of nodes within the original graph.

4.3.6.2 Mapping to Strings

The following algorithm maps graphs to strings. It again uses a memo table, this time to provide unique names for any types which occur more than once in the representation. Again, this deals both with cycles and with shared components in the graph representation.

startSymbol, *endSymbol*, and *separatorSymbol* are mutually distinct characters which do not occur within the strings found in graph nodes.

Before traversal starts:

1. Construct a new memo table
2. Initialise *nextMarker*, a procedure which produces a deterministic series of unique strings, a different string being produced on each call. These strings consist of characters which do not occur within the strings found in graph nodes, and do not contain the characters *startSymbol*, *endSymbol*, and *separatorSymbol*

Traversal:

First check in the memo table to see if the node with this identity has already been traversed. If it has, return the corresponding string stored with it. Otherwise:

- a. Store the node in the memo table, associated with the result of calling *nextMarker*.
- b. The result string is the concatenation of:
 - *startSymbol*
 - The node's *label* field in string format
 - *separatorSymbol*
 - The node's *names* field
 - *separatorSymbol*
 - The result of the recursive application of this algorithm to any component types
 - *separatorSymbol*
 - *endSymbol*

The use of the memo table is illustrated in Figure 4.3.3.

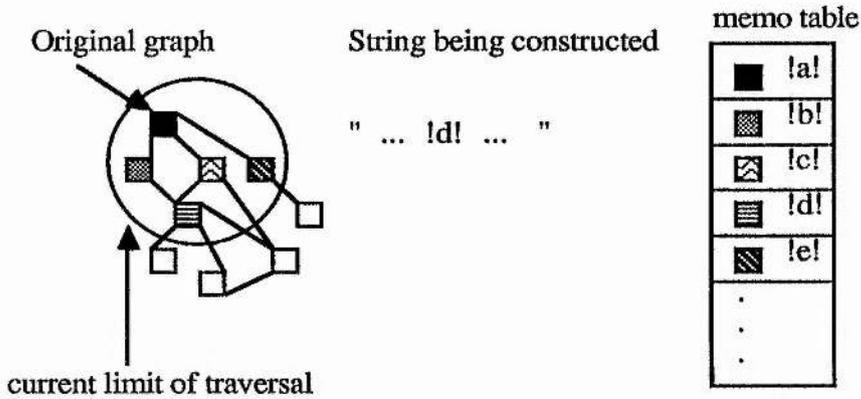


Figure 4.3.3 Use of a memo table in string production

The above algorithm both terminates and produces a unique form for any type. The unique markers used within the string for loops in the graph depend upon the traversal order of the graph; as the component types within any node are ordered as part of the normalisation process this order is always significant type information. The n^{th} marker produced by the *nextMarker* symbol corresponds to the n^{th} *startSymbol* which occurs within the string, and so these markers act as explicit references within the string. The string produced may thus be regarded as a normalised set of mutually recursive type definitions.

As these string forms are unique, the type equivalence relation is implemented by string equality. The important result is that an unstructured form exists for type representations, over which type equivalence may be implemented by a block comparison. This operation is already optimised in much conventional hardware, and so such a representation may be highly suitable for a prototype persistent system.

4.3.7 Comparison of Graphs and Strings

4.3.7.1 Speed

It has been shown in Section 4.3.5 that linear complexity can be achieved for equivalence algorithms over graphs as well as strings. This would imply that either representation is reasonable to build into a persistent system, as no dramatic slowdown would be associated with a programmer using more complex types in a program.

However, although the complexity may be the same, the hidden constant may be significantly different. The block comparison associated with strings would be faster than the graph traversal, even if both systems were constructed in hardware. In particular, conventional hardware is already optimised to perform block comparisons, and so the string representation should be substantially faster on an existing machine.

4.3.7.2 Space

The space occupied by an equivalent representation of a large type is significantly greater for graphs than strings. This is because each node in a graph carries the overhead associated with a persistent object, whereas a string may be implemented as a single object. Also, references to other nodes are persistent identifiers, which may be large depending upon the implementation of the persistent store. As the amount of information contained within any single node is relatively small, these overheads will be the major space cost of a graph representation.

The strings however require contiguous space, whilst the graphs do not. There are advantages on both sides of this. For large types, comparing strings may cause a large amount of volatile memory to be used up at one time, whereas the graph nodes may be fetched from non-volatile store in pairs if necessary to use only a

small amount of space. On the other hand, fetching the graph requires a large number of object faults, whereas only one is required for a string. The importance of these considerations depends upon the implementation of the persistent store.

As already stated however dynamic checking may be worthy of customised implementation, and it would be expected that both string and graph type representations would be adjusted to have similar characteristics using clustering, compression and fragmentation techniques as appropriate.

4.3.7.3 Sharing

So that the block comparison may be made over strings, all of the references to other types are converted so that they may be interpreted only within the context of the string. This means, unlike the graph representation, that sharing of common component type representations is not possible. This has serious implications for both time and space complexity.

For example, a program may require two values of related types from the store. This is common in persistent programming, where one procedure may generate an object of a complex type and others may use it in different ways:

```
type aType is ....  
let generator = proc( → aType ) ; ...  
let user1 = proc( aType → int ) ; ...  
let user2 = proc( aType → bool ) ; ...
```

Using graph representations, the type representations of these three procedures may use the representation already constructed for *aType*, and so only a few extra graph nodes are required to represent the more complex types. Using strings, however, complete new strings must be constructed for each procedure type, which duplicate all of the information already in the string constructed for *aType*, as the context-sensitive references may be different. This is because strings are a

truly anonymous representation, whereas graphs contain implicit naming information in their store addresses. The difference in space may be significant for programs which use a complex type in many different ways.

Another aspect of sharing components is that any memoisation performed over these components then carries over all types which use them. In the above example, it would be common practice for one program to define the three procedures and place them within the store and for another to subsequently access them. In this case, the memoisation performed by the graph equivalence algorithm will mean that the structural equivalence check is only performed once on the type *aType*, whereas the full structural check is required for each different procedure if a string representation were used. Again, this is a substantial saving for a large class of programs.

4.3.8 Measurements

To give an idea of the expected performance of these algorithms, some measurements taken from the Napier88 system are included. These measurements give some indication of the space overhead, performance, and complexity of the different schemes.

All measurements were made with the type of a Napier88 abstract syntax tree. This is an extended and revised version of PAIL [Dea87], and is a large, mutually recursive type, consisting of around one hundred and forty definitions.

The measurements of performance are hard to quantify, as they are highly dependent upon the implementation of the Napier88 system within which they were made. Suffice it to say that the best figures we have achieved for checking independently prepared versions of this type are at the rate of several per second

for a graph representation, with a substantial speedup to several hundred per second for a string representation.

The complexity measurements confirm the deduction that complexity at least as good as linear may be achieved for checking graph representations, with a suitable size of hash table. It is a reasonable assumption that a large enough table may be employed, as this is a fixed overhead per system. As explained, the table may be re-initialised if it becomes too large. In the case where the table may be large enough to contain all types used within the system, then the resulting memoisation achieves the same efficiency as name equivalence checking.

4.3.8.1 Space

The graph representation of the Napier88 PAIL type consists of 413 objects, with an average of just under nine words per object. The total size of this graph is 14,466 bytes. The string form of the graph is 2,206 bytes long.

These figures are all taken from relatively naïve representations; we have made no serious effort to compress the representations.

The benefits of space saving by sharing are hard to quantify, as they depend very much upon the manner in which the types are used. However, in our use of the abstract syntax tree type in building a Napier88 compiler, we found that the type is used in 276 different contexts. Each context requires a different string for its representation, as previously explained, but a graph may be shared by different contexts. Even if the system is fully optimised and only a single representation is used for each type, the total amount of store used to represent this type by strings is therefore more than 600,000 bytes, as oppose to a constant 14,466 bytes for the graph representation. Furthermore, each of these 276 representations requires at least one full structural check, whereas the first check using the graph

representation acts as a memo for any other check performed while the system is running.

4.4 Constructing type representations

In Section 4.3 it was stated that, for some type systems, values of the type shown in Figure 4.4.1 may be used to represent types for the implementation of type equivalence checking. In this section the discussion is about the construction of such type representations.

```
rec type TYPE is structure
(
  label          : int ;
  specificInfo   : string ;
  references     : list[ TYPE ]
)
```

Figure 4.4.1 The stated representation type

Napier88 is used as an example to show how a type system may be mapped onto this representation type [Con88]. The type system of Napier88 is sufficiently rich to encompass all of the type constructors in common use, and as such the description should show how type representations may be built to allow structural type checking over most other programming languages.

4.4.1 The Napier88 type system and algebra

The Napier88 type system is described by a set of base types and a set of constructors. The base types are integer, real, boolean, string, pixel, picture, file, and null. The Napier88 type constructors are:

- vector - One-dimensional array
- image - Rectangular array of pixels
- structure - Record type (labelled cross product)
- variant - Discriminated union (labelled disjoint sum)
- procedure - First class data type

- polymorphic procedure - Universally quantified procedures
- abstract data type - Structure type with existential quantification
- environment - Sets of bindings
- any - Infinite union of all types

The universe of discourse of Napier88 is defined by the closure of the recursive application of these constructors over the base types.

To increase the modelling power of the type constructors, Napier88 uses a simple type algebra. This algebra is fully evaluated before program execution.

- Types may be named, to allow abstraction within a program unit
- The definition of recursive types is allowed
- Parameterised types are provided to reduce program complexity

A full description of the types which may be described by the Napier88 type algebra is included as an appendix.

The introduction of both recursive and parameterised definitions may cause some problems with the decidability of structural equivalence [Sol78]. Some languages [Fai82, Car89] restrict the use of parameterisation to avoid these problems. The Napier88 type system is more general, allowing any combination of recursion and parameterisation for which a decidable algorithm is known. The class of recursive parameterised types for which no decidable equivalence algorithm is known corresponds to types which may not be represented by a finite graph of the representation type shown. The definition of such types is not allowed in Napier88, and will be detected before execution.

Figure 4.4.2 shows an overall view of the representation strategy for this type system and algebra. The columns in the table show in outline how the values for each type representation structure model the appropriate information for each constructed type. Only the types with label values from one to nine inclusive represent types which are used to describe denotable values within a language.

The other labels are needed during the construction of types as may be specified by the type algebra, and their purpose will be explained in detail later.

Although no formal justification will be given, it may be seen that deep equality over values created with these formats simulates type equivalence, as described in the previous section.

Constructor	Label	Specific Info	References
Base type	1	String description of base type	Empty
Vector	2	None	Single list element for element type
Structure	3	Representation of field names	One list element for each field type
Variant	4	Representation of branch names	One list element for each branch type
Procedures	5	Number of arguments	One list element for each argument, and one for the result
Polymorphic procedure	6	Number of universal quantifiers and number of arguments	One list element for each argument, and one for the result
Universal quantifier	7	Which one of the procedure's quantifiers	Polymorphic procedure bound to
Abstract data type	8	Number of existential quantifiers and representation of field names	One list element for each field type
Witness type	9	Which one of the abstract type's quantifiers	Abstract data type bound to
Parameterised type	10	Number of parameters and <i>label, specificInfo</i> of specialised type	<i>references</i> of specialised type
Type parameter	11	Which one of the parameterised type's parameters	Parameterised type bound to
Recursive type	12	Recursive name	Parameterisation information

Unbound universal quantifier	13	Which quantifier	Empty
Unbound witness type	14	Which witness	Empty
Type with free quantifier	15	None	Single list element of type with free quantifier

Figure 4.4.2 Outline of mapping from types to representations

4.4.2 Constructing type representations

The Napier88 compiler is built in a number of modules. One of these is the types module, which provides an interface which abstracts over the representation type. The module provides functions in its interface which cause the construction of appropriate representations, as well as the equivalence function shown in Section 4.3. The syntax analyser therefore has a relatively high-level view of the type system. One of the design aims of the type checking module was that it should be usable by the compilers of a number of different programming languages, rather than be specific to the Napier88 compiler.

The Napier88 types module is an abstract data type, with an interface which abstracts over the representation of types. For each type constructor, there will be an explanation of the relevant module interface procedures and how they should be used, followed by a detailed explanation of the implementation of these procedures.

For the examples, graphs will be drawn which correspond to the representation type as demonstrated in Figure 4.4.3.

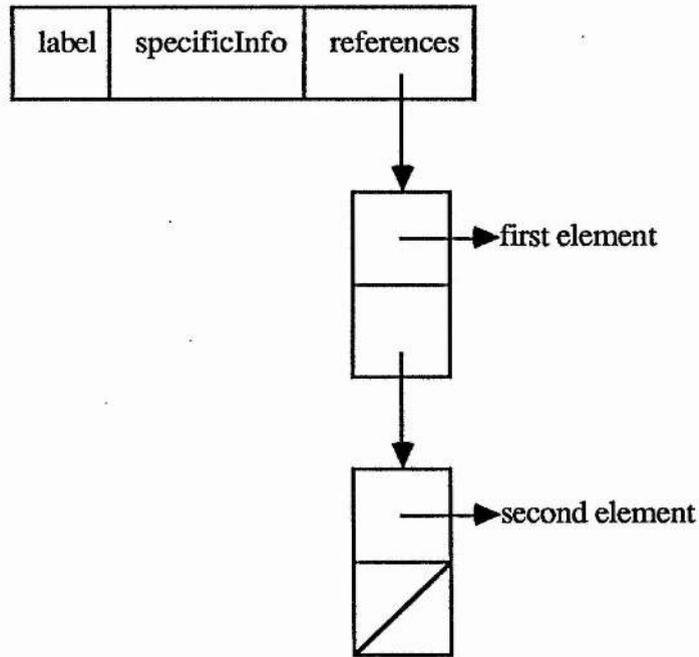


Figure 4.4.3 The graphical representation of type representations

4.4.2.1 Base types

To allow language independence, the types module allows base types to be defined, rather than exporting a fixed set of base types. For a particular language, a compiler would be expected to define the base types only once, and place these values in a persistent environment. These base type representations are created from outside the types module by an application of the function *baseType*, which takes a string parameter and returns a type representation.

Type description:

int

Possible module use:

let INT = baseType("int")

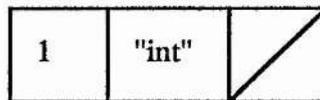


Figure 4.4.4 Base type representation

A syntax analyser may also use these representations for other purposes, for instance the manipulation of type descriptions corresponding to void and undefined expressions. Figure 4.4.4 shows an example of a base type construction.

4.4.2.2 Vectors

The *mkVector* function takes a type representation and returns the type of a vector with elements of this type. No *specificInfo* is required, and the *references* field consists of a list which always has a single element which points to the type of the vector elements. The associated selector function *elms*, which is used to check the type of indexing, takes the vector type and returns the element type. Figure 4.4.5 shows an example of a vector type construction. The syntax used to denote a vector type in Napier88 is an asterisk before the element type. This allows the concise definition of multi-dimensional vector types.

Type description:
***int**

Possible module use:
 mkVector(INT)

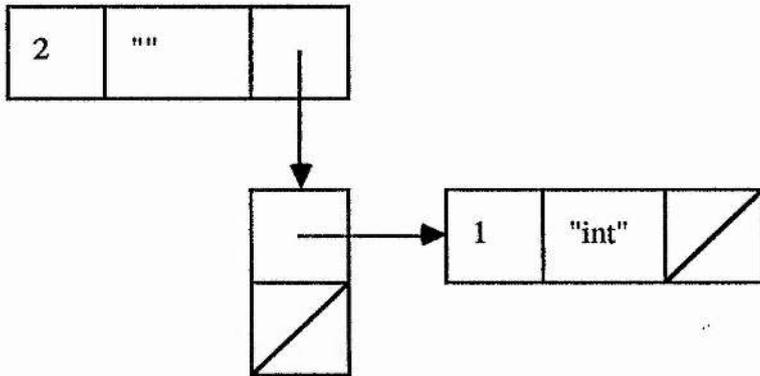


Figure 4.4.5 Vector representation

4.4.2.3 Structures

The types module supports two functions which build representations of structure types. The first, *newStructure*, is a function of no arguments which returns the

type representation for a structure with no fields. The second, *addField*, takes a structure type representation, a name, and a type, and augments the structure with the new name-type pair. The result reports whether the operation has been successful or not ; it will fail in the case of a name clash.

Two functions are provided to decompose one of these type representations. The first, *fieldType*, takes a structure type representation and a string, and returns the type associated with the name represented by the string. The second, *scanStructure*, is a scanning function, which takes a structure type representation as its argument and returns a procedure which returns a string. This returned procedure itself returns, on consecutive calls, each of the fieldnames of the structure in alphabetical order. When they are exhausted, it returns the empty string whenever it is called.

Type description:

structure(x : int ; y : bool)

Possible module use:

```
let new = newStructure()
let ok1 = addField( new, "x", INT )
let ok2 = addField( new, "y", BOOL )
```

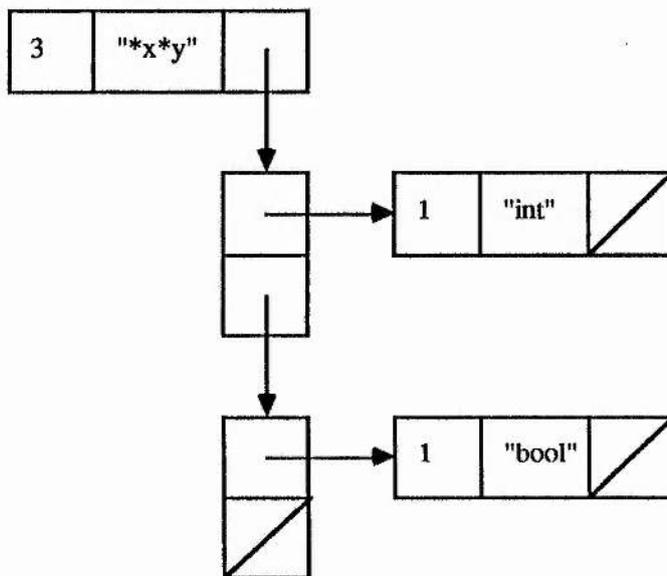


Figure 4.4.6 Structure representation

The *specificInfo* field contains a list of the structure's field names, each preceded by a reserved separator symbol. The *references* field contains a list of types, each element corresponding to one of the fieldnames, in the same order. Thus the *specificInfo* contains enough information for the correct interpretation of the list.

4.4.2.4 Variants

The type rules for variants are isomorphic to the type rules for structures, and the implementation is correspondingly isomorphic. Two functions are provided to build up a variant type, *newVariant* and *addBranch*. *newVariant* takes no parameters and returns a variant type with no branches. *addBranch* takes a variant type, a string, and a type and adds a branch to the variant type with a name corresponding to the string and the appropriate type.

Type description:

variant(x : int ; y : bool)

Possible module use:

```
let new = newVariant()
let ok1 = addBranch( new, "x", INT )
let ok2 = addBranch( new, "y", BOOL )
```

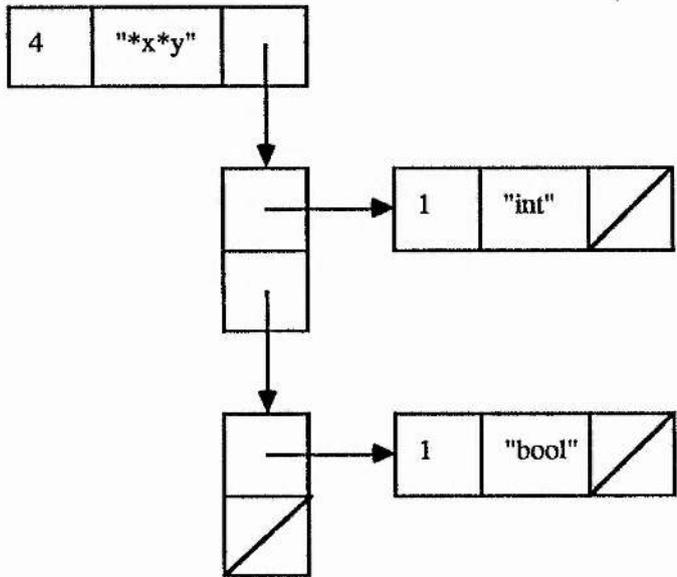


Figure 4.4.7 Variant representation

The selector function *branchType* takes a variant type and a string, and returns the type of the branch with a name corresponding to the string. *branchNumber* takes a variant type and a branch name as arguments, and returns an integer. The integers returned are calculated uniquely for each branch of the variant. This function is to allow the construction of tags for the branches of a variant.

The *specificInfo* field contains a list of the variant's branch names, each preceded by a reserved separator symbol. The *references* field contains a list of types, each element corresponding to one of the branch names, in the same order. Thus the *specificInfo* contains enough information for the correct interpretation of the list.

4.4.2.5 Procedures

The function to create a procedure type representation, *mkProc*, takes a list of type representations for the procedure's arguments and a type representation for its result, and returns a representation of the procedure type.

Of the selector functions, *args* takes a procedure type representation and returns its list of arguments, and *result* takes a procedure type representation and returns its result.

The *specificInfo* in this type contains the number of parameters for the procedure type. The *references* field points to a list of parameter representations, with the result type representation appended to the list.

Type description:
proc(int, bool → real)

Possible module use:
**let arguments = append(BOOL, append(INT, nilList))
mkProc(arguments, REAL)**

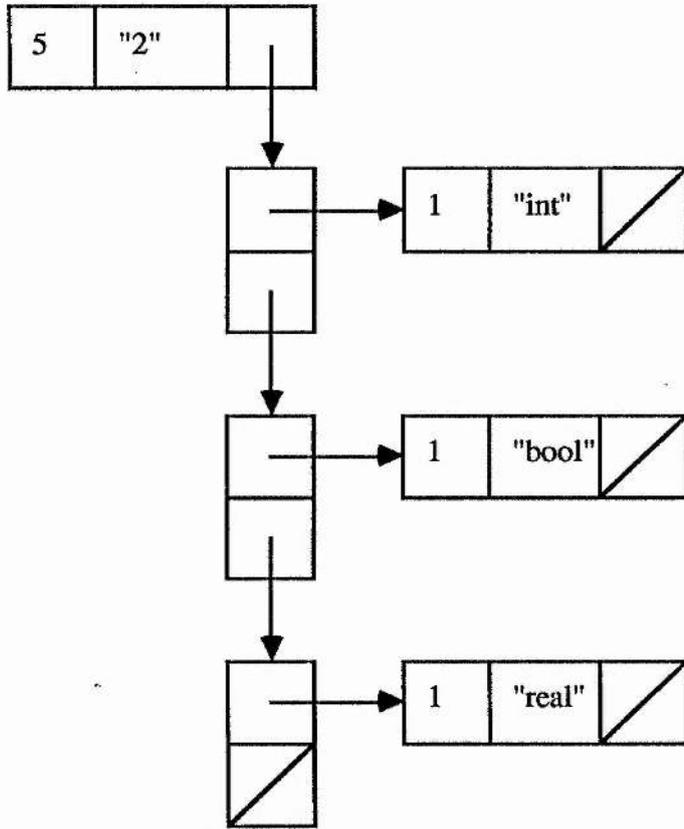


Figure 4.4.8 Procedure representation

4.4.3 Quantified types

In Napier88, the universe of discourse includes two different quantified types: polymorphic procedures and abstract data types. Figure 4.4.9 shows an example of a universally quantified procedure written in the Napier88 type algebra. This is the type of a procedure which has a parameter which may be of any type, and returns a result of the same type as the parameter.

type a is proc[t](t → t)

Figure 4.4.9 A universally quantified procedure type

The construction of a representation of a universally quantified procedure type by the Napier88 module occurs in three parts. First, appropriate quantifiers are constructed by use of the *mkQuantifier* function. This takes an integer argument, to denote which quantifier of the particular type is being constructed. Secondly, an appropriate procedure representation is created using the *mkProc* function as previously described. Finally, to bind the quantifiers to the procedure, the procedure *mkQuantified* is called, which takes as arguments a procedure type representation and a list of quantifier representations.

In the quantified procedure type representation, the *specificInfo* field must contain two pieces of information: the number of quantifiers and the number of parameters that the procedure has. These are represented in the same string, with a separator character between them. The *references* field represents the types of the procedure's arguments and result, as for a monomorphic procedure.

Each quantifier also has a type representation. The *specificInfo* field of these distinguishes between quantifiers in the case where a procedure has more than one quantifier. The *references* field of each quantifier contains a single list node which points back to the representation of the quantified procedure type.

Type description:
`proc[t](t → t)`

Possible module use:
`let t = mkQuantifier(1)
let p = mkProc(append(t , nilList) , t)
mkQuantified(p , append(t , nilList))`

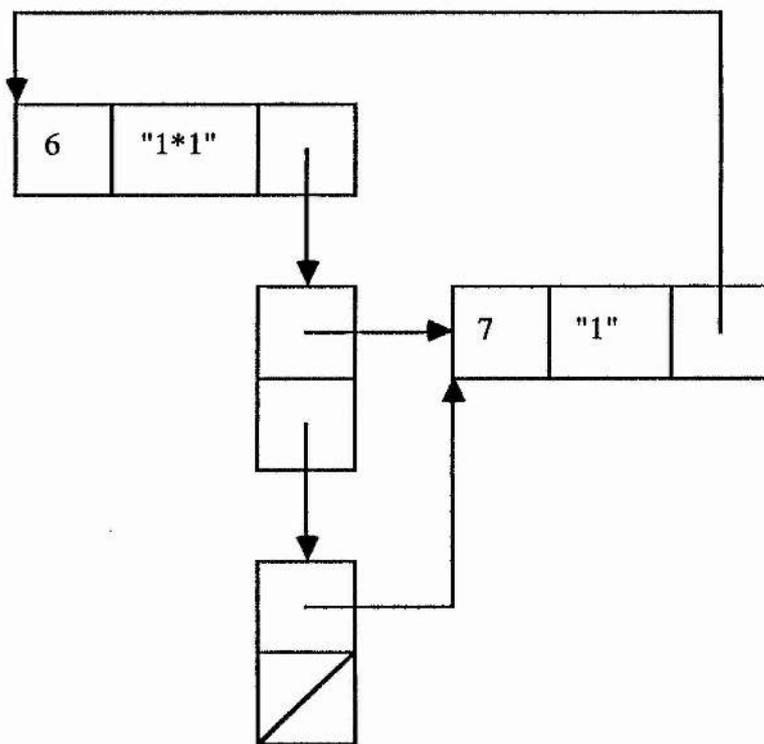


Figure 4.4.10 Universally quantified procedure representation

For the testing of equality, it is necessary for the quantifiers of these type representations to be associated with the representations of their quantified types. In types where different quantifiers are in scope at the same time, as in the example in Figure 4.4.11, it is essential to be able to distinguish them by the structure of the representations. This may be achieved by using cyclic graphs to represent quantified types, with the representations of the quantifiers pointing back to the representation of the type of which it is a quantifier. Figure 4.4.12 shows how this creates graphs with a different topology for the types shown in Figure 4.4.11.

type a is $\text{proc}[s](\text{proc}[t](s \rightarrow t) \rightarrow s)$
type b is $\text{proc}[s](\text{proc}[t](t \rightarrow s) \rightarrow s)$

Figure 4.4.11 Distinguishing quantifier bindings

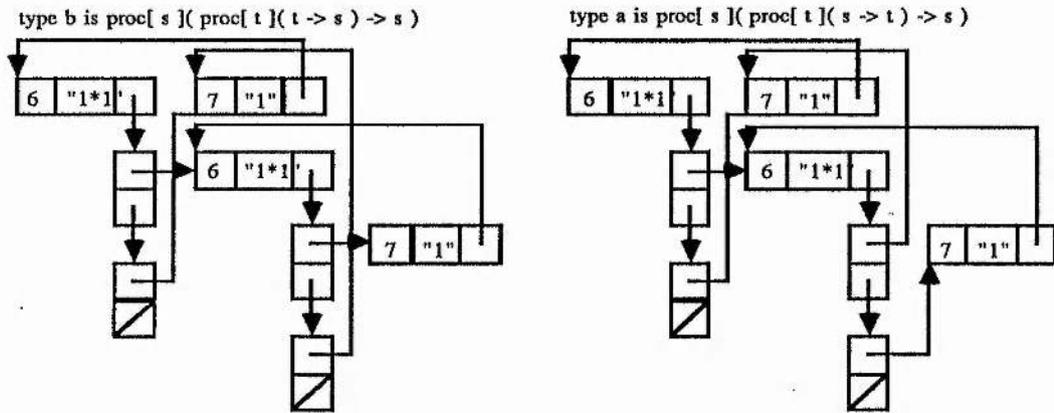


Figure 4.4.12 Different topologies for non-equivalent quantified procedures

Existentially quantified structure types are constructed in a similar manner. The interface procedures for existential types are called *mkWitness* and *mkAbstract*. The *specificInfo* field of an existentially quantified representation contains both the number of quantifiers and the concatenated field names, separated by asterisks. Figure 4.4.13 shows such a type in the Napier88 type algebra, and Figure 4.4.14 shows how its representation is constructed.

```
type b is abstype[ t ]( x , y : t )
```

Figure 4.4.13 An existentially quantified structure type

Type description:
abstype[t](x , y : t)

Possible module use:
let t = mkWitness(1)
let new = newStructure()
let ok1 = addField(new, "x", t)
let ok2 = addField(new, "y", t)
mkAbstract(new, append(t , nilList))

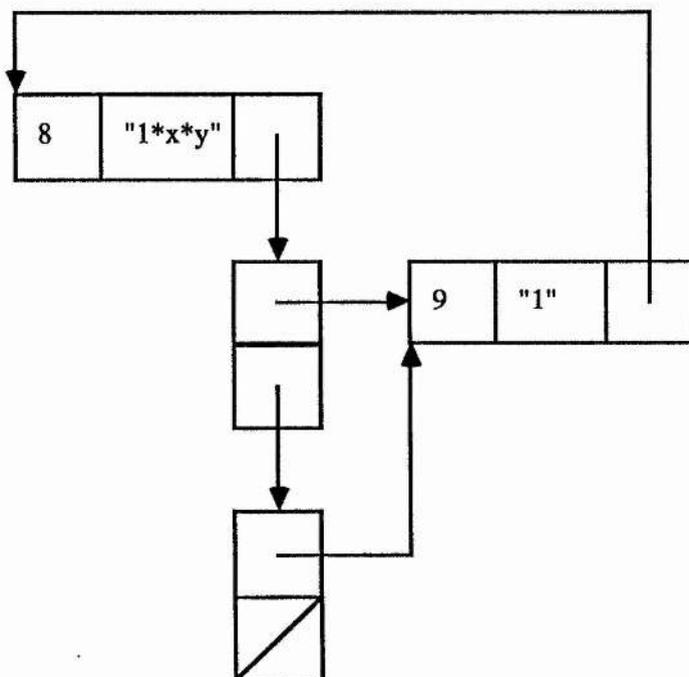


Figure 4.4.14 Existentially quantified structure representation

4.4.4 Parameterised types

The Napier88 type algebra allows abstraction over types in the form of parameterised type definitions. An example of a parameterised type definition and its use is given in Figure 4.4.15. Notice that *pair* is not itself a type, but a template from which types such as *intPair* may be constructed by specialisation. An equivalence check should never be performed over two parameterised definitions, but representation templates are built to allow the construction of representations for types constructed by specialisation, as described in Section 4.4.5.

```

type pair[ t ] is structure( fst, snd : t )
type intPair is pair[ int ]

```

Figure 4.4.15 A parameterised definition and use

The construction of parameterised representations is similar to the construction of quantified type representations. The interface procedures are called *mkParameter* and *mkParameterised*. Figure 4.4.16 shows how these may be used.

