This thesis should be referenced as:

Cutts, Q.I. “Delivering the Benefits of Persistence to System Construction and Execution”. Ph.D. Thesis, University of St Andrews (1992).

Delivering the Benefits of Persistence to System Construction and Execution

Quintin I. Cutts

Department of Mathematical and Computational Sciences

University of St Andrews

St Andrews

Fife KY16 9SS

Scotland

Declarations

I, Quintin Ivor Cutts, 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

I was admitted to the Faculty of Science of the University of St Andrews under Ordinance General No. 12 on 1st October 1988 and as a candidate for the degree of Ph.D. on 1st October 1988.

Signed Date

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

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

Abstract

In an orthogonally persistent programming system the longevity of data is independent of its other attributes. The advantages of persistence may be seen primarily in the areas of data modelling and protection resulting from simpler semantics and reduced complexity. These have been verified by the first implementations of persistent languages, typically consisting of a persistent store, a run-time system and a compiler that produces programs that may access and manipulate the persistent environment.

This thesis demonstrates that persistence can deliver many further benefits to the programming process when applied to software construction and execution. To support the thesis, a persistent environment has been extended with all the components necessary to support program construction and execution entirely within the persistent environment. This is the first known example of a strongly-typed integrated persistent programming environment.

The keystone of this work is the construction of a compiler that operates entirely within the persistent environment. During its construction, persistence has been exploited in the development of a new methodology for the construction of applications from components and in the optimisation of the widespread use of type information throughout the environment.

Further enhancements to software construction and execution have been developed that can only be supported within an integrated persistent programming environment. It is shown how persistence forms the basis of a new methodology for dynamic optimisation of code and data. In addition, new interfaces to the compiler are described that offer increased functionality over traditional compilers. Extended by the ability to manipulate structured values within the persistent environment, the interfaces increase the simplicity, flexibility and efficiency of software construction and execution. Reflective and hyper-programming techniques are also supported.

The methodologies and compilation facilities evolved together as the compiler was developed and so the first uses of both were applied to one another. It is these applications that have been described in this thesis as examples of its validity. However, the methodologies and the compilation facilities need not be inter-twined. The benefits derived from each of them are general and they may be used in many areas of the persistent environment.

Acknowledgements

Firstly, I would like to thank Ron Morrison, my supervisor, for endless encouragement, advice and patience and for his very significant rôle in my development as a researcher. He has constructed a research environment which is the object of envy and/or admiration to all who enter it.

Special thanks are also due to Fred Brown, Richard Connor, Alan Dearle and Graham Kirby for the many detailed discussions that have lead to work presented here. Not to mention their constant humour and enthusiasm.

Craig Baker, Ray Carrick and Dave Munro from the PISA group have also been involved with this work. Outside the group, advice and encouragement have come from Malcolm Atkinson, Dave McNally, Chris Marlin, John Rosenberg and Dave Stemple.

Fred, Richard and Ron also deserve great praise for reading and re-reading various drafts of the thesis.

Finally, I must thank Ron again for a very important contribution - allowing an Englishman into his group.

Contents

1 Introduction

The research presented in this thesis was undertaken in the context of the ESPRIT-funded FIDE project [FID90]. The major aim of the project is to facilitate the construction, maintenance and operation of large-scale long-lived and data-intensive application systems, referred to here as Persistent Application Systems (PAS). Typical examples of the systems under consideration are as follows:

- A social security system. The population recorded on such a system represent a very large body of long-lived and constantly-evolving data. The programs operating over that data will evolve as new legislation concerning social security is introduced.
- Management of aircraft design information. All details of the design are required from the initial concepts through to the finished product. Any subsequent modifications must also be recorded. By law, all of this information must be retained for tens of years after the final aircraft of the design has been manufactured in case of an aircraft failure. Design analysis programs may evolve as new techniques and aircraft characteristics are discovered.

The research of the FIDE consortium is concerned with reducing the cost and difficulty of designing, building and maintaining PAS. Lack of integration between conventional application-building components such as database systems, programming languages, design tools and operating systems unnecessarily increases the above intellectual and mechanical costs. System failure and the difficulties of recovery are also frequently exacerbated by this poor integration.

Improving the design, construction and operation of PAS has been planned within the FIDE project in two stages:

- Provision of a suitable language in which to write the components of a PAS. The aim here is to remove the discontinuity between database systems, programming languages and operating systems that must be tackled when constructing components of a PAS.
- Provision of a consistent and coherent support environment for the chosen language - a Fully Integrated Data Environment (FIDE). Such an environment can be realised by removing the discontinuities between the design and construction tools and the programming language itself.

Persistent programming languages are seen as an appropriate choice of PAS implementation language. Persistent languages remove the discontinuity between long and short-lived data exemplified by the traditional database, programming language divide. In conjunction with a powerful type system, persistent programming languages support increased power, safety and simplicity, all of which lead to optimisations in the programming process.

Early research in the FIDE project has supported the development of many different styles of persistent programming languages. These range from an embedded sub-language [LR89], a statically typed object-oriented language [ACO85], to strongly typed multi-paradigm languages [MBC+89,MMS92]. The attempts of C++ implementations [Str86,Car89] and of the Smalltalk implementations [GR83,BOP+89] in trying to integrate a programming language with an object store have the same basic philosophical aims as those of the FIDE project. Their success is discussed at the end of this chapter.

One of the current research areas of the FIDE project is the provision of a consistent and coherent programming support environment for these languages. Such an environment should support a harmonious framework in which a PAS may be designed, constructed, operated and maintained. Current programming language support environments fail to provide the required consistency. Whilst each individual component of these environments may be elegantly designed and well implemented, they are poorly integrated and it is left to the programmer to overcome the associated problems.

The required programming support environment will itself be implemented in some programming system. The requirement of the implementation system is that it should support within a consistent framework all the facilities for construction, compilation, binding to existing data and components and execution of programs in the supported language. In particular it must support the conversion of representations of programs in the supported language into representations that may be executed by the supporting language. This task is traditionally performed by a compiler.

This thesis assumes that a persistent programming language and its associated run-time environment are the appropriate implementation technology for an integrated programming support environment. It discusses the mechanisms necessary to transform a language that can manipulate a persistent store into a single integrated persistent programming environment and describes the implementation. Furthermore, it will be shown that the construction of the environment is simplified using persistence and subsequently that program construction and execution are enhanced within an integrated persistent environment. The expected benefits are as follows:

- The binding of software components is more flexible.
- The complexity of software construction and evolution is reduced.
- The execution of software may be made more efficient.

A bootstrap implementation of a persistent programming language will be used to construct the integrated programming environment supporting the construction and execution of programs in the same persistent language. The persistence attribute of the language is exploited wherever possible to enhance the construction and execution of the integrated programming environment.

As soon as the ability to write and execute programs within the environment has been provided the system may be developed from within. It is shown that these developments may take advantage of the persistent environment in order to support new techniques for software construction and execution within the system demonstrating new levels of power and efficiency.

The mechanisms given in this thesis have all been implemented but should be regarded as instances of methodological paradigms. Many more instances of the paradigms are already envisaged and it is expected that these and others will be implemented as use of the persistent environment becomes widespread.

1.1 Persistence

The persistence of a data item is defined as the length of time for which the item exists and is usable [ABC83]. In an orthogonally persistent system, the persistence of all data items manipulated within the system is independent of their other properties. A single programming language mechanism handles the longevity of all data items from micro-seconds to years.

The benefits of orthogonal persistence are well documented [ACC82,ABC+83, ABC+84,AM85,AMP86,AB87,MBC+87,Wai87,AM88,Dea88,Bro89,MBC+90,Con 91,Weg90]. A major advantage of persistent systems is seen in the removal of complexity. An inherent difficulty associated with the programming of data-intensive applications is the understanding of the mappings between the real world and the computational models. The programmer of an application manipulating long-lived data must fully understand all the models shown in Figure 1.1 as well as the mappings between them to be confident of constructing correct code.

In a persistent system only a single mapping is required between the programming language data model and the real world as shown in Figure 1.2. Hence the task of data modelling is greatly simplified.

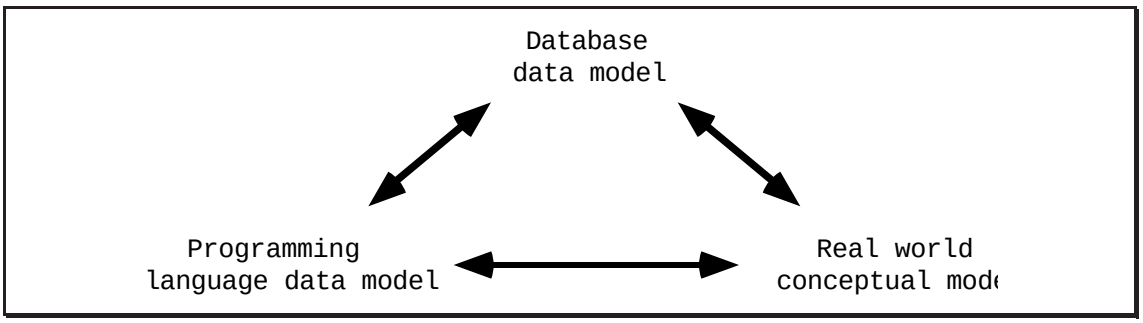

Figure 1.1 The models and mappings of a traditional system.

Figure 1.2 The models and mapping of a persistent system.

As a consequence of the single programming language model, the programmer need not write code to support the mappings that have been discarded. Measurements show that maintaining the mappings constitutes as much as 30% of all code in data-intensive applications [IBM78]. Thus persistence yields mechanical as well as intellectual benefits.

Some other advantages of orthogonal persistence are as follows:

- Type checking protection covers the entire environment. Since the only way to access the data in the environment is via the persistent language, no other protection mechanism is required
- Referential integrity is automatically supported. References between data values are maintained in a consistent state irrespective of the longevity of that data.
- Application components such as procedures and modules are first-class values in the environment, laying the foundations for the construction of large applications within the persistent environment.

1.1.1 Using a Persistent Environment to Cache Data

The persistent environment may be used as a cache for the storage of any data type supported by the persistent language. The general definition of a cache in this context

is a mechanism that may be used to store and later retrieve computed data and thus avoid its repeated recomputation.

There may be many factors involved in the operation of a cache:

- Which data should be stored in the cache?
- How is the data retrieved from the cache?
- Is data ever removed from the cache? There is a trade-off here between the space used to store the cache and the time saved using it.
- How fast may the cache be accessed? The cache access must be fast with respect to the time taken to compute the data for the cache to be worthwhile.
- How expensive in terms of space and time is maintenance of the cache? The trade-off here is against the time and space required to compute the data.
- How frequently will a cache search result in the required data? There is a trade-off here between the time taken to perform the search against the time to perform the original computation.
- The cache affects only the speed of the required computation.

The traditional hardware cache [PS87] is just one example of the general caching mechanism. A hardware cache stores the results of main memory lookups. The memory devices used for hardware caches are very high speed and so examination of the cache is much faster than a full main memory lookup. The cache lookup is performed in parallel with the main memory lookup in case the cache fails to produce the required result; otherwise the result of the main memory lookup may be ignored. The cache memory is very expensive and so is typically not large. Strategies are required to determine which data should be placed into the cache and at what point it may be removed. Hardware caching may run into difficulties when many individual operations all use the same cache, since a context switch may result in an unsuitable cache. Thrashing may occur when the contents of the cache are being repeatedly discarded. Cache switching can overcome this problem but suffers the extra penalty of reloading the cache on each switch.

The primary use of a persistent environment as a cache is as a store for structured data values. The computation that is avoided consists of the storage and retrieval of structured data to and from a storage mechanism that only supports flat unstructured data, typically a file system.

The persistent environment may also be used as a cache for the results of any other operations within the system. The results of complex calculations may be stored in a lookup table keyed by the operands of the calculation, for example. The factors involved in the operation of a cache described above should be considered before constructing a caching mechanism for a particular operation.

The thrashing problems of hardware caching described above may be avoided in the persistent environment since each operation may use a different cache. The persistent storage space is conceptually infinite and so the space used by the cache is considered cheap in comparison to the computation time that is saved.

Many instances of caching within the persistent environment will be demonstrated within this thesis. Since the aim is to optimise both the construction and execution of software, the programs that are caching data will be considered in the following four categories:

- Program composition.
- Compilation.
- Linking.
- Execution.

The data cached by a particular program invocation may be used for optimisation by another invocation of the same or any other program. There are many different and useful interactions between the four categories determined by the programs participating in the generation and use of cached data. For example, data cached during compilation of a program may be of use during its execution. Data cached during the execution of one program may be used in the composition of another. As shown in Figure 1.3, the persistent store may form the basis of a feedback mechanism where the results of one program's execution may be used by another.

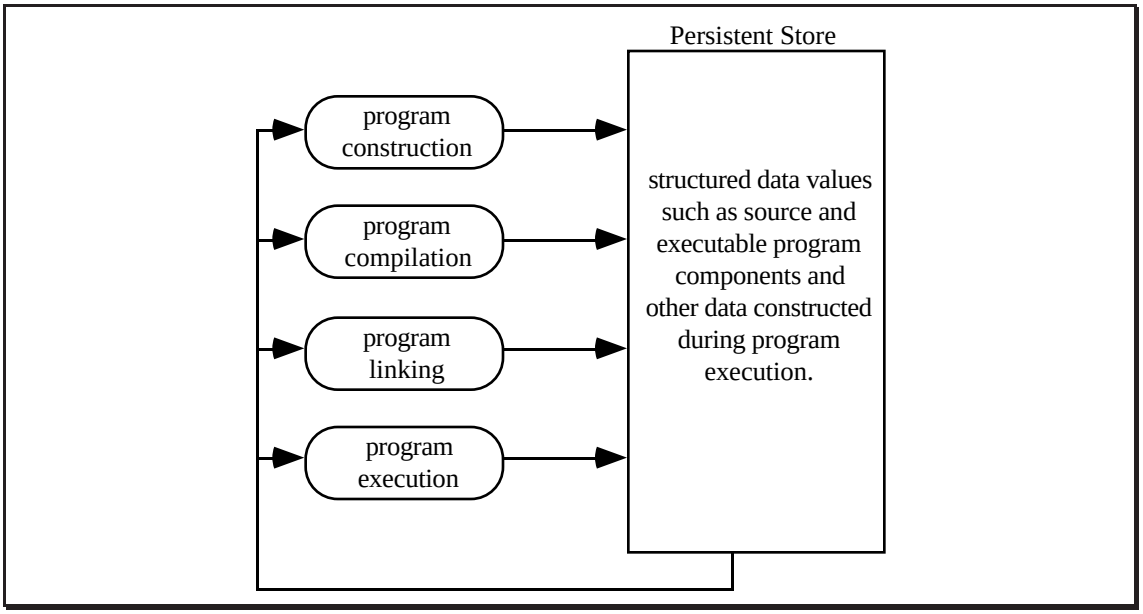

Figure 1.3 Caching data between programs via the persistent store.

There is no restriction over the type of the data that may be cached between programs. Examples of data that is cached in the various techniques to be described here are as follows:

- Structured program representations. These may be source, intermediate or executable code representations.
- Type representations and definitions to be used for type checking throughout the persistent environment.
- Execution profiles. An executing program may cache data that records various aspects of the program's performance in the persistent environment.

As with any caching technique, there is a trade-off between the space required in the persistent environment to hold the cached data and the time saved by not needing to reconstruct or recalculate it. However, many of the benefits described here apply to data that would otherwise have required storage in an external storage system had it not been cached. Significant extra space overall is not therefore required to support many of these techniques.

1.2 Building an Integrated Persistent Programming Environment

The early part of the FIDE project has supported the initial implementations of a number of persistent programming languages [MBC+89,AGO88,MMS92,BBB+88]. These typically consist of the following components:

- A persistent store implementation.
- A run-time system to support execution of programs in the language and to allow access to the persistent store.
- A compiler.
- Program construction tools.

As shown in Figure 1.4 which depicts an initial persistent language implementation the compiler is implemented outside the persistent environment. Programs are constructed and compiled using tools found in the enclosing environment and may then be executed against the persistent environment to which they can add and change values. In a programming language that allows both static and dynamic type checking, this configuration will result in type checking taking place both inside and outside the persistent environment.

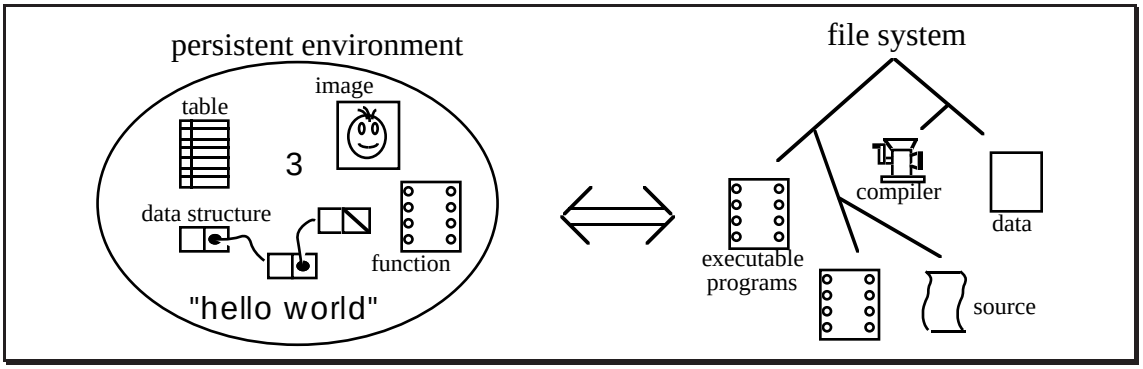

Figure 1.4 A typical early persistent system.

Such systems have been sufficient to demonstrate the validity of persistent systems in terms of greater programming simplicity and improved data modelling and protection mechanisms.

1.2.1 Conversion into an Integrated Persistent Programming Environment

An integrated persistent programming environment is achieved by supporting all programming activity with software constructed within a persistent environment. Such an integrated programming environment may be constructed using one of the initial persistent programming language implementations outlined above. The aim is to build within the persistent environment of such an implementation all the components required for construction, execution and maintenance of programs written within a persistent language. Such a system is depicted in Figure 1.5, where the integrated programming environment is just one program within the persistent environment which contains programming tools such as editors, compilers, browsers and window managers.

Where the implementation persistent language and the language of the constructed integrated environment are the same, a number of benefits are gained, as follows:

- Single consistent environment - simplifies the system.
- No conflict between the type system of the two languages.
- No partitioning of the store required between values of the two languages.

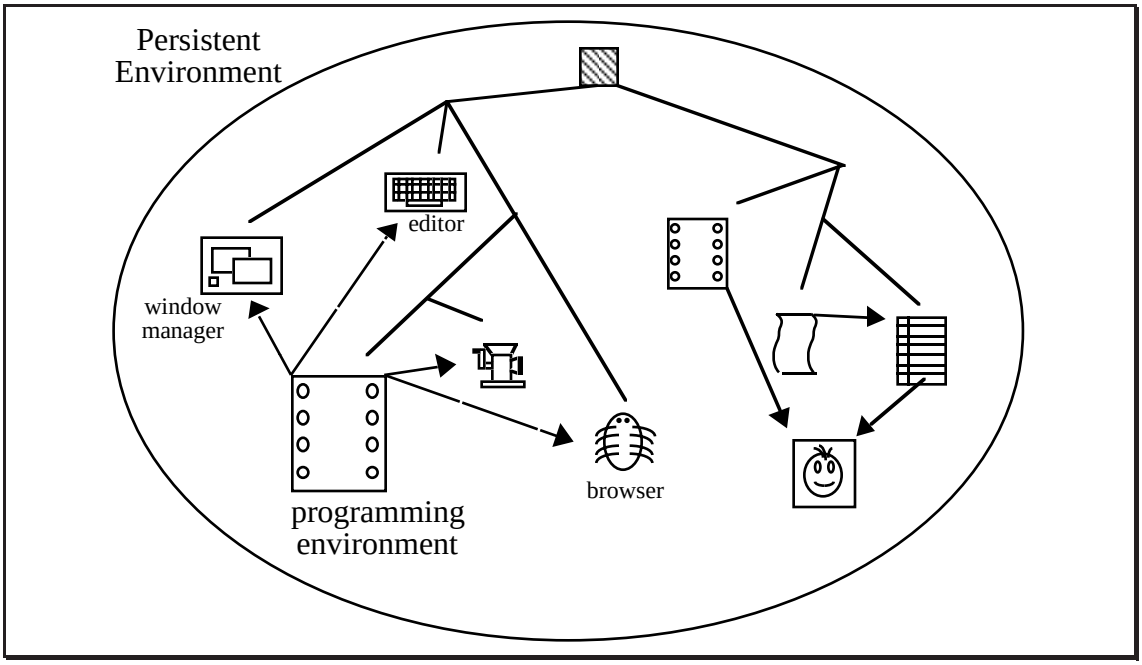

Figure 1.5 A fully integrated persistent programming system.

The key feature in this transfer process is the construction of a compiler within the persistent environment. As soon as a compiler is available to programs executing within the environment, evolution may take place from within the environment. The programs effecting the evolution may then take full advantage of the integrated persistent environment in which they are embedded in order to introduce new techniques for the construction and execution of software. These techniques demonstrate new levels of power and efficiency.

1.3 Constructing a Compiler within the Persistent Environment

The first half of the work presented in this thesis derives from the conversion of an initial implementation of a persistent programming language into a fully integrated persistent programming system. As stated above, the key to this operation is the construction of a compiler within the persistent environment of the initial persistent language implementation.

A state-of-the-art persistent programming language, Napier88, has been used as the basis for this work. Napier88 has been developed by the persistent programming research group at the University of St Andrews, one member of the FIDE consortium, and is the culmination of over ten years research into programming language design at St Andrews. The language supports orthogonal persistence, parametric

polymorphism and a strong type system of the kind described in [CW85]. A description of the Napier88 type system is given in Appendix 1.

Napier88 Release 1.0 on which the work is based is of a similar structure to the style of persistent programming language implementation depicted in Figure 1.4. This release consists of a stable store implementation, an interpreter for the Persistent Abstract Machine which is a byte coded interpreter operating over the stable store and supporting persistence and polymorphism, and a compiler constructed outside the Napier88 persistent environment. Program construction and compilation take place outside the Napier88 environment. The compilation techniques used in the compiler implementation may be found in [DM81] [Dea87] [MDC+91]. Further details of the Napier88 Release1.0 are not included here but may be found in [MBC+89].

The compiler for Napier88 is the largest application yet written in the initial Napier88 implementation. As such its construction has been a good test of the language's ability to support application construction and of the implementation techniques used in Release 1.0 of the language.

The new compiler is designed to support easy and efficient experimentation into language design and compilation techniques. The ability both to construct applications in a highly modular style and to perform easy and efficient change is required. The persistent environment was exploited during the construction process and in so doing a new architecture for the safe and flexible construction of applications from components has been developed.

The large number of individually compiled components making up the new compiler caused problems for the type checking implementation of the original Napier88 system. This is an effect of the two separate universes in which type checking takes place in that system. Firstly, independently-prepared components are statically type checked during compilation which takes place outside the persistent environment. Secondly, binding and execution of the components which takes place within the persistent environment may also require type checking. The requirement for type information to pass between the two environments causes severe efficiency problems in a strongly typed language where the amount of type information is typically large. Persistence was again exploited to optimise the type checking and a unified view of type system implementation within persistent systems was realised.

The research into the effect of persistence on application construction and type checking is summarised below.

1.3.1 Constructing Applications in a Persistent Environment

An application architecture has been developed that supports an evolutionary approach to the construction of software from individual components. The advantageous features of the architecture are efficiency and safety from failure during execution and the flexible evolution of components and applications. Existing construction architectures [Mil84,Car89,ACO85,MAE+62,GR83,Wir71,KR78,DOD83] attempting to satisfy these features fail to do so because the sharing of structured executable versions of applications between the construction and execution environments is not supported.

In a persistent system, the components of an application may be considered as data items [AM85]. In the construction architecture presented here, the persistent environment is used as a cache for components by programs that construct and bind them together to create and update applications. The strongly typed locations of a

persistent environment are used to hold the individual executable components of an application which are first class values in the environment. Component generation and binding programs cache the constructed components in these locations where they may be directly accessed during execution. They are also available for future invocations of the binding programs to permit flexible evolution of applications.

The architecture involves the sharing of cached data between the compilation, binding and execution of applications. Particular advantages of this architecture are as follows:

- Application components may be independently compiled.
- Individual components may be incrementally linked into an application.
- The type compatibility of the components is fully checked before application execution.
- Components may be reused in different contexts.
- Control over sharing and evolution of components and applications is supported as they progress through the software lifecycle.

One disadvantage is as follows:

- The use of bindings to locations results in less tightly bound applications than in a system using bindings to values [Str67]. Although the architecture guarantees the type of a component prior to application execution, the particular component contained in a location cannot be determined until execution. The control over sharing and evolution mentioned above helps to alleviate this problem.

1.3.2 Using Persistence to Optimise Type Checking

Type systems are well understood as mechanisms which impose static safety constraints upon a program. Within a persistent programming environment the type system provides all the data modelling and protection facilities for the environment. Elsewhere [CBC+90] it has been demonstrated that not all constraints on data may be captured statically. This leads to a judicious mixture of type checking times being employed to suit the particular requirements of an application. The spread of times allows a balance between static safety in large applications and dynamic flexibility in constraint expression and reuse.

The type system may be used to perform operations over program and data during any of the four software lifetime processes of composition, compilation, binding and execution. The point of interest here is the manner in which type information is shared among these four phases so that the relevant type system operations may take place. Much of the complexity associated with the implementation of type systems stems from a lack of support for the sharing of common type information across both independent components and lifetime processes. Few systems have overcome the technical difficulties associated with the transfer of complex type information between independent environments [ACP+91].

A single set of operations to create, manipulate and test representations of type information may be constructed within a persistent environment. The set may be shared by all programs in the persistent environment that perform type system operations. Instances of type representations may also be cached in the environment and shared across programs.

The only type system operation traditionally available outside the compilation environment is that of type equivalence. In order to maintain execution speed the efficiency of this operation is optimised [CBC+90]. The availability of a single set of

type system operations and the sharing of type representations between programs simplify the use of any type system operation in any of the construction, compilation, linking or execution contexts. The implementation of complex type operations such as those associated with parameterised and recursive types may require optimisation to prevent their use outside the compilation environment seriously affecting system performance. A format for the type information is required that can support efficient implementations of all the type system operations that are carried out in a persistent system. The persistent store may also be used to cache the results of inherently complex operations over types in order to improve performance.

The benefits of persistence to type system implementation are as follows:

- A single copy of type information may be cached in the persistent environment and shared by all programs.
- A single implementation of all type system operations may be cached in the environment and shared by all programs.
- The results of complex type checking operations may be cached in the persistent environment.

It is demonstrated within the thesis how all these benefits may be exploited.

1.4 Delivering the Benefits of a Persistent Programming Environment

The second half of the work presented in this thesis concerns some of the benefits that may be gained when it is possible to construct and execute programs within an integrated persistent programming environment. A number of general mechanisms for the enhancement of software construction and execution are outlined below. A detailed description and analysis of the first two mechanisms are presented in the thesis. The work leading to the research presented in this thesis also lead to the construction of the remaining mechanisms although they are not presented in detail here.

1.4.1 Using Persistence to Enhance Compilation

Embedding a compiler within a persistent environment creates a symbiotic relationship between the two. The relationship occurs in that the functionality of the compiler is enhanced and compiled code optimised using the persistent environment. The persistent environment benefits in that the execution of programs, including the compiler, may be made more efficient. The benefits of the relationship may be described in terms of the manner in which the compiler and other programs executing in the environment share cached data with one another, as follows.

Firstly the compiler, which is itself an item of data cached by the construction architecture, may be accessed by executing programs. An executing program may construct and execute new programs which manipulate the persistent environment. This ability of a program to alter its own environment during execution is a particular form of reflection [Mae87] known as run-time linguistic reflection [SSS+92]. It is of particular interest in persistent systems because it can allow long-lived data and programs to evolve in a type-safe manner.

Secondly, the functionality of the compiler may be enhanced by parameterising it with persistent values that are associated with the source. The number of stages at which programs are bound to data may be extended since the compiler itself can manipulate

that data. These stages are defined by the various times at which identifiers embedded in a program are resolved to their associated values, as follows:

- During program composition. The source contains embedded values. During compilation these values are incorporated into the compiled code. This technique is known as hyper-programming [KCC+92].
- During compilation. Free identifiers in the source are resolved using values passed into the compiler [FDK+92].
- Between compilation and execution. The resolution of the identifiers is performed in a separate phase involving an intermediate program representation and the associated values.
- During execution. Identifiers are resolved when the executing program accesses values in the persistent environment.

Thirdly, the compiler may manipulate values in the environment. There are two benefits here. The compiler may use the environment to optimise its own performance, for example by caching the results of operations repeated across invocations of the compiler. Also, the compiler may associate persistent values with compiled code in order to optimise the execution of that code.

To summarise, the advantages of embedding a compiler in the persistent environment are:

- Run-time linguistic reflection may be supported.
- A wide range of binding techniques between program and data may be supported.
- The performance of both the compiler and the code that it produces may be optimised.

1.4.2 Using Persistence to Enhance Execution

A persistent environment may be used by executing programs to record information detailing aspects of the their operation. This information may then be accessible to other programs executing in the environment. Optimisation within the environment may be performed by these programs based on the data collected during its operation, as follows:

- An execution profile of the program may be cached in the environment during program execution.
- At some later time a program designed to enhance the environment may analyse gathered execution profiles in an attempt to find optimisations that will improve the performance of the environment. The analysis is performed using a cost function that trades the cost of making an improvement against the potential benefit gained from it. The enhancer performs the optimisation if the analysis is favourable.
- Subsequent execution of the environment will be improved by the actions of the enhancer.

Dynamic clustering [BD90] is an example of this technique, where groups of stored values that are accessed at the same time are placed near each other on secondary storage to improve access time. The decision on which values to cluster is based on statistics collected during execution of the programs that access the values. A second example is the caching of the results of complex operations in the persistent environment, already described for type checking operations in Section 1.3.2. Where programs are considered as data there is the possibility of using the same technique to optimise program code in accordance with the dynamic needs of the program. Code

enhancement is possible since code generators make static trade-offs with regard to the space requirement and run-time execution speed of the code. Dynamic execution profiles can be used as a basis for changing these trade-offs. Dynamic query optimisation is a particular example of code enhancement, where execution of the query is optimised based on information recorded during previous query executions.

A particular benefit of this technique is as follows:

- Optimisations may be made to the operation of the persistent environment based on information not available statically.

An important consideration when using the architecture is as follows:

- The space and time required to record the execution profile may be significant. There is a trade-off here between the amount of recorded information and the cost of gathering it.

1.4.3 Further Beneficiaries of an Integrated Environment

Hyper-programming

Where the program composition process is supported inside the persistent environment, programs and data may be bound during composition. The programmer composes programs interactively by navigating the persistent environment and selecting data items to be bound into the programs. This requires that direct links to persistent data items are contained in program source code and that the compiler can manipulate such links. This style of programming is known as hyper-programming [KCC+92] and has been implemented on top of the integrated programming environment described here [Kir92].

Browsing technology

Advanced persistent object browsers [DCK90,FDK+92] have been developed using the technology described in this thesis. In particular, the ABERDEEN environment developed at the University of Adelaide [Far91] permits the browsing and tagging of persistent objects. Tagged objects may subsequently be incorporated into programs under construction. These tags are resolved during compilation providing another time at which program and data may be bound together.

Run-time Linguistic reflection

The provision of run-time linguistic reflection permits a system to evolve under its own control. This is achieved when an executing program constructs a program representation and passes it to a compiler. The compiler returns an executable value to the program which can then execute or store it as required. Reflective programs have traditionally been hard to understand because their source contains representations of language constructs that are in different formats and which will be executed at many different times. Work presented in [Kir92] attempts to simplify reflective programs by providing a construction methodology which highlights the different kinds of code contained in a reflective program. This work depends on the availability of a compiler within the programming environment where it may be accessed by executing programs.

1.5 Working Practices in the St Andrews Persistence Group

The work presented in this thesis was undertaken within the persistent programming research group at the University of St Andrews. During the time period in which this work was carried out, the group has consisted of the following researchers: Ron Morrison, Richard Connor, Alan Dearle, Fred Brown, Ray Carrick, Graham Kirby, Craig Baker, Dave Munro and myself. Ron Morrison supervised the work with Alan Dearle who was co-supervisor until he moved to the University of Adelaide at which point Richard Connor unofficially took over his role. All members of such a group contribute in some way to nearly all the work of the group. The work presented in this thesis has benefited from two significant collaborations as follows:

- The research into application building was performed in conjunction with Alan Dearle and Richard Connor.
- The optimisation of the type checking mechanisms took place jointly with Richard Connor who was responsible for the Napier88 Release 1.0 type checker.

It was my responsibility to provide the integrated persistent programming environment and demonstrate that persistence may be exploited to support optimisations within the environment. This then allowed others to use the facilities and for me to share in that work.

Jointly my supervisors and I designed the applications architecture which I then used to implement the compiler. This required the design of the new type representations and type checking optimisations, also undertaken jointly with my supervisors. Completion of this work lead to my designing and implementing the new compiler interface and the optimisation architecture. I have subsequently worked with Graham Kirby on the design and implementation of the hyper-programming system and with Alex Farkas on the ABERDEEN object store browser.

1.6 Related Work

There are no other known instances of a strongly-typed persistent programming language having been used as the implementation vehicle for an integrated persistent programming environment. However there are other programming language systems that could be used to support such an environment. In order to catalogue those languages, it is necessary to define exactly what is required to support the work described here.

The base point is a language and environment supporting the orthogonal persistence of strongly typed objects. In particular strongly typed executable code objects can be retained within the persistent environment. The language is used to construct an integrated set of tools within the environment that support the construction, compilation, linking and execution of programs written in the language. The key tool here is the compiler which can take representations of programs in the base language made up from the data structures of that same language and convert these into program components that are executable within the environment. This compiler is a function within the persistent environment available to other programs executing there.

A brief analysis of a number of languages with the basis philosophical aims found in the FIDE project is now given.

C++

The C++/Unix™ [Str86,Car89,RT78] world has succeeded in constructing integrated persistent programming environments. However the only data type that may persist in such environments is the byte. The benefits of strong typing such as improved safety and data modelling power over the whole environment cannot be achieved. An example of such a system is the combination of the language E [RCS89], an extension of C++, with the EXODUS Storage Manager [Car89]. The Storage Manager provides *storage objects* which are uninterpreted byte sequences of virtually any size. The aim is to build and extend database systems by writing the database system code in E. E inherits from C++ the lack of strong typing and so is unsuitable for the experiments described here.

Smalltalk

The object-based environments of Smalltalk systems [BOP+89] support the entire programming process from construction to execution. However, the compiler used in this environment is not implemented using the Smalltalk language and is not available to executing programs. In addition, Smalltalk implementations are dynamically typed which, while increasing flexibility for program construction and evolution, reduces the safety of completed applications. This is a significant factor in an environment designed to support large-scale and long-lived applications.

O2C

The O2 Database [BBB+88] is an object-oriented database server developed at Altair, France. An O2 sublanguage is embedded in an existing programming language in order to allow the creation of objects in and querying over the database. O2C [LR89] is an extended version of the C language [KR78] that can operate over the O2 Database. O2C programs are stored outside the database and the language is therefore unsuitable for the construction of system software within the O2 persistent environment. For example, since executable program components cannot be stored and manipulated within the environment, the construction architecture cannot be supported. Type checking is spread over both the external C environment and the internal O2 environment. Hyper-programming cannot be supported since source programs are not contained within the environment.

Galileo

Galileo [ACO85] is a strongly-typed object-oriented language developed at the University of Pisa in Italy. The language is supported by the Functional Abstract Machine (FAM) [Car83] with extensions to permit access to the Persistent Object Store developed by Brown at the University of St Andrews [BMM+92]. Any top-level declarations in an interactive Galileo session are retained within the persistent environment. The compiler is external to the persistent environment and is not available to executing programs. Executable code may be stored within the persistent environment in the form of first class functions and strongly typed persistent locations are also supported, so it should be possible to construct applications using the methodology outlined in Section 1.3.1. Until a compiler has been constructed within the language, which is possible, the remaining benefits of persistence described here cannot be supported. However it is hoped that collaborative research during the term of the FIDE project will permit these benefits to be demonstrated.

DBPL

Developed at the University of Hamburg in Germany, DBPL [MS89] supports an extended relational data model. "The final DBPL language design provides an orthogonal integration of sets and first order predicates into a strongly and statically typed programming language and the DBPL system supports the language with full database functionality including persistence, query optimisation and multi-user transaction management." [Atk92] A compiler has yet to be constructed within the DBPL language and so the benefits of persistence have yet to be realised. The designers of DBPL have progressed to the language Tycoon which has a more powerful type system in order to continue their experiments into persistent environments.

Tycoon

The typeful language Quest [Car89] was developed by Cardelli to explore the limits of strong typing within programs. The University of Hamburg combined Quest and the Persistent Object Store of Brown [BMM+92] to construct the language Tycoon [MMS92]. The Quest module mechanism inherited in Tycoon supports the construction of large applications from components which are first class values in the system. This module mechanism permits applications to be constructed that are more statically bound than those constructed using the application architecture outlined in Section 1.3.1. Similarly to Galileo, a compiler has yet to be constructed within the environment and so the benefits of persistence outlined in Sections 1.3.2, 1.4.1 and 1.4.2, whilst possible have yet to be realised. Again, it is hoped to demonstrate the benefits of persistence within a Tycoon integrated environment during the FIDE project.

1.7 Thesis Structure

The work described in this thesis is divided into four main chapters describing in detail the benefits of persistence to system construction and execution outlined above.

In Chapter 2, the application architecture outlined in Section 1.3.1 is described independently of any particular supporting language. This is followed by a description of an implementation of the architecture in the persistent programming language Napier88. The suitability of the architecture for other languages is discussed.

In Chapter 3, an analysis is made of the many different type system operations found in persistent programming systems and describes in which of the four program categories defined in Section 1.1.1 each operation is performed. It is shown how a single set of operations and type representations may be shared by all programs in the environment. A representation for type information is presented that supports efficient implementation of the type manipulation operations. The use of the persistent store as a cache for the results of complex type system operations is discussed.

In Chapter 4, a flexible approach to compilation and binding is described. An interface supporting this approach is defined. The construction of a compiler within the persistent environment using the construction architecture of Chapter 2 is described. The implementation of a protection mechanism for intermediate program representations and for type representations is discussed.

In Chapter 5, a particular instance of the optimisation technique of Section 1.4.2 is described that optimises the execution of the polymorphic procedures of Napier88. From the starting point of a complete but unoptimised implementation of polymorphic

procedures, it is shown how execution profiles may be collected and analysed in order to produce optimised versions of the procedures. These optimised versions are associated with the unoptimised versions and may be accessed under the appropriate conditions.

2 Constructing Applications in a Persistent Environment

2.1 Introduction

This chapter describes an application architecture designed to support an evolutionary approach to the construction of software from individual components. The desirable features of such an architecture are efficiency and safety from failure during execution and the flexible evolution of components and applications. Existing construction architectures attempting to satisfy these features fail to do so because the sharing of structured executable versions of applications between the construction and execution environments is not supported.

The strongly typed locations of a persistent environment may be used to hold the individual components of an executable application. The locations may be accessed by both the linking and execution processes. It is shown that an architecture supporting the construction of applications from components contained in these persistent locations may satisfy the desires of safety, efficiency and flexibility.

The architecture was developed during the construction of the Napier88 compiler within the Napier88 Release 1.0 persistent environment. The compiler was the largest software item yet constructed in Napier88 consisting of more than 10,000 lines of code and so construction in a single compilation unit was unacceptable. Using the binding mechanisms of Napier88, experiments were performed to realise various methods of binding Napier88 components together, each giving different trade-offs between safety, efficiency and flexibility. The result of these experiments is the application construction architecture given here which supports a balanced trade-off between these three characteristics.

2.1.1 Change and the Linking Process

Using an evolutionary approach to software construction, software evolves from a prototype to a production level system with frequent changes being made during implementation and testing. Once the software is released, changes are still needed when bugs are removed, the implementation of the software is improved or functionality is added as new requirements are perceived. Work by Lehman [Leh80] shows that many of the changes required after release are not caused by poor initial specification or construction but are the result of a revision in users' expectations brought about by the new software. Change is therefore an unavoidable process throughout the software lifecycle.

Where the source code for an entire software application is contained in a single compilation unit the process of making a change is straightforward entailing an update to the source followed by re-compilation and re-execution. However maintaining an application in a single compilation unit becomes increasingly difficult for large applications, for the following reasons.

- The complexity of the single unit of source code may be beyond one person's intellectual capacity.
- Only one programmer can work on the application at any one time.
- Current compilation technology strains to handle very large compilation units.

Breaking a large application contained in a single compilation unit into several individual compilation units along the boundaries of the application's logical components helps to overcome these problems. Constructing executable applications from components introduces its own difficulties however. When one component uses another component an identifier for the latter appears in the source code of the former. This identifier must be resolved into a link between the executable code of the two components before one component can use the other. In many languages a type check is also performed at the time of identifier resolution to ensure that the linked component is of the expected type. The combined operations of resolution and type checking will be referred to here as *linking*.

Linking complicates the process of change and affects the level of static safety in an application, as follows:

- Changing an individual component requires that each component referring to it must be relinked to the new version. The required recompilation and relinking after a change should be minimised.
- The linking operation introduces a new source of errors, since it will fail if the component to which an identifier refers cannot be found or if it is of the wrong type. It is desirable to avoid such errors during execution.
- In the context of one component the action performed by an external component cannot be guaranteed. It is a matter of programming convention to ensure that errors are not caused by unexpected component behaviour. Since this is a problem inherent in all linked systems, it will not be discussed further here.

Existing implementations of linking do not satisfactorily solve the first two problems. They fall into three categories, mainly determined by the time at which linking is performed.

The first group includes the implementations used in languages such as ML, Quest and Galileo [Mil84,Car89,ACO85]. In these languages, linking occurs during compilation removing the possibility of failure during execution. In addition, run-time efficiency is not affected, since the checks involved in linking have already been performed. However, separate compilation is not supported since compilation and linking take place simultaneously. Also, a change of a single component may require recompilation of the entire application in order to carry out the necessary relinking. Such an operation is expensive and it may even be impossible to propagate the changes to an application whose components are spread across a distributed environment.

At the other end of the spectrum are languages such as Lisp and Smalltalk [MAE+62, GR83] in which linking occurs during execution just before the link between components is required. This method loses the safety and efficiency provided by linking during compilation. However changing a single component requires minimal application reconfiguration since the new version will be relinked on the next execution of the application. A suitable architecture for the support of this style of linking is found in the Multics operating system [Org72].