Type description:

type pair[t] is structure(fst , snd : t)

Possible module use:

```
let t = mkParameter( 1 )
let new = newStructure()
let ok1 = addField( new, "fst", t )
let ok2 = addField( new, "snd", t )
let pair = mkParameterised( new, append( t , nilList ) )
```

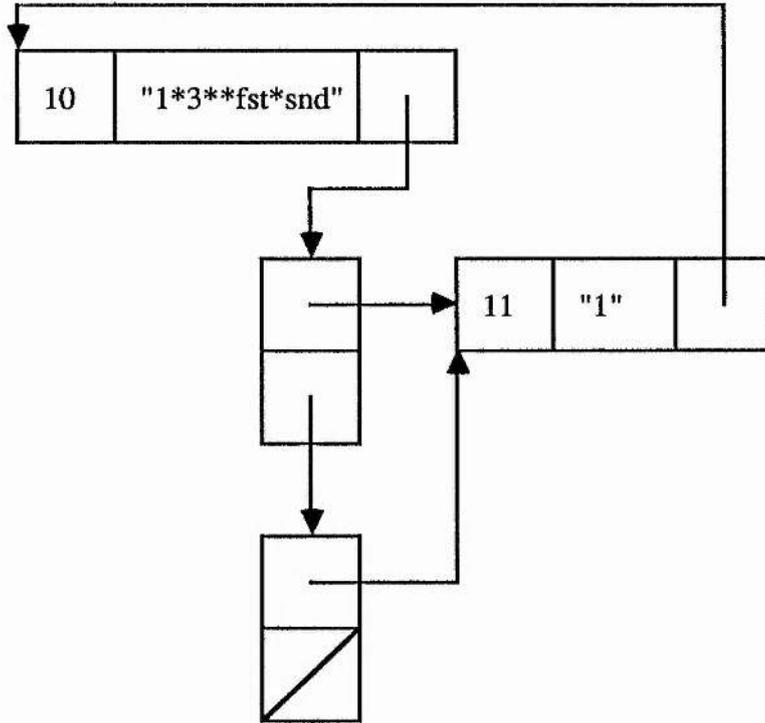


Figure 4.4.16 Parameterised type representation

The *specificInfo* field of a parameterised type contains the number of parameters, and the *label* and *specificInfo* required for instances of the type after specialisation.

4.4.5 Specialisation

Specialisation of parameterised type constructors is performed by the *substitute* procedure in the type module interface. This procedure takes as parameters the type representation to be specialised, and a list of type representations with which

to specialise it. This list should be the same length as the number of type parameters which the parameterised representation has associated with it.

The basic *substitute* algorithm is a graph copy of the parameterised type representation, as described in Section 4.3. However, instances of parameter representations are replaced by appropriate representations given in the specialisation list.

The copying starts by creating the *TYPE* node for the specialised type representation. The *label* and *specificInfo* field values can be derived from the *specificInfo* value of the parameterised type representation. The copying continues by traversing the *references* list, taking care to maintain the topology of the parameterised type representation.

Whenever a parameter type representation is encountered during the copy, its *references* field is tested for equality with the parameterised representation being specialised. If it is the same then, instead of the parameter being copied, its *specificInfo* is used as an index into the specialisation list, and this type representation is inserted into the new graph. Figure 4.4.17 shows the result of the specialisation shown in Figure 4.4.15.

Type description:

type intPair is pair[int]

Possible module use:

let intPair = substitute(pair, append(INT , nilList))

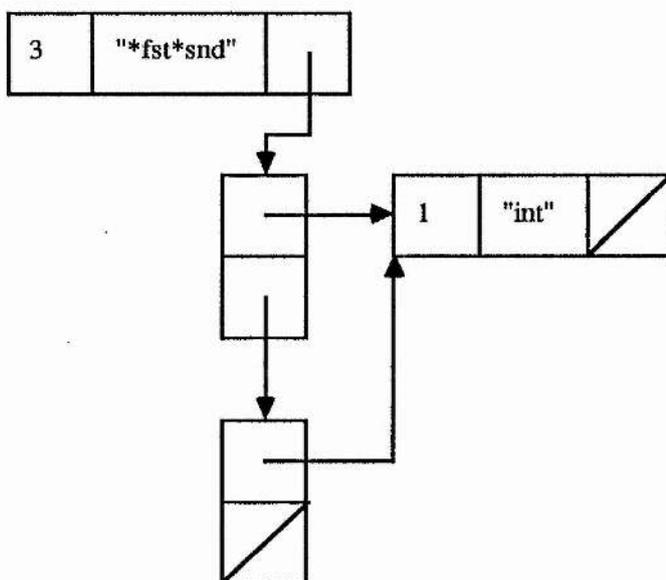


Figure 4.4.17 Specialisation of parameterised type

Specialisation is also necessary for both universally and existentially quantified types. This occurs when a universally quantified procedure is applied, or when an existentially quantified value is constructed. Figure 4.4.18 shows examples of these. In Napier88, the specialisation required in both instances is explicit, and uses the same syntax as the specialisation of a parameterised type. In languages where the type of the specialisation is inferred, the same type check must still be performed.

```

type universal is proc[ t ]( t, real → t )
type existential is abstype[ i ]( x : i ; y : real )

let useProc = proc( a : universal → int ) ; a[ int ]( 3, 2.2 )
let newAbs = existential[ int ]( 3, 2.2 )

```

Figure 4.4.18 Specialisation of quantified types

The substitute procedure described for the specialisation of parameterised representation also works, after minor adjustment, for the specialisation of quantified types. The adjustment required is for the quantified node to be initialised as the appropriate procedure or structure type, and then for the algorithm

to check for appropriate quantifier type representations instead of parameter type representations during the copy.

4.4.6 Recursive types

Many type algebras allow the recursive definition of types. For example, the type of a list of integers may be specified as in Figure 4.4.19.

```
rec type intList is structure( head : int ; tail : intList )
```

Figure 4.4.19 An integer list type

Mutually recursive definitions are also useful, as shown in Figure 4.4.20.

```
rec type intList is variant( cons : intNode ; tip : null )  
& intNode is structure( hd : int ; tl : intList )
```

Figure 4.4.20 An integer list type

Such types may be represented by graphs containing cycles. The equivalence algorithm described will function correctly over cyclic graphs. Here we are concerned with how such cyclic graphs may be constructed.

In general, it is not possible to resolve the meaning of any identifier within a set of mutually recursive type definitions until the end of the set has been reached. Representations of recursive types are constructed by the types module by the use of temporary representations for unresolved recursive types. These temporary representations may be used in the construction of other type representations. When enough information becomes available for the meaning of these type representations to be resolved, the fields within each representation are updated so that the representations becomes correct.

The procedures provided by the module are called *mkRecursive* and *replaceRecurSIONs*. *mkRecursive* is a procedure which takes a string parameter

and returns an unresolved type representation. *replaceRecursions* takes a lookup function from string to type. This procedure causes each of the recursive nodes already returned by *mkRecursive* to be internally modified so that they represent the correct types.

Whenever an identifier is encountered on the right hand side of a recursive declaration, *mkRecursive* is called with the identifier in string format as a parameter. The types module also keeps a list of every such type returned. When the end of the recursive definitions are reached the *replaceRecursions* procedure is called with an appropriate lookup function. This will normally be the syntax analyser's own lookup function which provides an index from identifiers to type representations. An example of the use of these procedures is shown in Figure 4.4.21, where *add_symbol* and *get_symbol* stand for abstractions over a compiler's symbol table.

Type description:

```
rec type  a is structure( x : b )
&        b is int
```

Possible module use:

```
let a = newStructure()
let ok1 = addField( a, "x", mkRecursive( "b" ) )
```

```
add_symbol( "a", a )
add_symbol( "b", INT )
```

```
replaceRecursions( get_symbol )
```

Figure 4.4.21 Use of a type name before its declaration

mkRecursive creates a type representation labelled as an unresolved recursive representation, whose *specificInfo* field is initialised to a string representation of the identifier used. Figure 4.4.22 shows the representations constructed by Figure 4.4.21 before the application of *replaceRecursions*. The *rec_list* is kept by the types module as a record of all unresolved representations.

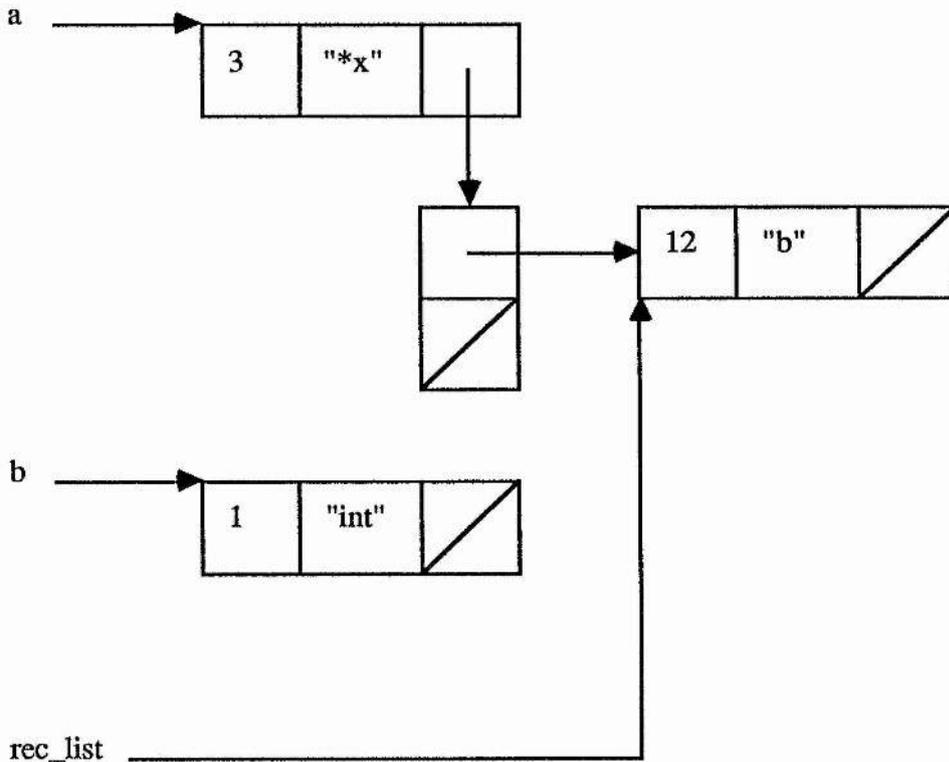


Figure 4.4.22 A recursive representation before resolution

During the execution of *replaceRecursions*, the *specificInfo* field of each node in the *rec_list* list is used in a call of the imported lookup function. The fields of the type representation returned are used to update all the fields, including the label, of the unresolved recursive representation. Thus all uses of an identifier in recursive definitions are updated in place so that they represent their meaning at the end of the set. Thus in the example of Figure 4.4.22, the fields of the unresolved representation inside the representation of *a* will be updated to the same values as those in the fields pointed to by *b*, as this is the representation that will be returned from the call of the lookup function.

One refinement of this basic algorithm is necessary where the type representation returned by the lookup function is itself a recursive type which has not yet been resolved. Such a situation could be caused by the example shown in Figure

4.4.23., in which the first attempt to resolve the recursive node *b* will result in the *get_symbol* function returning the recursive node *c*.

Type description:

```
rec type  a is b
&         b is c
&         c is int
```

Possible module use:

```
add_symbol( "a", mkRecursive( "b" ) )
add_symbol( "b", mkRecursive( "c" ) )
add_symbol( "c", INT )
```

```
replaceRecursions( get_symbol )
```

Figure 4.4.23 An unresolved representation may be returned by the lookup function

All that is required is for the algorithm, on encountering an unresolved representation, to move it to the end of the list and continue. The resolution is thus delayed until it is possible to perform.

Some check of finite progress should be made, to guard against definitions such as that of Figure 4.4.24. If no finite progress should occur, then a circular type definition has been made. This may or may not be an error, depending upon the language definition. The types module passes the information back to the syntax analyser, which deals with it as required.

```
rec type  a is b
&         b is a
```

Figure 4.4.24 A check for finite progress should be made

4.4.7 Parameterised recursive types

Some languages with both recursion and parameterisation in their type algebra do not allow the parameterisation of recursive types. In Quest [Car89] type operators are used to abstract over type definitions. Figure 4.4.25 shows how a recursive

operator *List* could be used to describe a generic list type. However, recursive operators are not allowed in Quest.

```
Let Rec List ( A :: TYPE ) :: TYPE =  
  Option  
    nil  
    cons with head : A tail : List( A ) end  
end ;
```

Figure 4.4.25 A (disallowed) recursive operator in Quest

The example shows a recursive operator which takes a type parameter *A* and returns the type of a list of *A*. The type of the result of *List* is a variant with labels *nil* and *cons*, where *cons* is a list node of the appropriate type. The definition of *cons* includes an application of the operator *List* itself, applied to the same parameter.

Although recursive operators are disallowed, an operator with the same apparent semantics as this may be declared in Quest using a non-recursive operator which returns a recursive type. This is because the recursive operator used as the *tail* of *cons* in fact specifies a fixed point, and this may be achieved without parameterisation as shown in Figure 4.4.26.

```
Let List ( A :: TYPE ) :: TYPE =  
  Rec( B :: TYPE )  
    Option  
      nil  
      cons with head : A tail : B end  
end ;
```

Figure 4.4.26 A non-recursive operator which returns a recursive type in Quest

Although Ponder [Fai88] does not have explicit type operators, the same restriction is imposed by a syntactic rule. Recursive type generators may be defined as shown in Figure 4.4.27, but all applications of *G* with *Body* must be to exactly the parameters $[T_1, \dots, T_n]$.

Rectype $G[T_1, \dots, T_n] = \text{Body}$

Figure 4.4.27 A recursive type generator in Ponder

This restriction means that all recursive parameterised definitions may be regarded as parameterised definitions over a μ -operator which binds to a type variable. This is shown in Figure 4.4.28, where F is a non-recursive type generator and G is a recursive type generator. Thus the syntactic restriction of Ponder gives precisely the same expressive power as disallowing recursive operators in Quest.

Rectype $G[X] = (\dots G[X] \dots F[X] \dots)$

\Rightarrow

Type $G[X] = \mu T. (\dots T \dots F[X] \dots)$

Figure 4.4.28 A recursive type generator in Ponder shown as a
fixpoint

The restriction is made in both cases to ensure that typechecking always terminates. However, it disallows a class of useful types for which a fully decidable algorithm is known [Sol78]. For example, it is not possible in these languages to express a generic type for a list of two alternating types. Napier88 allows all the class of types for which a decidable algorithm is known.

Figure 4.4.29. shows an example of a recursive parameterised type in Napier88. To deal with such types, the typechecker interface already described and its use may remain unchanged. However, the algorithms used by both *substitute* and *replaceRecursions* must be enhanced to allow for this form of interaction. This is because, with the algorithms described so far, *substitute* does not allow for recursive representations, and *replaceRecursions* does not allow for parameterised representations.

```

rec type list[ t ] is variant( cons : node[ t ] ; tip : null )
& node[ s ] is structure( hd : s ; tl : list[ s ] )

```

Figure 4.4.29 Recursive parameterisation

When a specialisation is specified on the right hand side of a set of recursive definitions, a call to *substitute* will occur with an unresolved recursive type representation. As the meaning of the recursive identifier is not known at this point, the algorithm is unable to perform any processing. However, it is known that the same node will eventually be processed by the *replaceRecursions* procedure, and so the substitution required may be postponed until this time. To mark the fact that the node is to be specialised after its resolution, its *references* field, which is otherwise unused, is set to point to the list of type representations with which the node is to be specialised.

Type description:

```

rec type a is b[ int ]
& b[ t ] is ...

```

Possible module use:

```

let b_use = mkRecursive( "b" )
let a = parameterise( b_use, append( INT, nilList ) )
... calls to construct b ...
replaceRecursions( get_symbol )

```

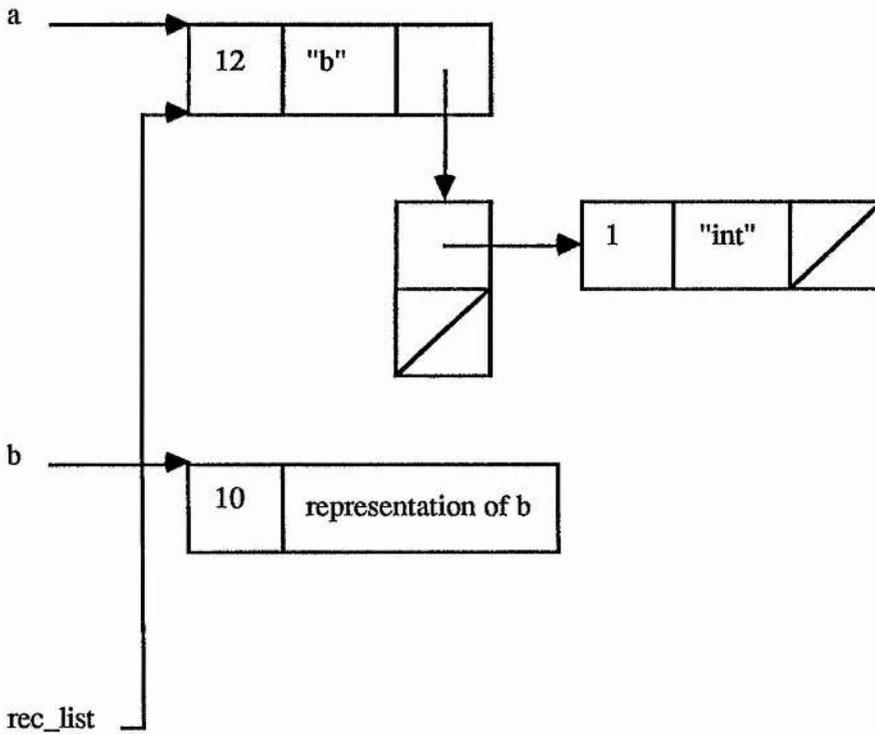


Figure 4.4.30 An unresolved representation

Figure 4.4.30 shows the unresolved parameterised representation, before the call of *replaceRecursions*.

During the execution of *replaceRecursions*, each unresolved recursive node is inspected to see whether its *references* field has a specialisation attached to it. If it has, then the lookup function should return a parameterised representation. To resolve the recursive use of a parameterised type, the *replaceRecursions* procedure makes a call to the *substitute* procedure, using the parameterised type returned by its lookup function and the parameters which are available from the *references* field of the recursive representation.

This however is not yet sufficient for the resolution of truly recursive parameterised types, as in Figure 4.4.31. In such cases, the *substitute* algorithm must also be adjusted to deal with unresolved recursive nodes, as they occur in the

graph which is to be copied. The type definition in Figure 4.4.31 will cause *replaceRecursions* to call *substitute* with the parameterised structure type representation built to represent *a*, which contains an unresolved recursive representation also of *a*.

```
rec type a[ t ] is structure( x : t ; y : a[ t ] )
```

Figure 4.4.31 More recursive parameterisation

This type is one of those allowed by languages such as Ponder, where parameterisation on the right hand side of a recursive definition may only act as a recursive fixpoint. When such a recursive node is encountered by the *substitute* algorithm, it terminates the local recursion and leaves a pointer to the unresolved node itself in the copy of the graph. This has the effect that, after the rest of the substitution has been effected and the fields of the recursive node are overwritten with the result, then a cycle which represents the fixpoint is left in the resolved recursive type representation.

After the type has been parsed, but before the call of *replaceRecursions*, the type representation in Figure 4.4.32 has been built for *a*.

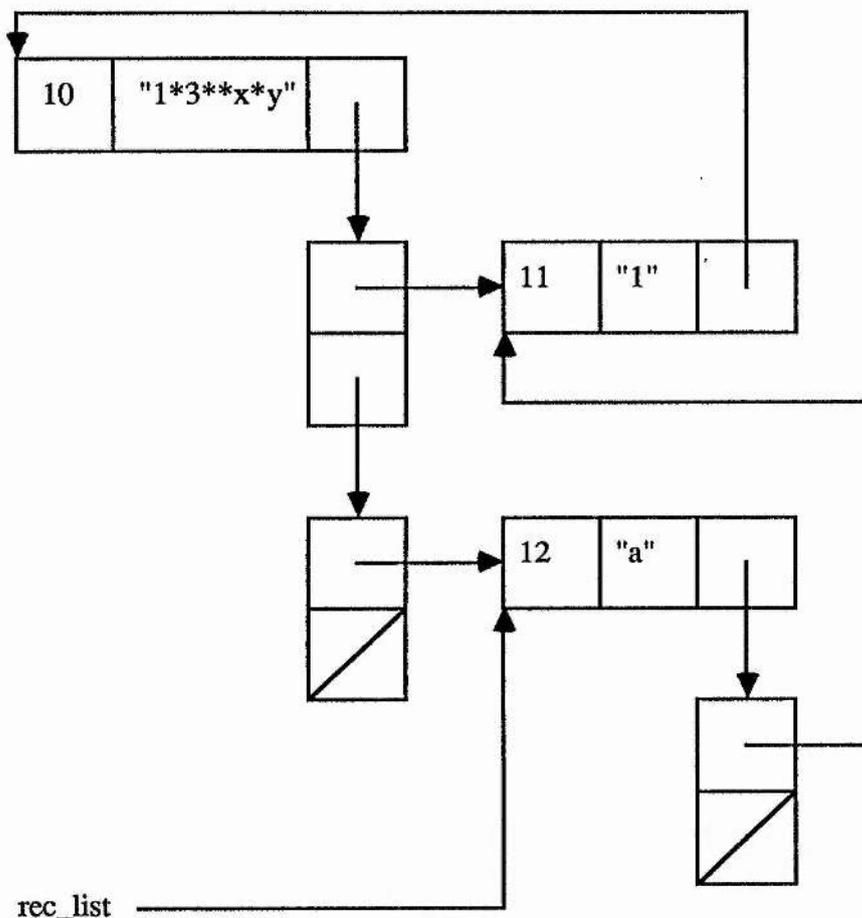


Figure 4.4.32 A representation containing an unresolved recursive node

replaceRecursions is now called to resolve the unresolved representation of *a* in the above graph. The lookup procedure called by *replaceRecursions* will yield a pointer to the whole parameterised representation, and this will cause a call of *substitute*. The recursive node representing *a* will itself be reached during the copying of the graph. This node is not copied, but a pointer to it is placed in the new representation. This leads to the situation, immediately after the internal call of *substitute*, shown in Figure 4.4.33.