The final category of languages, including Pascal, C and Ada [Wir71,KR78, DOD83], represent a compromise between safety and flexibility where linking is performed in a separate phase in between compilation and execution. Using this method all linking is performed before an application executes, giving the gains in efficiency and safety. In addition, changing an individual component requires the recompilation of at most the components that use it. Although this method is less expensive than linking during compilation, it is still significant in large programs. In general, existing implementations of separate linking do not allow a single changed component to be relinked to an application without forcing the entire application to be

relinked. The cost of a component update is therefore approximately proportional to the size of the whole application rather than the size of the changed component. An optimisation of separate linking is proposed in [QL91] where limited manipulation of linked executable applications is possible, but the cost here is still the same in the worst case.

2.1.2 Applying Persistence to the Separate Linking Process

Of the three existing styles of linking, the first does not support sufficient flexibility for change and the second does not support sufficient safety and efficiency. The third is a good compromise although it is still inflexible for change.

The inflexibility is caused by the manner in which the executable application is handled by traditional software engineering environments. A typical application consists of many logical components. The source code for an application is split into these components, some of which may be shared by other applications. Traditionally however the executable version of the application is not split up. It is a single self-referencing unit constructed by taking copies of the compiled versions of all the individual components and placing them together as a single unit. All references between components are made within the context of the unit.

This style of operation is caused by the separate environments in which the processes of compilation, linking and execution traditionally take place. The only means of communication between the processes is the file system supporting flat unconnected units such as compiled components and executable units. Placing the compiled versions of an application's components into a single file increases the efficiency and safety of the application during execution. It is the unstructured nature of that file however that makes it impossible in general to update a single component.

A persistent environment may be used to hold a structured version of the executable application which may be accessed by both the linking and execution processes. The components of this application are executable programs in their own right and may be shared using persistent addresses. The safety of the application is maintained since the environment is strongly typed.

A persistent environment holding the executable components of an application may support the separate linking phase described earlier since the components may be manipulated independently of one another. A change to a component requires only the relinking of components that directly or indirectly use it This may still be significant for large applications however and in the worst case entire applications will be relinked.

This chapter describes a new architecture for the construction and maintenance of large applications in which it is possible to safely and efficiently update single changed executable components. The architecture depends on shared strongly typed locations and the manipulation of individual executable components supported by a persistent environment. An alternative approach to the separate linking method is used which provides a different compromise between safety and flexibility. Linking is split into two phases as follows. All type checking and part of the identifier resolution are performed before application execution for safety and efficiency. The completion of the resolution operation is performed dynamically which gives the flexibility required for individual component update.

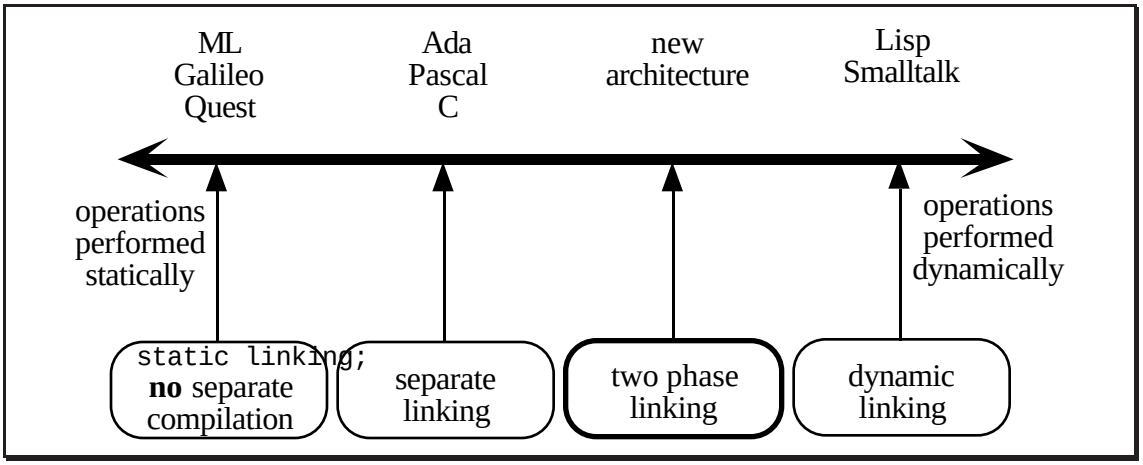

Figure 2.1 A comparison of linking mechanisms.

Figure 2.1 shows how this architecture relates to existing architectures for application construction. On each side are the extreme positions of entirely static or dynamic linking; in between are the existing and new compromises between safety, flexibility and efficiency.

The following section describes the proposed architecture independently of any particular programming language and shows how it may be used to support a software development environment. Section 2.3 describes a particular instantiation of the architecture using the persistent programming language Napier88 [MBC+89]. Section 2.4 makes comparisons with other languages that could be used to implement the architecture followed by some conclusions.

2.2 The General Architecture

An application consists of a number of logically separate components. The major aim of this architecture is to permit construction of applications from these components in a manner which is as safe as possible from dynamic failure whilst still allowing change to individual components throughout the software lifecycle with the minimum of application reconfiguration.

The desired safety and flexibility may be achieved using a two-stage linking process as follows. Every component of an application is contained in a separate typed location. Each component is linked to the *locations* of the other components that it uses, not to the components themselves. This initial linking operation from component to location takes place between compilation and execution and involves identifier resolution and type checking. When one component uses another during execution, a second linking operation is performed which retrieves the desired component from the relevant linked location. This is an inexpensive operation involving a dereference of a known location with no identifier resolution or type check.

The advantage of the two-stage linking process is that all type checking and partial identifier resolution are performed before application execution. Locations are guaranteed to contain a value of the correct type during execution. In addition an update to a component can be achieved by simply placing a new version into a location by assignment. The new version will automatically be used when the location is next accessed during execution.

This section elaborates on the basic principle, showing how applications are initially created and executed and how different styles of change may be achieved. The basic principle is then extended to show how other facilities desirable during the software development process such as software reuse and support for system building can be realised.

2.2.1 Application Construction and Execution

An example which contains two separate components, a text editor and a dialogue box, is now introduced. Since the architecture is strongly typed the two values must be assigned suitable types. As shown in Figure 2.2, the editor is a procedure taking as a parameter the initial text for the editor and returning the final edited text; the dialogue is a procedure taking the message text as a parameter and returning a boolean value. The types of the values are as follows:

dialogue box	**proc**(**text** → **bool**)
text editor	**proc**(**text** → **text**)

Figure 2.2 Component types.

This example has been chosen to show that the architecture can support mutual recursion: the dialogue box uses an editor to display the message; the editor uses a dialogue to request information. The process of constructing locations for these components, constructing the components themselves and executing them is now described. This will show how the architecture may be used to construct applications that cannot dynamically fail from identifier resolution or type checking errors.

Creation of Component Locations

Every component of an application is contained in a separate typed location. The first phase in the construction of a new application is to make the locations for these components. In order to do this the names and types of the locations must be known. Locations are always initialised containing a default value of the correct type in order to preserve the integrity of the location. For the editor/dialogue example, the new locations are shown in Figure 2.3: the locations are the ovals lying in the horizontal plane containing identifiers, types and default components.

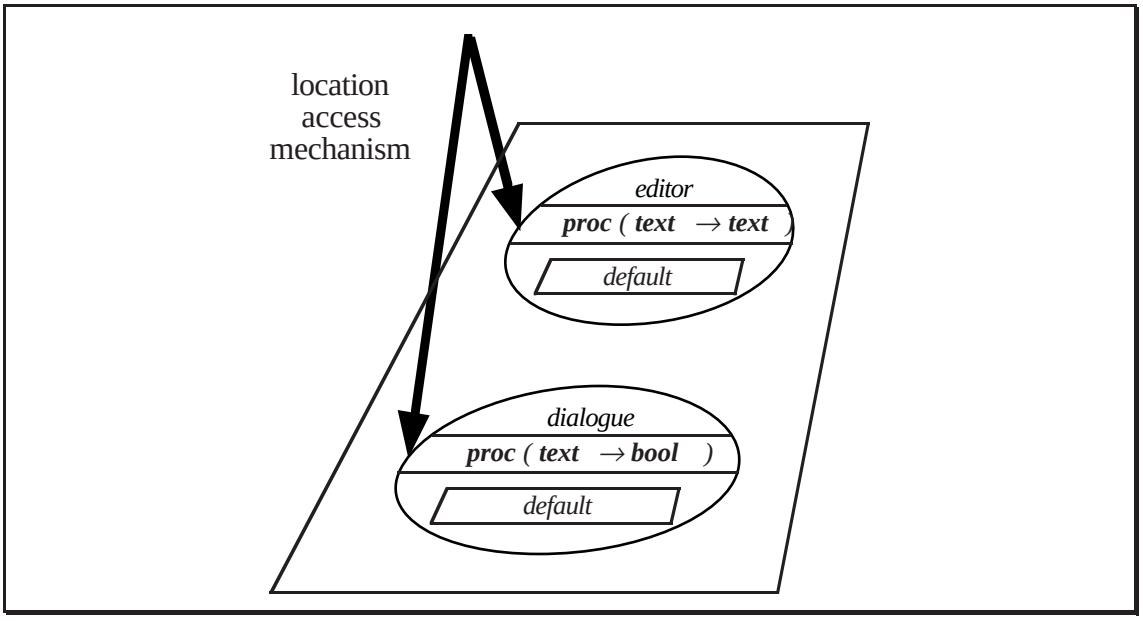

Figure 2.3 Initialised locations and access mechanism.

The application is not restricted to just this initial group of locations. Extra locations may be added at a later time if extra components are required.

A mechanism which will return a location given an identifier and a type is required to allow access to the locations during the subsequent stages of program construction and modification. The mechanism performs the identifier resolution and type checking required by the first linking stage mentioned in the introduction to this Section. The heavy arrows in the vertical plane of Figure 2.3 represent the location access mechanism.

Construction of Components

Once a component's location has been created, the component itself may be constructed. The initial linking phase is performed at this stage. The generation of an individual value involves three steps, as follows.

- **Find component locations**. Using the location access mechanism all the component locations used by the component under construction are identified. Type checking is performed by the access mechanism on the locations to ensure that each one is of the type expected by the new component.
- **Create new component**. A new component is created which may refer to any locations that were identified in the previous step. These locations are linked into the new component.
- **Store the new component**. Using the location access mechanism again the location of the new component is determined. The new component is stored by assigning it into this location which overwrites the dummy component.

Returning to the example, the generation of the dialogue procedure requires the identification of the editor location. When the new dialogue procedure has been created, containing a link to the editor location, it is assigned into the dialogue location, which must also have been identified. Figure 2.4a shows the situation after

the dialogue has been generated; Figure 2.4b shows the position after both components have been generated.

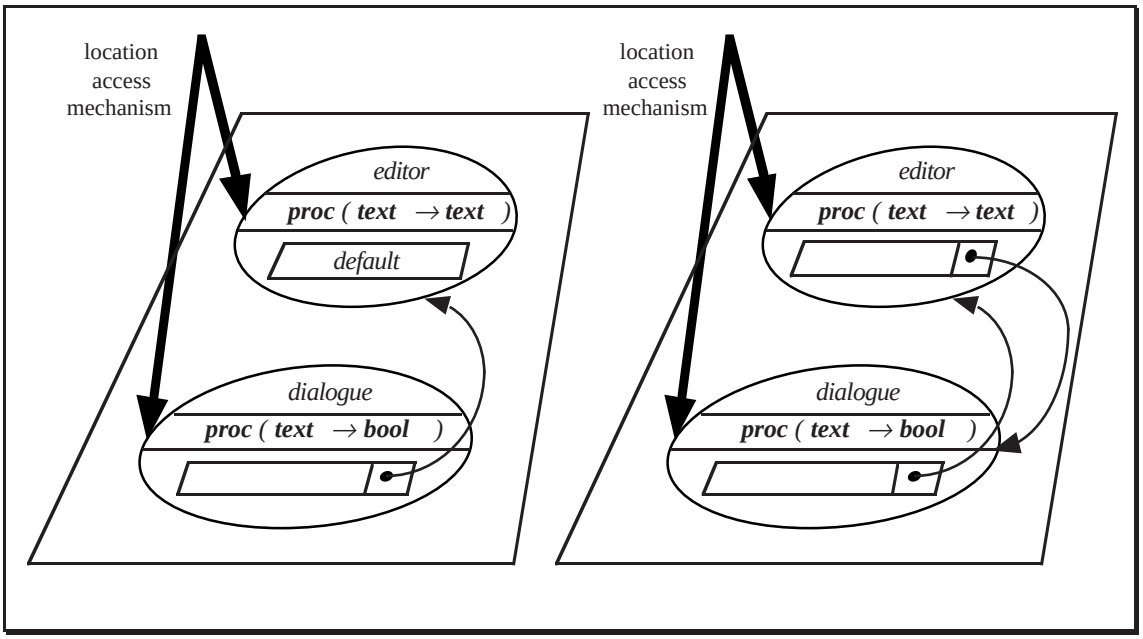

Figure 2.4a After dialogue construction. **Figure 2.4b** After construction of both.

When every component has been generated all links between components and locations will have been created. The components which are executable language values are represented by the rectangular boxes inside the locations shown in Figure 2.4b. Each component contains a link (or an address) to the location of the other component that it uses, represented by the arrows in the horizontal plane. The initial linking phase involving identifier resolution and type checking is therefore completed before application execution, which ensures that there can be no failure during execution due to missing component or type checking errors.

Execution of an Application

In order to execute an application the location of a suitable entry point in the call graph of the application is identified. This is achieved using the location access mechanism shown in Figure 2.3. The component itself is then retrieved from the location and executed. This retrieval is the second linking process mentioned in the introduction to Section 2.2, which is also used for all inter-component communication while the application is executing.

Any component of an application may be executed in the same way. This may be desirable during component testing. For example to test the dialogue box of Figure 2.4b the dialogue location is identified and the component retrieved. The component is then executed. If an editor is required by the dialogue box, the editor location to which the dialogue was linked at construction can be accessed.

Having identified the first component to be executed, all access between components is type safe, since the location access operation performed during execution does not need to perform identifier resolution or type checking.

An application may be executed before all components have been constructed, for example during a prototype phase. The dummy initialisation values will be used for those components that have not yet been generated.

2.2.2 Component Evolution

The architecture allows individual components to be updated with minimum disturbance to the remainder of the application. A mechanism for relinking single changed components is required to implement this. This section describes three different kinds of change that are all desirable and shows how they are supported by the architecture.

Changing a Component by Assignment

A change to a component may be achieved by assigning a new value of the correct type into the component's location. If other components use the changed component they will contain links to its location. When they next access the location dynamically the new component placed there by assignment will be retrieved.

The mechanism for this kind of change is identical to the operation to construct a component described in Section 2.2.1. It is impossible to place a component of an unsuitable type into the location since the assignment operation is type-safe. The change does not therefore compromise the level of type safety in an application provided by the architecture.

Type Changes

Often during the development of an application the type of a component is forced to change as new requirements are made of it. The architecture supports such changes without requiring major reconfiguration. Figure 2.5 describes an application as a large matrix of linked locations.

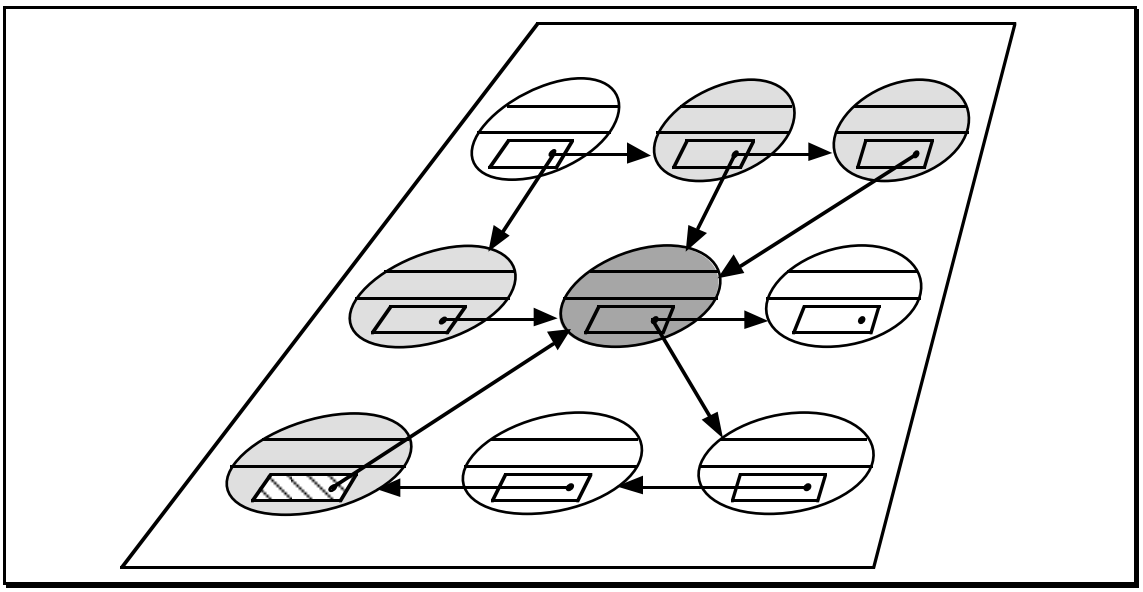

Figure 2.5 A linking matrix.

The ovals as before are locations containing component values of an application. The arrows denote the links made at construction time between components and the locations of other components to which they refer. The access mechanism has been omitted from this diagram. Consider the dark oval in the centre of the linking matrix. The components that are linked to that location have been shaded with diagonal stripes. If a change in type is to be made to the centre component then the striped components must also be changed to preserve the type correctness of the application. Even though these components must be changed internally, they will not themselves change type; consequently the changes that must be made to them may be handled by the mechanisms described earlier for changing a component by assignment. The mechanism to change the type of any component is therefore as follows.

- A new location is created of the appropriate new type.
- A value of the new type is created and assigned to the new location.
- Components dependent on the changed component are regenerated and linked to the new location, taking account of the new type.

This amounts to making a hole in the application binding matrix and then darning in a new component. In most cases the majority of the application will be unaffected and therefore not require any reconfiguration.

Changes During Application Execution

Sometimes it is undesirable or even impossible to halt the execution of an application in order to make a change. Systems administrators might not have such a bad name if they could install for example a new version of the mail server without bringing the whole operating system down. This type of change requires that both the application construction system and the application itself can execute concurrently. In addition, the construction system must be able to use the dynamic access mechanism to the application components at the same time as the application is using the statically constructed links.

Where the change to a component does not involve changing its type the location for that component may simply be updated with a new version. Any component of the application currently using the old version will continue to do so until the next access of the location at which point it will retrieve the new version.

A number of other components may also require alteration when a type change is made to a component. The change process is identical to that for a type change described above. As long as the changes are synchronised with the execution of the application, it will continue to operate correctly.

An application may change itself in an identical manner to that described above provided that it has access to its own locations and is able to manipulate components. This is a form of type-safe behavioural reflection [Mae87].

2.2.3 Supporting the Software Development Process

The architecture described so far may be used as the basis of a system to manage applications during their lifetime. This section describes some simple extensions to the basic architecture which add support for software reuse and application, as opposed to component, evolution.

Software Reuse

Software may be reused in two ways in this architecture. The most obvious way is to have many different applications using the same component, which is a straightforward extension of allowing many components in a single application to use the same location. For example there are many applications that use a dialogue box to request information from the user. These may all use the dialogue component described earlier. The access mechanism already described must be extended to reach all the components of all applications so that the generation process for a component can find any required location.

A second form of software reuse may occur when components are required that are identical except for the environment into which they are linked. For example two text editors may be identical except for the style of dialogue box that they display. The only difference between them is that they are linked to different dialogue locations. The duplication of these locations cannot be avoided; however the description of the generation process for each editor need not be duplicated.

The component generation process may be performed by an independent program. If the generator program for the text editor was parameterised by the locations that it required, it would not need to identify them itself. The generator program may now be reused since component creation is independent of any particular locations. The process of component generation is therefore as follows:

- **Find component locations**. Identify the locations that are to be used by the new value.
- **Call generator**. Call the generator program for this value, passing the locations as parameters. The new value is returned as the result of the generator.
- **Store value**. Store the value in the required location.

To support this, the architecture must contain a mechanism to allow the component locations of an application to be passed to generator programs.

A comparison can be made here between the construction operations of this architecture and the static scoping of block structured languages. A component is created in a block structured language in the context of an environment which is statically defined by the enclosing scope levels. The component may be linked to components contained in these scope levels. The passing of locations to generator programs that is performed in this architecture allows the enclosing environment of the new component to be created at construction time. During the execution of the application the appearance of components constructed using both methods will be identical - that of a component containing links to a number of locations in its closure. However greater flexibility in the construction of the enclosing environment is provided in this architecture.

Application Evolution

The degree of access required to applications and their constituent components changes during their lifetimes. Unrestricted access is desirable during the construction phase of components in order to permit sharing and evolution. The ability to evolve completed components may be restricted whilst access for sharing is still permitted, for example in a library of utilities to be used by developers. When an application evolves into a product, all access to the individual components of the software by its new users may be prohibited. This action, known as *sealing the system* [Car89], prevents accidental or malicious tampering with the software. Access can still be

provided to privileged users such as system engineers in order to permit maintenance of the software.

These levels of control over components and applications can be supported by the architecture if extra functionality is added to the location access mechanism.

The first step in achieving this goal is to partition the view of the component space provided by the access mechanism. This is analogous to the file systems of operating systems which provide a hierarchical access mechanism to a collection of files. This partitioning is shown in Figure 2.6. The component space itself represented by the rectangle in the horizontal plane is unchanged, with the same links between components and locations; it is the access mechanism which has been changed to provide the partitioning. Components may now be collected into logical groups relating for example to particular applications, utility libraries or data values.

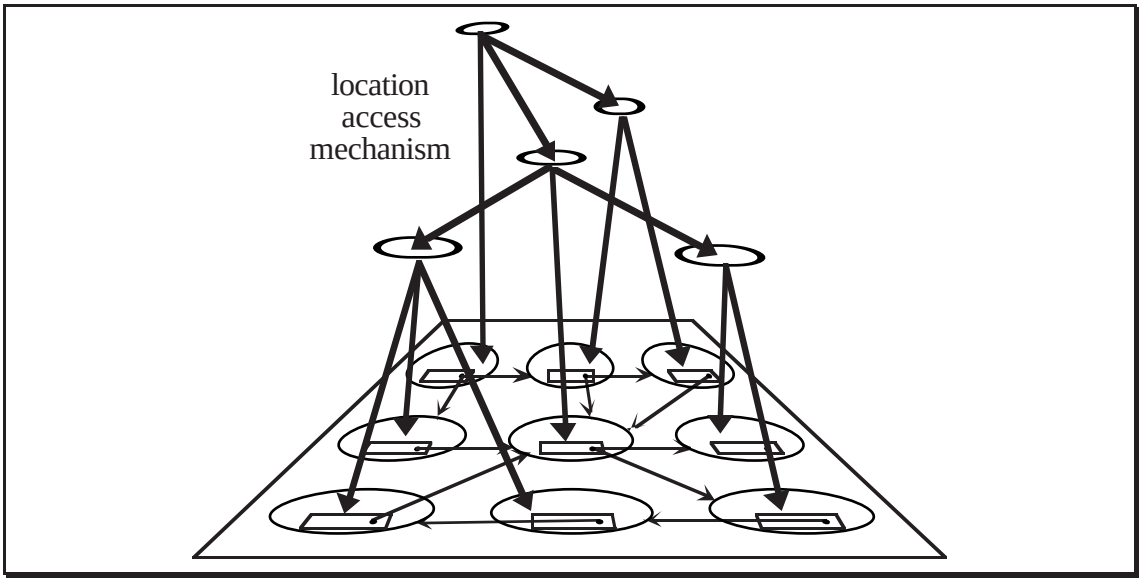

Figure 2.6 Partitioning the component name space.

Having partitioned the component space in this way the second step is to provide selective control over the sharing and evolution of components. This can be enforced by placing varying degrees of protection over particular access paths ranging from no protection at all to complete protection where its access path is removed altogether.

The restriction of access to components has no effect on the operation of applications that have already been constructed. Considering Figure 2.6, it is access to the vertical links that is being restricted, not the horizontal links made during component generation and used during application execution.

2.3 An Implementation of the Architecture

The architecture has been described so far in terms of components, shared typed locations and a linking mechanism that allows independently constructed components and locations to be linked together. A language to implement this architecture must therefore be able to support the following:

- The inclusion of application components within the value space.
- A type system sufficiently powerful to model the interfaces to the components.
- The ability to refer to the same location from different compilation units.

It should be noted that the language of the constructed applications does not have to be the same as the architecture's implementation language. For example, it is only a requirement that the components should be manipulable as values within the implementation language, not the constructed application's language. For the same reasons the application language need not be able to link components to locations

The ability to share strongly typed locations across compilation units is not shared by many languages. In a persistent system [Atk78], such locations may be held in a persistent store. A single persistent language and type system may be used both for the implementation of applications and the implementation of the architecture that supports their construction. This section describes an implementation of the architecture using the persistent programming language Napier88 which supports the construction of applications in that language. Following this, Section 2.4 shows how other languages may support the architecture.

2.3.1 Napier88

The facilities of Napier88 required to support the architecture are briefly introduced first. In Napier88, all values are first-class in accordance with the Principle of Data Type Completeness [Mor79]. Since application components must be included in the value space of the implementation language any Napier88 value can be a component.

Components constructed in different compilation units may be linked to the same location using the persistent store. Persistence in Napier88 is defined as reachability from a single distinguished root of persistence which is of the data type *environment*, or **env** [Dea89]. A value of this type is a collection of named typed locations. The persistent root may be retrieved by calling the predefined procedure *PS* of type **proc**(→ **env**). For example, the code in Figure 2.7 creates a new location in the persistent root environment with the identifier *textEditor* of type **proc**(**string** → **string**).

```
in PS() let textEditor := proc( initial : string → string ) ; "dummy result"
```

Figure 2.7 Creating a location in the persistent store.

Locations must be initialised on creation, in this case with the dummy procedure value following the *:=*. Having executed this program the location may be accessed in another program as shown in Figure 2.8.

```
use PS() with textEditor : proc( string → string ) in
        writeString( textEditor( "Here is some initial text." ) )
```

Figure 2.8 Retrieving a value from the persistent store.

The **use** statement makes an assertion that the persistent root contains a location named *textEditor* of type **proc**(**string** → **string**). Any uses of the identifier *textEditor*, available in the body of the statement following the **in**, may be statically checked with respect to this assertion. A dynamic check is performed to verify the

assertion when the **use** is executed: if successful the location is retrieved from the environment, otherwise the program fails. A dereference operation is implicit when the identifier *textEditor* is used, to give whichever value is in the associated location at the time of execution. The **use** statement and the dereference operation make up the two stage linking process described in Section 2. In this example the body of the **use** statement writes out the result of invoking the text editor on the supplied string.

Locations may be updated by assignment as shown in Figure 2.9 where the location with identifier *textEditor* is updated with a new procedure value. The assignment statement will not compile if the new value is of an unsuitable type.

```
use PS() with textEditor : proc( string → string ) in
        textEditor := proc( initial : string → string ) ; "a different dummy result"
```

Figure 2.9 Updating a location by assignment.

New empty environments may be created using the predefined procedure *environment* of type **proc**(→ **env**). Since environments are first class they may be placed into other environments such as the persistent root and are therefore used as a structuring tool over the values placed into the persistent store.

The three separate programs or compilation units above all manipulate the same text editor location during execution, as required. The use of these features in the implementation of the construction architecture will now be described.

2.3.2 Implementing the architecture

Creation of the Initial Locations

Following the phases of application construction covered in Section 2.2.1, the first step is to create the locations containing the components of an application. The dialogue/editor example will again be used.

```
!** Create a new environment.
let newEnv := environment()

!** Create locations containing dummy values.
in newEnv let dialogue := proc( message : string → bool ) ; false
in newEnv let textEditor := proc( initial : string → string ) ; ""

!** Place the new environment into the persistent store
in PS() let exampleEnv := newEnv
```

Figure 2.10 Creation of component locations.

Default instances of the components must be created at this stage since empty locations cannot be created in Napier88. The program shown in Figure 2.10 creates a new environment containing locations for the editor and the dialogue. This environment is then placed into the root environment of the persistent store.

Generators

Generators of the kind described in Section 2.2.3 may be modelled using Napier88 procedures parameterised by environments containing the component locations required for a particular generation. The generated component is returned by the procedure. Since these generators are first class values in Napier88 they may also be held in the persistent store. A program to create a generator for text editor components is given in Figure 2.11.

```
!** The generator takes one environment and returns an editor procedure.
let textEditGen := proc( dialogueEnv : env → proc( string → string ) )
begin
        !** Section 1: Retrieve the location required by the text editor.
        use dialogueEnv with dialogue : proc( string → bool ) in

                !** Section 2: Create the generator result, a first class
                !** procedure value.
                proc( initialText : string → string )
                begin
                        !** Code to construct the editor.
                        .....
                        !** The dialogue location may be used here
                        !** For example...
                        let reply := dialogue( "Save before closing?" )
                        .....
                        !** The edited text is returned as the result.
                        editedText
                end
end

!** Store this generator for future use in an environment named generators which
!** has been constructed already.
use PS() with generators : env in
        in generators let textEditGenerator := textEditGen
```

Figure 2.11 Generator procedure for a text editor.

Generators are usually coded in two sections: the first retrieves the required locations from the supplied environments; the second creates the value returned by the generator. A program to use the new locations created in Figure 2.10 and the generator of Figure 2.11 is shown in Figure 2.12. In this program, the editor generator and the example environment containing the new locations are retrieved; a new editor value is generated, which is then assigned to the correct location in the example environment.

```
!** Find the generator and the environment with the new locations.
use PS() with generators,exampleEnv : env in
use generators with textEditGenerator : proc( env → proc( string → string ) )
in
begin
        !** Create a new editor value
        let newTextEditor := textEditGenerator( exampleEnv )

        !** Update the textEditor location with the new value
        use exampleEnv with textEditor : proc( string → string ) in
                textEditor := newTextEditor
end
```

Figure 2.12 Program to create a new textEditor value.

Note that the value denoted by *newTextEditor* is linked to the location of the dialogue component at the time of construction, as required by the architecture. Any use of that location during execution requires no type or existence check. Referring back to Figure 2.4b, the checking is performed while traversing the vertical access paths; the links created during construction between components and locations are the arrows in the horizontal plane.

Execution

Components may be invoked by accessing their locations. For example, Figure 2.13 shows how the text editor component may be used in a program.

```
use PS() with exampleEnv : env in
use exampleEnv with textEditor : proc( string → string ) in
begin
        ....
        let editedText := textEditor( "Here is some text to edit." )
        ....
end
```

Figure 2.13 Executing components

Note that although access and type checking operations are performed to find the text editor location, the execution of the editor itself involves only a location dereference. Any references to other components during execution will be realised by similar location dereferences.

2.3.3 Support for the Features of the Architecture

Evolution of Components

To make a change to a component not affecting its type, the usual method is to edit, recompile and reload the generator for that component. Re-executing the program of the style of Figure 2.12 will cause the new generator to be used and hence a changed version of the component to be placed into its location. If a type change is required,

then a new location of the new type must be placed into the appropriate environment. Generators and programs using such locations must be adjusted accordingly.

Software Reuse and Control over the Component Space

The two styles of software reuse described in Section 2.2.3 are both supported. The first is the reuse of components in different contexts. This is supported since many different components may refer to the same location using persistent addresses. The second style of reuse allows the same code to be associated with different environments thus providing different functionality. The same generator may be used to create values bound to different locations by writing different versions of the program shown in Figure 2.12.

Techniques for handling application evolution were discussed in Section 2.2.3. These were partitioning the component space and restricting the access to component locations. The component space can be partitioned by the use of environments as a structuring tool in the persistent store as described in Section 2.3.1. The effect is identical to that shown in Figure 2.6.

There are many ways in which locations may be protected within a strong type system [MBC+90]: two are described here. The first involves the **drop** language construct, described by example in Figure 2.14.

```
use PS() with exampleEnv : env in
drop dialogue from exampleEnv
```

Figure 2.14 Dropping an access path.

This finds the *exampleEnv* environment in the root of the store and drops the binding between the identifier *dialogue* and the associated location. It is important to realise that if other values are bound to the location that is dropped then it will not be lost from the store. Only the ability to access the location via the environment *exampleEnv* has been lost. If Figure 2.4b described the situation before the drop, the situation is now as shown in Figure 2.15. Note that the links between values and locations are unchanged.

A second technique for restricting access is to place password protection over the environments. To protect *exampleEnv*, it may be enclosed in a procedure which returns it when supplied with the correct password. The code in Figure 2.16 creates this procedure and places it into the store. An empty environment is returned if the password is incorrect.

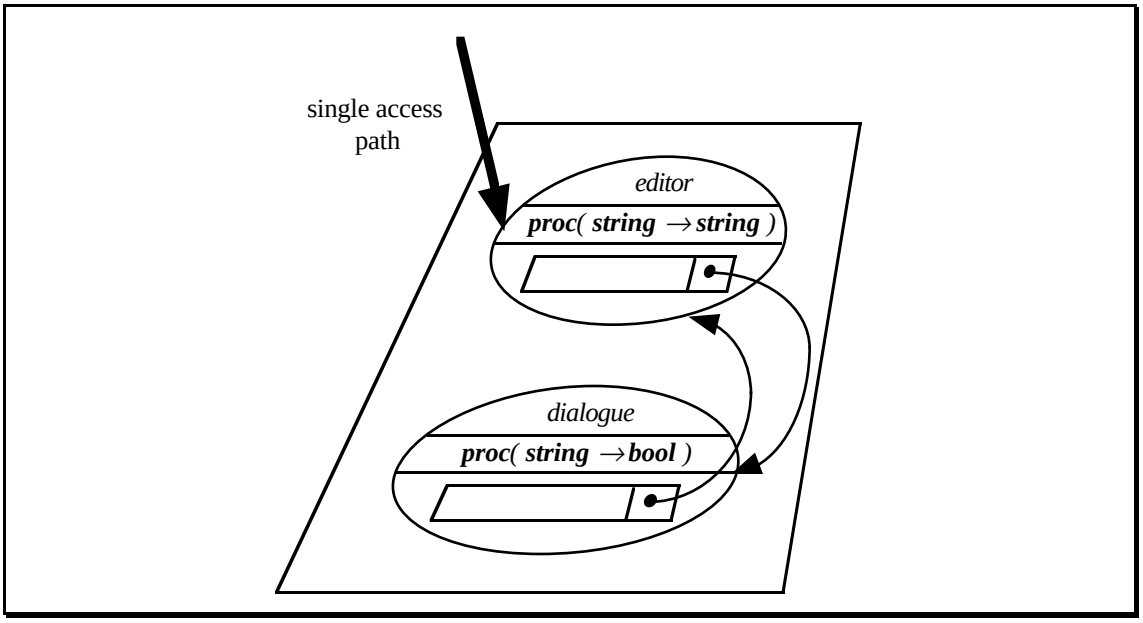

Figure 2.15 After the dropping of an access path.

```
let password := "abracadabra127"
in PS() let protectedEnv := proc( testPassword : string → env )
                                    if testPassword = password
                                    then exampleEnv
                                    else environment()
```

Figure 2.16 Password protecting an environment.

The environment may be accessed as shown in Figure 2.17.

```
use PS() with protectedEnv : proc( string → env ) in
use protectedEnv( "abracadabra127" ) with  .....
```

Figure 2.17 Using the password protection mechanism.

If the password is incorrect, the program will fail when locations are retrieved from the environment.

2.4 The Architecture Supported by Other Languages

Application components as first-class values and shareable strongly typed locations are the only major requirements of an implementation language for the architecture described here. These requirements are usually found in persistent environments but they may also be supported in non-persistent languages. The way in which these requirements are provided by different languages produces instances of the architecture with slightly different characteristics as described below.

A fore-runner of Napier88, PS-algol [PS88] can support the architecture using the fields of records as shareable locations. Dereference of these locations is explicit and

so the code for a component that uses external values will be noisier than in Napier88. Since the language is persistent and data type complete, applications will otherwise be very similar to those constructed using Napier88.

Standard ML supports the denotation of individual locations using the **ref** construct and allows code and other forms of data to be first-class values. It can therefore implement this architecture. SML is not persistent but most of the features of the architecture described above could be performed in a single invocation of the language's interactive system. The outermost scope level of the interactive system may be used to hold the shareable locations that are required by the architecture. The environment supporting the outer scope level behaves like a persistent environment for the duration of the invocation. Generating functions may be constructed taking locations as parameters which gives the same level of reusability as the generators of Napier88. Unlike Napier88, the use of an explicit location dereference operation is required in ML. There is also no simple method for removing access to locations or partitioning the location name space. Since the actions of an invocation of the interactive system do not persist beyond a single invocation, all components of an application will have to be re-entered and therefore recompiled when the system is next invoked.

Persistent ML [Har85] ensures that the bindings in the outermost scope level persist between invocations of the interactive system. This removes the requirement for application recompilation when the system is re-invoked. The persistent languages Quest and Galileo may also support the architecture in the same manner as Persistent ML. Quest and Galileo may also support protection of locations since access to them may be removed

The languages above can all be used as both implementation and application language in an instantiation of the architecture. Any language that can support shareable locations and model the values of another language should be able to implement the architecture for applications in the second language. For example, it may be desirable to write Pascal applications using this architecture in order to gain the benefits of the incremental linking. Pascal itself could not implement the architecture since it cannot support the requirements above. However Napier88 has a sufficiently powerful type system to model Pascal values and could therefore be used as the implementation language for an instantiation of the architecture that supported Pascal applications.

2.5 Conclusions

Systems supporting the construction of applications from individual components attempt to satisfy two conflicting desires. These are the desire to ensure efficiency and safety from failure during execution and the desire to permit flexible component and application evolution. Safety and efficiency may be gained by performing component linking operations before application execution. Flexibility is usually achieved by delaying these same operations until execution.

This chapter has introduced a new architecture which uses a blend of static and dynamic linking to satisfy both desires. Applications constructed using the architecture display the following features:

- Components may be independently compiled.
- Individual components may be incrementally linked into applications.
- Checks to ensure that components exist and are of the correct type are performed before application execution.

In addition, the architecture can be used as the basis of a software development environment with specific support for the following features

- Software reuse.
- Control over sharing and change of components and applications as they progress through the software lifecycle.

The flexibility for incremental linking is provided by a persistent environment supporting the sharing of strongly typed locations containing the executable versions of application components. The persistence mechanism is used to permit the type-safe sharing of structured executable versions of applications between the processes of linking and execution. Such sharing is not supported in traditional programming environments.

The architecture has been implemented using the persistent programming language Napier88, where it has been used to construct a compiler for Napier88 written in the language. This is an application consisting of more than 10,000 lines of code split between about 600 separate components and is described in detail in Chapter 4. It has also been used to construct window management, editing and browsing tools for the language. These components have been joined in an integrated programming environment written entirely in Napier88 which supports the construction of programs using the architecture [KCC+92].

3 Persistent Type Representations

3.1 Introduction

Type systems are well understood as mechanisms which impose static safety constraints upon a program. Within a persistent programming environment the type system provides all the data modelling and protection facilities for the environment. This role is similar to that of a traditional schema in a database system [ABM85]. Elsewhere [CBC+90] it has been demonstrated that not all constraints on data may be captured statically. This leads to a judicious mixture of type checking times being employed to suit the particular requirements of an application. The spread of times allows a balance between static safety in systems and dynamic flexibility in constraint expression and reuse. The type system may perform operations over program and data during the following processes which will be referred to as lifetime processes:

- Program construction and compilation. Type checking at compilation allows errors to be discovered earlier in the software lifecycle and eliminates the need for expensive type checks during execution.
- Linking. Type safe incremental linking may be supported in a persistent environment as described in Chapter 2. Type checking during the linking process allows programs to be constructed independently and linked as if they are a single unit. This is more flexible than the above with the disadvantage that a type error may occur later than compilation.
- Execution. Type checking during execution is the most flexible in terms of reuse and independent generation of program and data. It is however the least safe.

Program components and data are related by the common types associated with them. For example, the expected type of independently generated data must be known to the programmer so that valid operations may be constructed over it. The expected types of independent program components must also be available during construction so that the components can be correctly used by each other.

The focus of interest in this chapter is the manner in which type information may be distributed among independent programs and across the lifetime processes through which those programs pass. Traditionally, programs are completely independent and so each program must contain a copy of the type information that it requires. There is a conflict here between the type information and the code contained in each program: while the code is different in every program, the type information may be the same. In addition the various formats into which a single program is translated as it passes from source code to compile-time format to link segment to executable image may also require copies of the same type information. Much of the complexity associated with the implementation of type systems stems from a lack of support for the sharing of common type information across both independent components and lifetime processes. Few systems have overcome the technical difficulties associated with the transfer of complex type information between independent environments [ACP+91]. The ad-hoc language mechanisms supporting limited sharing of types between programs for example found in the languages C [KR78] and Quest [Car89] and the copying of type information between independent environments may be eliminated when the sharing of type information is supported between both components and processes.

A single set of operations to create, manipulate and test representations of type information may be constructed in a persistent environment. The operations may be

shared by the lifetime processes also supported by the persistent environment. Single instances of type information as opposed to copies may now be shared by programs and lifetime processes. The persistent environment is being used as a cache for type information from program construction through to execution.

Traditionally, the only type checking operation to be performed during the linking and execution processes has been type equivalence. In order to maintain execution speed the efficiency of this operation is optimised [CBC+90]. The availability of a single set of type system operations and the sharing of representations of type information between both programs and lifetime processes simplify the use of a type system operation. The implementation of other complex type operations may also require optimisation to prevent their use seriously affecting system performance. A format for the type information is required which may be used in efficient implementations of all the type checking operations that are carried out in a persistent system.

The discussion here relates to reasonably sophisticated type systems such as those found in ML [Mil84], Galileo [ACO85], Quest [Car89] and Napier88 [MBC+89]. The type rules are defined by a set of base types and a set of constructors. The universe of discourse is the closure of the recursive application of the constructors over the base types. Typically the constructed types contain labelled cross products (records), labelled disjoint sums (variants) and functions. Universal quantification of functions and existential quantification of records as well as recursively parameterised types are supported. In general the systems will use structural equivalence. The details of such type systems are found in [Con91]. Some of the operations associated with these systems are inherently expensive, irrespective of the type representation format. In addition to the caching of type information, the persistent store may also be used to cache the results of such inherently expensive operations in order to improve their performance.

The research presented in this chapter derives from the construction of the Napier88 compiler within the persistent environment. Napier88 Release 1.0 contains two independent type checking universes. The first is used to perform static type checking within the compiler which is implemented in PS-algol [PS88]. The second supports dynamic type checking during the execution of the Napier88 persistent environment.

In order to perform dynamic type checking, type information must be passed from the compilation environment to the execution environment. The initial implementation of this type information transfer reacted poorly with the application construction architecture of the previous chapter resulting in multiple copies of type information. The problems were so severe (tens of megabytes of redundant type information) that the type checking implementation had to be re-engineered in order that the construction architecture could be sensibly used.

A new type representation for the type information was designed for use within the persistent environment which represented a good trade-off between time to execute type checking operations and space to store the type information. The initial implementation of the compiler within the persistent environment was completed using an ad-hoc mechanism not described here to avoid multiple redundant copies of type within the environment. Reimplementation also allowed insight to be gained into the nature of type checking within an integrated persistent programming environment.