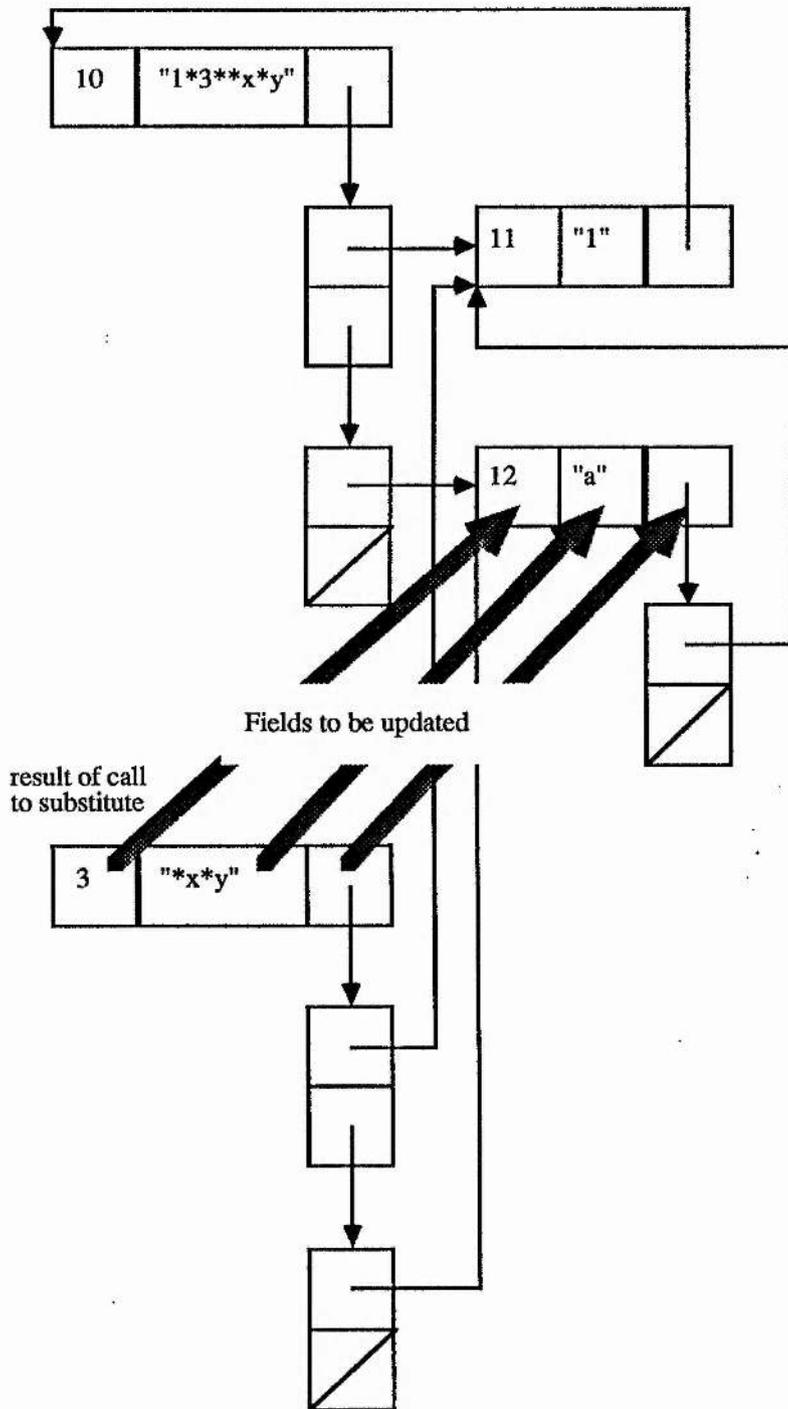


Figure 4.4.33 During recursive resolution

The next stage in the algorithm is to overwrite the fields of the recursive node with those obtained by the call of *substitute*. The result of this is shown, after garbage collection and rearrangement, in Figure 4.4.34.

type in the specialisation list is either the corresponding formal type parameter, or else is a type which contains no formal type parameters. Examples of such types are shown in Figure 4.4.35.

```

type a[ x, y, z ] is      ...   a[ x, y, int ]      ...
type a[ x, y, z ] is      ...   a[ structure( c : int ), y, z ]  ...
type a[ x, y, z ] is      ...   a[ x, y, z ]

```

Figure 4.4.35 "Right hand side" fixpoints

During the execution of *substitute* it is also possible for an unresolved recursive node which does not represent a fixpoint to be encountered. This will occur, for example, whenever mutually recursive parameterised types are declared, as in Figure 4.4.36. When such a node is encountered it is copied as usual, along with the specialisation list in its *references* field. Parameter substitution also occurs as normal during the copying of the specialisation. The new recursive node is then added to the recursion list, as no further processing may yet be performed.

```

rec type list[ t ] is variant( cons : node[ t ] ; tip : null )
&          node[ s ] is structure( hd : s ; tl : list[ s ] )

```

Figure 4.4.36 Mutually recursive parameterised types

In examples such as Figure 4.4.36, the substitution within specialisation lists leads to the creation of fixed points as the representation unfolds. In this example, the initial representation constructed for *list* contains an unresolved recursive representation of *node*, specialised with the representation of *t*. When *replaceRecursions* attempts to resolve this, during the call of *substitute* an unresolved recursive representation of *list*, specialised by *s*, is encountered. This is copied, but with the representation of *t* in place of the representation of *s*, and the new node is added to the *rec_list*. When this new node comes to be resolved, however, it is discovered to be a fixpoint and the recursive knot may be tied.

The algorithm as described so far does not terminate with all parameterised recursive definitions. In general, allowing any form of recursive parameterised types may lead to types whose equivalence is not known to be decidable. However, the class of types allowed so far may be extended.

The class of types for which no known decidable equivalence algorithm exists are those where the specialisation involves constructed types containing parameters. During the test for suitability described above, the types module detects such type descriptions. These are simply handled in the Napier88 system by disallowing programs which contain such descriptions. This restriction is implemented by a fully decidable algorithm, which ensures that static type analysis will always terminate. Figure 4.4.37 shows some examples of allowed and disallowed type definitions.

```
Disallowed:  
rec type array[ t ] is variant( simple : t ; complex : array[ *t ] )  
rec type oddList[ t ] is structure( hd : t ; tail : oddList[ structure( hd : t ) ] )  
  
Allowed:  
rec type altList[ s, t ] is structure( hd : s ; tl : altList[ t, s ] )  
rec type alt3List[ r, s, t ] is structure( hd : r ; tl : alt3List[ t, s, r ] )
```

Figure 4.4.37 Examples of allowed and disallowed type definitions

4.4.8 Recursive existentially quantified types

There is one further subtlety which must be dealt with in the substitution algorithm, which is a consequence of the combination of recursive type definitions and existential type. It is possible to define an existential type recursively in such a way that it refers to values of its own type. An example of such a type, along with its representation, is given in Figure 4.4.38.

Each element of the *heterogeneousList* may be a value of a different structure type which matches the signature. When a value of such a type is created, the proposed

concrete structure type is checked for equivalence with the appropriate specialisation of the existential type. Recursive instances, however, should not be specialised: for example, the *tail* field in Figure 4.4.38 should continue to be a representation of a *heterogeneousList*, and not its specialised form.

```

rec type heterogeneousList is abstype[ i ]
(
  head : structure( value : i ; print : proc( i ) ; change : proc( i → i ) );
  tail : heterogeneousList
)

```

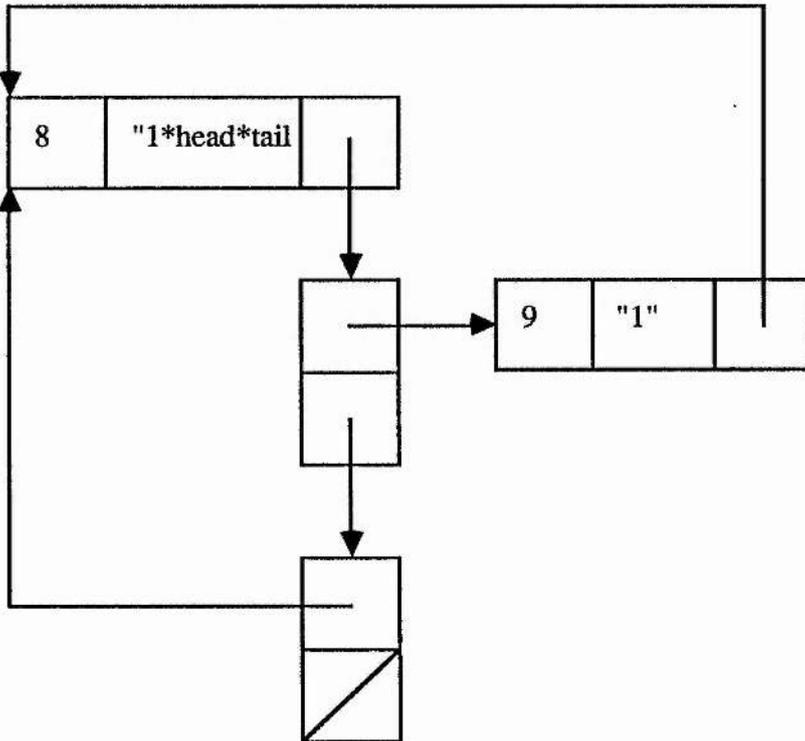


Figure 4.4.38 A heterogeneous list type and its representation

To achieve this, a further test is required in the substitution algorithm to ensure that the representation of the existentially quantified type being specialised is not re-traversed during the algorithm.

4.4.9 Free quantifiers

4.4.9.1 Static checking

In a language with quantified type values, structural type equivalence is not always a sufficient test for type safety. Consider the procedures declared in Figure 4.4.39.

```
let a = proc[ t ]( x : t )
begin
  let b = proc[ t ]( y : t )
  begin
    let new := y
    new := x          /* This must fail statically */
  end
end
```

Figure 4.4.39 Type equivalent procedures with non-equivalent quantifiers

In this example, the procedures declared as *a* and *b* are of equivalent types. The types of their quantifiers are not equivalent. However, for a recursive equivalence algorithm to succeed over procedure types such as these, their quantifiers must appear to be equivalent. The rule is that quantifiers should be regarded as equivalent in a context in which they are bound to their quantified type, but free quantifiers may only be equivalent if they derive from the same declaration.

To perform this test at each stage of the equivalence algorithm would add complexity and significant cost, and so a trick is used with the binding of the type representations for the quantifiers. When quantifiers are used in a free context, their representations are given a different label value. If two representations derive from the same declaration, then they will have the same identity. Identity of representations is checked for at the start of the equivalence algorithm, and so an equivalence test succeeds immediately. The only change that is then required to the equivalence test is to fail if a representation with this label is encountered.

The way the interface is defined to deal with universally quantified procedures makes this scheme easy to effect. The new quantifier representations are created at the start of the procedure with the free quantifier label, and when they are bound after the procedure body has been parsed their labels are updated to the bound quantifier value. Any type check on quantifiers inside the body of the procedure is performed on the free quantifier representation. This causes a type error to be reported in examples such as the one above. As soon as the end of the procedure body is reached, the quantifiers of the procedure become bound to the type of the procedure. This means that the recursive structural equivalence algorithm will perform correctly on the type of the procedure after it has been constructed.

With abstract data types, however, the syntax of the language does not help and it is necessary to provide extra interface procedures to achieve the different equivalence semantics. This is because name equivalence rules must be used during the use of an existentially quantified value, rather than during its creation. An example of this is shown in Figure 4.4.40.

The procedures provided are called *bindAbs* and *absFieldType*. *bindAbs* is used on the entry of a use clause by the syntax analyser. Its parameters are an abstract type representation and a list of new witness types. On entry to an abstract use clause, the syntax analyser must create a new set of unbound witness type representations, one for each witness type of the abstract type. These are put in a list which is used as the second parameter to a call of *bindAbs*. The result of *bindAbs* is a new abstract type representation, which contains both the original representation of the abstract type and a new copy made by substituting the new witnesses for the original ones.

Napier88 code:

```
type trivial is abstype[ i ]( x : i )

use trivial[ int ]( 3 ) as a in
use trivial[ string ]( "hello" ) as b in
begin
  let ok := a
  ok := b                /* This assignment should be allowed

  let notOk := a( x )
  notOk := b( x )       /* This must fail statically
end
```

Possible module use:

```
let i = mkWitness( 1 )
let temp = newStructure()
addfield( temp, "x", i )
let trivial = mkAbstract( temp, append( i, nilList ) )

let a = bindAbs( trivial, append( mkWitness( 1 ), nilList ) )
let b = bindAbs( trivial, append( mkWitness( 1 ), nilList ) )

let succeed = equalType( a, b )
let fail = equalType( absFieldType( a, "x" ), absFieldType( b, "x" ) )
```

Figure 4.4.40 Free existential quantifiers

absFieldType is parameterised by an abstract type representation and a string, and returns the type representation of the field associated with the string. The representation it returns is selected from the copied body of the abstract type, which contains the newly created unbound witness type representations. Therefore any witness type representations in the result will obey name equivalence semantics.

When the equivalence function is called on abstract data type representations, it checks for equivalence only down the pointer to the originally constructed type, and not the copy made during *bindAbs*, thus making sure that appropriate values are still type compatible.

Figure 4.4.40 shows the use of these procedures when free existential quantifiers are in scope. In this example the assignment to the variable *ok* is correct, as the type of this location contains no free quantifiers. However, the type of the location

denoted by *notOk* is of a free quantifier type, and so the assignment of a different free quantifier type to this location should fail statically. The two appropriate calls to *equalType* are shown, of which the first returns true and the second returns false.

4.4.9.2 Dynamic checking

No distinction has been made up to this point between static and dynamic type checking. For all the types shown, the time of checking is immaterial, as the same equivalence algorithm may be applied either during compilation or execution.

In general, dynamic type checking is required only when values are projected from unions. One reason for performing such a projection is when binding dynamically to values in the persistent store [ABC83], as shown in Chapter 2. As long as the types of both injection and projection are known statically, there need be no distinction between static and dynamic checking.

However, a complication occurs when a value is injected into a union type in a context where its type is abstracted over. For example, consider the program in Figure 4.4.41.

```
let injectAsA = proc[ t ]( e : env ; x : t )
  in e let A = x
```

Figure 4.4.41 Injection of a value with an abstracted type

A number of different semantic models are possible for this injection. One possibility is that the value should be injected with its concrete type. This would involve passing type representations around at execution time, but would be quite straightforward to implement. This is the semantics implied by universal quantification in the model proposed by Girard and Reynolds [Gir72, Rey74], as a

textual substitution of the type parameter takes place when a procedure is specialised.

The program in Figure 4.4.42 would then execute successfully, and print the successfully projected value.

```
let e = environment()
injectAsA[ int ]( e , 3 )
use e with A : int in print( A )
```

Figure 4.4.42 Injection with the concrete type

This is because the application of the procedure *injectAsA* takes place in two phases, the first being an application to the type parameter. This means that the procedure applied in Figure 4.4.42 is semantically identical to that shown in Figure 4.4.43.

```
let injectAsA = proc( e : env ; x : int )
  in e let A = x
```

Figure 4.4.43 Specialisation by Girard-Reynolds

However, this model also has the consequence that the concrete type of a quantified parameter may be discovered from inside the procedure body. An example of this is shown in Figure 4.4.44.

```
let discover = proc[ t ]( x : t )
begin
  let e = environment()
  in e let A = x
  use e with A : int in print( "It's an integer" )
  use e with A : string in print( "It's a string" )
  ...
end
```

Figure 4.4.44 Discovering a quantified type

This has a number of implications. The caller of a universally quantified procedure may not be sure that the parameters provided will not be manipulated within the procedure, thus losing some aspects of the protection discussed in Chapter 3. A different model must be employed for values of an existential quantifier type, complicating the behaviour of the programming language. Perhaps the most serious implication is that the procedure shown in Figure 4.4.41 is not a polymorphic procedure, as it does not abstract over the type of its operand. Therefore the Girard-Reynolds model of universal quantification does not model polymorphism in these circumstances, as it allows procedures to behave differently according to the type of their quantified parameters.

Another possibility, as defined in Quest [Car89], is to disallow the injection of values whose type is abstracted over. Thus the procedure in Figure 4.4.41 is disallowed during static program analysis. This is over-restrictive, however, as it means that a sequence of code which performs both an injection and a projection may not be abstracted over. An example of this is shown in Figure 4.4.45.

```

let store = proc[ t ]( x : t → proc( → t ) )
begin
  let e = environment()
  in e let A = x

  proc( → t )
    use e with A : t in A
end

```

Figure 4.4.45 Abstracting over injection and projection

The semantics of Napier88 lie between these two extremes, in that the procedure in Figure 4.4.44 will not discover the type of the operand, but the procedure in Figure 4.4.45 will execute correctly. When a polymorphic procedure contains an injection of a value whose type is abstracted over, a new type representation is created on each invocation of the procedure. This is treated by the type equivalence algorithm in the same way as the unbound quantifier representations

described earlier. Thus for a projection to succeed, the injection must have occurred during the same dynamic invocation of the polymorphic procedure. This scheme retains type safety without allowing the type information to escape.

This scheme is used for any type which contains a free quantifier. A new label is used for this representation. Although a free quantifier representation would give the correct equivalence semantics, more information is kept to allow a more meaningful error to be presented to the user on failure.

4.4.10 Conclusions

It has been shown how type representations may be constructed so that the equivalence algorithm defined in Section 4.3 reflects type equivalence in a programming language. The type constructors modelled are vectors, structures, variants, procedures, universally quantified procedures, and existentially quantified structures.

The complications of manipulating these representations according to a type algebra which contains both parameterisation and recursion have also been exposed. Of particular interest are the problems caused by the combination of parameterisation and recursion. This combination effectively gives a programmer the ability to write general functions over types. In most type systems these must be resolved statically, and the range of types whose equivalence may be decided is still an active research area. The algorithms shown are fully decidable, but allow only a restricted class of type definition to succeed.

Also described are the problems of typechecking with values whose types are abstracted over, and in particular the combination of these with union injection and projection. This is another active research area, and only a brief discussion has been given here.

4.5 Conclusions

Type systems in persistent programming languages are assuming an increasingly important role, and the traditional database schema is now commonly regarded as a type. This leads to a requirement for the efficient manipulation of types in a persistent system to allow provision of the facilities traditionally found in DBMS for schema editing, use and evolution. One aspect of schema manipulation, that of type equivalence checking, has been investigated here. The use of the common methods of name equivalence and structural equivalence have been studied in detail.

We have shown that while name equivalence schemes are easier to implement and are more efficient they still have to use structural checks to provide important facilities such as schema merging. On the other hand structural equivalence, generally more flexible and less efficient, can often achieve the same performance as name equivalence.

Given that the efficiency of name equivalence is adequate for our needs, the improvement of the performance of structural equivalence checking becomes important. A method of performing such checking has been described, and it has been shown that there is a balance between constructing efficient representations of types in terms of store and the speed of the equivalence algorithm.

We have shown how types may be represented by strings and by graphs, highlighting the difficulties in their construction and use. Some preliminary measurements are presented and our main conclusion is that where the type schema is large and involves the sharing of types, the graph representation will be much more efficient in terms of space. It may however be slower in terms of speed of checking depending on its use within the persistent store.

Finally, a method of constructing graphs for type representations has been shown in detail. While this is straightforward for most types, there are some difficulties with type algebras which include both parameterisation and recursion. Some outstanding difficulties with the dynamic typechecking of values whose type is abstracted over have also been outlined.

5 On the implementation of polymorphism

5.1 Introduction

Given that a polymorphic programming language may be correctly type-checked, there remains the problem of implementing a suitable run-time system. Compilers traditionally use type information for two purposes: the first is to ensure static safety constraints within a program, and the second is to generate appropriate code where this depends on the type of operands.

Values of different types frequently have different machine-level representations, for reasons of efficiency. The difficulty of implementing polymorphic systems stems from the fact that, as the type may be abstracted over, the representation of a value is not always known statically. This causes problems with all kinds of polymorphism. With parametric polymorphism, operations which may manipulate a value of any type are defined over values whose type, and therefore representation, is not known. With inclusion polymorphism operations such as record dereference may be defined over values whose type is partially abstracted over. Representation-dependent information, such as record offsets, may not always be calculated statically. With ad-hoc polymorphism there is a requirement for operations with different semantic meanings to be executed according to the type of the operands, which may be abstracted over in some systems.

A number of solutions to these problems have been proposed and successfully implemented [Mil83, Lis81, DD79, Mat85, BMS80, Fai82, Tur87, ACO85, SZ86, GR83, Hud90]. Many of the solutions however have been in non-persistent languages, and for various reasons are not well suited to the implementation of polymorphism in a persistent programming system. This chapter proposes two new mechanisms, for the implementation of parametric and

inclusion polymorphism. Both of these are suited for use in a persistent programming system.

5.1.1 Compilation, architectures, and polymorphism

For the purpose of discussion, a simplified model of programming language implementation is assumed. The terms compilation and architecture will be used throughout. The assumed model is of a program, which consists of text, being used as input to another program, a compiler. The compiler transforms the text into a sequence of instructions which may be executed upon a machine architecture. This architecture is assumed to use an evaluation stack and a heap storage mechanism.

Implementation techniques for polymorphism may be broadly divided into two main categories, according to whether or not the architecture supports instructions which perform differently according to the type of their operands. Architectures which do support such instructions will be referred to as tagged architectures, and those which do not as untagged architectures.

These two categories may be further subdivided. Untagged architectures may be used to support polymorphism in two ways, defined as uniform and textual implementations [MDC90]. Uniform implementations are those where a compiler can produce an instruction sequence without using any knowledge of the types being manipulated. This is only possible if all language operations which are legal on values of all types perform the same series of instructions at the architectural level. Textual implementation refers to any implementation of polymorphism which is neither tagged nor uniform, and must therefore rely upon executing different instruction sequences in different calls of a polymorphic procedure.

Tagged implementations may also be divided into two categories, according to whether all values are tagged or tagging is restricted to values whose type is unknown statically.

The given classification of implementation techniques is illustrated in Figure 5.1.1, along with examples of programming languages or architectures which use each method. The classification is somewhat oversimplified, as a combination of these techniques may be used within a single system. This Figure will be discussed in detail in the conclusions of this chapter.

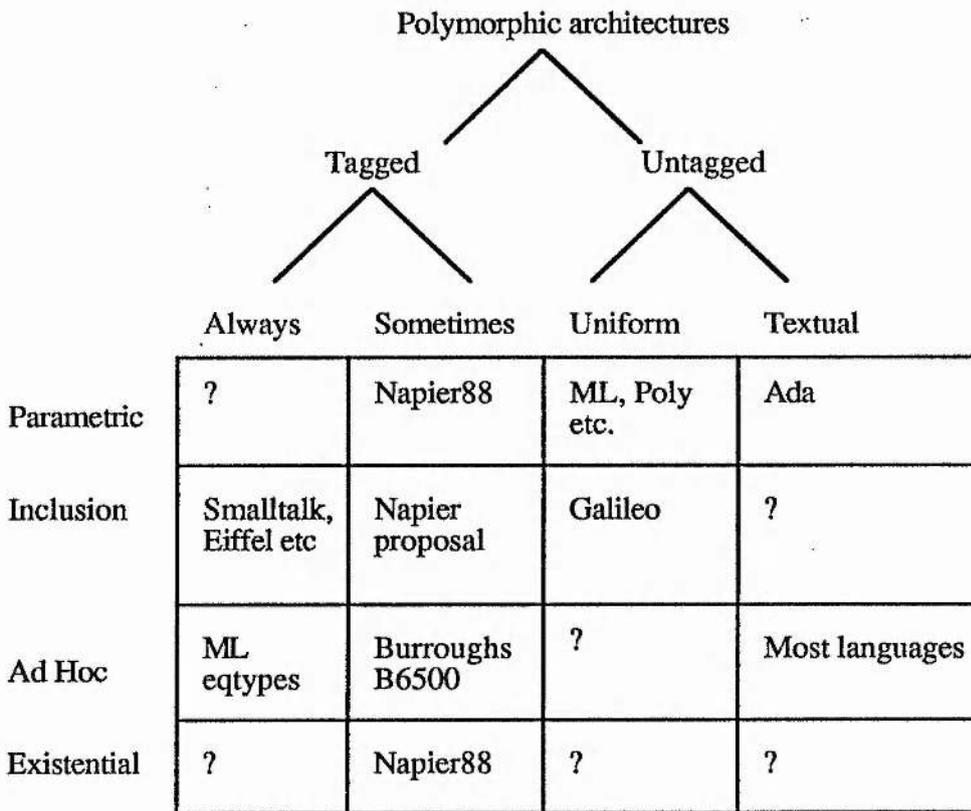


Figure 5.1.1 A Classification of polymorphic implementation

Notice that this classification is entirely of implementation, and bears no relation to the semantics of a language. However, the same semantic model may indicate different implementation techniques depending partly upon the way it is used

within a language and partly upon the other classes of computation expressible within a language. The suitability of different implementation techniques for some different styles of programming language will also be discussed in the conclusions.

The essence of this chapter is the description of two new tagged architecture mechanisms, one of which supports parametric polymorphism [MDC90], and the other of which supports inclusion polymorphism [CDM89]. The first of these is fully implemented, and is used to support the language Napier88. The second is currently under construction. Both mechanisms restrict the use of tagging to those cases where it is strictly necessary, thus allowing the compiler to produce efficient code for programs which do not use these features.

Different architectural mechanisms are described for the different language models of parametric and inclusion polymorphism. There is no reason, however, why both mechanisms may not be supported by a single architecture. Parametric and inclusion polymorphism are discussed in separate sections, each section consisting of a review of other implementation methods followed by the new method.

5.2 Parametric polymorphism

5.2.1 Machine-level type systems

Most machines, both hardware and software, have their own type systems. These are normally low-level and not checked for safety. For example, many machines represent integers as a single word value and floating point numbers as a double word value. For simplicity, most machine designers choose a small number of object formats, perhaps in the order of ten.

The use of different formats necessitates the use of different instructions for operations applicable to values of all types, such as binding and parameter passing.

Thus program segments such as

```
a := 7
```

and

```
a := 7.0
```

must be represented by different instruction sequences. The different space demands of integers and floating point numbers mean that different load literal and assignment instructions must be generated. The stack address calculation for the rest of the program will also be affected differently.

The difficulty with different machine formats for different classes of types is that a system with type abstraction does not always have enough information to generate different instructions for the different formats. The problem is that any operation defined over all data types should also be defined over values of a type which is not known statically. For example, Figure 5.2.1 shows a Napier88 procedure which updates the value of a location whose type is abstracted over.

```
let problem = proc[ t ]( x , y : t )  
begin  
    x := y  
end
```

Figure 5.2.1 Assignment to a location whose type is unknown statically

This procedure is universally quantified over a type denoted by t . The procedure takes two parameters of this type, which at the time of procedure call are used to initialise the locations bound to the identifiers x and y . The procedure then updates the location bound to x with the value contained by y . As these are statically known to be the same type, the assignment is valid.

Although x and y are known to be the same type, it is not known what size their representations are, or whether they are pointers or scalars. This means that the correct assignment instruction is not known: even the stack addresses of x and y may not be statically determined. The same problem occurs with existential data types, where a type signature may also be abstracted over.

Different machine formats may also cause difficulty with some data structures, as knowledge of the types contained by them may not be statically available within polymorphic code. For example, the procedure shown in Figure 5.2.2 is again universally quantified over t , but this time the single parameter is an array of values of this type, and the result is a single value of this type. The procedure returns the value of the second element of the array. However, the offset and size of the required element cannot be statically determined, as they depend upon the type of the array elements.

```
let second = proc [ t ]( x : array[ t ] → t ) ; x( 2 )
```

Figure 5.2.2 A polymorphic addressing problem

The problems of implementing parametric polymorphism have been successfully addressed by the implementors of the Ada programming environment [Ich83], the Functional Abstract Machine [Car83], designed primarily to implement ML, and the Persistent Abstract Machine [BCC88, CBC89], designed primarily to implement Napier88. The different requirements of these languages have led to three quite different solutions. The Ada solution is textual, the ML/FAM solution is uniform, and the Napier88/PAM solution is partly tagged. Each solution is examined in turn, along with reasons for its adoption.

5.2.2 Textual implementation

Textual polymorphism is defined as an implementation of polymorphism where different instruction sequences are produced by the compiler for different calls of the same polymorphic procedure. This method is commonly used for the implementation of ad hoc polymorphism.

If data types do have different representations, then polymorphic procedures must execute different code according to the types with which they are specialised. One method of implementing such a scheme is for the compiler to translate polymorphic procedures into an intermediate form. When a specialisation of the polymorphic procedure is compiled, the intermediate form is used to generate code specific to the specialisation type. This generated code may be in the same language, in which case the compiler will then compile the generated code. This technique is known as type reflection [SFS90]. Alternatively, the intermediate form can be used to generate specific machine code directly.

The major advantage of textual polymorphism is that the compiler can always produce optimum code, and an architecture which supports different data formats for any reason does not cause a problem. The major disadvantage is that the amount of code produced by the compiler may be large. A number of cases in which this can become severe are now discussed.

5.2.2.1 Multiple parameters

It is not necessary to compile a different version of the procedure body for every call of a polymorphic procedure. If the type of two calls share the same representation, then the code may be shared between these calls. In general, an upper bound of the number of different versions required is the number of different machine formats.

If a polymorphic procedure abstracts over more than a single type, however, this upper bound becomes exponentially greater with the number of type parameters. For example, supposing that an architecture supports ten different machine formats. Then a procedure of type

proc[t](t → t)

will have an upper bound of ten different bodies compiled for it. However, for a procedure of type

proc[s,t](s,t → t)

the upper bound is one hundred, and for a procedure of type

proc[r,s,t](r,s,t → t)

the upper bound is one thousand. In general, the upper bound is n^m , where n is the number of different machine formats and m is the number of types abstracted over. It should be remembered, however, that these examples of upper bounds are not necessarily representative of the amount of compiled code required. If the parameter types of all calls to such a procedure are known statically, then the number of calls of a polymorphic procedure within a program is a more realistic upper bound to the number of code versions required.