This chapter analyses the many different type system operations found in persistent programming systems and determines in which lifetime processes each is found, demonstrating the widespread use of the type system. The operations may be split into those that make type information available in the system (the producers) and those that directly manipulate the information itself (the consumers). Implementation of the producers in a persistent environment is described, showing how a single set of operations and single instances of type representations may be shared by all programs

and software processes. A representation for type information is presented that supports efficient implementation of the type manipulation operations. The final section discusses the use of the persistent store as a cache for the results of inherently complex operations over types.

3.2 Type System Operations in Persistent Systems

Type information is associated with all applications. It describes the data being manipulated in terms of the data model of the programming language supporting the application. The type information exists in three distinct forms:

- **Type concepts**
  These are the type models created by the programmer that describe the data to be manipulated by the application. They form the programmer's understanding of the type information associated with the application.

- **Type denotations**
  A means of passing type concepts from the programmer to the programming system is required in order to perform mechanised type checking. This is achieved by translating the type concepts into formal denotations that may be interpreted by the system. A denotation in this context is a textual representation of a valid type concept in the type system. In addition the system may pass type information back to the programmer via denotations.

- **Type representations**
  Type denotations are converted by the system into internal representations over which type checking operations may be performed.

There are a number of operations associated with these three forms which are performed either by the programmer or by the programming system. They fall into nine groups as depicted in Figure 3.1. Only a single format for denotations and for representations is considered at the moment.

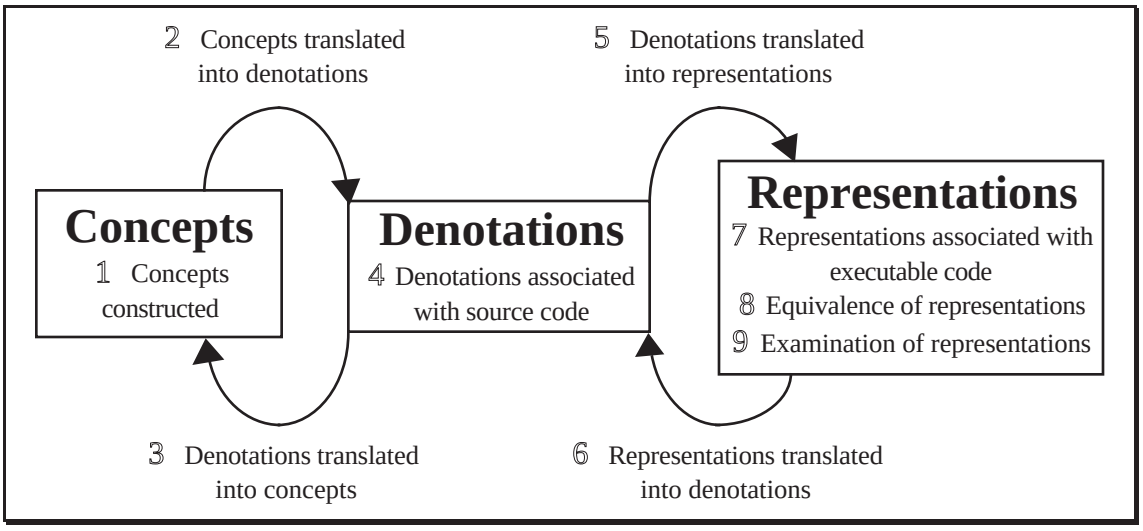

Figure 3.1 Operations over type information.

3.2.1 The Type System Operations

The nine groups of type system operations may be described as follows.

1 Construction of Type Concepts

Using the language data model the programmer designs the types that describe the data associated with an application. For example, a record type may be created to represent a person, an instance of which contains a name and an age of the base types *string* and *int* respectively. Further data models may be constructed with reference to the person type. For example, a type to describe a research grant may be a record with a grant holder of the person type and a grant award of type *int*.

2 Conversion of Type Concepts into Type Denotations

The programming language supports a type algebra in which type concepts may be formally expressed. Formal expression of type information is required so that the type concepts created by the programmer may be submitted to the system. The programmer converts type concepts into denotations in the type algebra of the language.

Figure 3.2 gives the formal denotation of a type describing a person. This and all other denotations given here are written in the type algebra of Napier88. The Napier88 type system is outlined in Appendix 1. The algebra supports the naming of data types to avoid repeated denotations of the same type information. The identifiers assigned to data types form no part of the type information. In this case the identifier for the type is *Person*.

```
type Person is structure( name : string ; age : int )
```

Figure 3.2 A type denotation.

Denotations may contain references to other denotations already created. The denotation in Figure 3.3 denotes a research grant type and refers to the person type using the identifier *Person*.

```
type ResearchGrant is structure( holder : Person  ; award : int )
```

Figure 3.3 Referring to an existing denotation.

Parameterised type definitions may be used to construct type denotations that are similar but not identical in structure, reducing the complexity of multiple descriptions. For example the denotation identified by *Related* in Figure 3.4 describes a record type containing two fields of the same type. The exact type of the fields is abstracted from the denotation and referred to by the identifier *t*.

```
type Related[ t ] is structure( first,second : t )
```

Figure 3.4 Parameterised type *Related*.

Whilst not a type itself, *Related* specifies an infinite group of types, each of which may be realised by specialising the definition with a type to replace the abstracted type. Figure 3.5 shows two such specialisations.

```
type FamilyRelationship is Related[ Person ]
type SimilarResearch is Related[ ResearchGrant ]
```

Figure 3.5 Specialising a parameterised type.

FamilyRelationship may be used to model relationships between two people in a family; *SimilarResearch* may model pairs of research grants for related work areas.

It is not essential that all types be named. Anonymous type denotations may be included in the source code in situations where the trouble of writing the anonymous denotation is no greater than that of writing the named denotation and then referring to it. A typical example is **int* which is the denotation for a vector of integers in Napier88. This denotation is so simple that naming it is unnecessary.

3 Conversion of Type Denotations into Type Concepts

The construction of denotations must be a reversible process so that the programmer is able to understand type information returned from the system.

4 Association of Type Denotations with Program Source Code

A mechanism is required that associates type denotations with program source code where they may be found by the compilation system. Denotations may be included in program source code or constructed separately and then referred to from the source code.

5 Conversion of Type Denotations into Type Representations

Type denotations are translated into type representations usually at or before compilation so that they may be manipulated more efficiently in the type checking process. This may entail parsing the type denotations and using existing representations to create new representations. For example, the construction of a representation for the type *Person* in Figure 3.2 requires only parsing of the denotation since there are no references to any types previously constructed. However, constructing a representation for *ResearchGrant* in Figure 3.3 requires access to the representation for *Person* in order to determine the correct information for the new representation. Further, the specialisations in Figure 3.5 require access to representations for both the parameterised type constructor *Related* and the specialising types. The specialised representation is constructed by copying the parameterised constructor, replacing the parameterised types found in the constructor by the specialising types.

The system ensures that the denotations constructed by the programmer represent valid types according to the data model of the language.

6 Conversion of Type Representations into Denotations

The system occasionally makes type information available to the programmer, requiring a conversion from a representation to a denotation. This may occur for

example when the programmer is manipulating language values with a tool such as a browser [DB88,Far91] or when type errors occur.

With structural equivalence there are an infinite number of denotations for a given representation since the choice of identifiers for named types is arbitrary and they are not part of the type information. However, the identifiers used in a particular set of denotations are usually given semantic information by the programmer relating to the expected use of the types they represent. For example, values associated with the equivalent types *moonRocket* and *fish* in Figure 3.6 may be freely intermixed in any program manipulating structured values containing two fields named *length* and *noOfFins* of type *int*. However, receiving a denotation called *moonRocket* in an angling program will probably lead to confusion.

```
type moonRocket is structure( length,noOfFins : int )
type fish is structure( length,noOfFins : int )
```

Figure 3.6 Equivalent types.

The point here is that denotations produced by the system should contain information that is not misleading. Where the system cannot guarantee that a type created in one context will not end up in another context it should ensure that identifiers for type denotations contain no contextual information.

7 Association of Type Representations with Executable Code

A mechanism is required that associates type representations with executable code where they may be found for type system operations performed during execution.

8 Type Equivalence Checking

Type equivalence checking is the primary type checking operation that occurs in persistent systems. In order to perform a structural type equivalence check, a representation defining the set of operations over values of the type is required for each type. The check may be expensive since a complete traversal of both type representations is required. This traversal is of similar complexity to a check for equivalence over two graphs, although structural equivalence is not necessarily implemented in this way. The expense of the traversal is also related to the amount of information stored in the representation which is dependent on the data models supported by the type system. The optimum efficiency for equivalence checking may be achieved if two representations can be shown to be identical instances since a full structural check is then not necessary.

9 Examination of Type Representations

The system examines the information contained in the representations in order to carry out many operations. The information may be used for type checking purposes or for other operations such as address calculation and the generation of code to manipulate values of the type. The system requires operations over type representations to extract information such as the constructor used to build a particular type, the names and types of fields in a record type, the number and types of the parameters to a procedure or the type of the elements in a vector.

3.2.2 Using the Type System Operations

The use of the type system operations in a persistent environment will be demonstrated here in the framework of the three lifetime processes of program construction, compilation and execution. As described in Chapter 2, the linking process may be supported in a persistent environment by executing programs that manipulate first class values in the environment. It is therefore included in the execution process here.

Two programs that independently generate and manipulate shared data will be used to illustrate the type system operations. The program shown in Figure 3.7 creates a value of the type *Person* described in Section 3.2.1 and places it into the persistent environment. The identifier *Person* from the type denotation is being used as a constructor function for values of that type. Initialising field values of *"quintin"* and *25* are supplied to the function. ***in*** *PS()* ***let*** indicates that the declaration of the identifier *quintin* is to be made in the persistent environment and not in the local scope.

```
in PS() let quintin := Person( "quintin",25 )
```

Figure 3.7 Creating a persistent value.

The program shown in Figure 3.8 accesses the value placed in the persistent environment by the program of Figure 3.7 and increments its *age* field. To ensure safe application of operations to the value some type checking must be performed during execution. The mechanism used in Figure 3.8 depends on the compiler's making and verifying assertions about the expected types of values in the persistent environment at execution. The first line of Figure 3.8 makes an assertion that the persistent store will contain a value named *quintin* of type *Person* when the program executes. The code following the ***in*** is type checked during compilation with respect to this assertion. During execution the assertion must be verified before the program can access the data and the code can safely be executed.

```
use PS() with quintin : Person in
    quintin( age ) := quintin( age ) + 1
```

Figure 3.8 Accessing a persistent value.

The type system operations associated with the application of each process to these programs will now be described.

Construction

Before program construction the programmer must create the type concepts modelling the data to be used in the program. The concept in the example programs is the model of a person described by a name and an age of the base types *string* and *int* respectively.

The concepts and denotations are used by the programmer to aid the construction of valid code. For example, to write code that constructs a value of the *Person* type the programmer must know the types of the fields. To access the fields of a *Person* value the field names must be known.

The concepts are translated by the programmer into denotations and made available for program construction. The type denotation for the example programs is shown in

Figure 3.2. The denotation is associated with both programs. For example this may be performed by writing the denotation in each program.

An alternative method of determining type concepts may occur in a persistent environment since values in the environment may be accessible during program construction. The system may allow the programmer to browse over these values, displaying denotations of the type representations associated with the values. Having determined that the type concepts associated with these existing denotations are suitable, the programmer may associate these denotations with the programs under construction. This is the first step towards hyper-programming [KCC+92] where existing language values as well as existing type denotations may be associated with program source code.

Of the operation groups described in Section 3.2.1, those used during the construction of programs are as follows:

1 Creation of type concepts.
2 Conversion of concepts into type denotations.
3 Conversion of denotations to concepts.
4 Association of denotations to source programs.
6 Conversion of type representations to denotations.
9 Manipulation of type representations (during browsing).

Compilation

Type representations are constructed from the denotations at any time before they are required for type checking purposes although traditionally the conversion takes place during compilation. Checks during conversion ensure that the denotations represent valid types according to the data model of the language.

Code constructed by the programmer may be checked for type correctness during compilation if the type of the data can be determined by an examination of the source code. The type representations describing the data are the arguments to the type checking operations. Compile-time type checking may be performed on the program of Figure 3.7 to ensure that the types of the initialising values of the new *Person* record are equivalent to the types of the fields to which they are being assigned. That is, checks are performed to ensure that the type of the value *"quintin"* is equivalent to *string* and that of the value *25* is equivalent to *int*. Using a representation for *Person*, type checking in the program of Figure 3.8 ensures that *Person* has a field named *age*. The type of this field must be equivalent to the type of the first argument to the addition operation. Finally, the result type of the addition operation must be equivalent to the type of the *age* field in order to permit the assignment.

Type errors detected during program compilation are reported back to the programmer. The type representations involved in the type error must be reconverted into denotations for display so that the programmer can understand the error.

Both example programs require that a type representation for *Person* is associated with the executable code produced by the compiler. Values reachable in the persistent environment are self-describing in order to allow the late binding of independently prepared program and data. The executable version of the program in Figure 3.7 requires a representation for *Person* that will be associated with the value placed into the persistent environment during execution. The program of Figure 3.8 requires the representation so that verification of the compile-time assertion on the type of the value associated with the identifier *quintin* may be made.

The operation groups used during the compilation of programs are as follows:

3 Conversion of denotations to concepts.
5 Conversion of denotations into representations.
6 Conversion of representations into denotations.
7 Association of representations to executable code.
8 Type equivalence checking.
9 Manipulation of type representations.

Execution

When the program of Figure 3.7 executes, the type representation associated with the executable version of the program must in turn be associated with the newly-created *Person* value as it is placed into the persistent environment.

Verification of the assertions over the expected state of the persistent environment may be made during execution when the actual state has been discovered. The verification operation is one of type equivalence between the representation expected by the executing program and the representation associated with the value being accessed. For example in the program of Figure 3.8, equivalence is determined between the *Person* representation associated with the program and the representation associated with the value identified by *quintin* in the persistent environment.

Subsequent execution of the program may continue if the verification of the assertion is successful. Otherwise the program terminates. The representations of the types involved in the failed verification are converted into denotations and reported to the programmer.

The contents of the persistent environment may grow in an arbitrary manner according to the programs that are executed against it. Tools such as browsers may be required to allow users to discover the contents of the store. During the construction of a browsing program the programmer cannot anticipate the types of all values that will be found in the persistent environment during execution. The browser is therefore an adaptive program that incrementally learns during execution how to browse values with types that have not previously been encountered. The browser requires the ability to manipulate type representations in order to browse these new values. Conversion of representations to denotations is also required so that the browser can display the types to the user.

Some representations cannot be constructed until program execution because they depend on language values that are not available before that time [Con92]. In these cases, some of the operations used to convert denotations into representations and to manipulate representations may be required during execution.

The operation groups used during the execution of programs are as follows:

3 Conversion of denotations to concepts.
5 Conversion of denotations into representations.
6 Conversion of representations into denotations.
8 Type equivalence checking.
9 Manipulation of type representations.

3.2.3 Sharing Type Information and Operations

Section 3.2.2 shows that the same type information may be used in more than one program and in more than one of the processes through which programs pass. In

addition, the type information may be used in more than one of its different forms in each process. Considering the program of Figure 3.8, the concept and denotation for the type *Person* are used during program construction, the denotation and representation are used during compilation and all three forms may be used during execution depending on the result of the type verification operation.

Sharing of type information may therefore occur in three dimensions:

- The type information may be used in the three different forms of concept, denotation and representation.
- Co-operating programs such as those making up an application may operate over shared data and will therefore share the type information associated with that data.
- Type information associated with a single program may be used in and therefore shared by more than one of the lifetime processes through which the program passes.

In addition, many of the type system operations are used in each process. For example the examination of type representations may occur during program construction, compilation and execution.

3.2.4 Summary

This section has demonstrated that type system operations occur throughout the lifetime processes. The objective of this chapter is to show how they may be efficiently implemented. The nine operation groups associated with a type system that have been described here may themselves be split into three groups as follows:

1. Operations involving type concepts, groups 1 - 3.
2. Operations that make type information available throughout the system, groups 4, 5, 6 and 7. These are the producers of type information and involve the translation of type information between denotation and representation, the sharing of type information between programs and the transfer of type information between the lifetime processes.
3. The remaining operations are concerned with representations only, groups 8 and 9. These are the consumers of type information and involve type equivalence over and information retrieval from type representations.

The operations over concepts are carried out in the programmer's head and are not of concern in this discussion. Of interest here is the efficient implementation of the type operations performed by the system supporting a persistent programming environment. Efficient production of type information is essential since its use is widespread in the different forms, lifetime processes and programs. Similarly, efficient execution of type system operations over representations throughout the system is desirable. The remaining sections discuss the implementation of the type system operations in groups 2 and 3 using persistence to support the implementation.

3.3 Type Information Producers

The ease with which the various system components such as the lifetime processes and programs may refer to the type denotations and representations depends on the availability of type information throughout the system. The disjoint collection of components that are traditionally used in programming environments do not permit efficient inter-component references and so a number of ad-hoc mechanisms have been

constructed to effect the required distribution of information. These usually involve multiple copies of the same type information in different representations each requiring its own set of type system operations.

The implementation of a programming environment may be based on a persistent environment in which references between system components and complex type information are maintained by the persistence mechanism. Here a single set of type system operations may be used by all processes. Instances of denotations and representations constructed by the operations may be shared by the lifetime processes and also by the programs supported by the programming environment.

The manner in which type information is made available to the system will now be described in more detail according to the three dimensions of sharing discussed in Section 3.2.3.

Conversion Between Denotation and Representation

Complex type denotations used throughout an application are usually constructed and named by the programmer before construction of the programs. Reference may be made from the source code of the programs to the shared denotations. The type representations associated with these denotations may be constructed at any time before the representation is required for type checking purposes. Since these types are usually complex the denotation can be linked to the representation to avoid repeated execution of the conversion operation. This also avoids excessive space consumption which can be caused by multiple representations since each one may be retained for use during execution.

Anonymous type denotations are included in the source code of many programs in situations where it is inconvenient to name the type. Conversion of these denotations to representations is required on every compilation. However, they usually denote simple types or types that extend an existing shared type, for example ***proc(*** → *complexType)* where *complexType* is a large named type. The conversion of these types need not be expensive providing that the representation for the shared types may be shared in the new representation.

Making a link from representations to denotations in order to effect the reconversion operation may lead to confusion over the names used to refer to component types as described in Section 3.2.1. A conversion operation is therefore required that maps representations to denotations whose inter-component references contain no contextual information.

In order to ensure a consistent use of names between all components in a denotation without using a system-wide naming scheme, each type representation in the system in general requires a complete self-contained denotation. Attempting to cache all these denotations is prohibitively expensive and so denotations are constructed from representations when required. The time spent constructing the denotation is not important since the system only requires denotations at points of interaction with the user when speed is not essential.

Sharing Type Information Between Software Processes

Figure 3.9 illustrates the transformations on the persistent store as the program of Figure 3.7 passes through the lifetime processes of construction, compilation and execution using a single shared type denotation and representation, $\mathbb{TD}$ and $\mathbb{TR}$, that have already been constructed for the type *Person*.

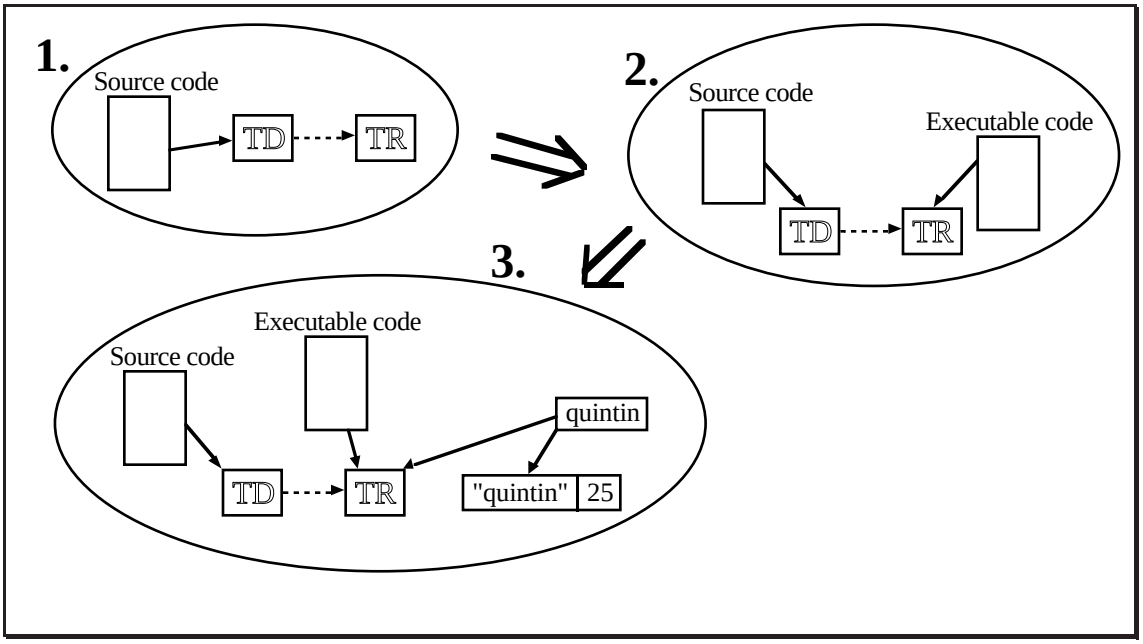

Figure 3.9 The persistent store during the development of a program.

The numbered stages are described as follows:

1. The source code of the program is created and linked to the type denotation.
2. The compilation process can access the type representation for type checking purposes via the type denotation and produces an executable version of the program linked to the type representation.
3. Having executed the program, the store contains an entry with the identifier *quintin* linked to both a value of type *Person* and the type representation.