5.2.2.2 Calls within calls

In general, the types of all the parameters at a procedure call are not always known statically. If the body of a polymorphic procedure itself contains a call to another polymorphic procedure then the amount of code necessary is more than one version for each static instance of a procedure call. For each instance of a procedure call where the types are known, specialised procedure bodies for all polymorphic procedures which are reachable through a calling chain must be produced. This has a multiplicative effect of the production of specialised code.

5.2.2.3 First class procedures

In a language with first class procedure values the problems may become much greater. This is because it may not be possible to tell statically from a procedure call which instance of a procedure is being called. Under these circumstances it is not possible for the correct specialisation to be performed statically by the compiler. For example in the program in Figure 5.2.3, the compiler can not statically provide a procedure body for the call of *either*, since it does not know which version will be called when the program is executed.

```
let either = if <condition>
    then proc[ t ]( a , b : t → t ) ; a
    else proc[ t ]( a , b : t → t ) ; b

either[ int ]( 2,3 )
```

Figure 5.2.3 Calling a non-manifest polymorphic procedure

There are two possible solutions: either the compiler is called dynamically, or all versions of such procedures which could possibly be required are compiled statically, with the compiler inserting a dynamic lookup to access the correct version. This lookup could be keyed using some kind of tag for the appropriate machine representation. The first of these, although slow, limits the production of code to only that which is used. The second demands sophisticated static analysis and, in a complex system, will produce volumes of code which approach the upper bounds shown above.

5.2.2.4 Persistence

Even this last scheme is not suitable for use in a persistent system with first class procedures. The use of a first class polymorphic procedure in such systems may extend beyond a single compilation unit, and a compiler is then unable to perform any static analysis of its use. Such a system therefore degenerates into a worst case of the above, and either all calls to persistent polymorphic procedures must

dynamically invoke the compiler, or every possible procedure body must be available in the persistent store.

5.2.2.5 Conclusions

Textual implementations of polymorphism produce optimum instruction sequences and do not impose restrictions on the design of an architecture. However, the technique may introduce serious problems with the amount of code produced, or the number of dynamic calls to the compiler.

Despite its apparently serious drawbacks, the technique has been used successfully to support generic types in the programming language Ada. Ada, however, does not support either persistence or first class polymorphic values. For programming languages with such features this implementation technique is unsuitable as a general implementation mechanism, although it may still be useful as an optimisation technique.

5.2.3 Uniform implementation

A uniform implementation of polymorphism is one in which the polymorphism is exhibited both by the compiler and by the architecture: any operation which is legal on values of all types always generates the same code and performs the same series of instructions independently of its particular use. This can only be achieved by using a single representation for all data types, both on the run-time stack and within the language's data structures.

This has two consequences: all manipulated values must be represented in a fixed size, and pointers and non-pointers must be manipulated identically both on a run-time stack and within any heap object which represents a data structure.

5.2.3.1 Fixed size value representations

Having all data values of a fixed size loses efficiency for applications which do not use polymorphism. Different data types have different space requirements for their representations: for example, real numbers usually use twice, or sometimes four times, the space of integers. If the representations of all data types are restricted to the same size, then either they must all be the size of the largest, with the consequent wastage in space, or else some values must be wrapped in heap objects, and operations specific to their type must go through an otherwise unnecessary level of indirection.

5.2.3.2 Finding pointers

Many architectures require to distinguish between object identifiers and scalar values at the store level. This allows activities such as garbage collection, clustering, and pointer swizzling [BC85] to be performed without a high-level knowledge of a language's run-time system. In general, pointers may be detected either by tagging or by restricting them to certain locations which may be discovered from store conventions. One problem with pointer tagging is that it is hard to implement efficiently in software, and few machines provide hardware support. Even if hardware support is available, keeping pointers in known locations removes the need for searching and may therefore increase the efficiency of garbage collection and pointer swizzling.

In some systems, pointers within heap objects are always distinguished by being restricted to a contiguous area with boundaries somehow delimited within the object. Pointer and non-pointer values may be distinguished during program evaluation either by keeping a bitmap of the stack or by using two separate stacks, one for pointers and one for non-pointers [BMM80, PS85] Both of these cases,

however, require the generation of different machine instructions for operations such as loading and assignment, which are defined over all values.

There are two possibilities which may be used to allow the architecture to find pointers in a uniform system. The first of these is by marking all data values as either pointer or non-pointer, which may be achieved by restricting the ranges of possible values. This is not efficient to implement on a conventional architecture. Not only are the number of possible values in each category reduced, but to guarantee strong typing all instructions which manipulate either category of value must be checked in software for overflow from the restricted range.

The second possibility is for non-pointer values to be encapsulated in heap objects. The machine can then find pointer values, as all values are represented both on the stack and within data structures as pointers. Heap objects must also contain headers which allows the machine to dynamically determine which contained values are pointers, so that arbitrary pointer following may be achieved below the language level. This scheme means that in general all operations on non-pointer values must use an extra level of indirection, although certain optimisations may be made.

Notice that the ability to determine some aspect of a data value's meaning is only necessary for utilities which operate below the level of a programming language. Instructions generated by a compiler never need to test a value, as type-dependent operations may only be generated where sufficient type information is available statically.

5.2.3.3 Conclusions

The main advantage of the uniform polymorphism scheme is that the model of polymorphism is simple and elegant. This not only makes compilers easy to write,

but may also increase a user's understanding of a language by use of an abstract architecture as a reference model.

The main disadvantage of this scheme is the requirement for unnecessary levels of indirection for at least some data within the system. If the architecture is to be simulated on a conventional machine, then either this requirement becomes greater or software tagging must be used. The cost of the extra indirection is twofold: operations over data are more expensive, and creation of new data involves extra heap object creation. All data values in the system must pay this price, whether or not polymorphism is used.

The technique of uniform polymorphism has been successfully used to implement ML and many other functional programming languages. It is perhaps less suitable for the implementation of a persistent language due to the extra levels of indirection which are required. As these are required for all data within the system, and not just those whose type is abstracted, all access to the persistent store would potentially involve extra indirections. In a persistent system, object faulting is a major execution cost [Bai89]. This is in contrast to functional programming systems, where most data is transient and implementation is normally geared to the production and reclamation of large numbers of temporary heap objects [Car83].

5.2.4 Semi-uniform, partly tagged implementation

The uniform implementation of polymorphism as described above relies upon an architecture being able to perform the same instructions for the same logical operation on a value of any type. This prevents the use of different formats within an architecture. We now show a semi-uniform implementation, which allows a machine to use different formats to represent different data types. The essence of the solution is to provide a further format which is used for values whose type is not known statically. The compiler may therefore generate uniform code using this

format when values of abstracted types are manipulated. On entry to a polymorphic context, values are converted to this format, and reconverted on exit.

5.2.4.1 Semi-uniform implementation

The semi-uniform implementation requires a machine to support a format suitable for the representation of values whose type is abstracted over. Values of such a type must be converted into this format on procedure entry, and reconverted to their original format on procedure exit.

The requirement for a polymorphic format is that any other format may be mapped to and from it. That is, the information space is large enough to hold all the information in any other format. It may be assumed that the operations which map values to and from the polymorphic form will be applied consistently, and so the polymorphic form itself does not need to hold information related to the type of its contained value.

Any data format which may be used for a uniform implementation of polymorphism may also be used for this technique. The difference is that values are represented in this format only when their type is abstracted over. The extra cost of this technique over uniform implementation is in the conversion for the execution of polymorphic code. The advantage is the extra flexibility in the rest of the architecture, where any number of different formats may be used.

As an example of this technique, imagine a language which supports only two data types, integer and real. Integers are represented by a single word, and reals by two words. One possible polymorphic representation would be as heap objects. Integers could be represented as a single-word object, and reals as a double-word object. The conversion and reversion routines are different for the two types, consisting of the appropriate wrapping and unwrapping of the values. Within a

polymorphic procedure both types are represented by a single-word pointer, and this is known statically by the compiler.

In this example a more efficient solution may be to represent a polymorphic value always as two words, which do not need to be wrapped in a heap object. Values of both possible types may be contained in this space. The operation to convert a parameter of type integer to the polymorphic format may be simulated by an increment of the stack pointer, and the reversion by a decrement. No operation is required to convert and reconvert real values, as they are represented by the same format as polymorphic values.

As an example of this technique, consider the program in Figure 5.2.4. Two procedures are declared and called; both return their first argument. Both are applied to pairs of integers. The only difference between the procedures is their type; *intFirst* may be applied only to integers, and *polyFirst* may be applied to any two operands of the same type.

```
let intFirst = proc( x , y : int → int ) ; x
let polyFirst = proc[ t ]( x , y : t → t ) ; x
let a = intFirst( 2 , 3 )
let b = polyFirst[ int ]( 2 , 3 )
```

Figure 5.2.4 Identical procedures with different types

The procedure *polyFirst* is compiled using the polymorphic format. As *x* and *y* are of type *t*, which is statically known to be an abstracted type, both parameters are known to be represented in the polymorphic format. All relative stack addresses may be calculated statically given this information.

The conversion to and from this format depends on the format of the type being applied, which in this case is known statically. When a non-polymorphic

procedure with two integer parameters is called the integers are evaluated one at a time and left upon the stack. The frame of the invoked procedure uses these two stack locations as its parameters. As this call is to a polymorphic procedure, however, after each parameter is evaluated it must also be converted to the polymorphic form. This is achieved, depending upon the chosen polymorphic format, either by wrapping it as a heap object or by incrementing the stack pointer.

When the procedure finishes executing, the result will be left on the stack in the polymorphic format. Again, the actual type of the result is known from its context, and the integer format may be regained by application of the appropriate reconversion operation.

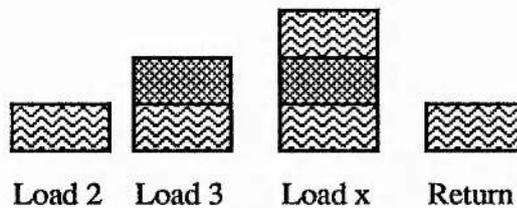


Figure 5.2.5 Stack manipulation during the call of `intFirst`

Figure 5.2.5 shows the manipulation of the stack during the application of the procedure `intFirst`. Figure 5.2.6 and 5.2.7 show the different manipulation during the application of the procedure `polyFirst`. Figure 5.2.6 shows the polymorphic format as a heap object, and Figure 5.2.7 as a double word value. Both procedures successfully result in the same word being left on the stack.

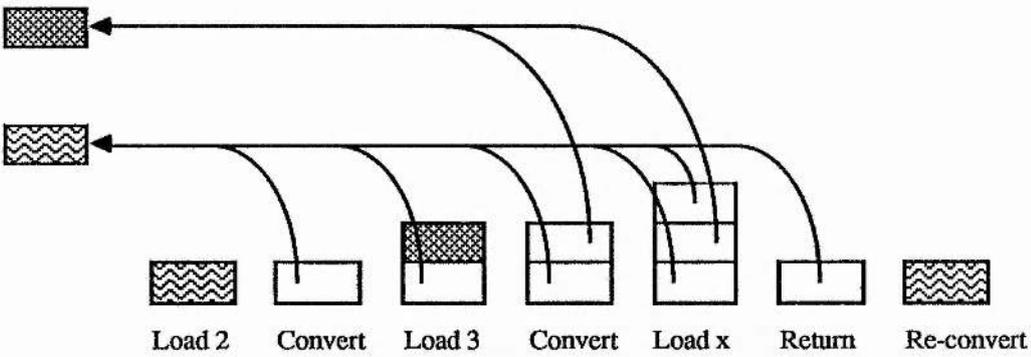


Figure 5.2.6 Stack manipulation with pointer polymorphs

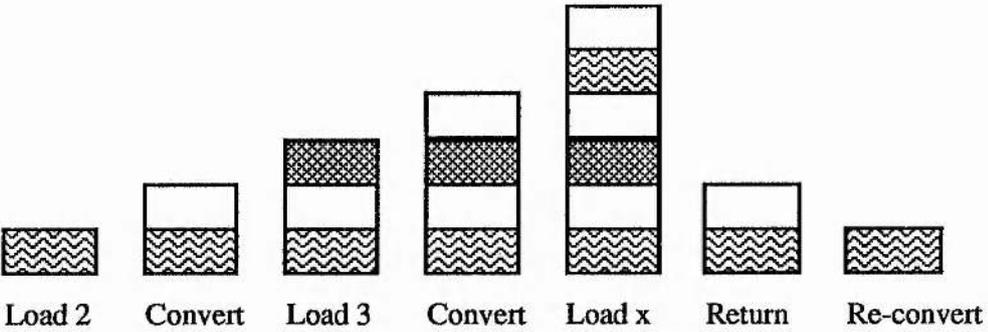


Figure 5.2.7 Stack manipulation with two word polymorphs

5.2.4.2 Conversion by procedural encapsulation

The above description of polymorphic conversion and re-conversion carefully avoids mention of whether the code which performs these operations is located with the polymorphic procedure or with its call. It is always more efficient to perform the expansion and contraction at the procedure call, as the type of operands are known statically. If the conversion code were inside the procedure, then knowledge of the formats of its operands would have to be passed with them, making both the conversion routine and the procedure call less efficient.

For conversion to be possible outside the procedure it must always be possible to tell statically the format of the operands that the procedure is expecting. This will always be the case if procedures are manifest values, but is not so in general with first class procedures. For example, Figure 5.2.8 shows a procedure where an identifier *first* is initialised with a procedure value, of type **proc(int → int)**. However, the value of this procedure depends upon a dynamically evaluated condition, and whether it is a polymorphic procedure value or not cannot be determined statically. At its call, therefore, it is not known statically whether or not polymorphic conversion should be performed on its operands.

```

let polyFirst = proc[ t ]( x , y : t → t ) ; x
let intFirst = proc( x , y : int → int ) ; x

let first = if <condition>
            then intFirst
            else polyFirst[ int ]

let a = first( 2 , 3 )

```

Figure 5.2.8 Calling a procedure of unknown derivation

The problem is in fact more general than this. If the polymorphic conversion were performed inside the polymorphic procedure, then parameters would always be passed in their non-polymorphic formats. However, it would then be necessary to pass the information of the representation types every time a procedure was called, whether it was polymorphic or not. Although this would be straightforward to arrange, code which did not use polymorphism would have to pay a significant extra cost for procedure calls.

In some languages, polymorphic procedures are not type compatible with any of their monomorphic forms without the explicit application of a type parameter. One way of allowing a polymorphic procedure to execute correctly in contexts with different type views is to implement this partial application as a procedure application. Polymorphic procedures are compiled as higher-order procedures

which take the necessary representation information as a parameter and return the appropriate specialised procedure.

```
let polyProc = proc[  $\alpha$  ]( Sig1[  $\alpha$  ]  $\rightarrow$  Sig2[  $\alpha$  ] )
=>
let polyProc = proc( tag(  $\alpha$  ) : tag  $\rightarrow$  proc( Sig1[  $\alpha$  ]  $\rightarrow$  Sig2[  $\alpha$  ] ) )
  proc( Sig1[  $\alpha$  ]  $\rightarrow$  Sig2[  $\alpha$  ] )
  begin
    <convert all instances of  $\alpha$  to polymorphic form>
    <evaluate body using polymorphic form>
    <convert results from polymorphic form>
  end
```

Figure 5.2.9 Representing a polymorphic procedure by encapsulation

Figure 5.2.9 shows how this could work in principle. The type of the specialisation is denoted in Figure 5.2.9 by α , and the parameters and result of the polymorphic procedure are type signatures which use the type α . It is only the format used for this type, rather than the type itself, which is required for the correct execution of the enclosed procedure. As the polymorphic procedure must already have been correctly type checked, no dynamic manipulation of the full type information is required.

The encoding of representation formats is unimportant, and is represented here by the type **tag**. The most likely encoding would be as an integer, for reasons of efficiency. When a polymorphic specialisation is executed, the outer-level procedure is called with the encoding appropriate to the specialising type. This is denoted by $\text{tag}(\alpha)$ in Figure 5.2.9. The format information is all that is required for the procedure which is returned by the specialisation to be able to execute correctly.

Figure 5.2.10 shows the details of the conversion for the *polyFirst* example used above. The presence of only two machine formats, one word and two words, is

still assumed. The formats are encoded in this example as strings, "one" for a single word and "two" for a double word. Although it is not possible to write this procedure in a high-level language, there is no problem in producing the code within a trusted compiler. Explicit architectural support is not necessary, but may be required to increase efficiency. Although the conversion code looks very bulky in this example, on close examination it may be seen to be of relatively minor cost.

```
let polyFirst = proc[ t ]( x , y : t → t ) ; x
```

=>

```
let polyFirst = proc( tag : string → proc( α , α → α ) )
  proc( x , y : α , α → α )
  begin
    ! convert all instances of α to polymorphic form, i.e.
    ! parameters should be at words one and three,
    ! with stack pointer at four
    case tag of
      "one" : begin
                !parameters are at words one and two,
                !with stack pointer at two
                move stack 2 to stack 3
                set stack pointer to 4
              end
      "two" : begin
                !parameters are at words one and three,
                !with stack pointer at four
                !no conversion is required
              end
      default : error( "no such polymorphic tag" )
    <evaluate body using polymorphic form>
    case tag of
      "one" : begin
                !result should be a single word
                !is currently polymorphic form
                set stack pointer to 1
              end
      "two" : begin
                !result should be two words
                !no conversion is required
              end
      default : error( "no such polymorphic tag" )
    end
  end
```

Figure 5.2.10 Compiler expansion of polyFirst

Notice that the effect of specialising the same polymorphic procedure with a number of different types is not to create new instances of the returned procedure, but only new instances of its closure. Thus only a single copy of the polymorphic code is ever necessary.

With this scheme, a polymorphic procedure must always be specialised with a concrete format before being called. This means that, if a polymorphic procedure is called within the context of another polymorphic procedure, then the parameters must be converted to their concrete format before the call. The necessary tag information is available for this purpose, and the compiler must plant code to convert the polymorphic parameter back to its concrete format for this purpose, as well as for returning the result of the procedure.

There are in fact four different type views possible at the time of the call of a polymorphic procedure. For completeness, these will be listed, and the above scheme will be demonstrated to work in each case. There are four possibilities as the type of the parameter values may be known statically or not, and the procedure may have been specialised or not. Thus the four cases are as follows:

1. concrete parameters passed to unspecialised procedure
2. abstracted parameters passed to unspecialised procedure
3. concrete parameters passed to specialised procedure
4. abstracted parameters passed to specialised procedure

Figure 5.2.11 shows an example of each of these cases. The polymorphic identity procedure declared as *id* is called in four different contexts, with or without prior specialisation and with concrete and abstracted parameters.

```

let id = proc[ t ]( x : t → t ); x

let case1 =          ! concrete parameter and unspecialised procedure
  id[ int ]( 3 )

let callId = proc[ t ]( x : t )
begin
  let case2 =        ! abstracted parameter and unspecialised procedure
    id[ t ]( x )
end

let intId = id[ int ]
let case3 =          ! concrete parameter and specialised procedure
  intId( 3 )

let callId2 = proc[ t ]( x : t )
begin
  let idT = id[ t ]
  let case4 =        ! abstracted parameter and specialised procedure
    idT( x )
end

```

Figure 5.2.11 The four cases of polymorphic procedure call

All four cases are in fact dealt with by the same methodology, by making the call of a polymorphic procedure appear identical to the call of a non-polymorphic procedure. The encapsulating procedure is always called before the polymorphic procedure, and abstracted parameters are always reconverted to their concrete formats before a call.

Although the above mechanism has been described in terms of a language model of explicit specialisation, it may also be used in languages where explicit specialisation is not forced. In languages with type inference, for example, polymorphic procedures are type compatible with any of their monomorphic forms. However, in any statically typed language it may always be detected when a procedure is used in a more specialised context, and the necessary partial application may always be performed safely at this point.

This scheme is general, but has two efficiency considerations which must also be addressed. These are:

- the cost of procedural encapsulation, and polymorphic conversion occurring within the procedure
- the cost of converting parameters of complex types

These problems will be addressed individually, the first in Sections 5.2.4.3 - 5, and the second in Section 5.2.4.6.

5.2.4.3 Cost of procedural encapsulation

Conversion to polymorphic format is more efficient at the procedure call, rather than inside the procedure. This is because the format of the values being converted is known statically, whereas a lookup is required from inside a procedure. The representation of polymorphic procedures as encapsulations does not necessarily mean that conversion may never be performed outside the procedure, as a boolean flag may also be passed as a guide to whether conversion is necessary or not. However, an extra cost has been introduced in that the calling of a polymorphic procedure causes the architecture to perform two procedure calls, rather than one.

There are cases where the above scheme may be seen to be unnecessarily inefficient. For example, whenever a polymorphic procedure calls another polymorphic procedure, then all abstracted parameters are converted to their concrete format when they are passed, and then reconverted to the polymorphic format in the procedure which has been called. Furthermore, two procedure calls will occur each time, as partial specialisation may not be performed outside the context of the first procedure body.

An extreme example of this is a recursive polymorphic function, such as *contains* in Figure 5.2.12. This function uses recursion to determine whether an item is in a list of an appropriate type. In a uniform implementation of polymorphism the cost of applying *contains* to a list of three elements which does not contain the item is ten function calls. With the method of encapsulation, the number of function calls is doubled, and each all may involve conversion to and from the polymorphic

form. Although most of these conversions may be avoided by optimisation, at least some computation is required to check whether they are necessary or not.

```
rec let contains = proc[ t ]( x : t ; l : list[ t ] → bool )
  if empty[ t ]( l ) then false else
    head[ t ]( l ) = x or
    contains[ t ]( x , tail[ t ]( l ) )
```

Figure 5.2.12 A recursive polymorphic function

However, the general efficiency considerations of this model depend to a great extent upon the expected usage of a system. In the model of software reuse in persistent systems given in Chapter 2, for example, polymorphic code is brought from the persistent store, specialised once, and called many times. With this style of use, significant numbers of extra procedure calls are not generated.

5.2.4.4 Reverse encapsulation

Encapsulation is an appealing model to solve the problem of different type views of the same procedure, with the encapsulating procedure being called to specialise the type view. However, encapsulation may also be also used to implement different type views by applying it in the reverse sense. That is, instead of a polymorphic procedure being represented as an encapsulation, the specialised type view of it may instead be implemented as the encapsulation. This allows conversion to and from polymorphic formats to be arranged from the outside of polymorphic procedures, where the type formats are always known statically.

Still using *polyfirst* as an example, it may now be compiled as a procedure which operates over polymorphic formats, instead of also being responsible for the conversion and reversion of its parameters. Then, in the simple example of a call of the procedure as shown in Figure 5.2.13, the code to convert to polymorphic form is placed at the procedure call. The advantages of this are that

only a single procedure call is performed, and the representation type is known statically, so no testing is required for the conversion.

```
let polyFirst = proc[ t ]( x , y : t → t ) ; x
let b = polyFirst[ int ]( 2 , 3 )
```

Figure 5.2.13 A simple polymorphic call

As shown, however, the procedure *polyFirst* may be used in a context where it is not known to be polymorphic. The way this can be implemented is for the compiler to cause it to become encapsulated by a procedure of the specialised type when its specialisation occurs. For example, the specialisation of *polyFirst* to integer would be compiled as in Figure 5.2.14. As previously explained, this technique may be used whether or not a language uses explicit specialisation.

```
polyFirst[ int ]
=>
proc( x , y : int → int )
begin
    convert x to polymorphic format
    convert y to polymorphic format
    polyFirst( x , y )
    convert result from polymorphic format
end
```

Figure 5.2.14 The compilation of specialisation using reverse encapsulation

The efficiency of this technique again depends upon the use of the system. When a procedure known to be polymorphic is called then only a single procedure call is executed, and the conversion to polymorphic form is more efficient. If a polymorphic procedure calls a procedure also known to be polymorphic, only a single call and no conversion is required. In the case where a polymorphic procedure is specialised once and called many times, then twice as many procedure calls occur as in the previous scheme. However, the operations which perform the

conversion are generated in a context where the concrete format is known statically, and so these may be very much more efficient.

The reverse encapsulation scheme does not require any tagging at all, as no operations now depend dynamically upon the format of an operand. However, this is not claimed as a result of the scheme as tagging will be reintroduced for other reasons of efficiency. The tag may be passed as an extra hidden parameter to polymorphic procedures, rather than as a parameter in the encapsulating procedure.

To return to the example of the *contains* function in Figure 5.2.12, the reverse encapsulation scheme has the same cost in procedure calls as for uniform polymorphism, and requires only a single conversion to polymorphic form. The representation type for this conversion is known statically. This is approximately the same cost as the uniform implementation, but the mechanism retains the flexibility of allowing the architecture to use different value formats.

5.2.4.5 Cost comparison of encapsulation and reverse encapsulation

The schemes of encapsulation and reverse encapsulation are not fundamentally different, but have different costs according to the use of polymorphic procedures. There are four different situations where a polymorphic procedure call gives rise to different costs with the two models. These depend upon whether the type of the parameter values is known statically or not, and upon whether the procedure has been specialised or not. The four cases, as shown earlier, are as follows:

1. concrete parameters passed to unspecialised procedure
2. abstracted parameters passed to unspecialised procedure
3. concrete parameters passed to specialised procedure
4. abstracted parameters passed to specialised procedure

With the first model of encapsulation, all four cases are dealt with in the same way, by making the call of a polymorphic procedure appear identical to the call of a non-polymorphic procedure. The encapsulating procedure is always called before the polymorphic procedure, and abstracted parameters are always reconverted to their concrete formats before a call.

With reverse encapsulation, the situation is optimised for the call of a procedure which has not been specialised. This executes only a single procedure call. The polymorphic conversion is also more efficient in Case 1, as the type being converted from is known statically. In Case 2, it may be determined statically that no conversion is required. However, to ensure that the system will still work where a polymorphic procedure is specialised, the specialisation must be represented as a procedure encapsulation. This is in contrast to the procedure call for the first model. This means that calling such a procedure after specialisation involves two calls instead of one. The conversion, however, is again more efficient. Case 4 is the same as Case 2 in the reverse encapsulation scheme; the partial specialisation in this context requires no computation, as the parameters are guaranteed to be in the polymorphic format.

It may then be seen that reverse encapsulation is more efficient where a procedure is known to be polymorphic, but probably less efficient in the case where a procedure is specialised once and called many times. Which scheme is more appropriate thus depends entirely upon the way in which a polymorphic system is used.

5.2.4.6 Lazy conversion using tagging

So far the discussion of both methods of encapsulation has been limited to simple parameters which may be easily converted to the polymorphic form. However, converting all instances of an abstracted type inside a data structure may not be

sensible if, for example, some such values are never manipulated by the procedure. Figure 5.2.15 shows an example of this. This procedure is quantified over a type denoted by t , and returns the first element of an array of this type. In such cases it would be sensible to perform the polymorphic conversion lazily, as a value of the abstracted type is accessed.

```
let first = proc[ t ]( x : array[ t ] → t ); x( 1 )
```

Figure 5.2.15 Eager conversion not sensible

If conversion to polymorphic form is to be performed lazily, then it must be performed by code within the body of the polymorphic procedure. This means that an encoding of the concrete format must be accessible by the procedure. In the case of normal encapsulation, this information is already available in the procedure closure. With reverse encapsulation it may be provided by compiling a polymorphic procedure with an extra hidden parameter, in which the tag information is passed. When the compiler encounters a call of a polymorphic procedure it can determine statically either the value or the address of the value to be passed as this parameter.

Code may be planted which accesses this information to perform appropriate conversions, as has previously been shown in the example of procedural encapsulation in Figure 5.2.10. This may be required, for example, to convert a value to polymorphic format after it has been dereferenced from an array, or to reconvert it from polymorphic format before it is used to update a location in an array.