Sharing Type Information Between Programs

Many programs resident in the store may share the same instance of a type denotation and representation. For example consider the persistent store depicted in Figure 3.10. This contains the source and executable code of the programs in both Figures 3.7 and 3.8 and the value placed in the store during the execution of the first of those programs. Note that all of these items contain links to a single type denotation - representation pair.

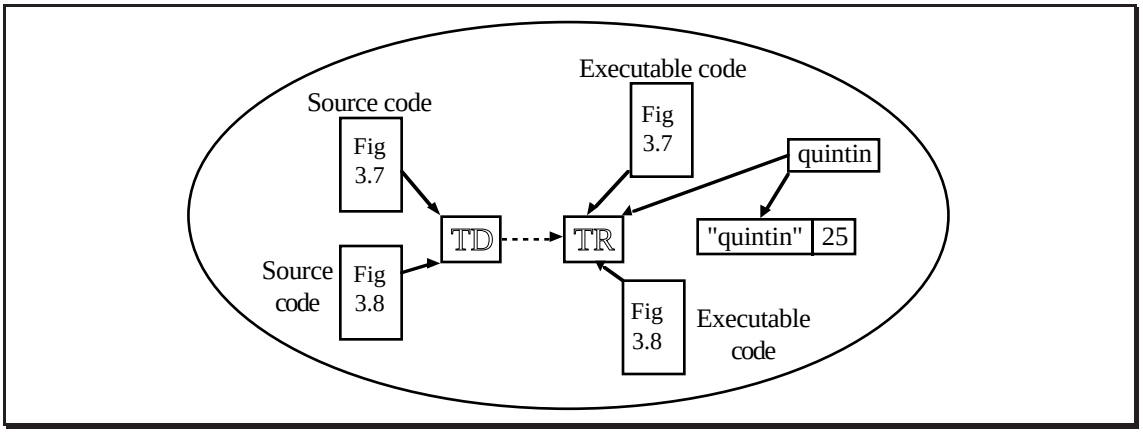

Figure 3.10 Separate programs sharing a single type representation.

3.3.1 Assessing the Implementation

The advantages of the techniques described above all stem from the ability to share instances of denotations and representations:

- Multiple instances of the same denotations and representations are not required in each program.
- Different denotations and representations of the same type information are not required in each lifetime process.
- Only a single set of type operations is required since there is only one representation over which to operate. The operations are shared by all lifetime processes.
- The results of converting complex types from denotation to representation may be cached.
- The efficiency of structural type equivalence checking is optimal. Considering Figure 3.10, the type equivalence check performed by the program in Figure 3.8 to verify the assertion over the type of the value identified by *quintin* in the store will succeed on an identity check since both the executable program and the identifier *quintin* are linked to the same type representation instance.

In addition, completely independent construction of program and data is still possible since the system does not preclude the construction of independent but equivalent type denotations and representations. Where independent construction is required the only effect will be a loss in the efficiency of the type checking operations. The reduction in efficiency may be minimised by using the persistent store to cache the results of type checking operations as described in Section 3.5.

3.3.2 Multiple Type Representations

The use of a single type representation shared by programs and lifetime processes has been considered so far. Alternatively, multiple representations of the same type information may be used by different type system operations in different lifetime processes. Greater efficiency may be achieved since the representations can be optimised for the operations applied to them. For example a compact representation may be desirable during execution to improve system operation as opposed to another used during compilation from which type information may be more efficiently retrieved.

As shown in Section 3.2 type system operations may be performed over type representations from different lifetime processes. For example the compilation of a program may involve type representations created during both the construction and execution processes. When multiple representations are used operations may be required over type information stored in different representations. There are two ways in which these operations may be carried out:

- The representations may be translated into a single format suitable for the operation being performed. The space consumed by multiple copies and the cost of translation may be significant.
- Specialised versions of operations may be constructed for each combination of representations encountered. This may cause a code explosion if there are many combinations.

An analysis of system operation is required to determine the trade-off of increased efficiency in the execution of the type system operations against the space required for multiple representations and operations.

3.3.3 Summary

The persistent store has been used here as a cache for type denotations and representations. The caching avoids multiple copies of the same type information appearing in different programs and the lifetime processes through which they pass. In addition the results of conversion operations between the denotations and representations of complex types are also cached.

The persistent store may also be used as a cache for the results of those operations that are inherently expensive or those less well-suited to a particular representation. The results of operations over multiple representations may also be cached. An analysis of the system will show whether the gain in efficiency justifies the expense of maintaining the cache.

Section 3.4 describes a single type representation that in conjunction with the caching of operation results described in more detail in Section 3.5 provides at least satisfactory efficiency during system operation.

3.4 A Type Representation

The persistent environment is used in Section 3.3 to make type information available throughout the system by supporting a single set of type operations over denotations and representations that may be shared among both programs and lifetime processes. In this section a type representation is described which gives at least satisfactory performance for those operations acting on type representations. These are:

- Construction of representation instances, in particular construction involving the use of existing instances.
- Type information retrieval from a representation instance.
- Type equivalence of two representation instances.

The representation to be described here has been used in the implementation of the Napier88 programming environment, the type system for which is outlined in Appendix 1. It is suitable at the least for any language whose type system supports a subset of the Napier88 type constructors. These are vectors, labelled cross product types, labelled disjoint union types, universally quantified procedures, existentially

quantified abstract data types and infinite union types. Parameterised and recursive types are also supported.

Protecting the Integrity of Type Representations

It should be noted that the integrity of the whole Napier88 system depends on the integrity of the type information used by type checking operations. Traditionally type checking occurs during compilation and the compiler is trusted to manipulate type representations in an appropriate manner. Where type representations are to be manipulated by untrusted code, a protection mechanism is required to ensure their integrity. Suitable protection may be provided using the Napier88 abstract type mechanism described in Section 4.4.1. The Napier types module [Con88] is an abstract data type and the type representation manipulated in the persistent system is a witness type of this module.

The use of values of witness type for this style of protection depends upon a particular style of witness type checking. This style is more flexible than that of Napier88 release 1.0 [MBC+89] but is still statically checkable. It is discussed along with its implementation in Section 4.4.

3.4.1 Choosing Representation Characteristics

There are a number of possible characteristics of a representation that affect operations over representations:

- The construction of and information retrieval from representations may be improved where they are constructed using other representations as components. Structural type equivalence checks and the use of space may also be more efficient.
- It should be possible to traverse representations efficiently since this improves the efficiency of information retrieval.
- Representations should be compact since a reduction in the use of space improves the operation of the system as a whole.

There is a conflict here between the first two characteristics and the third characteristic which concerns the addressing between components making up a particular representation. A typical representation consists of a number of component parts containing references to one another. Efficient construction and traversal may be achieved using a structured representation in which persistent addresses are used to link the components of a representation together. Alternatively, a flat representation is likely to be more compact since the context of the addresses of the various components of the representation is limited to the representation itself and so the addresses themselves may be much smaller than a system wide persistent identifier. The majority of the information in most representations records inter-component references and so the difference in size between a flat and a structured representation is likely to be significant.

Global addressing permits the sharing of independently constructed representations which may also be a major factor in space consumption. Because of the limited scope of the references inside a flat representation it is impossible to share components between representations.

Both structured and flat representations therefore reduce space consumption but in different ways. The types involved in a persistent programming system such as that supported by Napier88 are typically large as demonstrated later in this section. To prevent the space used by the intrinsically large representations associated with these

types from seriously affecting the performance of the system, it is important to find the representation that is most space efficient overall. Some measurements are included on two different type representations. The first is a terse textual representation and the second permits the sharing of component instances.

The measurements were made using representation instances of a Napier88 abstract syntax tree. This is a variation of PAIL [Dea87] and is defined as a set of mutually recursive types containing about 140 component types.

The size of the textual representation of the abstract syntax tree is 2,206 bytes compared to 14,466 for the structured representation. In the construction of the Napier88 compiler the type is used as a component type of 276 other types. If no sharing of component types is possible then each of the 276 types requires a separate copy of the abstract syntax tree type representation. This amounts to a space overhead of over 600,000 bytes for the textual representation which does not support sharing of components as opposed to a constant 14,466 bytes for the structured representation which does support such sharing. Therefore, depending on the use of the representation instances, the ability to share components has a more significant effect on overall space overheads than the size of an individual type representation.

The speed of a full structural equivalence check over the textual representation is faster than that over the structured representation. Checking independently constructed versions of the abstract syntax tree representation is of the order of several hundred times faster for the textual representation than the structured representation. However, full structural checks are rarely required when the sharing of representation instances is possible since they will frequently succeed immediately on an identity check. The structured representation's ability to share components promotes the sharing of representations throughout the system. In a persistent system where the sharing of instances is promoted across all programs and lifetime processes, a structured representation appears to satisfy all of the desirable characteristics of easy construction, traversal and compactness and it is therefore this style of representation that is described here.

3.4.2 **Type** *Type*

The format used for all type checking in the Napier88 system may be written down as the following Napier88 type:

```
rec type Type is structure( label,misc,random  : int ;
                            name               : string ;
                            others             : var )
&        var is variant( none,unique : null ; one : Type ; many : *Type )
```

Figure 3.11 Type *Type*.

A Napier88 *structure* is a labelled cross product type; a *variant* is a labelled disjoint union type. The interpretation of the fields is as follows.

label This records either that the representation is of a base type or if not then the kind of constructor it is.

name,misc These form the information specific to this type, for example the name of a base type, the concatenated field names of a record type, the number of branches in a variant type or the number of parameters to a procedure.

random This is a pseudo-random number used to optimise type caching as described in Section 3.5.

others This is a field of the variant type *var*, recording the component types making up this type.

The interpretation of the variant labels is as follows:

none There are no component types for this type

one There is a single component type for this type

many There are many component types. They are stored in a vector, denoted by the * in the type description.

unique There are situations where a type equivalence check should only succeed if the identity of the representations is the same, that is to say, strict name equivalence is required. This occurs when checking universally and existentially quantified types. If the *unique* label is discovered during an equivalence check, then name instead of structural equivalence is used for this type.
It is possible to implement some classes of dependent types using this label, since a one to one correspondence between type representations and values may be constructed.

Some instances of the structure of representations for Napier88 types using this format are illustrated in Figure 3.12:

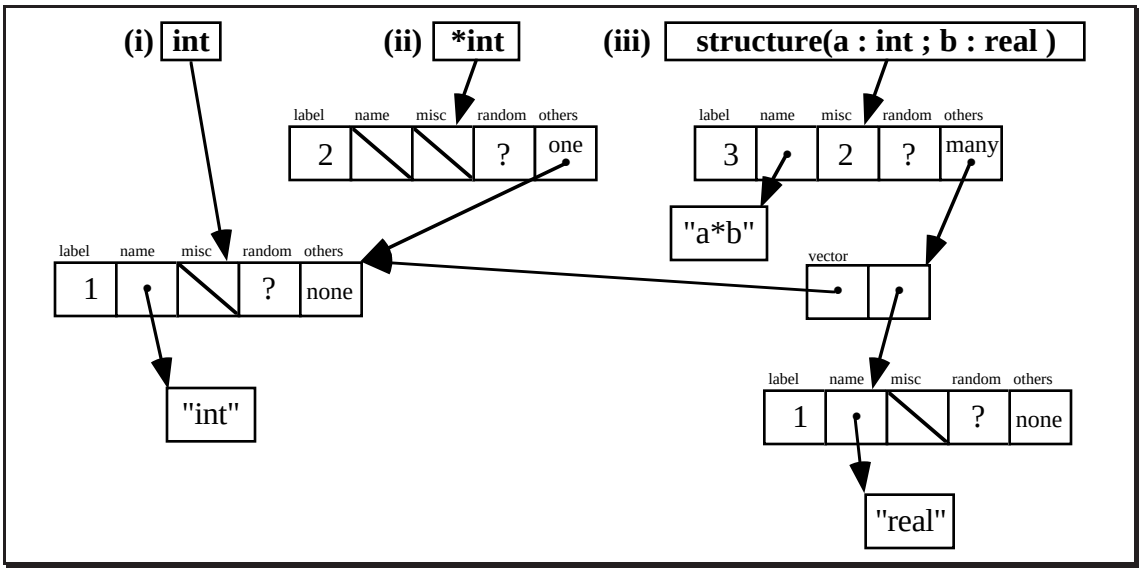

Figure 3.12 Instances of the universal representation format.

The first instance *(i)* is for the base type *int*. The *1* contained in the *label* field specifies that the representation is a base type. The information associated with a base type is its name which is contained in the string attached to the *name* field.

The second instance *(ii)* is for a vector whose elements are of type *int*, denoted **int*. The *label* is *2* denoting a vector constructor. The information associated with the vector is the type of the elements of the vector which is recorded using the *one* branch of the variant *var* and points to a type representation of the type *int*. Note that the representation for *int* used here is the same as that in the first instance. The vector type representation is using an existing type representation for its component type.

The third instance *(iii)* is for a structure containing two fields named *a* and *b* of types *int* and *real* respectively. The *label* is *3* denoting a structure constructor. The *name* field contains the field names of the structure concatenated into a string, separated by * characters. The *misc* field records the number of fields in the structure, two in this case. The component types of the structure, *int* and *real*, are recorded using the *many* branch of the variant *var* which points to a vector containing one entry for each component in field order. Sharing of the component type representation for *int* is taking place between all three instances.

3.4.3 Using *Type* for all Type Checking Operations

The design of the *Type* representation is based on a trade-off between space and time. The representation may be used by all type system operations to achieve at least satisfactory performance whilst some achieve good performance. The space used by the representation is low because of its ability to share components.

The use of space has also been minimised by reducing the number of objects created by the system for a particular representation. In the Napier88 system, the maximum number of objects per component type is three. Every component requires an instance of the *Type* structure, which is implemented as one object. The name field is a string which is implemented as a separate object. The variant is stored within the *Type* object. However if the variant contains a value that is implemented as a separate object then this will necessarily be external to the *Type* object. The third object that can occur in the representation of a single type is required when the type contains many components. The variant field is instantiated to the *many* branch when this occurs. The vector required for this branch is implemented as a separate object.

In order to minimise the number of objects in a representation instance, some type information is densely encoded. This increases the complexity of some operations such as type decomposition. An example is the concatenation of the field names of a record type into a single string. Whilst this improves the efficiency of full structural checking, complex string manipulation functions are now required to determine individual field names. The complexity is manageable since it may be restricted to a few of the type manipulation functions encapsulated inside the implementation of the type system operations.

In this section a representation was described that supported at least satisfactory performance of operations over representation instances. The results for each of the three operations are as follows:

- Composition of type representation instances. The ability of a single representation instance to be a component of many others, as shown in Figure 3.12, simplifies the composition of new instances.

- Retrieval of type information. The structured nature of the representation simplifies the traversal of components that is usually required when gathering information from representations. However, analysis of information in a single representation is also required which is more complex because of the dense encoding of type information.

- Type equivalence checking. The representation is designed for a persistent environment in which sharing of instances is frequently achieved. Equivalence checking over a shared instance gives optimal efficiency. Some features of the representation are included to optimise the performance of full structural checking, since it is still required when programs are constructed in complete independence. In particular, the

number of objects created for an instance is small and the operations required to check the equivalence of the contents of objects are not complex. The type equivalence algorithm is described in more detail in Section 3.5.3.

3.5 Caching the Results of Type System Operations

Some of the type system operations are expensive no matter what representation is used. For example both the specialisation of parameterised type constructors and full structural type equivalence are intrinsically expensive since they perform operations on every component encountered during complete traversals of type representations.

It has already been shown how persistence may be used to cache the construction of type representations, achieved by making an existing representation accessible in the persistent environment. Persistence may also be used to cache the results of other intrinsically expensive operations.

3.5.1 The General Technique

Each operation to be optimised is linked to a cache containing the results of the operation's invocations. The cache is keyed on some unique characteristic of the operands to the operation. On each invocation of the operation the following takes place:

- The cache is examined to check if the operation has already been performed on these operands. If the cache contains an entry for the particular combinations of operands then it is returned as the result of the operation. Otherwise,
- The result is calculated, stored in the cache and then returned.

There are a number of factors that should be taken into account when considering the use of such a cache:

- The time taken to access the cache compared with the time taken to perform the operation.
- The frequency of successful searches over the cache. This depends on the repeated invocation of the operation on the same operands. If the frequency of success is low, the time taken to perform the cache access must also be small with respect to the time taken to perform the operation in order to make the use of the table worthwhile.
- The expense of maintaining the cache. Very large caches may affect the operation of other parts of the system.
- The class of results held in the cache. It may be possible to store only the results of the operation over a certain class of operands. This may reduce the expense of maintaining the cache or increase the access speed on the cache.
- The correctness of an operation's algorithm does not depend on the contents of the table - it is just an optimisation. The table may be emptied at any time if it becomes too large.
- Non-existence of a result in the table for a set of operands proves nothing about the result of the operation over those operands.

This remainder of this section describes the caching optimisation of the following two operations as examples of the technique:

- Specialisation of parameterised types.
- Structural type equivalence checking.

Both these operations take type representations as the operands. If the representation contains no ordered key over which the type may be indexed then the cache will in fact be a long list of (representation instance, result) tuples searched linearly on the identity of the representation instances. This would cause the lookup time to be proportional to the number of entries in the table which is unsatisfactory for a large number of results.

The *random* field in the *Type* representation of Section 3.4 contains a pseudo-random number calculated during the creation of the representation. It is included for use as a hashing key into hash tables implementing the caches described here. The pseudo-random nature of the hash key should ensure an even distribution of results in the table. With a good hashing function and a large enough table, access into the cache may be performed in a near constant time [CBC+90].

3.5.2 Optimising the Specialisation of Parameterised Types

The first example of a type checking operation whose results may be cached is the specialisation operation that takes place over the parameterised type constructors described in Section 3.2.1.

The type *parameterised* in Figure 3.13 is parameterised by two parameter types, *t1* and *t2*. The type itself is a structure containing two fields of each parameter type respectively. The type *specialised* is a specialisation of the parameterised type constructor using the specialising types *int* and *real* and after specialisation is structurally equivalent to the type *concrete*.

```
type parameterised[ t1,t2 ] is structure( a : t1 ; b : t2 )
type specialised is parameterised[ int,real ]
type concrete is structure( a : int ; b : real )
```

Figure 3.13 A parameterised type declaration and a specialisation.

Figure 3.14 depicts the representation instances for the type *parameterised* and the specialisation of that type to the types *int* and *real*. The parameterised instance is indicated by the label *6*. The number of parameters is specified in the *misc* field, in this case two. The parameterised instance refers to the instance of the type being parameterised via the *others* field. Where this instance contains components that represent parameter types, instances are used that are denoted by the label *11*. A parameter instance specifies which parameter it was in the original parameterised declaration as well as a reference to the parameterised declaration itself. The name of the parameter is not part of the type information. The specialised type has a similar structure to the parameterised type and is obtained by traversing every node of the parameterised instance making a copy of it in which the parameter nodes are replaced by the specialising type instances.

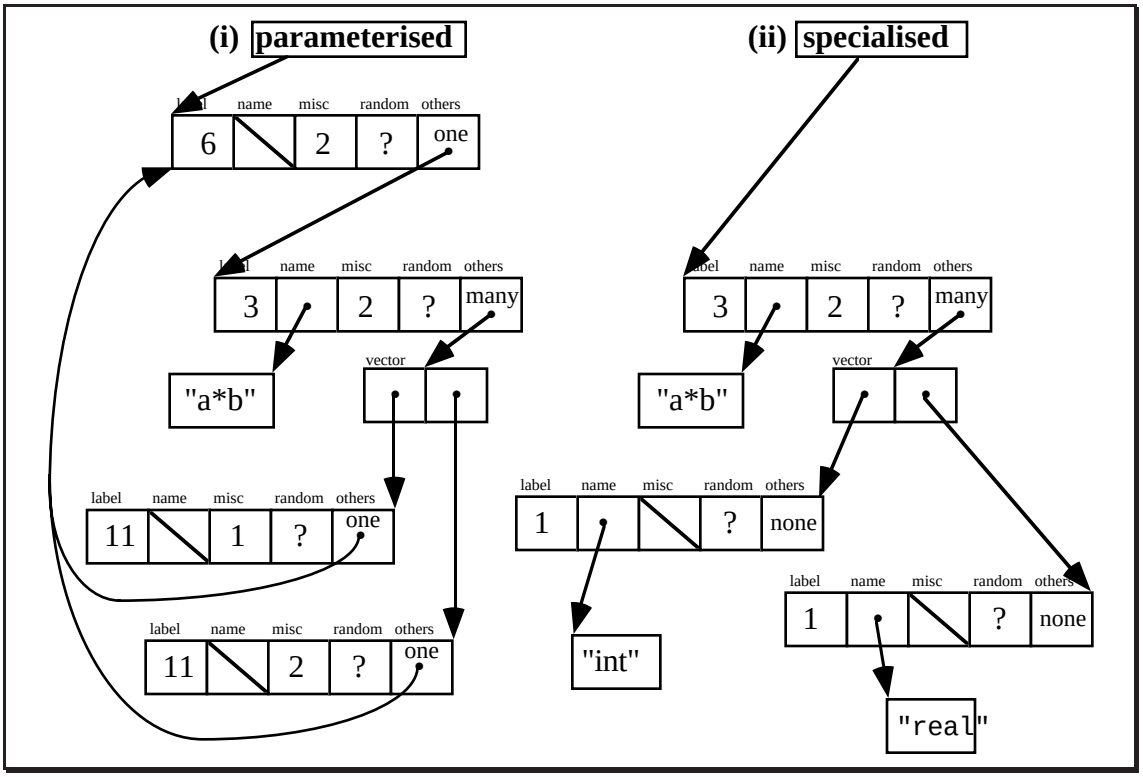

Figure 3.14 A representation of a parameterised type and a particular specialisation.

Specialisation is an intrinsically expensive operation which may occur repeatedly with the same types. A typical example is the use of a parameterised type declaration that is global to many programs, such as a tree or a list type. The type is repeatedly specialised in the local programs with the same specialising types and also in a single program if the denotation is clearer than a type identifier. The cost of repeatedly reconstructing the specialised version may be avoided by using a persistent cache that records the results of specialisation operations. The cache maps from parameterised type and specialising types to the appropriate specialised version. The cost of searching the cache to find out if a particular specialisation has already been performed will in general be small in comparison to the full specialisation operation. As well as avoiding the cost of the specialisation, the cache also promotes efficiency in the rest of the type system since it allows a single representation instanced to be shared by many programs.

3.5.3 Optimising Structural Type Equivalence Checking

Although the type representation described in Section 3.4 is optimised for structural equivalence checking such checking is still inherently expensive. The objective here is to avoid the expense incurred by re-checking pairs of types that have already been checked wherever this may have occurred in the lifetime processes.

The type equivalence function is used frequently during the compilation, linking and execution processes. Recording all the results of its execution would result in a cache that consumed excessive space and slowed the cache lookup. At the same time, the comparison of many representation pairs would not be significantly improved since the full equivalence check is fast for certain representation pairs. An example is those pairs that can be shown to be non-equivalent near the root of the representation graphs.

A strategy is required for recording just the results of the representation pairs whose comparison causes the most expense. This occurs when the types being checked are:

- Equivalent, since complete traversals of the type representations are required.
- Nearly equivalent, differing only at the leaves of the representation graphs, since a near complete traversal is required to show the difference.

An Equivalence Cache

Recursive data types require representations that are cyclic. Type equivalence checking must terminate when checking these cyclic structures. To recognise the cycles a caching mechanism is used.

An empty cache is created on each invocation of the equivalence algorithm. Pairs of type representations already encountered during a particular invocation are stored in the cache. The rechecking of cycles is then avoided by an examination of the cache.

If the result of the equivalence test is successful then the pairs of instances in the cache are also equivalent. Making this cache persist between invocations of the algorithm provides successful component type equivalence results with almost no extension of the equivalence algorithm. This section describes the implementation of the equivalence algorithm and the mechanism for recursion checking for the *Type* representation described in Section 3.4.

Structural equivalence checking involves simultaneous traversal of the representations of the types being checked. The structural check over non-cyclic instances is recursively defined in five stages, as follows:

1. Check whether the instances are identical. Return true if they are. Otherwise,
2. Check that the *label* fields of the two instances are the same. Return false if they are not. Otherwise,
3. Check that the specific information of each type, made up from the *name* and *misc* fields, is compatible. Return false if they are not. Otherwise,
4. If the *others* fields of the instances contain the *unique* branch of the variant, the test fails immediately. Types with the *unique* branch are defined to be equivalent only to themselves, in which case the identity check in stage 1 above would have succeeded. If the field was not *unique*,
5. Apply the algorithm recursively to any type instances contained in the *others* field to check that the component types are equivalent.

To illustrate the algorithm, consider the equivalent but not identical instances in Figure 3.16 of the type denoted in Figure 3.15 which is a record type with one field named *a* of type *real*.

```
structure( a : real )
```

Figure 3.15 A non-recursive type.

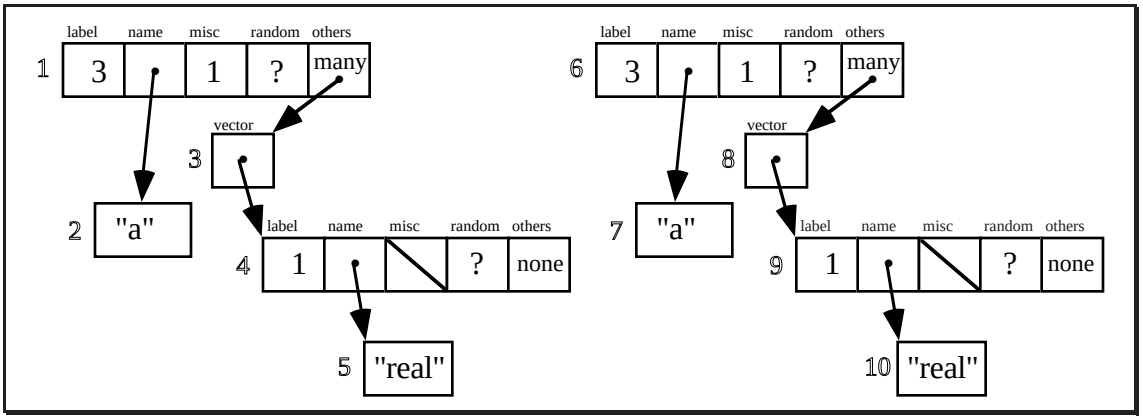

Figure 3.16 Equivalent but not identical representations.

The objects making up the instances in the diagram have been numbered. An equivalence check over these instances would initially compare object 1 with object 6. The two objects are not identical and so the check continues. The labels are the same, as is the specific information consisting of the number of fields and the field name. The algorithm is then recursively applied to the types accessible from the *others* field, in this case the types found in the vector objects 3 and 8. Each of these vectors contains just one type instance and so the algorithm is applied to these, objects 4 and 9. Again the objects are not identical. The labels and specific information are equivalent and this time there are no further type instances accessible from the *others* field and so the recursion of the algorithm is successfully grounded.

Equivalence checking in the presence of recursive data types is more complicated since their representation instances contain cycles. For example, consider the two cyclic instances in Figure 3.18 of the recursive type denoted in Figure 3.17 which is a record type containing one field named *x* of the same record type and two fields named *y* and *z* of type *int*.

```
rec type a is structure( x : a ; y,z : int )
```

Figure 3.17 A recursive type

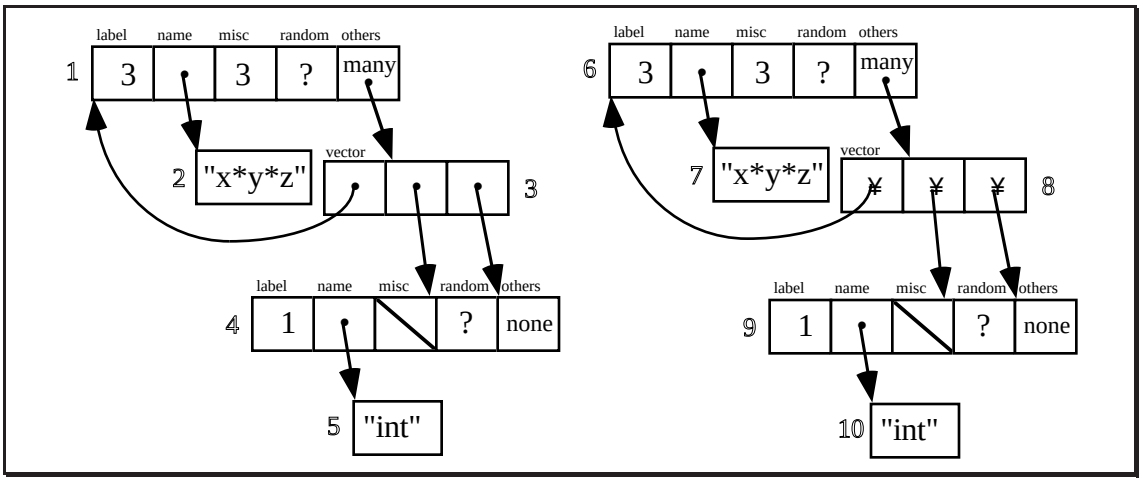

Figure 3.18 Equivalent cyclic representations.

In order to check for cycles during an equivalence check over two instances, a cache is maintained which records each pair of component types that have so far been encountered during the traversal of the instances. Every time the algorithm is recursively applied to a pair of components, the cache is scanned to find out if the identical pair is already in the cache. If the pair is not present then it is placed in the cache and the equivalence of the instances is determined according to their structure. If the pair is found in the cache then one of two situations has been detected:

1. The pair of instances has already been fully checked. If this is the case, the check must have been successful otherwise the algorithm would already have terminated The recursive application of the algorithm to these types may therefore be successfully terminated.
2. A cycle has been detected in which case the pair of types may safely be assumed to be equivalent. The assumption is based on the recursive application of the algorithm to other components of the type. If these applications succeed then the assumption is correct, otherwise the types will have been shown to be non-equivalent and the type check will fail. In either case the re-traversal of the cycle has been avoided.

The algorithm may be demonstrated using the two instances in Figure 3.18. On the first call the cache is empty and so the pair of types represented by the non-identical objects 1 and 6 is placed in the cache. Comparison of the label and the specific information of the two types is successful since they are both structures and the field names are the same. The algorithm is then recursively applied to the component types, in this case three applications to the field types of each structure. The first application is to objects 1 and 6. A cycle occurs here and at the start of the new application this cycle is detected since the object pair 1 and 6 is found in the cache. Condition 2 above applies and so no further checking of these types is attempted. The second application is to the non-identical objects 4 and 9. The pair is stored in the cache since it is not present and then structurally checked for equivalence which is successful. The third application is also to objects 4 and 9, and so the pair is found in the cache this time. Condition 1 above now applies and so rechecking of the instances is avoided. The algorithm now terminates with the result *true*.

Storing and accessing component pairs in the cache may become the most significant operation in the checking of large type instances. Using the *random* field of the *Type* representation as described in Section 3.5.1 ensures that the cost of checking for cycles need not be significant.

The cache prevents the re-traversal of pairs of components that are encountered more than once during the execution of the algorithm. This is the basic requirement of the cache for type equivalence results. Making the cache persistent makes this caching beneficial to shared components across all representation instances and across all invocations of the equivalence algorithm since it will then record all non-identical, equivalent type representations so far encountered. The cache does not record the results of all equivalence tests that have been performed, only those performed over non-identical but equivalent instances. Use of the cache is not quite as flexible as with the general caches described in Section 3.5.1 since a cache is required for the correct operation of the equivalence algorithm. In addition very frequent clearing of the cache may result in no finite progress being made.

To maintain the correctness of the check for cycles, instances not proven to be equivalent must not remain in the cache across invocations. If no special action is taken these may be present after a failed invocation of the algorithm. One method of solving this problem is to make a record of the new pairs entered into the cache on each new invocation of the equivalence algorithm. Should it turn out that the two types being checked are not equivalent these new pairs must be removed from the cache.

A particular version of a persistent cache for successful type equivalence tests may therefore be achieved using the mechanism to check for recursive types. The cache stores all non-identical equivalent pairs of representation instances that have been seen by the equivalence algorithm.

Including Non-equivalent Pairs in the Cache

The structural checking of pairs of type representations that are nearly equivalent is almost as expensive as the checking of equivalent types. The difference between the representations is only detected at the leaves of the representation graphs which means that a near-complete traversal of the representations is required.

In order to include pairs of representation instances that are nearly equivalent in the cache, a 'measure of equivalence' is required that can be cheaply calculated during the execution of the equivalence algorithm. On equivalence failure, only those pairs that are nearly equivalent are placed into the cache. Each entry in the cache must also record whether the pair are equivalent or not. The equivalence algorithm must be adjusted slightly since accessing the cache can now produce three results - the pair being checked are equivalent, the pair are not equivalent or they are not present in the cache.

3.6 Conclusions

As the scope of the type system's application has grown from just the compilation environment to all lifetime processes, type system implementations have gained ad-hoc extensions to cope with the expansion. Both programs and software development processes are traditionally independent of one another and the only way to distribute the type information now universally required has been to make copies throughout the system. In persistent programming systems where the type descriptions are typically large, this copying is a complex operation and there are few implementations that permit system-wide distribution of type information.

It has been shown here how the ability of a persistent environment to cache the construction of complex data structures may be used to transform the implementation of the type system operations associated with a persistent programming system. Instead of making copies of the type information, a single instance of a structured type description may be shared by all programs and all software processes through which the programs pass.

The implementation of a type system now requires only a single set of operations to construct, manipulate and test type representations. The type storage, copying and conversion routines associated with traditional type system implementations as well as the ad-hoc language mechanisms to promote limited type sharing may be dropped since the function that they attempted to fulfil is now carried out by the persistence mechanism.

The use of complex type checking operations has traditionally been restricted to the compilation environment where speed may not be critical. As the influence of the type system extends to all parts of the system, the use of these complex operations is also spreading. Since almost any type system operation may now be used in any part of the software development process, it is essential that a type representation is used that permits efficient implementation of all type operations. The description of a particular type representation has been given here.

Some type system operations are inherently expensive to implement no matter what type representation format is used. It has been shown here how the persistent store

may be used to cache the results of these operations in order to improve the performance of the type system implementation.

The use of persistence to support the implementation of a type system as described here greatly simplifies the use of the type system operations in any of the lifetime processes.

4 Using Persistence to Enhance Compilation

4.1 Introduction

Embedding a compiler within a persistent environment creates a symbiotic relationship between the two. The benefits of the relationship may be described in terms of the manner in which the compiler and the environment interact with one another, as follows.

Firstly, the compiler may be accessed by executing programs. An executing program may construct and execute new programs which manipulate the persistent environment. This ability of a program to alter its own environment during execution is a particular form of reflection [Mae87] known as run-time linguistic reflection [SSS+92]. It is of particular interest in persistent systems because it can allow long-lived data and programs to evolve in a type-safe manner.

Secondly, the functionality of the compiler may be enhanced by parameterising it with cached persistent values that are associated with the source. The number of stages at which programs are bound to data may be extended when the compiler can manipulate that data. These stages depend upon the times at which identifiers embedded in a program are resolved to their associated values, as follows:

- During program composition. The source contains embedded complex structured values. During compilation these values are incorporated into the compiled code. This technique is known as hyper-programming [KCC+92].
- During compilation. Free identifiers in the source are resolved using values passed into the compiler [FDK+92].
- Between compilation and execution. The resolution of the identifiers is performed in a separate phase involving an intermediate program representation and the associated values.
- During execution. Identifiers are resolved when the executing program accesses values in the persistent environment.

Thirdly, the compiler may manipulate values in the environment. There are two benefits here. The compiler may use the environment to optimise its own performance, for example by caching the results of operations repeated across invocations of the compiler. An example of this was seen in the implementation of type checking seen in Chapter 3. Also, the compiler may associate persistent values with compiled code in order to optimise the execution of that code. An example of this for optimising the implementation of polymorphism is described in Chapter 5.

The symbiotic relationship occurs in that the compiler uses the persistent environment to enhance its functionality and to optimise code for use within the environment. The persistent environment benefits in that the execution of programs, including the compiler, is made more efficient.

This chapter describes the facilities that are required in order to deliver the benefits described above. They are:

- A flexible compiler interface. The interface supports reflective programming techniques and flexible binding strategies.

- The construction of the Napier88 compiler within the Napier88 persistent environment. The construction architecture of Chapter 2 is used to achieve this.

As described in Chapter 1, construction of a compiler within a persistent environment is the major task in converting a persistent programming language implementation into an integrated persistent programming environment. Construction of the Napier88 compiler within the Napier88 persistent environment was the keystone on which all of the research in this thesis is based.

The flexible compiler interface was developed once the initial implementation of the compiler was complete. The impetus for the interface was a number of new component binding styles implemented independently during the development of the persistent environment. These included hyper-programming [KCC+92], compile-time binding[FCK+92] and binding between compilation and execution. The location binding used in the construction architecture of Chapter 2 was also incorporated. The new compiler interface cleanly supports all of these binding styles.

4.2 A Flexible Compiler Interface

Placing the compiler within the persistent environment establishes a requirement for greater flexibility in the use of the compiler and therefore demands more facilities from it. These facilities are described as a series of ever richer interfaces to the compiler which can be used in the appropriate context.

4.2.1 Linguistic Reflection

Linguistic reflection is defined as the ability of a program to generate new program fragments and to integrate these into its own execution. The importance of linguistic reflection is that in conjunction with strong typing it may be used to provide a type safe mechanism for the production and evolution of programs and data in a persistent environment. In current systems it has been used to attain high levels of genericity [SFS+90], accommodate changes in systems [DB88,DCK89], implement data models [Coo90a,Coo90b], optimise implementations [CAD+87,FS91] and validate specifications [FSS92,SSF92].

The focus of interest here is the manner in which linguistic reflection may be used to support the programming process entirely within the persistent environment. To achieve this, a particular style of linguistic reflection known as run-time linguistic reflection is used. This style is concerned with the construction and manipulation of new program components during the execution of an existing program.

Run-time linguistic reflection involves the use of a compiler that can be called dynamically to compile newly generated program fragments. Programs may access a compiler during execution since it is a procedure value within the persistent environment. The type of this procedure is of interest. Traditionally, a compiler takes a source code representation and produces executable code. The source representation may be a string and the executable code may be represented by a void procedure. In such cases the type of the compiler is as in Figure 4.1.

```
type compiler is proc( string → proc() )
```

Figure 4.1 The type of the compiler procedure.

An example of the use of this procedure to support run-time linguistic reflection is given in Figure 4.2 in order to demonstrate that the whole programming process, from program construction through compilation to execution, may be carried out within a persistent environment. The example is a fragment from a program that supports a very simple programming environment in which the user is prompted to type in the source text of a program. The source text is then passed to the compiler procedure which returns executable code for the source program encapsulated inside a void procedure. The reflection occurs at this point. The resulting executable procedure may then be stored or executed.

```
!** The new program is constructed.
writeString( "Please type in a program: " )
let source = readString()

!** The source text is compiled, using the procedure compile,
!** to give a void procedure.
let executable = compile( source )

!** Do something with the executable version of the program.
writeString( "Store or execute the program? " )
if readString() = "store"
then !** storeProc stores the source and associated executable code.
     storeProc( source,executable )
else !** The new program may be executed by calling the procedure.
     executable()
```

Figure 4.2 Using a compiler procedure within a program.

The user may type the program of Figure 4.3 when the program of Figure 4.2 is executed. The user's program is compiled into a procedure that will write out *Hello World* when called. The user may execute the procedure immediately or store it for later use.

```
writeString( "Hello World" )
```

Figure 4.2 A simple program.

By extrapolating the program fragment of Figure 4.2 it can be seen how a complete programming environment may be supported if the reflective ability to turn a program representation into executable code is provided within the persistent environment.

The compiler interface given in Figure 4.1, used to introduce the concept of reflection, does not specify the full behaviour of a conventional compiler since there is no indication of the action taken when the compiler detects an error in the source program. The interface may be extended to handle compilation errors as shown in Figure 4.4. The compiler returns a variant type *compilerResult* which may indicate a compilation failure along with an appropriate message or else a successful compilation along with the void procedure result.

```
type compilerResult is variant( fail : string ;
                                ok   : proc() )
type compiler is proc( string → compilerResult )
```

Figure 4.4 Handling compilation errors.

The subject of compilation errors raises an important point about the nature of the compiler in a persistent environment. The compiler is the only program with the power to convert program representations into executable values within the environment. A particular component within the compiler, known as the *magic* module, performs this task. The compiler is trusted to ensure that only valid representations of programs are passed to the *magic* module. This is particularly important in a persistent system where the type system may be the only protection mechanism over data. It is therefore essential both that the compiler is implemented correctly and that it is protected from corruption.

4.2.2 Supporting Flexible Binding Strategies

In persistent systems, an executing program and the persistent data over which it operates combine to form a complex graph structure within the persistent object store. Where program construction and compilation is independent of the persistent environment however, source and executable program representations are restricted to being unstructured values. Therefore the binding of program and persistent data may only be achieved as follows:

- An executable program may contain code which causes the data to be accessed during execution.
- If the data can be translated into a flat representation then a flat copy of the data may be taken from the persistent environment and included in the program. Referential integrity over the data is lost in this case.

When the program construction environment is contained within the persistent environment, both source and executable program representations may be represented by structured persistent values. This promotes a range of techniques by which programs may be bound to data. The trade-offs associated with these styles of binding may be found in [FDK+92].

The compiler interface described in Section 4.2.1 reflects the functionality of a compiler external to the persistent environment. A number of extensions are required to the interface in order to support new binding techniques. Each technique is described in conjunction with the necessary extension to the interface. The different interfaces described here are simplified for the sake of clearer discussion.

Composition-time Binding

Where the program composition process is supported inside the persistent environment, programs and data may be bound during composition. The programmer composes programs interactively by navigating the persistent environment and selecting data items to be bound into the programs. This requires direct links to the persistent data items to be represented in the program source. This style of programming is known as hyper-programming [KCC+92].

The source representation passed to the compiler may be extended to include direct links to persistent values or locations as shown in Figure 4.5. An instance of *sourceRep* is a list of source components, each element of which may be either a language lexeme or an embedded item contained in the persistent environment. Whether the embedded item is a value or a location is determined using the *value* field of *valueOrLocation*. The embedded item may be of any valid type and so is injected into the infinite union type *any*.

```
rec type compiler is proc( sourceRep → compilerResult )
&        sourceRep is list[ sourceComponent ]
&        sourceComponent is variant( lexeme   : string ;
                                     embedded : valueOrLocation )
&        valueOrLocation is structure( value : bool ; item : any )
```

Figure 4.5 Compiler interface with extended source representation.

When the compiler encounters an embedded item, its type is determined from the *any* in which the item is contained. The value or location itself is directly embedded into the executable code produced by the compiler.

As described in Chapter 3, type representations found in the persistent environment may also be included in the source representation to promote optimisations in type checking. *sourceRep* may be used unchanged to hold type representations since they are persistent values. The compiler can always determine by context whether an *embeddedValue* is to be used as a value or as a type. A problem specific to Napier88 is the overloading of types as constructor functions. Further complexity is required to handle this which is not of interest here.