There are also addressing issues concerned with the format of values within data structures. For example, in Figure 5.2.16 the dereference operation itself must access the format information to calculate the offset of the second element of the array. Although this is relatively straightforward for arrays, it becomes more

complex for structure or record types, as the format of some values may affect the offsets of others. However, knowledge of the tags always conveys enough information for the compiler to plant code to calculate the correct offsets.

```
let second = proc[ t ]( x : array[ t ] → t ) ; x( 2 )
```

Figure 5.2.16 Lazy conversion may include address calculation

One special case of polymorphic conversion is when parameter and result types contain procedures. The conversion of a non-polymorphic procedure to one which operates over polymorphic formats and vice-versa would be very costly, and a different scheme is adopted for this.

Conversion of values with procedure types is not straightforward. The difficulty is that the same procedure may be required to execute with parameters of concrete or polymorphic representations, depending upon its context. A further complication, which rules out some potential solutions, is that procedures may be declared within the polymorphic context. It is not always statically detectable in which context a procedure has been declared. Figure 5.2.17 shows an example of this. The procedure *y* which is passed as a parameter has been declared outside the context of the polymorphic procedure, and the procedure *new* is declared inside. These procedures however are type compatible, both in the context of the procedure and after being returned as its result. Depending upon the dynamically evaluated condition, either procedure may be called both during the execution of the procedure and after it has returned its result.

It would be possible to transform the low-level code of the procedure appropriately, but this is potentially a very complex and expensive operation.

```

let useProcs = proc[ t ]( x : t ; y : proc( t → t ) → proc( t → t ) )
begin
  let new = proc( z : t → t ) ; z

  let either = if <condition> then y else new

  let unknown = either( x )

  either
end

```

Figure 5.2.17 Procedures of abstracted type

A more sensible solution is to convert the format of the procedure parameters before a call, rather than the procedures themselves. No conversion is then necessary for procedures which are either passed as parameters or returned as results. On the application of a procedure whose parameter types are abstracted over, the parameters are converted back to their concrete formats using the tag information. If all dereferences of abstracted types from constructors are performed lazily, then any such operations within the procedure declared outside the polymorphic context will continue to execute correctly inside its context.

The only problem remaining is that of procedures declared within a polymorphic context. They must somehow be able to execute correctly when they are passed parameters which are not of polymorphic format. This may be achieved by using the scheme initially introduced for procedural encapsulation of polymorphic procedures. The procedure declared contains the necessary code to perform the polymorphic conversion and re-conversion of its parameters and results. As it is declared within the context of the polymorphic procedure, the compiler can statically calculate the address where the tag may be found. If such a procedure is exported from the polymorphic context, the tag value will be retained in its closure, and it will be indistinguishable from a procedure declared over the concrete type.

5.2.4.7 Conclusions

The semi-uniform, partly tagged scheme has been introduced as a method of allowing polymorphism with both first class procedure values and with an architecture that uses different value formats. The model has been logically introduced at a high level by a description of the implementation using procedure closures. This introduces a cost to the system of extra procedure calls, but it has been shown how the extra calls may be avoided by reversing the encapsulation strategy. The conversion to and from polymorphic format has also been shown to be inexpensive, perhaps only a single instruction on many architectures. The model seems, for the most part, to capture the efficiency of polymorphic procedure call displayed by uniform polymorphism but without disallowing different value formats and their associated efficiency. The model is less efficient than that of textual polymorphism, but continues to work in the presence of first class procedures.

This scheme, without reverse encapsulation, has been successfully implemented in the Napier88 system, with architectural support provided by the Persistent Abstract Machine [BCC88, CBC89].

5.2.5 Conclusions

Three different models of implementing parametric polymorphism have been discussed in depth. Of particular interest is the suitability of the models for use in a persistent system.

Textual polymorphism depends upon the compiler producing different versions of code for different uses of a polymorphic procedure. This is appealing in that it factors out all of the cost of polymorphism at compilation time, although it can

cause the production of a very large volume of code. However this method alone seems unable to support first class or persistent procedures.

Uniform polymorphism provides an elegant model for polymorphism, and an efficient implementation of polymorphic procedures. However, this is at the cost of disallowing multiple value formats in an architecture, which leads to the generation of otherwise unnecessary heap objects. This implementation of polymorphism is therefore very suitable for purely functional languages, where function call is the major activity and the system must already be geared to the production and garbage collection of very large numbers of transient heap objects. Such an implementation is less suitable for persistent systems in general, where the number of object faults is a major factor in the execution cost of a program.

The new implementation method described here has been called semi-uniform, partly tagged. The main point of the method is that it allows different value formats by introducing a new format for polymorphic values, with values being converted dynamically between formats. Tagging is required only when the conversion is more efficient dynamically, and it is polymorphic procedures themselves which are tagged, rather than values. The method may be implemented using a conventional architecture, and does not therefore place constraints on the implementation of other language features.

5.3 Inclusion polymorphism

In general, programming systems which exhibit subtype inheritance allow some contexts where a data value may be used in place of one with less functionality. One type may be a subtype of another if all operations allowed on the second type are also allowed on the first. Inclusion polymorphism is the most general use of

subtyping, where in any context the specified use of a type may be applied to any of its subtypes.

There are a number of models of subtype inheritance which are less general than pure inclusion polymorphism, as they have extra restrictions in the type compatibility rules. These include, for example, systems with explicit or single type inheritance [ACO85, GR83]. However, any implementation technique which works for inclusion polymorphism will also work for any other form of subtyping.

5.3.1 Subtype inheritance

Inclusion polymorphism gives the ability to abstract partially over the type of values. This gives a different set of implementation problems from parametric polymorphism, in which type abstraction is all-or-nothing.

Most languages with subtyping describe their type system as a lattice, based upon the partial ordering of the subtype relation. This subtype relation may be predefined by a language, according to the primitive operations defined over type constructors, or explicitly defined by a programmer. Implicit subtype relations are normally the most general possible according to the primitive operations defined over each type constructor. To preserve correctness, any explicitly defined subtype relation must always be a sub-lattice of this most general subtype relation.

For example, Figure 5.3.1 shows the declaration of some record types in a system with predefined subtyping rules. One record type is a subtype of another if it has all, and perhaps more, of its labels, and the types associated with each label are recursively in the subtype relation. *student* and *employee* are both subtypes of *person*, *demonstrator* is a subtype of both *student* and *employee*, and *student* and *employee* are not comparable with each other.

```

type person is record( name , address : string )
type student is record( name , address : string ; matricNo : int )
type employee is record( name , address : string ; payrollNo : int )
type demonstrator is
  record( name , address : string ; matricNo , payrollNo : int )

```

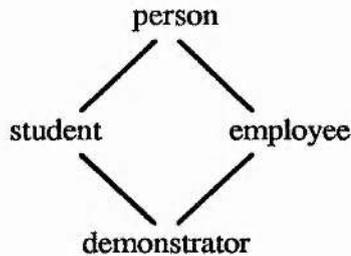


Figure 5.3.1 A lattice of types with implicit subtyping

In terms of the lattice, the type compatibility rules for a system with inclusion polymorphism are that an operation defined over a type at any point in the lattice may be applied to a type anywhere below this point. In Figure 5.3.1, any operation defined over values of type *person* may also be applied to values of any of the other three types, and operations defined over values of type *student* or *employee* may also be applied to values of type *demonstrator*.

The essential difference between parametric and inclusion polymorphism is that parametric polymorphism allows a programmer to abstract fully over a type, whereas inclusion polymorphism gives the ability to partially abstract over a type so that only those properties relevant to a particular context need to be specified. The implementation problems of the two models reflect this difference: systems with parametric polymorphism must implement a fixed set of operations with no static type information, whereas systems with inclusion polymorphism must implement some type-specific operations with partial static type information.

5.3.2 Value representations and partial type abstraction

To achieve a better understanding of the general problems associated with the implementation of inclusion polymorphism, the way in which values may be represented within an architecture will first be examined.

In the description of implementation techniques for parametric polymorphism no details of value representations were required for the discussion. The only assumption made in the description of the semi-uniform method was that a mapping exists from any type to its representation, and that this mapping may be determined statically for any value.

To simplify the discussion of inclusion polymorphism, a two-tier protocol for value representation will be assumed. The first tier of this protocol is a mapping from type constructors to a fixed size value format within an architecture. The second tier is the set of conventions which describe how this fixed size format is used to represent a particular instance of a constructed type. For example, record types may be represented at the first tier by a fixed size heap pointer, and at the second tier each record type may be represented differently within the heap object pointed to. In the discussion of this section, the terms general format and specific format will be used to refer to these two tiers.

This model is used to emphasise the different implementation problems of parametric and inclusion polymorphism. With parametric polymorphism, the implementation problems are to do with the general format of a value not being known statically. However, the restricted nature of the operations possible on values whose type is fully abstracted means that specific format information is never required. With inclusion polymorphism, the general format is normally known statically, and so operations which are allowed on values of all types present no problem. However, some information about the statically unknown

specific format is also required, as a number of type-specific operations are allowed on such values.

5.3.2.1 General formats

With the exception of types which represent the top and bottom of a type lattice, the subtype relation may not be defined over any two types which have different type constructors. As every constructor has a different implicit set of operations defined over it, the behaviour of values with different type constructors is not comparable. This is not true for some trivial examples, for instance the type of a record with no fields may have the same operations defined over it as that of a variant with no branches. Such examples are of little practical use within a subtype relation, however, and are not normally included.

As the general format of a value within an architecture is defined according to only the type constructor, the general format of a value is usually known statically. Although the full type information may not be available, the type constructor is always known because of the subtyping rules. In fact most languages with any form of subtype inheritance have a type model which is a number of unconnected lattices, rather than a single lattice. Therefore a value's general format may always be deduced statically.

There is an exception to this general rule in languages which define types to represent the top and bottom of the type lattice. This closes the lattice, so that only a single lattice is required to describe the whole type system. The implication of this is that any value may be used in place of a value of type top, and a value of type bottom may be used in place of any other value.

Values of these types must always be dealt with as special cases. As no operations are available on a value of type top, no information need be passed if a value is to

be viewed as this type. If a value of type bottom exists, then every operation in the language must in any case perform a dynamic check on each operand, to check whether it is this value. In a statically typed language, the value of type bottom may be simulated by a number of reserved values, one for each different general format in the architecture.

In an architecture which relies upon a uniform implementation of parametric polymorphism, only a single general format is in any case defined for all type constructors.

5.3.2.2 Specific formats

The specific format of a data value is usually decided according to rules associated with each different type constructor. These will depend upon the different operations associated implicitly with values of the particular constructor type. For example, a type constructor for records may be defined as

"For mutually unique identifiers $i_1..i_n$, and types $t_1..t_n$, **record**($i_1 : t_1; \dots; i_n : t_n$) is the type of a record with field i_i associated with type t_i for $i = 1..n$."

The particular operations available upon a type constructed with this rule depend in this case upon the identifiers which specify the field names, and their related types. For example, the type

record($a : \text{int} ; b : \text{real}$)

will implicitly infer operations which select and perhaps update the fields associated with the labels a and b . If records are always implemented as heap objects, then the specific format convention for record types will need to allow the appropriate offsets to be determined from each field name. In a system with no inclusion polymorphism, the above type could, for example, be represented as a three word structure, with the first word representing the value of a , and the second and third representing the value of b .

In a type system with inclusion polymorphism, there is a further requirement that the specific format must allow all of the implicit operations upon a type to succeed even where the full type has been partially abstracted over. For example, a value which is viewed as a particular record type may be a partial abstraction over a type with more fields. If record types are implemented as single heap objects, and their specific format is determined according to their type at creation, then a compiler is not always able to statically determine an offset for either field.

5.3.2.3 Bounded universal quantification

The separate implementation problems are clear in consideration of the model of bounded universal quantification [CW85]. This model partly unifies the two models of polymorphism by allowing a maximum type to be specified in a universal quantification. For example, Figure 5.3.2 shows a procedure which is specified over any value of type *t*, with the condition that *t* is a subtype of *person*, and returns the value of its *name* field.

```
let name = proc[ t ≤ person ]( x : t → string ) ; x( name )
```

Figure 5.3.2 An example of bounded universal quantification

Parametric polymorphism is a subset of this model, as it may be expressed by universal quantification bounded by type *top*; Figure 5.3.3 shows the example *first* from the previous section written in this style.

```
let first = proc[ t ≤ top ]( x , y : t → t ) ; x
```

Figure 5.3.3 An example of bounded universal quantification

For any type bounded only by *top*, the general format is not known statically, but no more specific operations are available. The implementation techniques described for parametric polymorphism will therefore suffice for such a procedure.

For quantification bounded by any other type lower in the lattice, the general format will be known and more specific operations are possible. Bounded universal quantification may itself be implemented by the combination of any two techniques which respectively implement parametric and inclusion polymorphism.

5.3.2.4 Implementations

Three categories of implementations which solve the problems of specific formats will be discussed. These fit into the categories introduced earlier in this chapter. The first is uniform, where the compiler and the architecture both behave in the same way independently of the specific type of the operand. The second is tagged, where the compiler always produces the same instruction for the same logical operation, but the interpretation of this instruction by the architecture depends upon information associated with the instance of the operand. The third category is partly tagged, where, as in the partly tagged solution for parametric polymorphism, not all data values are tagged and maximum use is made of static type information to increase the efficiency of the polymorphic operations.

There is no discussion of a textual implementation for inclusion polymorphism, as no such implementations are known to exist. This is perhaps for historical reasons. There are two main classes of language with subtyping schemes. The first of these are the object-oriented languages, such as Smalltalk [GR83]. In many such languages subtyping is inextricably mixed with a model of dynamic binding and separate compilation. All other languages which display subtype inheritance also have first-class procedure values. The lack of textual implementations of inclusion polymorphism may therefore be related to the historical order in which these concepts became understood by the research community.

The general discussion of implementation will first be concentrated upon the problem of field addressing in record types. This is the only manifestation of inclusion polymorphism where a number of different implementations are known and may be contrasted. Subtyping may also be sensibly defined over a number of other type constructors, including variants, procedures, and integer subranges. A more general discussion of the techniques will also be presented.

5.3.3 Uniform field addressing

With a uniform implementation of polymorphism, the compiler and the architecture which support a language must behave in the same manner when operations are specified over values whose full type is not known statically. In the context of record addressing, this means that the architecture must support the logical operation of finding the value or location which is referred to by a record label. This implies that the specific format of record values does not depend upon the particular type of a record.

Such a scheme is implemented in the programming language Galileo [ACO85]. Records whose type may be partially abstracted are implemented by a linked list of pairs, each pair containing a string and a value. The strings represent the labels of the record fields, and their associated values are held as the other element of each pair. Thus the low-level operation to find the value associated with a label takes a string and a record as parameters, and chains down the list until the string matches the label. This operation is independent of the particular record type. As the static type system will ensure that a record has no two identical labels, the list may be held in any order.

```
let joe = employee( "Joe Doe" , "1 Assignment Boulevard" , 100000 )
```

Figure 5.3.4 A value of type employee

For example, consider the value of type *employee* created in Figure 5.3.4. Using this uniform representation scheme, the structure represented in Figure 5.3.5 will be built.

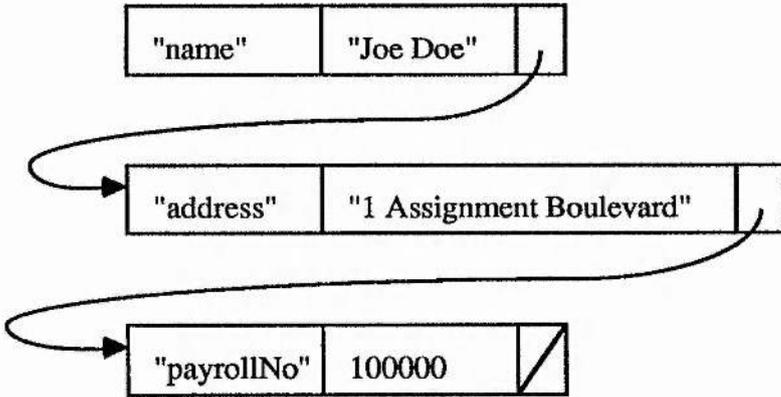


Figure 5.3.5 A linked list record representation

This scheme is clearly inefficient for both creation and addressing, but may be useful either as a reference model or if other operations must be supported. In Galileo, for example, this scheme is adopted mainly to allow the extensibility of such values.

A variation of this scheme would be to use an indirection through an address map in each record value. The address map, which may be located at the start of every object, contains offsets for the fields belonging to that particular subtype. For example, the value created in Figure 5.3.4 would be represented as shown in Figure 5.3.6. Creating such a value entails creating the address map as well as the fields of the record.

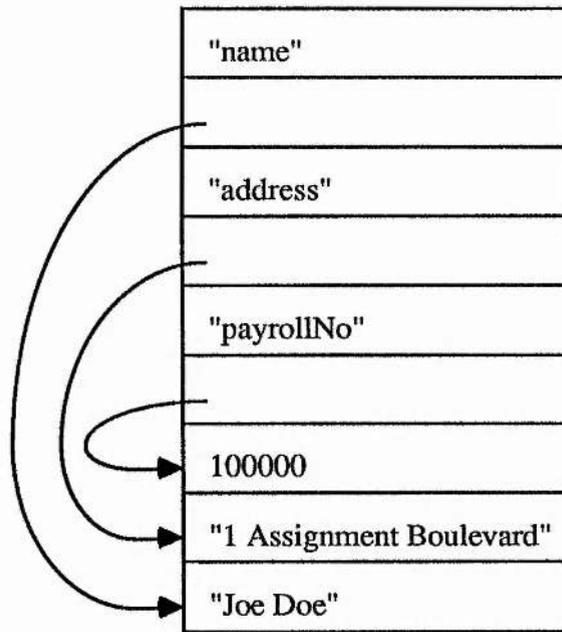


Figure 5.3.6 A record with its own address map

Record selection may be resolved, at execution time, by looking up the appropriate string in the address map to yield the correct field offset.

As stated in the introduction, the categorisation of implementation techniques is simplified for descriptive purposes. This representation could arguably be classified as uniform or tagged, depending upon the level of abstraction at which architecture instructions are described.

5.3.4 Tagged field addressing

An immediate optimisation of the address map technique is to have only one copy of the address map for every type of record, as in Figure 5.3.7. The record itself is now formatted according to its specific type, but contains a pointer to the map. This pointer acts as a tag representing the record's specific type information. This arrangement may also have some advantages in identifying pointers for garbage collection.

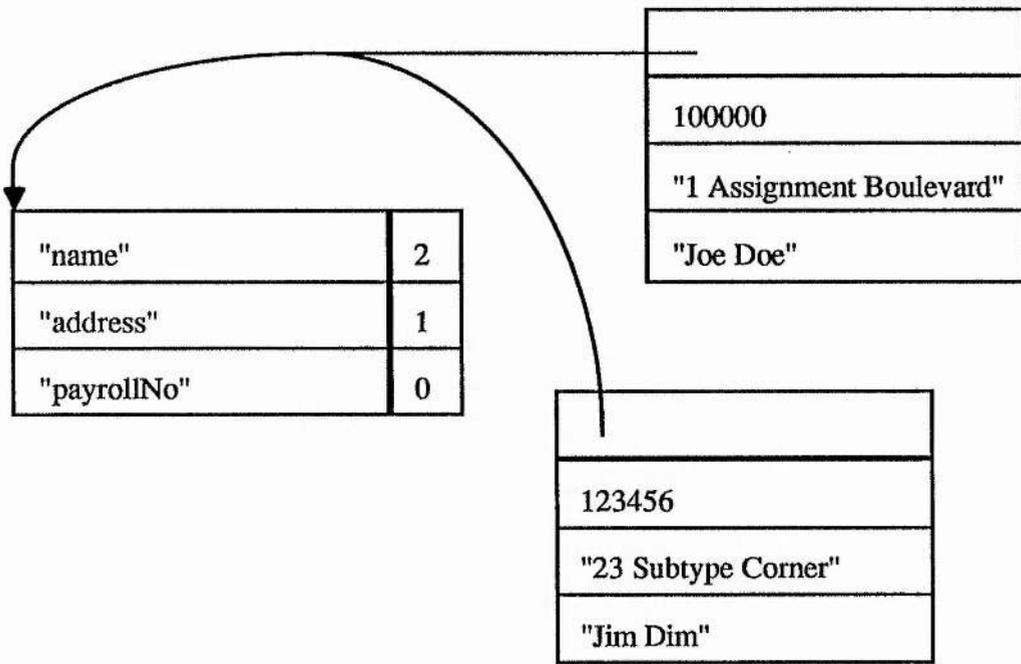


Figure 5.3.7 Sharing the address map

An advantage of these techniques is that strings match across compilation units which means that there is no need for a centralised dictionary of field identifiers and addresses. Each compilation unit can contain its own address map and still work consistently with independently prepared data. In object-oriented database systems [CL88, BBB88], and distributed systems, this aspect is a major consideration.

5.3.4.1 Variants of string address maps

The string address maps shown above may be optimised by the use of a hashing function. However, a further improvement in the efficiency of the address maps would be gained if they could be keyed by an integer instead of by a string. To allow this the compiler could keep a central dictionary of field names, in which it can statically allocate a unique integer key for each field name. Thus

X(name)

can be translated, at compile time, to key (name) field of X. This key, which is an integer, can be used at run time to search the address map.

The drawback of the method is that it requires a centralised dictionary for field keys. In an object oriented database system, or a distributed system, the number of keys may become large and access to the dictionary holding the keys may constitute a bottleneck in the system.

A further variation is found in the ObServer system [SZ86]. This is an object oriented database with name equivalence and explicit subtyping. That is, a type is a subtype of another only if it is declared to be so in the database schema. Consider the following schema:

```
type thing is ( age : int )  
type limbedThing is thing with ( arms,legs : int )  
type sightedLimbedThing is limbedThing with ( eyes : int )
```

In the above, the fields may be grouped as: (age, (arms,legs), eyes). Thus the address map need only contain entries for these groups. The above schema may be used to create an object of type *sightedLimbedThing* by

```
let Ron = sightedLimbedThing( 42,2,2,2 )
```

Each set of common fields may be allocated an address 'slot'. Within the slot (arms,legs), the field offsets of *arms* and *legs* can be calculated statically. This is shown in Figure 5.3.8.

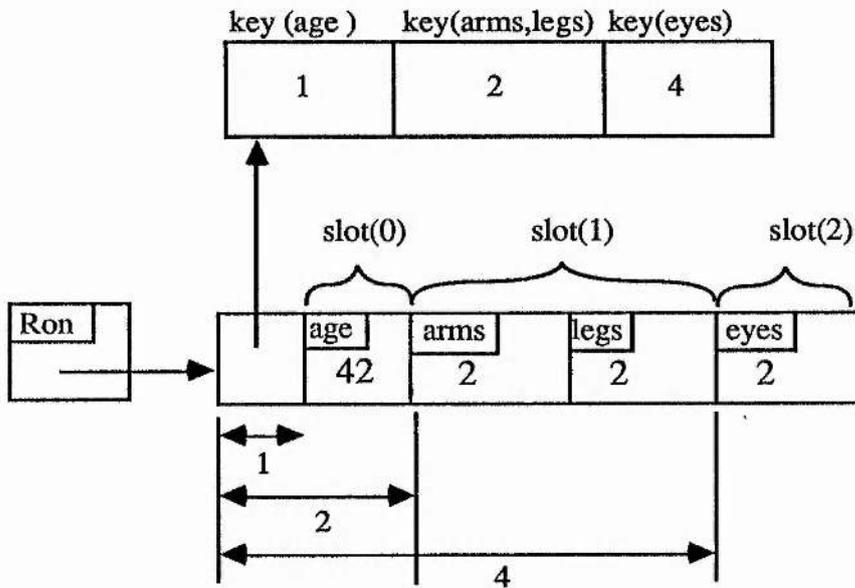


Figure 5.3.8 Fixed Slots for Subtypes

There are two advantages to this scheme. First, slots may be placed on different volumes, or distributed, an important consideration for large databases. Second, the addresses within the subtypes may be calculated statically. The disadvantage is that it works only for name equivalence with explicit subtyping, and may run into severe reorganisational overheads if the schema is edited to alter the subtype hierarchy.

5.3.5 Partly tagged field addressing

An implementation technique will now be described which uses an integer mapping for record addressing.

An implementation for languages with bounded universal quantification is first described. This technique uses the same features as the encapsulation technique described for parametric polymorphism. Bounded universal quantification restricts where subtype values are compatible, in such a way that it is possible to tell

statically the precise type of an object except where it is explicitly abstracted over. Encapsulation may be used at these points to contain the relevant field addresses.

This technique is not however sufficient to implement inclusion polymorphism in general. This is because it may not always be statically determined whether a value's type is abstracted or not. The description goes on to extend the implementation by encapsulation to allow for the more general case.

5.3.5.1 Bounded universal quantification

For the examples in this section, the type hierarchy described in Figure 5.3.9 will be used. This describes an implicit subtype relation between things, animals, things with legs, and animals with legs.

```
type thing is structure( age : int )  
type animal is structure( age : int, food : string )  
type leggedThing is structure( age : int, noOfLegs : int )  
type leggedAnimal is structure( age : int, food : string, noOfLegs : int )
```

Figure 5.3.9 An implicit subtype relation

In such a system, there is an implicit function *noOfLegs* with which we may wish to find the number of legs belonging to either a table or a dog, or indeed any object which is a subtype of *leggedThing*. Figure 5.3.10 shows a procedure which tests whether an object with legs is stable, and two calls of this procedure.

```
let fallsOver = proc[ t ≤ leggedThing ]( x : t → bool )  
    x( noOfLegs ) ≤ 2  
  
let unsafe = fallsOver[ leggedThing ]( myDesk )  
let dogmeat = fallsOver[ leggedAnimal ]( myHorse )
```

Figure 5.3.10 A procedure with bounded quantification

To determine the correct address fields, it is possible to use information that is statically available at the call of such a procedure. In the above examples, at each

procedure call the type of the operand is known statically, and the field offsets can be calculated. Calculating this information statically saves most of the cost of the associative lookup required with the lookup table solution.

Since the type of the procedure being called is known statically, it is possible to tell which fields of the operand object may be required during the execution of the procedure. For example, the procedure *fallsOver* is restricted to an operand of type *leggedThing*, and so only the *noOfLegs* and *age* fields may be required, no matter how many other fields the operand may contain.

The information may be encapsulated without altering the run-time support for the language being implemented, so long as this includes support for higher-order functions [AM85, BCC88, CBC89]. When the quantified procedure is compiled, it is compiled as two nested procedures, with the field offsets being parameters to the outer procedure and appearing as free variables within the inner procedure. The inner procedure, corresponding to the quantified one, is returned as the result of the outer. This scheme is exactly parallel to the one proposed earlier for unbounded universal quantification.

```
let fallsOver = proc[ t ≤ leggedThing ]( x : t → bool )
    x( noOfLegs ) ≤ 2
=>
let fallsOver_wrapper =
    proc( age_offset, noOfLegs_offset : int → proc( α → bool ) )
        proc( x : α → bool )
            x( noOfLegs_offset ) ≤ 2
```

Figure 5.3.11 Compiling an encapsulation with specific format information

Figure 5.3.11 shows how such an example is treated by the compiler. Notice that, whatever the type of the parameter, only the fields *age* and *noOfLegs* may be

accessed. This is therefore all the information about the specific format that needs to be passed.

At the point in the program where the procedure is called, the compiler can plant the integer values required as parameters to the wrapper procedure as literal values in the code stream, and no dynamic lookup is required. This is possible as the compiler may statically determine which offsets are required and the precise type of the operand. Thus the matching program transformations in Figure 5.3.12 make the scheme work correctly. When the *noOfLegs* field is accessed in the procedure, the second field of *myDesk* and the third field of *myHorse* will be looked up as required.

```
fallsOver[ leggedThing]( myDesk )
fallsOver[ leggedAnimal ]( myHorse )

=>

( fallsOver_wrapper( 1,2 ) ) ( myDesk )
( fallsOver_wrapper( 1,3 ) ) ( myHorse )
```

Figure 5.3.12 Compiling calls to the encapsulated procedure

The type of the operand is not always known statically, but because of the generality of the solution no extra work is required for these cases. For example, Figure 5.3.13 shows a procedure with a nested call to another quantified procedure.

```
let atConference = proc[ t ≤ leggedAnimal ]( x : t → bool )
  x( food ) = "curry" and fallsOver( x )
```

Figure 5.3.13 A nested quantified call

In this example, the type of the *x*, supplied to the *fallsOver* procedure as a parameter, is not known statically as it has already been abstracted over. It is clear, however, that *x* is a subtype of *leggedThing* and as such may be used as the

operand of *fallsOver*. The compiler does not know statically the offsets it requires to pass to the *fallsOver* wrapper procedure, but it does know where they can be found, as they must be a subset of the offsets provided by the *atConference* wrapper procedure. The procedure then compiles as shown in Figure 5.3.14. It may be seen that the correct values are introduced for the required offsets.

```

let atConference_wrapper =
  proc( age_offset, food_offset, noOfLegs_offset : int → proc( α → bool ) )
    proc( x : α → bool )
      x( food_offset ) = "curry" and
      ( fallsOver_wrapper( age_offset, noOfLegs_offset ) ) ( x )

```

Figure 5.3.14 Compiling a nested call

As before, the technique of compiling higher-order functions has even greater advantage if the field lookups are performed many times with operands of the same type. When this happens the wrapper procedure is called only once, and once the free variables are in place no more work is needed to provide the correct offsets for use within the procedure. For example, the procedure in Figure 5.3.15 incurs a slightly greater fixed cost, but has no penalty in proportion to the size of the array, when compared to the same procedure declared for only one of the allowed types.

```

let allFallOver = proc[ t ≤ leggedThing ]( a : array[ t ] → bool )
begin
  let res := true
  for i = 1 to max do
    if ( a( i )( moOfLegs ) > 2 do res := false
  res
end

```

Figure 5.3.15 Many calls during a procedure execution

If a procedure is going to be used many times with the same type of operand, the same efficiency can be achieved by only calling the wrapper procedure once and using the returned value for all other instances of the call.

5.3.5.2 Inclusion polymorphism

The above scheme relies upon the fact that the compiler knows the precise type of the object supplied as a parameter at the point of type abstraction. In the general case of inclusion polymorphism, this is no longer the case. We now extend the above solution to allow for this.

Unlike bounded universal quantification, inclusion polymorphism allows a location of a particular type to contain a value of any of its subtypes. For example, if a location is initialised with a value of type *thing*, then it may be updated with a value of type *animal*, as *animal* is a subtype of *thing*. The location continues to be viewed as type *thing*: the operation not only updates but also throws away type information, as the object may no longer be used as an *animal* but only as a *thing* from its new reference. For example, the last statement in the program in Figure 5.3.16 is incorrect.

```
let a := thing( 129 )
let b := animal( 23,"petrol" )
a := b
let d = a( age )

let e = a( food ) !** this line is not statically correct
```

Figure 5.3.16 Losing static type information

This is because the *food* field of the animal object can not be accessed through the location *a*, as this location is of type *thing*. Notice that it may still be accessed from the location *b*, as may the *age* field. The two locations have a different type view of the same object.

It is now no longer possible for the compiler to determine statically where to find the named fields of an object stored in a particular location, as this location may be updated with an object whose real type the compiler may not even know about at

the time of compilation. The addressing information must be accessed dynamically.

However, the solution already given may be extended to allow this. The semantics of assignment are similar to the replacement of the formal parameter of a quantified function by the actual parameter, with the corresponding type widening that occurs. This technique works by allocating space with the formal parameter location where the addressing information required may be accessed; similar space may be allocated instead with each record type location.

A straightforward way to achieve this is to implement a record type location with a double pointer, instead of a single one. Thus, we have one pointer which references the original record object, and another which points to an address map for the fields. This may appear superficially similar to a more conventional address map solution, but the following points should be noted:

- The address map is not a conceptual part of the object, but its associated semantically object may be viewed through a number of different address maps. (Figure 5.3.17)
- The map contains only the addresses within the object which may be accessed through that location, and has no information about any other fields which may be in the object. (Figure 5.3.17)
- The same map may be shared by many different locations, not necessarily restricted to locations of the same type. (For example, Figure 5.3.20)

Use of the address map is as follows. The compiler statically calculates the field offset as if it were dealing with an object of the known supertype. If it knows that the object is of precisely this type, then this value is used to index the object directly. Otherwise, the offset is used instead to index into the local address map for the object, which will contain the correct address of the required field. The penalty for a dereference is thus at most a single indirection.

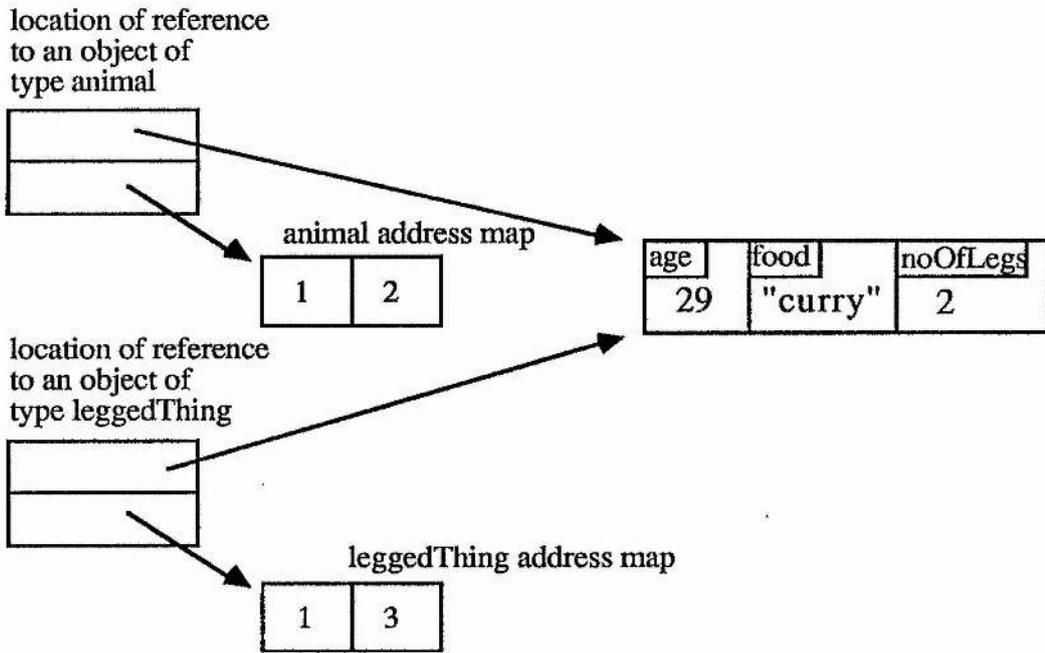


Figure 5.3.17 Location Address Maps

When records are assigned, it is normally necessary only to perform a straightforward assignment of both the pointer to the record and the pointer to the current address map. This is not expected to incur any penalty on most machine architectures, which already support double word assignment.

More work is required only when an assignment involves the loss of type information. Where this occurs in a program is statically detectable. When it does happen, a new address map is constructed. This must map the indexes calculated for the supertype into the values contained in the appropriate positions in the subtype's address map, and may be constructed by performing the first level indirection for each field accessible by the supertype. The code to construct this mapping may then be generated statically.

Consider the example in Figure 5.3.18. Here the location *lThing* of type *leggedThing* has had an object of type *leggedAnimal* assigned to it; this is legal as

far as the type rules are concerned, since *leggedAnimal* is a subtype of *leggedThing*. After the assignment, the location *lThing* must have an address map which correctly maps the fields *age* and *noOfLegs* to the appropriate addresses in the new object. Notice that in general the object referenced by *lAnimal* may have other fields which are not accessible from the location *lAnimal*, since it could itself be any subtype of *leggedAnimal*.

```
let a = proc( lThing : leggedThing ; lAnimal : leggedAnimal )
begin
    .
    .
    lThing := lAnimal
    .
end
```

Figure 5.3.18 Assignment of statically unknown subtypes

Where no optimisation is possible, as in this case, the new address map for *lThing* must be created dynamically. The size of the map is known, as it only need contain addresses for the fields which may be accessed from the new location, in this case *age* and *noOfLegs*. The address map objects however must be created dynamically, as a new one is required every time the piece of code is executed.

Code is planted by the compiler to first construct a new address map object of the required size. Then, for each field in the type being assigned to, in this case *age* and *noOfLegs*, two offsets are calculated by the compiler: let us call them X, the calculated offset into an object of the type being assigned to, and Y, the calculated offset into an object of the type being assigned. Code is then planted to copy the contents of the Yth location in the old address map into the Xth location in the new one. When this has been done for all the addressable fields, the new address map is complete. An example of this is shown in Figure 5.3.19.

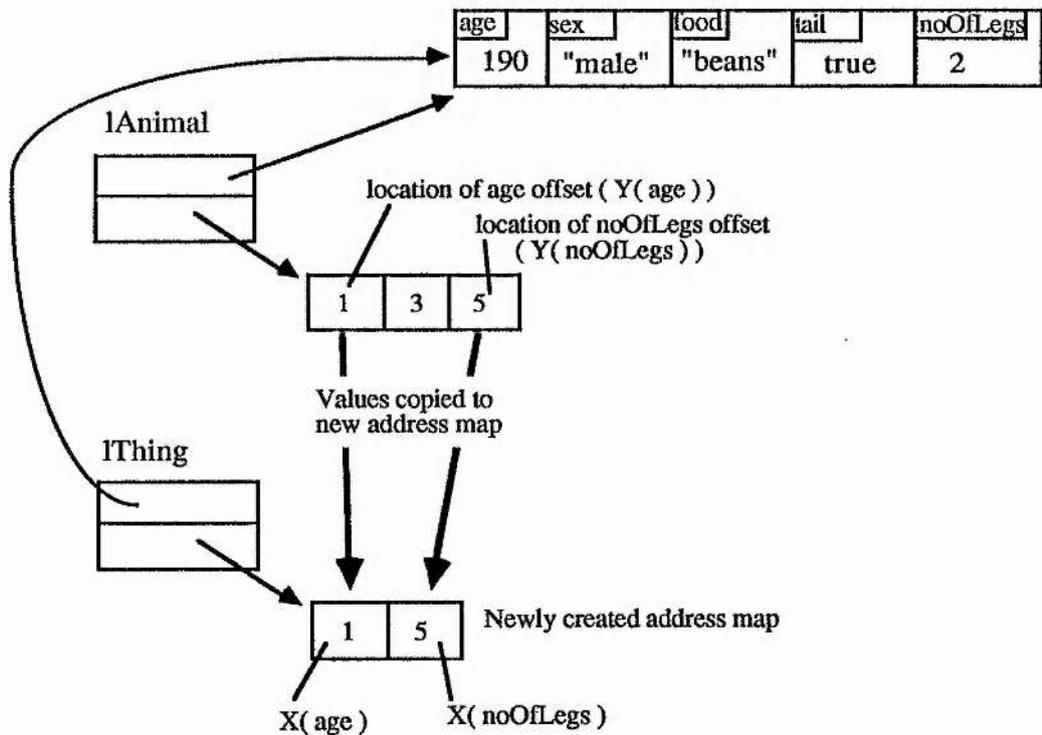


Figure 5.3.19 The Mechanics of Assignment

If the precise type of the assigned value is known statically, then this process may be factored out and the address map may be constructed by the compiler rather than during execution. Notice that as these maps are immutable, they are required at most once per static assignment, rather than dynamically, and also that they may be shared between objects of different types.

The mechanisms described above for creating and calling bounded universally quantified procedures remain the same. As values within these procedures may still be assigned to supertypes, all addressing must be done using the same indirection techniques. The same technique of compiling 'wrapping' procedures is still required to act as temporary storage from which to build new address maps for the quantified parameters: as type widening is occurring, they cannot be simply assigned.

An optimisation of this technique may be obtained by realising that when a record is originally created, the address map required is normally a simple isomorphism: that is the i th address will contain the value i . There need therefore be only one of these maps for the entire system, so long as it is at least the length of the largest record. On initialisation, every object location shares this single isomorphic map. Assignment and construction of new address maps may then take place in the manner described above.

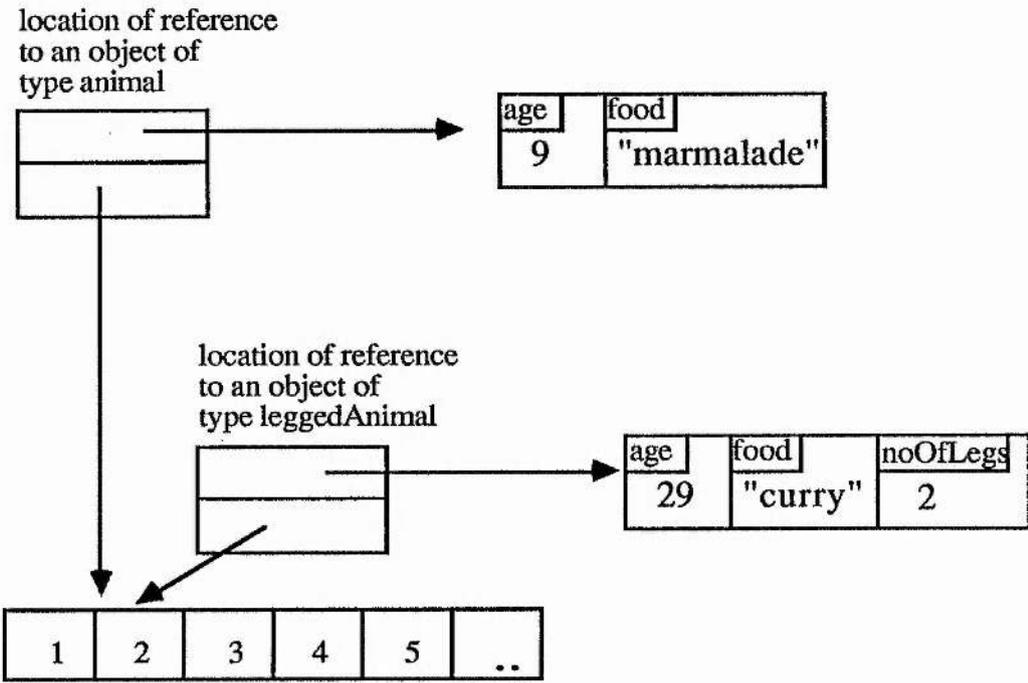


Figure 5.3.20 A Single Shared Address Map for Object Creation

This optimisation is most important to the efficient working of the scheme, as it means that an object creation never involves the creation of a new address map. Notice also that this is not a constraint for a distributed system: although only one of these maps is necessary, any number may be used to suit the implementation of the system.

In fact, this optimisation is most efficiently implemented by reserving a system pointer value to signify this "nil" address map. The dereference operation will check for this value, and if it is present then the object may be accessed directly with the statically calculated offset. Values therefore need have no address map associated with them unless their type is partially abstracted. This is the reason that this solution is classified as partially tagged and not fully tagged. In a program where no partial abstraction occurs, no values will ever be tagged.

Another important optimisation can be made when, on assignment, the fields of the supertype object are identical to those at the start of the subtype object. If this is the case, the original address map may be assigned, as it will still work correctly. This will always be true in languages with explicit single inheritance schemes.

5.3.6 Comparisons

In this section the various merits of the schemes discussed will be considered. As shown, the uniform implementations are intrinsically inefficient except for some special cases of language. Such implementations are not considered further in terms of efficiency. The emphasis here is on comparing the tagged implementation of string symbol table address maps with the new partly-tagged addressing mechanism. The different costs in four areas will be compared between the two general schemes: space overhead, assignment, object creation, and indexing.

In both schemes the space needed to store a reference to the address map is the same – namely the space required by one pointer. However, using the string address map scheme the pointer is associated with the object instances. In the partly tagged scheme the pointer to the addressing information is associated with the locations at which object references are stored. In most cases the number of object instances and the number of locations storing object references will be of the same order of magnitude.

However, the string address map scheme requires the names of the fields to be stored in the address map. This may involve the construction of a simple table with low space overhead or an elaborate hash table. Using the new scheme the names are no longer necessary with consequent space savings.

A very much smaller number of address maps is required in a system using the new scheme. In particular, objects which are not assigned to a location occupied by a supertype never need have additional address maps created for them. Therefore space is saved due to a more compact address mapping and the ability to limit the manufacture of address maps. There may be some space lost due to extra pointers associated with objects.

The traditional object address map scheme has no additional cost associated with assignment. The time cost of the new scheme described in this section depends on the kind of assignment being performed. In those cases of assignment where a value of some type is assigned to a location of exactly the same type, the only additional cost is of an extra word assignment. The assignment of two contiguous words is not expected to be significantly more costly than the assignment of a single word. When a subtype is assigned to a location of a supertype, the operation depends on whether the type of the object to be assigned is known statically or not. If it is, then the new address map required may be created statically and the extra dynamic cost is a single pointer update. If it is not, then the map must be built dynamically, and the cost is one dereference and integer assignment for each field in the supertype.

In both schemes discussed the additional cost associated with object creation is low. Using the traditional technique the address map would normally be created at the time of establishment of the type or class of the object. In the new scheme the

address map objects are not required upon creation, as a single system-wide map may be used until such time as a subtype assignment occurs.

The new addressing mechanism described in this section has been optimised for indexing performance. Indexing encompasses indexing objects to retrieve values, assigning values to locations within objects, and method selection in object-oriented languages. In the worst case there is a single level of indirection for an index, and in the best case a statically planted offset is possible.

The traditional address mapping mechanism uses a hash table to look up names in the address map. Even using a very efficient hashing mechanism such as that used by the Eiffel language implementation [Mey88], the speed of an indirect address can never be equalled. Even if a very high hash hit rate is achieved there is still an associated cost with performing the hashing function in addition to the index. The technique described also allows for non-uniform field sizes and for the dynamic reorganisation of field order.

All of the examples given here describe first order inclusion polymorphism, and the records shown contain only simple objects. It should be noted that this was for ease of description only, and that the technique is sufficiently general to work for any order of multiple inheritance with implicit subtyping.

5.3.7 Generalisation of implementation techniques

Although the discussion has been restricted to record field addressing, the techniques involved may be extended to any type constructor which has a well-defined subtype rule. The general features of each technique will be isolated as far as possible. As an example, these general techniques will be shown in outline to support different methods of implementing subtyping among variant types.

5.3.7.1 Variants and subtyping

Variants are a type constructor which allow modelling with finite unions. They are usually constructed with a set of labels each of which refer to a type, and the variant represents the labelled disjoint sum of these types. A variant type constructor may be defined as

"For mutually unique identifiers $i_1..i_n$, and types $t_1..t_n$, **variant**($i_1 : t_1; \dots; i_n : t_n$) is the type of a variant with label i_i associated with type t_i for $i = 1..n$."

Figure 5.3.21 shows two types constructed using this rule. It is assumed that *CarType*, *VanType*, and *BusType* are types which represent respectively the entities of the three types of vehicles. Nothing further need be known about these types for the purposes of this example.

```
type commercial is variant( van : VanType ; bus : BusType )  
type vehicle is variant( car : CarType ; van : VanType ; bus : BusType )
```

Figure 5.3.21 Two variant types

In the subtype relation defined by Cardelli for such types [Car84], *commercial* is a subtype of *vehicle*. The intuition for this is that any procedure which will work given a value of type *vehicle* will also work given a value of type *commercial*. Further details of this subtype relation are not important here.

A static assertion may be made about which branch of the variant the value is in. This is required to preserve strong typing if the value is to be used as this type. Such an assertion requires a dynamic check. In a language without subtyping, this check may be made efficient by a simple encoding of the labels. This encoding may take advantage of the finite and usually small number of labels declared in a particular type.

An example of this is given in Figure 5.3.22, where a procedure *isVan* is declared. This procedure tests the parameter, specified as type *vehicle*, to determine whether or not it is a van. This procedure may be legally applied to values of type *vehicle*, and also values of type *commercial*.

```
let isVan = proc( v : vehicle → bool )
    v is van

let a = isVan( myVehicle )
let b = isVan( myCommercial )
```

Figure 5.3.22 A variant projection

When a value of a variant type is created, the appropriate label must be known statically, and its encoding may be stored with the value. When a projection to a label is specified, the code will contain an equality test of the encoded label stored with the value against the encoding of the specified label, which may be planted in the projection code. The specific type information for values of a variant type thus may include the method by which its labels are encoded.

In a system with subtyping, however, this information may not be known statically. For example, values of type *commercial* and *vehicle* may both be projected onto the label *van*, and the encoding of *van* may be different in the two types. An implementation of variants under inclusion polymorphism must therefore either be able to dispense with or calculate dynamically the details of this encoding.

5.3.7.2 Uniform implementation

In a uniform implementation, the specific format of a value must be independent of the specific type of a value. This is so that instructions may be specified in the architecture which perform logical operations on any value of the general type. For some type constructors this may mean that a uniform representation is

inherently inefficient, as no specific type information may be used to optimise such implicit operations.

Such a representation may be achieved for variant types by representing variant values as a pair. The first element of the pair could be a string representation of the label, and the second the value. Thus the implementation of the logical operation for label testing is a string comparison of the expected label with the string contained in the pair.

Figure 5.3.23 shows how any value with a label *van* of type *VanType* would be represented in such a scheme.

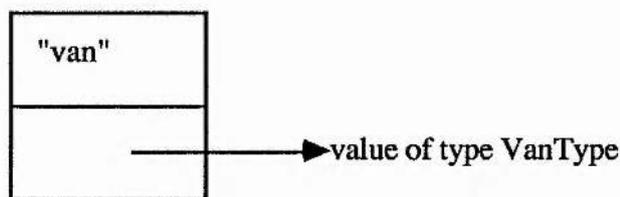


Figure 5.3.23 Implementation of any variant containing a van

The difference in efficiency between this representation and the encoded version described above is that the labels are represented in an unbounded value space. This makes the equality test more expensive, and the space occupied by the representation larger. In the example of *vehicle* there are only three different labels, which could therefore be encoded in two bits. The three character string, although not seemingly a great overhead, requires more space and is harder to organise.

5.3.7.3 Tagged implementation

In a tagged implementation, the specific format of a value is one which would be suitable in a system with no type abstraction, but with an extra pointer to a

representation of the specific format information. To perform an operation on a value whose specific type is not known statically, the compiler produces code which accesses this information dynamically. The cost of an operation in this scheme is approximately the cost of the same operation in a static system plus the extra cost of this dynamic lookup. The information looked up is specific to the type with which the value is created, and as such may be shared by any number of values created with this type.

For variant types the specific type information may be stored by placing the label encodings in a string lookup table. The encoded form of the label will be stored with the type. To test a variant value for a particular label this table must be used to obtain the specific encoding of the string label. This encoding is then checked against the encoding held with the variant value.

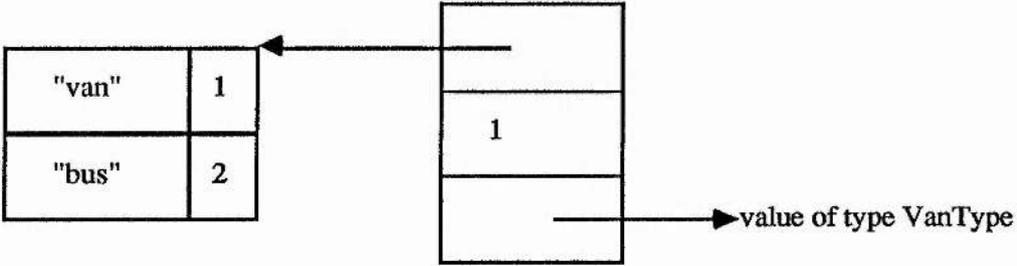


Figure 5.3.24 Implementation of *commercial* containing a van

This method of implementation for variants seems over-elaborate and generally inefficient, but is included for completeness. As always, the best implementation technique depends largely upon the way in which a system is used, and so it is not necessarily surprising that the technique most widely adopted for record addressing is not well suited for variant label testing.

5.3.7.4 Partly tagged implementation

The general strategy for partly tagged implementation is also to have a specific format which is suitable for a language with no type abstraction. Information specific to a particular type view, however, is kept in association with the type view itself, rather than with the value. This allows values to remain untagged except in contexts where their type is partially abstracted.

To perform an operation on a value whose type is partially abstracted, the compiler again produces code which dynamically accesses information specific to a type. As this information is associated with the particular type view, however, some of the computation required may be partially evaluated during compilation.

In the case of variant implementation, the compiler may statically evaluate the encoded label information that would be produced for the local type view. At execution time, if no tag exists for a particular context then this encoding is known to be correct and may be applied directly to the variant value. If tag information does exist, then the encoding calculated statically may be used to index a table in which the appropriate encoding for the particular value is stored. If there is no related encoding, then the value is known not to be of this label as the label did not exist in the context of the value's creation. These tables must be maintained whenever loss of type information occurs.

Figure 5.2.25 shows how a variant created as type *commercial* is represented from two type views, one of type *commercial* and one of type *vehicle*.

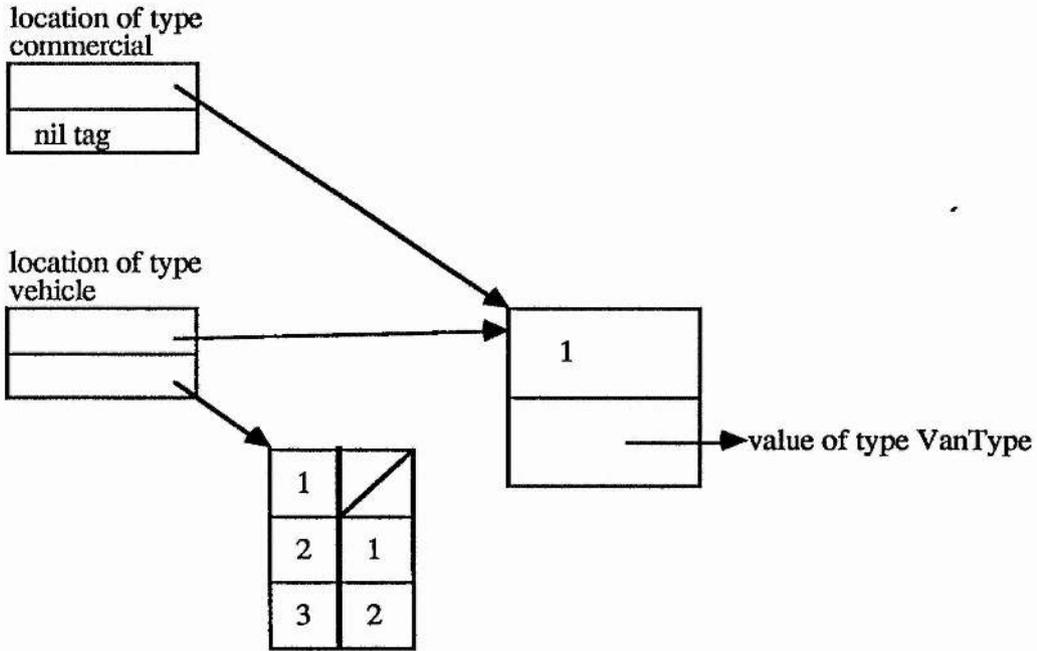


Figure 5.3.25 Partly tagged implementation of a *commercial*

This implementation of variants will be efficient for some systems, notably those where subtyping is only used occasionally. Once again, however, the importance of the example is in the generalisation of the different techniques, and whichever implementation is best for a particular type constructor and system will depend upon many factors.

5.3.8 Conclusions

In implementation terms, inclusion polymorphism in a programming system means that some type-specific operations are allowed upon values whose type may be partially abstracted over. The associated problems arise because many language implementations gain efficiency in the implementation of type specific operations by formatting values according to their specific type. This information must then be made dynamically available when an operation is performed upon a value whose full type is not known statically.

The three strategies for the implementation of inclusion polymorphism discussed were uniform, tagged, and partly tagged. Textual implementations of inclusion polymorphism may be most efficient, but are not suitable for systems with separate compilation or first-class procedure values. No textual solutions were discussed as no such implementations are known.

Uniform implementations do not rely upon specific type information for type specific operations, but may lose efficiency because of this. Tagged implementations allow a more efficient format, but must look up the relevant information dynamically. Partly tagged implementations also look up information dynamically, but some of this information may be partly evaluated statically, and values whose type is not abstracted do not require tagging.

Which of the techniques is most suitable depends upon both the type constructor, and the way in which a particular system is used. However, the given classification may be useful to a system implementor in deciding upon a suitable technique.

5.4 Other forms of polymorphism

Polymorphism may be defined as the ability of a single value to be viewed as more than one type. Thus with parametric polymorphism a universally quantified procedure may also be viewed as any of its specialised types, and with inclusion polymorphism any type may also be viewed as any of its supertypes.

There are some other programming language constructs which allow more than one type view, which may also be classified as polymorphism for this reason. These include overloading, often referred to as ad hoc polymorphism, and existential data types. Implementations of these forms of polymorphism may be

categorised in the same framework as solutions of parametric and inclusion polymorphism.

5.4.1 Ad hoc polymorphism

According to Strachey [Str67], in ad hoc polymorphism there is no single systematic way in which to determine the type of the result of a function from the type of its arguments. Burstall and Lampson [BL84] give an operational definition, stating that in ad hoc polymorphism the code executed depends on the type of the arguments. As an example, a **print** procedure executes different code when evaluating

```
print 42
```

and

```
print "42"
```

This procedure is polymorphic since it is defined over more than one type, but the code executed may be different in each case. Most ad hoc polymorphic procedures are "built in", that is supplied by the system, but both Keas [Kea88] and Wadler and Blott [WB89] have shown how user defined ad hoc polymorphism also may be defined.

Most programming languages contain some predefined ad hoc polymorphic procedures, such as overloaded arithmetic operators. However, very few languages allow abstraction over the type of the operands of such procedures. Notable exceptions to this are Napier88, where the equality operator may be used over any two values whose type is known to be the same, and Haskell [Hud89] where the user has the power to define ad hoc polymorphic functions. Standard ML also defines ad hoc equality over restricted types. These are the only known

examples of a system where the types of parameters to ad hoc polymorphic procedures may be abstracted over.

All other languages with overloading are able to compile different code for different instances of an overloaded operator, and as such may be classified as textual implementations.

It is interesting to note that architectures capable of supporting such type abstraction were available long before a type system which could model it. The Burrough's B6500 machine [HD68] supports ad hoc polymorphism for arithmetic operations. The architecture supports some polymorphic operations such as plus, minus, times, equal and not equal which operate according to a value's data tag. For example, both integers and reals have a times operation defined over them. On execution the machine instruction inspects the tag and performs either integer or real multiplication. Indeed, the real numbers themselves may be single or double precision, constituting a further variation.

Once again, the general classification of polymorphic architectures may be used to highlight the differences in the implementation techniques used for the three languages listed above. ML uses a tagged architecture, and Napier88 a partly tagged architecture. Haskell has a solution which does not fit neatly into any of the categories. However, it is stated in [WB88] that "the new system could be added to an existing language ... simply by writing a pre-processor", and so for now it will be described as textual.

One point that is immediately clear is that a uniform implementation is not feasible for ad hoc polymorphism, by definition. This would require an architecture to perform the same instructions for any application of a polymorphic operator; this is

incompatible with the definition of ad hoc polymorphism, which stated that a different semantic operation occurred according to the type of the operands.

5.4.1.1 Fully tagged ad hoc implementation

The language ML did not originally allow any abstraction over the types of parameters of ad hoc polymorphic functions. The language did contain ad hoc polymorphism, for arithmetic operators and equality, but the type inference system could not infer types for functions which contained such operations in their bodies. In Standard ML, it is allowable to write

```
3 + 4 ;
```

and

```
3.1 + 4.2 ;
```

but an attempt to abstract over such expressions in a function such as

```
fun plus ( a , b ) = a + b ;
```

fails. This is because the type of the overloaded operator "+" is statically unresolvable.

Equality in ML is defined over all data types except for functions. However, the same treatment of the equality operator was considered over restrictive. This would make it impossible, for example, to write a polymorphic function which determined whether a certain item was contained in a list of the same type. Such a function would be undefined for a list of functions, but useful for a list of any other type.

The problem was resolved by the introduction of "eqtypes", which allowed static type safety to be maintained. However, the discussion here is of implementation rather than static type checking, and so the problem addressed is how a test of

equality over values whose type is not known may be implemented within the uniform architecture of the Functional Abstract Machine.

The semantics of equality in ML require that values are tagged so that the equality operation may determine the correct treatment of the machine representations. As equality is "deep", it must be possible for the operator to distinguish between pointers and non-pointers. Furthermore, pointers which represent updatable locations are treated differently from pointers to data values, and so these must also be distinguished. This however presents no serious problems in the implementation of ML. As already shown, pointers must in any case be distinguishable by the architecture so that garbage collection may be performed. Apart from updatable locations, whose values may be restricted for identification, this is all the information that the equality operator needs to access.

5.4.1.2 Partly tagged ad hoc implementation

Napier88 has no static type mechanism which allows any general abstraction over ad hoc polymorphic operators. However, the operator "=" is defined in Napier88 over all types, and so its use is statically allowable within a universally quantified procedure. Napier88 has store semantics, and equality is defined as identity on all types but scalars. This allows the definition of equality over procedure types, as such values have an identity.

It is therefore allowable in Napier88 to write a procedure which abstracts over the type of the operands of an equality operation, for example

```
let equal = proc[ t ]( x, y : t → bool ) ; x = y
```

and the implementors must address this problem.

In the Napier88 implementation, as in ML, the solution is already largely provided by the mechanisms which implement parametric polymorphism. In ML, a

mechanism already existed to allow the distinction between pointers and non-pointers. Slightly more information than this is available in the Napier88 implementation, as any type which is abstracted over is represented by a tag in a known location within the procedure closure. Therefore the special case of equality may be implemented by building in a machine instruction which is able to dynamically determine all the necessary information about its operands.

5.4.1.3 Textual ad hoc implementation

Both the ML and Napier88 solutions for ad hoc implementation were for a single operator. Haskell has a type system which allows a user to define ad hoc polymorphic functions, and so a more general solution than either of these is required.

Haskell introduces type classes, which extend the Hindley/Milner type system used in Standard ML. The essence of the mechanism is to allow the definition of type classes over which overloaded operators may be introduced. For example, the class *Num* may be introduced over which the arithmetic operators for both integers and reals may be declared. This allows the declaration in Haskell of the *plus* function disallowed in Standard ML, as shown in Figure 5.4.1.

```
plus      :: Num a => a -> a
plus a b  = a + b
```

Figure 5.4.1 Using a type class in Haskell

The run-time implementation used by Haskell may be described as textual, in that all polymorphism is factored out by the language compiler in the translation of the concrete syntax to lambda-expression.

It has been stated that a textual implementation is not suitable for a polymorphic system with first-class procedure values, whereas Haskell is a functional language

with first class functions. In fact this classification is not entirely accurate, as the only reason that textual polymorphism is possible is that the implementation of parametric polymorphism in Haskell is uniform. Thus the compiler may treat any value of the abstracted type in the same way, and only the different code for the overloaded operators needs to be accessed according to the type. In fact, this is an implicit assumption in the claim that the technique may be used for any language.

Functions whose type is inferred as over a type class are compiled with an extra formal hidden parameter. This stands for a dictionary of all overloaded operators defined over the type class. A call to such a function must also pass the relevant dictionary, according to the type of the call. Where a function passes into a different type view, the function must be partially applied to the appropriate dictionary. As Haskell has no update, this partial application causes no problems.

The implementation of Haskell may also be likened to the implementation of parametric polymorphism in Napier88, with the difference that the partial application of a type parameter passes a dictionary of operators instead of a type tag. This dictionary acts as a high-level tag of the specific type information. Thus the implementation of ad hoc polymorphism in Haskell is really a mixture of all the different categories.

5.4.2 Existential quantification

Although not normally described as such, existentially quantified data types as proposed by Mitchell and Plotkin [MP88] are a form of polymorphism. They give the ability for a type to be abstracted over and so a value may be viewed from a number of different type contexts.

The details of implementation of such types will not be exposed in any detail for two reasons. The first is that the only known implementation of existentially

quantified types is in the language Napier88, and so no contrasting implementations may be described. The second reason is that the form of type abstraction available with existential types is the same as that with parametric polymorphism. Thus a value's type may be abstracted over in an all-or-nothing manner, and as such any implementation of parametric polymorphism may be extended to existential types.

This is the case in Napier88, where a value of a free existential quantifier type is treated in exactly the same manner as a value of a free universal quantifier type. As abstraction occurs within a structured value, instead of within a procedure invocation, the appropriate type tag may be attached to the value instead of being passed as a parameter. Thus the compiler is still able to calculate a static address at which the tag may be found. Apart from this difference, the implementations match precisely [MDC90].

5.5 Conclusions

Polymorphism is the ability within a system of a value to be viewed as more than one type. In all cases, this implies the ability to abstract over types. Most systems use different formats for values of different types, and rely upon type information for the generation of appropriate code. In a system with any kind of polymorphism, the necessary type information may be abstracted over and therefore not available statically.

There are two main categories of implementation techniques which may support polymorphic systems. The first of these is tagged systems, where the type information which is not available statically is somehow accessed dynamically by some dynamic indicator of type. This tag information may be associated either with a value or with the context of its abstraction. Tagged systems have been

further divided into those where all values are tagged, and those where tagging occurs only with the occurrence of type abstraction.

The second category of implementation techniques is untagged. In such systems, the type information which is not available dynamically is somehow dispensed with. These again fall into two main subcategories. Uniform implementations are those where no type information is required for an operation to be executed upon a value; this means that all values are represented with a single uniform format and the level of description of the architecture must be at the semantic level of the language. The other subcategory, textual implementations, are those in which the different type formats are somehow factored out during the compilation, so that different code may be specified for the same operation in different static contexts. The diagram from the introduction of this Chapter is reproduced as Figure 5.5.1.

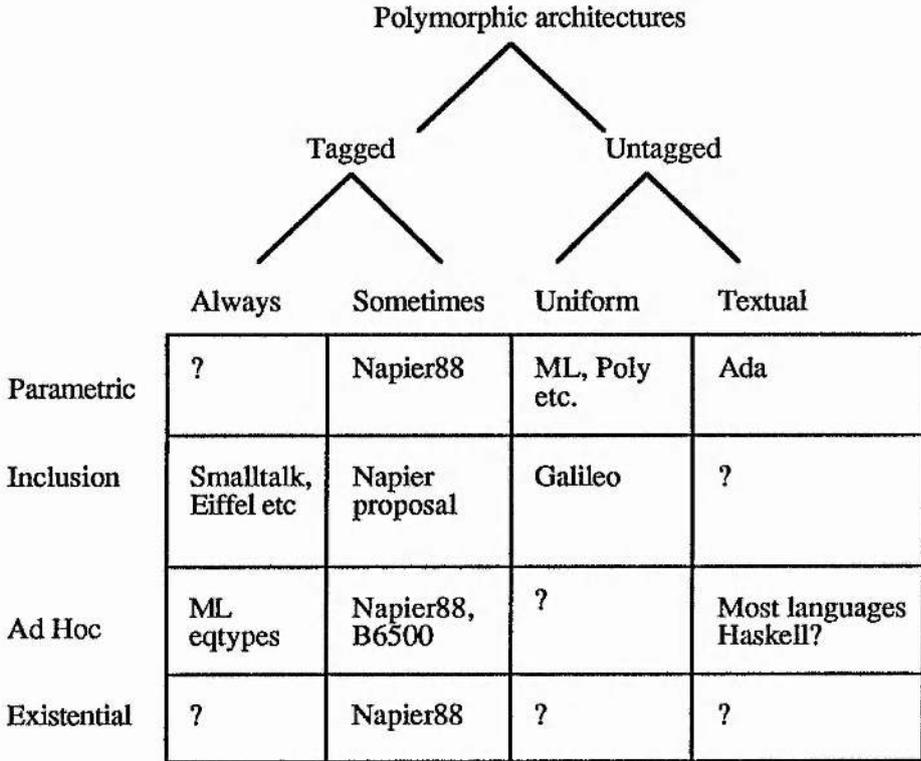


Figure 5.5.1 A classification of polymorphic implementation

Four categories of polymorphism within programming languages have been examined. Of these, the implementation techniques used in different languages for parametric and inclusion polymorphism have been discussed in detail. New techniques have been described for the implementation of all four categories of polymorphism; every one uses an implementation where data values are tagged by their context only when their types are abstracted. This has the major benefit that programs which do not use polymorphism pay little or no cost for its inclusion in a programming system.

The diagram in Figure 5.5.1 gives only a few examples of the implementations of polymorphic programming languages. The implementation of most of these languages has been discussed. It is interesting to consider the categories where there are no known implementations.

The reason for no textual implementation of inclusion polymorphism has already been proposed as a historical accident: the only languages implemented with inclusion polymorphism have other features which make a textual implementation unsuitable. No uniform implementation of ad hoc polymorphism has also been stated as a consequence of the definition of ad hoc polymorphism, which makes a uniform implementation impossible. For existential polymorphism, only a single implementation is known. The last gap, no fully tagged implementation of parametric polymorphism, is also not surprising. Such an implementation would be feasible, but the manipulation of values of unknown format implies that no static information would be available to a compiler for addressing information.

The classification is not intended to be absolute, and as stated is a simplification. A demonstration of this is the implementation of ad hoc polymorphism in Haskell, which may be partly placed in three different categories. In fact, all of the languages described are known to be implemented on untagged Von Neumann

machines with different formats for integers and floating point numbers, and may therefore be classified as textual at some level of abstraction! However, the categories are believed to be useful for descriptive purposes.

6 Conclusions

The motivation for the research described in this thesis is the development of technology to help with the construction of large and long-lived applications building systems. The intrinsic problems of building applications which deal with large bodies of long lived data have been identified, and these have been contrasted with unnecessary problems introduced by the failure of current technology. The lack of integration within the various support systems used in a traditional applications building environment leads to avoidable complexity for the applications builder.

Persistent programming systems have been identified as one way of avoiding this complexity. Persistent programming languages originated to solve the problems of a programmer having to maintain more than a single conceptual map between real world data and its representation in a computer system, and have evolved to a model which may support all desired computation.

For a persistent system to be able to specify all programming activity on large bodies of data, it requires a flexible binding mechanism and a rich type system. To preserve the integrity of a persistent store, the type system must be strongly enforced. This is best achieved by making the type system constraints as static as possible, although at least some dynamic type checking is considered necessary to allow programs which change the type of the persistent store. The use of mostly static typing may cause a loss of flexibility, but the effects of this may be limited by the use of polymorphism.

The issues of persistence and binding have been investigated before in detail. The ideas introduced in this thesis concentrate on the issues concerned with adding a rich type system to a persistent system with a flexible binding mechanism.

Two major themes have been discussed. Chapters 2 and 3 concentrate upon the methodologies which may be used for building applications systems from a richly typed persistent system. Chapters 4 and 5 concentrate upon implementation techniques for the building of such persistent systems.

First, a short synopsis of the research upon which the thesis is based, and some more which may derive from the topics in it, will be given.

6.1 Related work

6.1.1 Past

The persistence of data was first identified as being orthogonal to its other attributes by Atkinson in [Atk78]. After some attempts to provide persistence in other languages, S-algol [Mor82] was identified as a suitable candidate and the first implementation of PS-algol, the first persistent language, was completed in 1980 [ACC81]. Napier, a language based on PS-algol but with a much richer type system, was first described in general terms in [AM85a], and the Napier88 system [MBC89] was released in 1988. A plethora of other persistent languages has also appeared since 1985 [MS89, Eve85, RC89, ACO85, Mat85, DM90, Car85, GMD85, HS89].

The need for flexible, incremental binding mechanisms in a persistent system was identified in [ABC83], and the relationship between binding and typechecking in [ABM88]. Binding mechanisms in Napier were first discussed in [AM85a], and refined to the form of Napier88 environments [Dea89]. The description in Chapter 2 uses the Napier88 environment mechanism as an example in a more general discussion of software system construction paradigms. The particular example of a construction method given is due to [CD90].

Structural typechecking algorithms of the kind described for the Napier88 system in Chapter 4 appeared in the implementation of S-algol, and later PS-algol. Both of these systems build graph representations of types during compilation, and a similar recursive algorithm traverses the graphs to test equality. Both languages use a string type representation to check types dynamically. However, the languages have relatively simple type systems, which do not include for example recursive, parameterised or quantified types. The description in this thesis mainly shows how the technique may be used for a more descriptive type system and algebra.

The discussion of implementation techniques for polymorphic programming languages draws from the descriptions of some of the many successfully implemented languages [Mil83, Lis81, DD79, Mat85, BMS80, Fai82, Tur87, ACO85, SZ86, GR83, Hud90], as well as a number of claims, counter-claims, admissions and denials during conversations with implementors and their friends.

6.1.2 Future

In the FIDE (Esprit II Basic Research Action 3070) project, a number of partners are working on the problems of type systems for programming in the large. Work at the University of Pisa on objects and modules [AGO90] relates to Chapters 2 and 4. Collaborators at INRIA [ALR90] are working on a general polymorphic model of bulk data types, which may relate strongly to ideas in Chapters 4 and 5.

Binding is still an important issue in persistent systems, particularly in the context of system evolution. The discussion in Chapters 2 and 3 shows how existential types may be used both as an abstraction over implementation, and as a protection mechanism. The combination of these concepts may lead to a further model of binding in persistent systems which could allow an "organic" system, where meta-

data may evolve incrementally and outdated schema information could gradually disappear from the system.

Some of the results shown in Chapter 4 suggest that the limit of expressiveness of current type systems is being reached. One way of writing strongly-typed programs which may exceed the expressiveness of a given type system is by the use of reflection [DB88, SFS90]. The disadvantage of this technique is that it appears to be intrinsically difficult for a programmer to understand. A worthy area of investigation is in the interaction between the expressiveness of a type system and its use in a reflexive system.

It is shown in Chapter 5 how the least general form of polymorphic implementation, that of source code manipulation, is also the most efficient. Efficiency in all the other implementation techniques shown is a major problem, and programmers are known not to use constructs that they know to be inefficient. There is no reason why a mixed implementation strategy should not be used, perhaps with commonly used polymorphic constructs being optimised incrementally. This topic of incremental optimisation of persistent code is still in its infancy.

6.2 Applications building systems

6.2.1 Software reuse

6.2.1.1 Dynamic L-value binding

The use of units of dynamic and L-value binding, along with first-class persistent procedure values, may be used to solve some of the problems associated with the incremental construction of large programming systems. In particular, software componentry may be shared, rather than copied, whilst maintaining the benefits of

a strongly typed programming language. Apart from orthogonal persistence, two other language facilities are required to allow this.

A flexible, incremental binding mechanism should be able to model a number of different methodologies for the binding of program components. In particular, such a mechanism should allow the incremental construction and evolution of a complex software system. So that a system constructed incrementally may be released, it should also be possible to bind the components together tightly at a later stage.

Much of the description of binding and Napier88 environments is based on research by Dearle [Dea89], and the given example of a system construction paradigm is due to Cutts and Dearle [CD90]. However, this is the first general description of the way that the units of binding may be composed to provide the flexibility shown.

The examples also show a use of structural type equivalence checking over independently prepared programs, as discussed in Chapter 4.

6.2.1.2 Quantification over dynamic bindings

Type abstraction is also necessary to maximise the sharing of software componentry, as it allows the specification of procedures that do not depend upon a full type description of their parameters. Universal quantification allows abstraction over the type of a procedure itself, and existential quantification over the type of a parameter's implementation. Once again, the examples show instances where these types must be checked dynamically for structural equivalence.

6.2.2 Protection and viewing

6.2.2.1 Using types to hide information

The integrity of persistent data is of the utmost importance, and must be protected as far as possible. Traditionally protection is provided by a two-level model, with different schemes in operation for data within programs and permanent data. In a persistent system this difference does not exist, and a single protection mechanism must be used for all data within the system.

As all data within a persistent system are subject to type system constraints, no unlawful access to the data may occur except in the event of system failure. This gives the aspect of type safety to permanent data, which may not be achieved in a non-persistent system.

A further effect of the type safety of all data is that any protection which is programmed may not be revoked by a lower level of technology. This means that access protection and software constraints may be programmed as suitable for the individual data. A number of different language mechanisms may be used to protect permanent and shared data, including subtype inheritance, procedural encapsulation, and abstract data types [MBC90].

6.2.2.2 Viewing with existentially quantified types

The protection of data in database systems is normally achieved by integrity constraints and viewing mechanisms. In a persistent system, however, such mechanisms do not need to be specially provided as they may be programmed within a persistent programming language. Integrity constraints may be programmed within procedural encapsulation, and viewing mechanisms may be programmed with a combination of encapsulation and existentially quantified data types.

Existential types have a number of advantages over traditional viewing mechanisms. They are statically type checkable, and as only the type of data and program is abstracted over no change in efficiency need be associated with the mechanism. The referential integrity provided by a persistent store ensures that views occur over the same instance of data, and the copying of data, with its associated integrity problems, is not necessary. Lastly, the same existentially quantified type may abstract over more than one representation type in the same context, which allows data representations to change incrementally.

6.3 Programming system implementation

6.3.1 Typechecking across persistent systems

6.3.1.1 Type models

If the traditional database schema is regarded as a type, there is a requirement for the efficient manipulation of types in a persistent system. This is to allow provision of the facilities traditionally found in DBMS for schema editing, use and evolution [CBC90].

Two models of type equivalence are in common use: name equivalence and structural equivalence. It has been shown that while name equivalence schemes are easier to implement and are more efficient they still have to use structural checks to provide important facilities such as schema merging. On the other hand structural equivalence, generally more flexible and less efficient, can often achieve the same performance as name equivalence.

The details of schema merging still remain an outstanding problem: it is clear that structural checking is necessary, but not sufficient. A related topic for future research is the sharing of type definitions between modules. This is necessary for

name equivalence systems, and causes problems with the resolution of the meaning of a type name.

6.3.1.2 Structural checking across store

One important topic of research is how to improve the performance of structural equivalence checking. How such checking may be performed has been fully investigated. The balance between constructing efficient representations of types in terms of store and the speed of the equivalence algorithm in comparing two representations is described.

The combination of parameterisation and recursion allows much expressive power in a type algebra. The algorithms described for the equivalence testing of such types are believed to be fully decidable for a greater class of types than those described elsewhere. Some useful types are still disallowed, and it is not even known whether a decidable algorithm exists for such types. It is clear that a greater class of types could be decidable checked for equivalence, if not all types expressible with such an algebra.

6.3.2 Implementation of polymorphism

6.3.2.1 General problems and classification of solutions

Values of different types frequently have different machine-level representations for reasons of efficiency. The difficulty of implementing polymorphic systems stems from the fact that, as the type may be abstracted over, the representation of a value is not always known statically. This causes problems with all kinds of polymorphism.

Solutions to the general problems of type information loss have been categorised into a system which is useful at least for the description of such implementations.

The four categories of implementation have been described as textual, uniform, sometimes tagged and always tagged. Two new mechanisms, both classified as "sometimes tagged", are proposed here for the implementation of polymorphism.

6.3.2.2 Parametric implementation

One of these implementation mechanisms solves the problems encountered when a fixed number of operations is available on a value whose type is fully abstracted over. This is suitable for the implementation of parametric polymorphism, ad hoc polymorphism, and existential data types, and works by changing values into uniform formats only in contexts where their type is abstracted over. This mechanism has been successfully used in the implementation of Napier88.

6.3.2.3 Inclusion implementation

The other proposal is for a mechanism which allows type-specific operations to be performed on a value whose type is partially abstracted over, and may therefore be used to implement inclusion polymorphism. This mechanism may also be useful in object-oriented languages as a solution to object addressing with multiple inheritance. An implementation of this mechanism has not yet been built, mainly because the writing of this thesis became more important. An implementation of inclusion polymorphism within the Napier system is proposed for the near future.

6.4 Some final words

The motivation for the research contained in this thesis is the development of technology to help with the construction of better applications building systems. Applications, however, are built by programmers, who write them using systems provided by computer manufacturers. This leaves the outstanding question of whether computer manufacturers will ever invest in persistent systems.

Even if they do not, the work will not have been in vain. Some of it is applicable in the general context of large applications systems, and some of it is applicable to systems which are not large. Most importantly, it has provided, and will continue to provide, interesting problems to wake up to each day [Mor90].

But will computer manufacturers invest in persistent technology? Large manufacturers seem unlikely to, as they must follow and not lead market demand, and therefore continue to produce outdated technology such as Fortran compilers and *qwerty* keyboards.

However, recent developments have shown that, in the computer market, a small forward-thinking computer company which backs a winning concept from the research community may become a large company in a very short time. Examples of such winning concepts may be seen in graphical user interfaces and networked workstations.

So meanwhile we will continue to work on our interesting problems, in the hope that, one day, all computer systems will be made this way...

Appendix I

Napier88 Context Free Syntax

Session:

<session> ::= <sequence>?
<sequence> ::= <declaration>[;<sequence>] | <clause>[;<sequence>]
<declaration> ::= <type_decl> | <object_decl>

Type declarations:

<type_decl> ::= **type**<type_init> |
rec type<type_init>[&<type_init>]*
<type_init> ::= <identifier>[<type_parameter_list>]is<type_id>
<type_parameter_list> ::= <lsb><identifier_list><rsb>

Type descriptors:

<type_id> ::= **int** | **real** | **bool** | **string** | **pixel** | **pic** |
null | **any** | **env** | **image** | **file** |
<identifier>[<parameterisation>] |
<type_constructor>
<parameterisation> ::= <lsb><type_identifier_list><rsb>
<type_identifier_list> ::= <type_id>[,<type_identifier_list>]
<type_constructor> ::= <star><type_id> | <structure_type> |
<variant_type> | <proc_type> | <abstype>
<structure_type> ::= **structure**([<named_param_list>])
<named_param_list> ::= [**constant**]<identifier_list>:<type_id>
[;<named_param_list>]
<variant_type> ::= **variant**([<variant_fields>])
<variant_fields> ::= <identifier_list>:<type_id>[;<variant_fields>]
<proc_type> ::= **proc**[<type_parameter_list>]([<parameter_list>]
[<arrow><type_id>])
<parameter_list> ::= <type_id>[,<parameter_list>]
<abstype> ::= **abstype**<type_parameter_list>
(<named_param_list>)

Object declarations:

<object_decl> ::= **let**<object_init> |
rec let<rec_object_init>[&<rec_object_init>]*
<object_init> ::= <identifier><init_op><clause>
<rec_object_init> ::= <identifier><init_op><literal>
<init_op> ::= = | :=

Clauses:

<clause> ::= <env_decl> |
if<clause>**do**<clause> |
if<clause>**then**<clause>**else**<clause> |
repeat<clause>**while**<clause>[**do**<clause>] |
while<clause>**do**<clause> |
for<identifier>=**clause****to**<clause>
[**by**<clause>]**do**<clause> |
use<clause>**with**<signature>**in**<clause> |
use<clause>**as**<identifier>[<witness_decls>]
in<clause> |
case<clause>**of**<case_list>**default** :<clause> |
<raster> |
drop<identifier>**from**<clause> |
project<clause>**as**<identifier>
onto<project_list>**default**:<clause> |
<name>:=<clause> |
<expression>
<signature> ::= <named_param_list>
<witness_decls> ::= <type_parameter_list>
<case_list> ::= <clause_list>:<clause>;[<case_list>]
<raster> ::= <raster_op><clause>**onto**<clause>
<raster_op> ::= **ror** | **rand** | **xor** | **copy** | **nand** | **nor** | **not** | **xnor**
<project_list> ::= <any_project_list> | <variant_project_list>
<any_project_list> ::= <type_id>:<clause>;[<any_project_list>]
<variant_project_list> ::= <identifier>:<clause>;[<variant_project_list>]
<env_decl> ::= **in**<clause>**let**<object_init> |
in<clause>**rec let**<rec_object_init>
[&<rec_object_init>]*

Expressions:

<expression> ::= <exp1>[**or**<exp1>]*
<exp1> ::= <exp2>[**and**<exp2>]*
<exp2> ::= [~]<exp3>[<rel_op><exp3>]
<exp3> ::= <exp4>[<add_op><exp4>]*
<exp4> ::= <exp5>[<mult_op><exp5>]*
<exp5> ::= [<add_op>]<exp6>

<exp6>	::=	<literal> <value_constructor> (<clause>) begin <sequence> end {<sequence>} <expression>(<clause><bar><clause>) <expression>(<dereference>) <expression>'<identifier> <expression><lsb><specialisation><rsb> <expression>([<application>]) <structure_creation> <variant_creation> <clause> contains [constant <identifier>[:<type_id>] any (<clause>) <name>
<dereference>	::=	<clause>[,<dereference>]
<specialisation>	::=	<type_identifier_list>
<application>	::=	<clause_list>
<structure_creation>	::=	<identifier>[<lsb><specialisation><rsb>] ([<clause_list>])
<variant_creation>	::=	<identifier>[<lsb><specialisation><rsb>] (<identifier>:<expression>)
<name>	::=	<identifier> <expression>(<clause_list>)[(<clause_list>)]*
<clause_list>	::=	<clause>[,<clause_list>]

Value constructors:

<value_constructor>	::=	<vector_constr> <structure_constr> <image_constr> <subimage_constr> <picture_constr> <picture_op>
<vector_constr>	::=	[constant] vector <vector_element_init>
<vector_element_init>	::=	<range> of <clause> <range> using <clause> @<clause> of <lsb><clause>[,<clause>]*<rsb>
<range>	::=	<clause> to <clause>
<image_constr>	::=	[constant] image <clause> by <clause><image_init>
<image_init>	::=	of <clause> using <clause>
<subimage_constr>	::=	limit <clause>[to <clause> by <clause>] [at <clause>,<clause>]
<structure_constr>	::=	struct ([<struct_init_list>])
<struct_init_list>	::=	<identifier><init_op><clause>[,<struct_init_list>]
<picture_constr>	::=	<lsb><clause>,<clause><rsb>
<picture_op>	::=	shift <clause> by <clause>,<clause> scale <clause> by <clause>,<clause> rotate <clause> by <clause> colour <clause> in <clause> text <clause> from <clause>, <clause> to <clause>,<clause>

Literals:

<literal>	::=	<int_literal> <real_literal> <bool_literal> <string_literal> <pixel_literal> <picture_literal> <null_literal> <proc_literal> <image_literal> <file_literal>
<int_literal>	::=	[<add_op>]<digit>[<digit>]*
<real_literal>	::=	<int_literal>.[<digit>]*[e<int_literal>]
<bool_literal>	::=	true false
<string_literal>	::=	<double_quote>[<char>]*<double_quote>
<char>	::=	any ASCII character except " <special_character>
<special_character>	::=	<single_quote><special_follow>
<special_follow>	::=	n p o t b <single_quote> <double_quote>
<pixel_literal>	::=	on off
<null_literal>	::=	nil
<proc_literal>	::=	proc [<type_parameter_list>]([<named_param_list>] [<arrow><type_id>]);<clause>
<picture_literal>	::=	nilpic
<image_literal>	::=	nilimage
<file_literal>	::=	nilfile

Miscellaneous and microsyntax:

<lsb>	::=	[
<rsb>	::=]
<star>	::=	*
<bar>	::=	
<add_op>	::=	+ -
<mult_op>	::=	<int_mult_op> <real_mult_op> <string_mult_op> <pic_mult_op> <pixel_mult_op>
<int_mult_op>	::=	<star> div rem
<real_mult_op>	::=	<star> /
<string_mult_op>	::=	++
<pic_mult_op>	::=	^ ++
<pixel_mult_op>	::=	++
<rel_op>	::=	<eq_op> <co_op> <type_op>
<eq_op>	::=	= ~=
<co_op>	::=	< <= > >=
<type_op>	::=	is isnt
<arrow>	::=	->

<single_quote>	::=	'
<double_quote>	::=	"
<identifier_list>	::=	<identifier>[,<identifier_list>]
<identifier>	::=	<letter>[<id_follow>]
<id_follow>	::=	<letter>[<id_follow>] <digit>[<id_follow>] [<id_follow>]
<letter>	::=	a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
<digit>	::=	0 1 2 3 4 5 6 7 8 9

Appendix II

Napier88 Type Rules

Meta-rules

type arith is int | real
type ordered is arith | string
type literal is ordered | bool | pixel | pic |
 null | proc | file
type nonvoid is literal | image | structure |
 variant | env | any | abstype |
 parameterised | poly | *nonvoid
type type is nonvoid | void

Session :

<sequence> : void ? => void
t : type, <declaration> : void ; <sequence> : t => t
t : type, <clause> : void ; <sequence> : t => t
t : type, <clause> : t => t

Object Declarations :

<declaration> => void
where <object_decl> ::= [in<clause> : env]let<object_init> |
 [in<clause> : env]rec let<rec_object_init>
 [&<rec_object_init>]*
where <object_init> ::= <identifier><init_op><clause> : nonvoid
where <rec_object_init> ::= <identifier><init_op><literal> : nonvoid
where <init_op> ::= = | :=

Clauses :

<clause> : env contains [constant]<identifier>[:<type_id>] => bool
if <clause> : bool do <clause> : void => void
t : type, if <clause> : bool then <clause> : t else <clause> : t => t
repeat <clause> : void while <clause> : bool [do <clause> : void] => void
while <clause> : bool do <clause> : void => void
for <identifier>=<clause> : int to <clause> : int
 [by<clause> : int]do<clause> : void => void

t : pixel, <expression> : t <pixel_mult_op> <expression> : t => t
 where <pixel_mult_op> ::= ++
t : literal, <literal> : t => t
t : nonvoid, <value_constructor> : t => t
t : type, (<clause> : t) => t
t : type, **begin** <sequence> : t **end** => t
t : type, { <sequence> : t } => t
<expression> : string (<clause> : int <bar> <clause> : int) => string
<expression> : image (<clause> : int <bar> <clause> : int) => image
<expression> : pixel (<clause> : int <bar> <clause> : int) => pixel
t : nonvoid, <expression> : *t (<clause> : int) => t

Value constructors:

t : nonvoid, **vector**<range>**of**<clause> : t => *t
t : nonvoid, **vector**<range>**using**<clause> : proc(int -> t) => *t
t : nonvoid, **vector**@<clause> : int **of**<lsb><clause> : t
 [, <clause> : t]* <rsb> => *t
 where <range> ::= <clause> : int **to** <clause> : int
image <clause> : int **by**<clause> : int **of** <clause> : pixel => image
image <clause> : int **by**<clause> : int **using** <clause> : image => image
limit<clause> : image [**to**<clause> : int **by**<clause> : int]
 [**at**<clause> : int , <clause> : int] => image
struct(<struct_init_list>) => structure
 where <struct_init_list> ::= <identifier><init_op><clause> : nonvoid
 [, <struct_init_list>]
<lsb><clause> : real , <clause> : real <rsb> => pic
shift<clause> : pic **by**<clause> : real , <clause> : real => pic
scale<clause> : pic **by**<clause> : real , <clause> : real => pic
rotate<clause> : pic **by**<clause> : real => pic
colour<clause> : pic **in**<clause> : pixel => pic
text<clause> : string **from**<clause> : real , <clause> : real
to<clause> : real , <clause> : real => pic

Literals :

[<add_op>]<digit>[<digit>]* => int

<int_literal>.[<digit>]*[e<int_literal>] => real

true | **false** => **bool**

<double_quote>[<char>]*<double_quote> => string

on | **off** => pixel

nil => null

t : type, **proc**[<type_parameter_list>]([<named_param_list>]

[<arrow><type_identifier> : t]);<clause> : t

nilpic => pic

nilimage => image

nilfile => file

Appendix III

The Napier88 Typechecker Interface

```
rec type list[ t ] is variant( cons : structure( hd : t ; tl : list[ t ] ) ; tip : null )
```

```
type NapierTypes is abstype[ TYPE,TypeName ]
```

```
(  
    Vector,Structure,Variant,Proc,  
    Parameterised,Parameter,  
    Quantified,Quantifier,  
    Abstract,Witness,  
    Recursive          : TypeName ;  
  
    BaseType          : proc( string -> TYPE ) ;  
    MkVector          : proc( TYPE -> TYPE ) ;  
    NewStructure      : proc( -> TYPE ) ;  
    AddField          : proc( TYPE,string,TYPE -> TYPE ) ;  
    NewVariant        : proc( -> TYPE ) ;  
    AddBranch         : proc( TYPE,string,TYPE -> TYPE ) ;  
    MkProc            : proc( list[ TYPE ],TYPE -> TYPE ) ;  
    MkParameterised,  
    MkQuantified,  
    MkAbstract        : proc( TYPE,list[ TYPE ] -> TYPE ) ;  
    MkParameter,  
    MkQuantifier,  
    MkWitness         : proc( int -> TYPE ) ;  
    MkRecursive       : proc( string -> TYPE ) ;  
  
    Elms              : proc( TYPE -> TYPE ) ;  
    Args              : proc( TYPE -> list[ TYPE ] ) ;  
    Result            : proc( TYPE -> TYPE ) ;  
    ScanStructure     : proc( TYPE -> proc( -> string ) ) ;  
    FieldType         : proc( TYPE,string -> TYPE ) ;  
    AbsFieldType      : proc( TYPE,string -> TYPE ) ;  
    BranchType        : proc( TYPE,string -> TYPE ) ;  
    BranchNumber      : proc( TYPE,string -> TYPE ) ;  
    NoOfSubstitutions : proc( TYPE -> int ) ;  
  
    Substitute        : proc( TYPE,*TYPE -> TYPE ) ;  
    BindAbs           : proc( TYPE,*TYPE -> TYPE ) ;  
    ReplaceRecursions : proc( list[ TYPE ],proc( string -> TYPE ) ) ;  
    Commute           : proc( TYPE -> TYPE ) ;  
  
    IsType            : proc( TypeName,TYPE -> bool ) ;  
    EqualType         : proc( TYPE,TYPE -> bool ) ;  
  
    Display           : proc( TYPE -> string )  
)
```

References

- [ABC76] M.M. Astrahan et al.
"System R: A Relational Approach to Data Management"
ACM ToDS 1, 2 (June 1976) pp 97 - 137
- [ABC83] M.P. Atkinson, P. Bailey, K.J. Chisholm, W.P. Cockshott and R. Morrison
"An Approach to Persistent Programming"
The Computer Journal 26, 4 (1983) pp 360 - 365
- [ABM88] M.P. Atkinson, O.P. Buneman and R. Morrison
"Binding and Typechecking in Database Programming Languages"
Computer Journal 31, 2 (March 1988) pp 99 - 109
- [ACC81] M.P. Atkinson, K.J. Chisholm and W.P. Cockshott
"PS-algol: an Algol with a Persistent Heap"
ACM SIGPLAN Notices 17, 7 (July 1981) pp 24 - 31
- [ACO85] A. Albano, L. Cardelli and R. Orsini
"Galileo: a Strongly Typed, Interactive Conceptual Language"
ACM ToDS 10, 2 (1985) pp 230 - 260
- [AGO90] A. Albano, G. Ghelli and R. Orsini
Discussion with other collaborators at FIDE Types Club Meeting,
Pisa, March 1990.
- [ALR90] M.P. Atkinson, C. Lécluse and P. Richard
Discussion with other collaborators at FIDE Types Club Meeting,
St Andrews, November 1990.

- [AM85] M.P. Atkinson and R. Morrison
"Procedures as Persistent Data Objects"
ACM ToPLaS 7, 4 (October 1985) pp 539 - 559
- [AM85a] M.P. Atkinson and R. Morrison
"Types, Bindings and Parameters in a Persistent Environment"
In M.P. Atkinson, O.P. Buneman and R. Morrison (editors)
"Data Types and Persistence", Springer - Verlag (1988) pp 1 - 24
- [AMP86] M.P. Atkinson, R. Morrison and G.D. Pratten
"Designing a Persistent Information Space Architecture"
Proc. Information Processing 1986 (September 1986) pp 115 -
119
- [ANS78] American National Standard Programming Language Fortran
ANSI X3.9-1978, New York (1978)
- [Atk78] M.P. Atkinson
"Programming Languages and Databases"
Proc. 4th International Conference on Very Large Data Bases,
Berlin
In S.P. Yao (editor), IEEE (September 1978) pp 408 - 419
- [Bai89] P. Bailey
"Performance Evaluation in a Persistent Object System"
In J. Rosenberg and D. Koch (editors)
"Persistent Object Systems, Newcastle, Australia 1989"
Spinger-Verlag (1990) pp 373 - 385

- [BBB88] F. Bancilhon, G. Barbedette, V. Benzakin, C. Delobel, S. Gamerman, C. Lecluse, P. Pfeffer, P. Richard and F. Velez
"The Design and Implementation of O₂, an Object-Oriented Database System"
In K.R. Dittrich (editor)
"Advances in Object-Oriented Database Systems"
LNCS Vol. 334, Springer - Verlag (September 1988) pp 1 - 22
- [BBH63] D.W. Barron, J.N. Buxton, D.F. Hartley, E. Nixon and C. Strachey
"The Main Features of CPL"
The Computer Journal 6 (1963) pp 134 - 143
- [BBO89] V. Breazu-Tannen, O.P. Buneman and A. Ogori
"Can Object-Oriented Databases be Statically Typed?"
In R. Hull, R. Morrison and D. Stemple (editors)
Proc. 2nd International Workshop on Database Programming Languages
Morgan - Kaufmann (1989) pp 226 - 237
- [BC85] A.L. Brown and W.P. Cockshott
"The CPOMS Persistent Object Management System"
Universities of Glasgow and St Andrews PPRR-13-85 (1985)
- [BCC88] A.L. Brown, R. Carrick, R.C.H. Connor, A. Dearle and R. Morrison
"The Persistent Abstract Machine"
University of St Andrews PPRR-59-88 (1988)

- [BDM73] G.M. Birtwistle, O.J. Dahl, B. Myrhaug and K. Nygaard
"SIMULA BEGIN"
Auerbach (1973)
- [BL84] R. Burstall and B. Lampson
"A Kernel Language for Abstract Data Types and Modules"
Proc. International Symposium on the Semantics of Data Types
LNCS Vol. 173, Springer - Verlag (1984)
- [BMM80] P.J. Bailey, P. Maritz and R. Morrison
"The S-algol Abstract Machine"
University of St Andrews CSR-80-2 (1980)
- [BMS80] R. Burstall, D. McQueen and D. Sanella
"Hope: an Experimental Applicative Language"
ACM Lisp Conference, New York (1980) pp 136 - 143
- [Car83] L. Cardelli
"The Functional Abstract Machine"
Polymorphism (The ML/LCF/Hope Newsletter) 1, 1 (January
1983)
- [Car84] L. Cardelli
"A Semantics of Multiple Inheritance"
Proc. International Symposium on the Semantics of Data Types
LNCS Vol. 173, Springer - Verlag (1984) pp 51 - 67
- [Car85] L. Cardelli
"Amber"
Technical Report AT7T, Bell Labs, Murray Hill, USA (1985)

- [Car89] L. Cardelli
"Typeful Programming"
DEC SRC Technical Report No. 45 (May 1989)
- [CBC89] R.C.H. Connor, A.L. Brown, R. Carrick, A. Dearle and R. Morrison
"The Persistent Abstract Machine"
In J. Rosenberg and D. Koch (editors)
"Persistent Object Systems, Newcastle, Australia 1989"
Springer-Verlag (1990) pp 353 - 366
- [CBC90] R.C.H. Connor, A.L. Brown, Q.I. Cutts, A. Dearle, R. Morrison and J. Rosenberg
"Type Equivalence Checking in Persistent Systems"
Proc. 4th International Workshop on Persistent Object Systems, Martha's Vineyard, Massachusetts (September 1990) pp 151 - 164
- [CD90] Q.I. Cutts and A. Dearle
Private communication
- [CDM89] R.C.H. Connor, A. Dearle, R. Morrison and A.L. Brown
"An Object Addressing Mechanism for Statically Typed Languages with Multiple Inheritance"
OOPSLA89 - 4th International Conference on Object-Oriented Programming: Systems, Languages and Applications
in ACM SIGPLAN Notices 24, 10 (October 1989) pp 279-286

- [CDM90] R.C.H. Connor, A. Dearle, R. Morrison and A.L. Brown
"Existentially Quantified Types as a Database Viewing Mechanism"
Proc. International Conference on Extending Database Technology,
Fondazione Cini, Venice (March 1990) pp 301 - 315
- [CL88] T. Connors and P. Lyngbaek
"Providing Uniform Access to Heterogeneous Information Bases"
In K.R.Dittrich (editor)
LNCS 334, Springer-Verlag (September 1988) pp 334-339
- [CM88] L. Cardelli and D. McQueen
"Persistence and Type Abstraction"
In M.P. Atkinson, O.P. Buneman and R. Morrison (editors)
"Data Types and Persistence", Springer - Verlag (1988) pp 31 - 41
- [Con88] R.C.H. Connor
"The Napier Type-Checking Module"
University of St Andrews PPRR-58-88 (1988)
- [CW85] L. Cardelli and P. Wegner
"On Understanding Types, Data Abstraction and Polymorphism"
ACM Computing Surveys 17, 4 (December 1985) pp. 471 - 523
- [DD79] A. Demers and J. Donahue
"Revised Report on Russell"
Cornell University TR79-389 (1979)
- [Dea87] A. Dearle
"A Persistent Architecture Intermediate Language"
University of St Andrews PPRR-37-87 (1987)

- [Dea88] A. Dearle
"On the Construction of Persistent Programming Environments"
Ph.D. Thesis, University of St Andrews (1988)
- [Dea89] A. Dearle
"Environments: a Flexible Binding Mechanism to Support System Evolution"
Proc. 22nd Hawaii International Conference on Systems Sciences,
Hawaii (January 1989) pp 46 - 55
- [DB88] A. Dearle and A.L. Brown
"Safe Browsing in a Strongly-Typed Persistent Environment"
Computer Journal 31, 2 (March 1988) pp 540 - 544
- [DM81] A.J.T. Davie and R. Morrison
"Recursive Descent Compiling"
Ellis - Horwood Press (1981)
- [DM90] A.J.T. Davie and D.J. McNally
"The Staple Language Reference Manual"
University of St Andrews CS/90/16 (1990)
- [DvH66] J.B. Dennis and E.C. van Horn
"Programming Semantics for Multiprogrammed Computations"
CACM 9, 3 (1966) pp 143 - 145
- [Eve85] M. Evered
"Leibniz - a Language to Support Software Engineering"
Ph.D. thesis, Technical University of Darmstadt (1985)

- [Fai82] J. Fairbairn
"Ponder and its Type System"
University of Cambridge Technical Report 31 (1982)
- [Fai88] J. Fairbairn
"A New Type-Checker for a Functional Language"
In M.P. Atkinson, O.P. Buneman and R. Morrison (editors)
"Data Types and Persistence", Springer - Verlag (1988) pp 69 - 87
- [Fel79] S. I. Feldman
"Make – A Program for Maintaining Computer Programs"
Software – Practice and Experience 9 (1979) pp 255 - 265
- [FID90] FIDE Esprit II Basic Research Action 3070
Course on Database Programming Languages written by
collaborators
- [Gir72] J.-Y. Girard
"Une extension de l'interprétation de Gödel à l'analyse, et son
application à l'élimination des coupure dans l'analyse et théorie des
types"
Proc. 2nd Scandinavian Logic Symposium (1972) pp 63 - 92
- [GM79] H.I.E. Gunn and R. Morrison
"On the Implementation of Constants"
Information Processing Letters 9, 1 (July 1979) pp 1 - 4

- [GMD85] P.M.D. Gray, D.S. Moffat and J.B.H. Du Boulay
"Persistent Prolog: A Searching Storage Manager for Prolog"
In M.P. Atkinson, O.P. Buneman and R. Morrison (editors)
"Data Types and Persistence", Springer - Verlag (1988) pp 353 -
368
- [GR83] A. Goldberg and D. Robson
"Smalltalk-80: the Language and its Implementation"
Addison-Wesley (1983)
- [Har85] R. Harper
"Modules and Persistence in Standard ML"
In M.P. Atkinson, O.P. Buneman and R. Morrison (editors)
"Data Types and Persistence", Springer - Verlag (1988) pp 21 - 30
- [HD68] E.A. Hauck and B.A. Dent
"Burroughs B6500/B7500 Stack Mechanism"
AFIPS SJCC 32 (1968) pp 245 - 252
- [HS89] A.J. Hurst and A.S.M. Sajeev
"A Capability Based Language for Persistent Programming"
In J. Rosenberg and D. Koch (editors)
"Persistent Object Systems, Newcastle, Australia 1989"
Springer-Verlag (1990) pp 186 - 201
- [Hud90] P. Hudak et al.
"Report on the Programming Language Haskell"
University of Glasgow (1990)

- [IBM78] IMS/VS Publications
IBM, White Plains, N.Y. (1978)
- [Ich83] Ichbiah et al.
"The Programming Language Ada Reference Manual"
ANSI/MIL-STD-1815A-1983 (1983)
- [Kea88] S. Keas
"Parametric Overloading in Polymorphic Programming Languages"
LNCS Vol. 300, Springer - Verlag (1988) pp 131 - 144
- [KR78] B.W. Kernighan and D.M. Ritchie
"The C Programming Language"
Prentice Hall (1978)
- [Lis81] B.H. Liskov
"CLU Reference Manual"
LNCS Vol. 114, Springer - Verlag (1981)
- [MAB90] R. Morrison, M.P. Atkinson, A.L. Brown and A. Dearle
"On the Classification of Binding Mechanisms"
Information Processing Letters 34 (February 1990) pp 51 - 55
- [Mat85] D.C.J. Matthews
"Poly Manual"
University of Cambridge Technical Report 65 (1985)

- [MBC87] R. Morrison, A.L. Brown, R. Carrick, R.C.H. Connor, A. Dearle and M.P. Atkinson
"Polymorphism, Persistence and Software Reuse in a Strongly Typed Object-Oriented Environment"
IEE & BCS Journal on Software Engineering (December 1987)
- [MBC89] R. Morrison, A.L. Brown, R.C.H. Connor and A. Dearle
"The Napier88 Reference Manual"
University of St Andrews PPRR-77-89 (1989)
- [MBC90] R. Morrison, A.L. Brown, R.C.H. Connor, Q.I. Cutts, A. Dearle, G. Kirby, J. Rosenberg and D. Stemple
"Protection in Persistent Object Systems"
In "Security and Persistence", J. Rosenberg and L. Keedy (eds), Springer-Verlag (1990) pp 48 - 66
- [MBD85] R. Morrison, P. Bailey, A. Dearle, A.L. Brown and M.P. Atkinson
"The Persistent Store as an Enabling Technology for Integrated Support Environments"
Proc. 8th International Conference on Software Engineering, Imperial College, London (August 1985) pp 166 - 172
- [McC62] J. McCarthy, P.W. Abrahams, D.J. Edwards, T.P Hart and M.I. Levin
"The Lisp Programmers' Manual"
M.I.T. Press, Cambridge, Massachusetts (1962)

- [MDC90] R. Morrison, A. Dearle, R.C.H. Connor and A.L. Brown
"An Ad Hoc Approach to the Implementation of Polymorphism"
University of St. Andrews CS/90/3
Accepted by ACM ToPLaS January 1991
- [Mey88] B. Meyer
"Object-Oriented Software Construction"
Prentice - Hall (1988)
- [Mil83] R. Milner
"A Proposal for Standard ML"
University of Edinburgh CSR - 157 - 83 (1983)
- [Mil83a] R. Milner
"How ML Evolved"
Polymorphism (The ML/LCF/Hope Newsletter) 1, 1 (January
1983)
- [Mor82] R. Morrison
"S-algol: a Simple Algol"
Computer Bulletin 2, 31 (March 1982)
- [Mor90] R. Morrison
"The Meaning of Life, and what we are Doing Here"
Private communication
- [MP88] J.C. Mitchell and G.D. Plotkin
"Abstract Types have Existential Type"
ACM ToPLaS 10, 3 (July 1988) pp. 470 - 502

- [MS89] F. Matthes and J.W. Schmidt
"The Type System of DBPL"
In R. Hull, R. Morrison and D. Stemple (editors)
Proc. 2nd International Workshop on Database Programming
Languages
Morgan - Kaufmann (1989) pp 219 - 225
- [NW74] R.M. Needham and R.D. Walker
"Protection and Process Management in the CAP Computer"
Proc International Workshop on Protection in Operating Systems,
INRIA (August 1974) pp 155 - 160
- [OB89] A. Ohori and P. Buneman
"Static Type Inference for Parametric Classes"
SIGPLAN Notices 24, 10 (October 1989) pp. 445 - 456
- [Org72] E.I. Organick
"The Multics System: an Examination of its Structure"
M.I.T. Press (1972)
- [OTC90] A. Ohori, I. Tabkha, R.C.H. Connor and P. Philbrow
"Persistence and Type Abstraction Revisited"
Proc. 4th International Workshop on Persistent Object Systems,
Martha's Vineyard, Massachusetts (September 1990) pp 137 - 149
- [Pow85] M.S. Powell
"Adding Programming Facilities to an Abstract Data Store"
Proc. Persistence Data Type Workshop, Appin, Scotland
Universities of Glasgow and St Andrews PPRR-16-85

- [PS85] "The PS-algol Abstract Machine Manual"
Universities of Glasgow and St Andrews PPRR-11-85 (1985)
- [PS88] "The PS-algol Reference Manual (Fourth Edition)"
Universities of Glasgow and St Andrews PPRR-12-88 (1988)
- [Rey74] J.C. Reynolds
"Towards a Theory of Type Structure"
Proc. Paris Colloquium on Programming (1974) pp 408 - 425
- [RC89] J.E. Richardson and M.J. Carey
"Implementing Persistence in E"
In J. Rosenberg and D. Koch (editors)
"Persistent Object Systems, Newcastle, Australia 1989"
Springer-Verlag (1990) pp 302 - 319
- [Sch77] J.W. Schmidt
"Some High-Level Language Constructs for Data of Type Relation"
ACM ToDS 2, 3 (1977) pp 247 - 261
- [SFS90] D. Stemple, L. Fegaras and T. Sheard
"Exceeding the Limits of Polymorphism in Database Programming Languages"
Proc. International Conference on Extending Database Technology,
Fondazione Cini, Venice (March 1990) pp 269 - 285
- [Sol78] M. Solomon
"Type Definitions with Parameters"
Proc. 5th ACM Symposium on Principles of Programming Languages (1978)

- [SSS88] D. Stemple, A. Socorro and T. Sheard
"Formalizing Objects for Databases"
2nd International Workshop on Object-Oriented Database Systems,
LNCS 334 (1988) pp 110 - 128
- [Str67] C. Strachey
"Fundamental Concepts in Programming Languages"
Oxford University Press (1967)
- [SWK76] M. Stonebreaker, E. Wong, P. Kreps and G. Held
"The Design and Implementation of INGRES"
ACM ToDS 1, 3 (1976) pp 189 - 222
- [SZ86] A. Skarra and S.B. Zdonik
"An Object Server for an Object-Oriented Database System"
Proc. International Workshop on Object-Oriented Database
Systems, Pacific Grove (September 1986) pp 196 - 204
- [Tur87] D. Turner
"Miranda System Manual"
Research Software Ltd., Canterbury, England (1987)
- [vWMP69] A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck and C.H.A.
Koster
"Report on the Algorithmic Language ALGOL 68"
Numerische Mathematik, Vol. 14 (1969) pp 79 - 218

- [WB89] P. Wadler and S. Blott
"How to Make Ad-Hoc Polymorphism Less Ad Hoc"
16th ACM Symposium on Principles of Programming Languages,
Austin, Texas (January 1989) pp 60 - 76
- [WZ89] P. Wegner and S. Zdonik
"Models of Inheritance"
In R. Hull, R. Morrison and D. Stemple (editors)
Proc. 2nd International Workshop on Database Programming
Languages
Morgan - Kaufmann (1989) pp 226 - 237