The integrity of the whole persistent environment depends on the manipulation of type representations by the compiler. The compiler as described previously is a trusted program and must construct and manipulate type representations correctly to ensure that only legal operations are applied to data. Where the compiler is the only trusted program in the system it is potentially dangerous to allow type representations to escape the control of the compiler and then be returned to it in a different context. A mechanism is required to ensure both that type representations supplied to the compiler did originate there and that type representations cannot be corrupted. Such a mechanism is described in Section 4.2.3.

Compile-time Binding

Where the compiler can access and manipulate persistent values, binding between program and data may also be performed during compilation.

The binding mechanism may be based on the resolution of free identifiers within the program by the compiler. The use of an identifier that is not declared within a program is detected by the compiler. Traditionally an error is reported in such cases. To support compile-time binding the compiler may be parameterised by a table of persistent values or locations keyed by identifiers. When a free identifier is detected the compiler attempts to resolve the identifier against the table of values. Successful resolution causes the associated value to be included in the executable program. Unsuccessful resolution causes a compilation error.

The interface to the compiler is extended as shown in Figure 4.6. *table[string,any]* is the type of a table package that holds persistent values injected into type *any* keyed by values of type *string*.

```
type compiler is proc( sourceRep,table[string,valueOrLocation] →
                                                    compilerResult )
```

Figure 4.6 Compiler interface suitable for compile-time binding.

It should be noticed that the compiler interface has been extended to allow both composition and compile-time binding to co-exist.

Separating Checking and Binding

By permitting the binding of programs to values during compilation, the compiler is now performing the following two activities:

- Compilation, consisting of lexical analysis, syntax analysis, type checking and code generation.
- Binding.

Greater flexibility and efficiency may be achieved if the binding is performed in a separate phase. Binding may take place after compilation and before execution. Efficiency is gained since the program passes through the compilation phase only once. Flexibility is increased since a compiled program forms a generic specification which may be specialised in many different ways provided that the generic version is allowed to pass through the binding phase many times. On each occasion a different executable program may be produced according to the values supplied to the binding phase. Flexibility may be further increased where the binding can be performed in incremental stages, each stage producing a more completely resolved program. Execution of the program can take place when all the bindings are completely resolved.

The compiler interface may be split into two sections as in Figure 4.7 to support separate compilation and binding phases. Note that the compiler may produce executable code if the source code contains no free identifiers.

```
rec type    compiler is proc( sourceRep,table[ string,valueOrLocation ] →
                                                              result )
&           result is variant(  fail          : string ;
                                stillUnbound  : unboundDetails ;
                                executable    : proc() )
&           unboundDetails is structure(  code  : intermediateRep ;
                                          free  : list[ id ] )
&           id is structure( name : string ; embedded : valueOrLocation )
&           intermediateRep is .....
&           binder is proc( intermediateRep,table[ string,valueOrLocation ] →
                                                              result )
```

Figure 4.7 Separating compilation and binding.

Although the values associated with free identifiers are not resolved until the binding phase, the types associated with the identifiers must be known during the compilation phase. The *table* passed to the compilation phase is used for this purpose. The values in the *any*s are ignored by the compiler; it is the specific types that are used by the compiler.

The compilation phase may detect an error in which case an appropriate message is returned. Otherwise either no free identifiers are detected in which case the compiler returns an executable procedure or else an intermediate program representation is returned along with details of the free identifiers contained in the program. The format of the intermediate representation is not of concern here.

The binder phase takes an intermediate representation as parameter along with a table of values. If the representation contains a free identifier for which the table contains an entry and the entry is of the type specified during checking, then the identifier is resolved. If all free identifiers are resolved then the binder produces an executable procedure. Otherwise a new intermediate representation is produced along with details of the remaining free identifiers. If the table contains entries for which the program has no corresponding free identifiers or if the entries are of the wrong type then the entries are ignored.

The compiler and binder procedures may be used independently from one another. Interfaces may be defined over them however to provide various styles of compilation. For example a procedure that supports the integrated compiling and binding interface of Figure 4.6 is shown in Figure 4.8.

The source is passed to the procedure along with the values table. The result of the compiler is projected onto the possible branches of type *result*. A compilation error may be detected, or else the compilation is successful. In the latter case the values table may be empty in which case the compiler returns an executable procedure. Otherwise the intermediate program result is passed to the binder procedure along with the values table. Since the same values table is used in both compilation and binding, this operation is always successful and the resulting executable procedure is returned.

```
let integratedCompiler = proc(  source  : sourceRep ;
                                values  : table[ string,valueOrLocation ] →
                                                                compilerResult )
begin
    let compRes = compiler( source,values )

    project compRes as X onto
        fail          : compilerResult( fail : X )
        executable    : compilerResult( ok : X )
        stillUnbound  : begin
                            let binderRes = binder( X( code ),values )

                            project binderRes as Y onto
                                executable  : compilerResult( ok : Y )
                            default         : {...}
                        end
    default           : {}
end
```

Figure 4.8 A compiler interface over the compiler and binder procedures.

An alternative interface may be constructed that allows free identifiers in a program to be resolved against the values found in a list of environments, as illustrated in Figure 4.8. An extra procedure is required that converts an environment into a suitable table of values. If the compilation is successful then the intermediate program result is repeatedly bound against the values found in each environment in the list. If the environments do not contain all the required values, then the compilation fails and a message is returned to that effect.

```
let envToTable : proc( e : env → table[ string,valueOrLocation ] ) ; { ..... }

let compileWithEnvs = proc(  source  : sourceRep ;
                             envs    : list[ env ] ;
                             freeIds : table[ string,valueOrLocation ] → compilerResult )
begin
    let compRes = compiler( source,values )

    project compRes as X onto
      fail          : compilerResult( fail : X )
      executable    : compilerResult( ok : X )
      stillUnbound  : begin
                          let intRep := X
                          let finished := false
                          let result := compilerResult( fail :"Couldn't bind all free identifiers" )
                          while envs isnt tip and ~finished do
                          begin
                              let values = envToTable( envs'cons( hd ) )
                              let binderRes = binder( X( code ),values )

                              project binderRes as Y onto
                                  executable    : begin
                                                      result := compilerResult( ok : Y )
                                                      finished := true
                                                  end
                                  stillUnbound  : begin
                                                      intRep := Y
                                                      envs := envs'cons( tl )
                                                  end
                                  default       : begin
                                                      finished := true
                                                      result := compilerResult( ok :
                                                                    "error in binding" )
                                                  end
                          end
                          result
                      end
    default       : { .... }  !** cannot occur
end
```

Figure 4.8 Compiling against a list of environments.

An intermediate representation is another example of a value constructed by the compiler that may escape its control, to be returned to the compiler at another time. Corruption of the representation may result in the production of an unsafe program. Consequently it is necessary to protect the representation while it is outside the compiler.

4.2.3 The Compiler Interface as an Abstract Data Type

As discussed above, type representations and intermediate program representations must be protected to avoid potential corruption of the persistent environment. Protection is only required when they are outside the control of the compiler. The required protection may be provided using the abstract data types of Napier88 [CDM+90]. The compiler interface may be supported using the abstract data type shown in Figure 4.10.

```
rec type compilerPackage is abstype[ intermediateRep ]
(
  compiler  : proc( sourceRep,table[ string,valueOrLocation ] →
                                                  result[ intermediateRep ] ) ;
  binder    : proc( intermediateRep ,table[ string,valueOrLocation] →
                                                  result[ intermediateRep ] )
)
&       result[ IR ] is variant( fail          : string ;
                                 stillUnbound  : structure( code : IR ;
                                                            free : list[ id ] ) ;
                                 executable    : proc() )
&       id is structure( string,valueOrLocation )
```

Figure 4.10 The compiler abstract data type.

The abstract type specifies a type known as the witness type, in this case *intermediateRep*, which is the type that is abstracted over. The interface contains two procedures, *compiler* and *binder*, which return values of the *result* type which is parameterised by the witness type. Outside the abstract data type the internal structure of values of the witness type may not be accessed. However these values may be stored and passed around. When they are re-presented to the binder then their internal structure may be accessed again within that procedure with the confidence that it has not been altered in an unsafe manner. A value of the abstract type is constructed by specifying a concrete type for the witness type and supplying two procedures operating over that concrete type. A suitable concrete type for the intermediate representation used by the compiler might be a record containing a list of instruction codes and a list of unresolved identifiers and their positions of use in the instruction sequence. It is this data structure that is protected while outside the compiler.

The above description of the use of values of witness types depends upon a particular style of witness type checking. This style is more flexible than that of Napier88 release 1.0 [MBC+89] but is still statically checkable. It is discussed along with its implementation in Section 4.4.

4.3 Constructing the Compiler within the Persistent Environment

The benefits described so far depend on the availability of a compiler within the persistent environment. A compiler must therefore be constructed inside the environment. The binding style used in the construction architecture of Chapter 2 may be used to support the implementation since it gives appropriate levels of safety and flexibility. The construction of the Napier88 compiler will be used as an example of how a complex application may be mapped onto that software architecture.

4.3.1 The Napier88 Compiler

The Napier88 compiler uses the recursive descent compilation technique [Amm73,DM81]. Recursive descent is a single-pass technique which is centred around the syntax analysis phase of the compilation. This phase is split up between a number of recognition procedures, each one performing the syntax analysis of a particular production in the grammar defining the language.

Embedded in each recognition procedure are calls to others to recognise the appearance of sub-productions. For example the procedure recognising the declaration of an object uses the procedure that recognises clauses in order to parse the object itself. The technique is mutually recursive since within one recogniser the use of the same recogniser is frequently required at a lower level. This follows the mutually recursive definition of the language. For example, the definition of a procedure may contain the definition of encapsulated procedures.

No explicit parse trees are constructed using recursive descent. All information about the compilation is retained in the stack frames of the recursive procedure calls.

The recognition procedures perform the syntax analysis phase of the compilation. A number of other procedures are required to perform lexical analysis, type checking, code generation, and error handling. Calls to these procedures are embedded at appropriate positions within the recogniser procedures.

The compiler produces a low level intermediate code representation. A second pass is required to convert this representation into target machine code. This pass is performed incrementally at the end of the compilation of each procedure. There are two existing implementations of the second pass for PAM [BCC+88] code and for SPARC [Sun87] code.

4.3.2 Building the Compiler on the Construction Architecture

When no compiler is available in the persistent environment, one must be constructed using an external program construction environment. The binding styles described in Section 4.2 are therefore not available. However the flexible binding style of the architecture described in Chapter 2 may be supported using the generator programs described there. Having bootstrapped a compiler into the environment, it may be reimplemented using the binding techniques of Section 4.2. Both the bootstrap and one possible reimplementation technique are described here.

General Requirements of the Architecture

Using the construction architecture of Chapter 2, an application is divided into a number of components. Each component is contained in a typed location and bound to the locations of other components to which it refers. This style of binding gives both safety and flexibility at the cost of a small degradation in the efficiency of inter-component references.

The locations are also bound together by a mechanism that allows access to the locations so that they and their contents may be manipulated. In Napier88 this access mechanism is supported using the environment data structure. The environments holding the locations of the compiler are tree-structured as shown in Figure 4.11 in order to impose some logical structure over the 600 or so components that make up the compiler. The tree structure is reachable from the persistent root. Each name in the diagram is an environment containing a number of components grouped according to the compilation operation for which they are used. There are also a number of constant values that are treated as separate components.

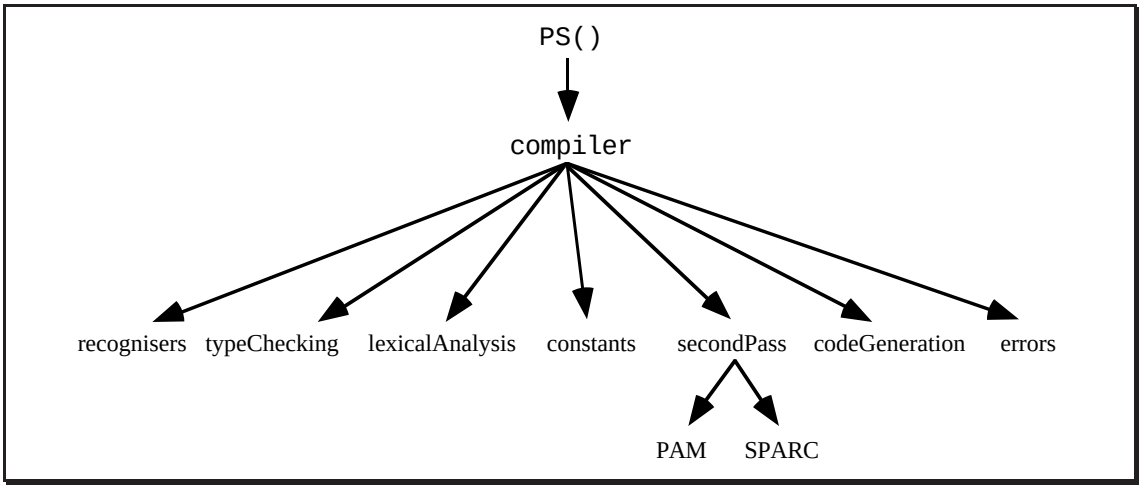

Figure 4.11 The environment tree structure containing component locations.

Figure 2.6 is reproduced here in Figure 4.12 as a reminder of the two styles of binding that are used in the construction architecture. The environment structure of Figure 4.11 is the location access mechanism represented in the vertical plane of Figure 4.12. The bindings in the horizontal plane are constructed when the components are created and as described above are bindings from component to location.

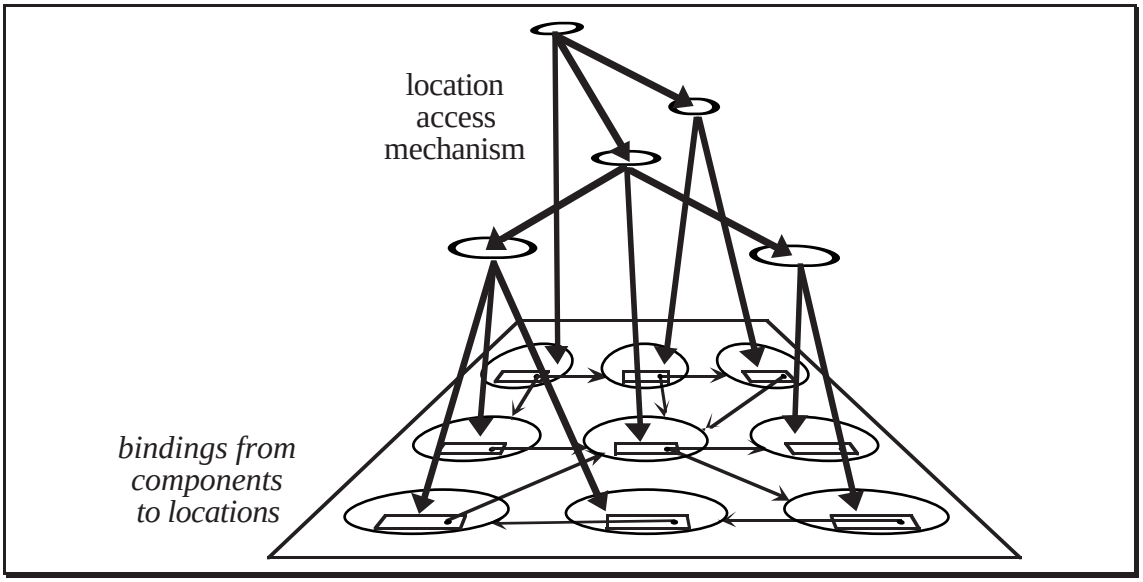

Figure 4.12 The two styles of binding in the construction architecture.

Bootstrapping the Compiler into the Persistent Environment

When the Napier88 program construction environment is external to the persistent environment, binding between program and persistent data is achieved by including code in programs to access the data during execution. Using the construction architecture of Chapter 2, programs bind to persistent data during the generation of new components in order to construct the required component to location bindings. There are two kinds of program that should be considered.

Firstly, initialiser programs are required to construct the environment structure and the associated locations shown in Figure 4.11. The program of Figure 4.13 constructs the *compiler* environment. The procedure *environment* available in the root environment of the persistent store may be used to create new environments.

```
use PS() with environment : proc( → env ) in
      in PS() let compiler := environment()
```

Figure 4.13 Creating the *compiler* environment.

There is an initialiser program for each sub-environment to create that environment and its associated typed locations. The locations initially contain default values that are not of interest here. Figure 4.14 shows the program that is used to set up the *recognisers* environment and the locations contained therein. Initially the program binds to the *compiler* environment and to the procedure to create new environments. A new environment *newEnv* is created and then a number of locations of suitable types are created in the new environment. The exact types are not of interest here. *sequence* and *objectDecl* are two of the many productions in the language grammar. Finally the new environment is placed into the *compiler* environment.

```
use PS() with  environment  : proc( → env ) ;
               compiler     : env in
begin
      let newEnv = environment()
      in newEnv let sequence := proc( .... )
      in newEnv let objectDecl := proc( .... )
      ....
      in compiler let recognisers := newEnv
end
```

Figure 4.14 Creating an environment and associated locations.

Secondly, generator programs are required to fill the locations with the appropriate components. Figure 4.15 shows a program to generate a new version of the *sequence* recogniser procedure. The *sequence* procedure may be bound to any of the components in the environment structure. Among others, *sequence* uses the *objectDecl* procedure, also found in the *recognisers* environment. Once the generating program has been bound to the required locations in the persistent store, a new version of the *sequence* procedure is constructed and assigned to the *sequence* location.

When generator programs for all components have been executed the compiler is ready for use. The construction of a compiler interface package of the style shown in Figure 4.10 is outlined in Figure 4.16.

```
use PS() with compiler : env in
use compiler with recognisers, types, cgen, ..... : env in
use recognisers with  sequence   : proc( .... ) ;
                      objectDecl : proc( .... ) ;
                                   ....  in
use types with ....
....
begin
      sequence :=  proc( .... )
                   begin
                         ....
                         !** objectDecl may be used within sequence
                         .... objectDecl ....
                         ....
                   end
end
```

Figure 4.15 Generating a new component.

```
type intermedRep is structure( .... )       !** Details unimportant here.
type compilerPackage is abstype[ ... ]( .... )      !** As in Figure 4.10.

use PS() with compiler : env in
use compiler with recognisers : env in
use recognisers with sequence : proc( .... ) in
....
begin
      let compiler = proc(  s          : sourceRep ;
                            freeIds    : table[ string,valueOrLocation ] →
                                                  result[ intermedRep ] )
      begin
            ....
            !** The compiler part of the package is called using sequence
            .... sequence( .... )
            ....
      end

      let binder = proc(  i : intermedRep ;
                          freeIds : table[string,valueOrLocation] →
                                                  result[ intermedRep ] )
      begin
            ....
      end

      !** A new compiler interface is constructed using intermedRep as the
      !** specialising concrete type.
      let compilerInterface = compilerPackage[ intermedRep ]( compiler,binder )

      !** Use or store the interface.....
end
```

Figure 4.16 Generating a compiler package.

First a concrete type for the intermediate program representation is declared. Having retrieved the necessary components from the environment, versions of the compiler and binder procedures are constructed that operate over the concrete intermediate representation. At some point, the compiler procedure calls *sequence* since it is *sequence* that compiles the top-most production in the language grammar. Other details of *binder* and *checker* are not important here. Having generated the two procedures, a new instance of the compiler interface abstract type may be constructed. This is achieved by specifying the concrete type and appropriate interface components.

Using the Flexible Compiler Interface to Generate a Compiler

Once a compiler has been constructed in the store, the flexible compiler interface described in Section 4.2 may be used to reimplement the compiler. As an example, composition-time binding will be used to show how the programs of Figures 4.13, 4.14 and 4.15 are adjusted. The bindings in a program using composition-time binding may be denoted using boxes surrounding identifiers for the embedded values. Full details of an interface which allows these composition-time bindings to be constructed and examined is given in [KCC+92].

The program of Figure 4.13 which creates the *compiler* environment is altered to that shown in Figure 4.17. Note that the *environment* procedure was found during program composition and bound into the program representation.

```
in PS() let compiler = environment()
```

Figure 4.17 Creating the *compiler* environment using composition-time binding.

The program of Figure 4.14 to create the *recognisers* environment is altered to that of Figure 4.18. Both the *environment* procedure and the *compiler* environment have been bound during program composition.

```
let newEnv = environment()
in newEnv let sequence := ...
in newEnv let objectDecl := ...
...
in compiler let recognisers := newEnv
```

Figure 4.18 Creating the *recognisers* environment using composition-time binding.

The most radical reduction in complexity is the version of the program to generate a new *sequence* component using composition-time binding. This is shown in Figure 4.19.

```
sequence := proc( ..... )
            begin
                    ....
                    !** objectDecl may be used within sequence.
                    .... objectDecl ....
                    ....
            end
```

Figure 4.19 Creating the *sequence* component using composition-time binding.

Of the binding styles introduced in Section 4.2, only composition time binding has been shown here. For increased flexibility, the compiler may be implemented using generic specifications of each component in which the exact components to which they refer are left unspecified. Compilers with different behaviour may be produced according to the particular values used to specialise the components.

4.4 Dynamically Checked Witness Types

As described at the end of Section 4.2, Napier88 abstract data types may be used to protect both type and intermediate program representations when they are outside the control of the compiler. This is a requirement in the system since it is only the compiler that is trusted to manufacture and manipulate these representations. The representations are viewed as values of the witness type of the compiler abstract data type while external to the compiler. As such only the basic operations available over all types may be performed on the representations such as equality and assignment.

This section describes the construction and use of Napier88 abstract data types in detail. Some limitations over witness type checking encountered in Napier88 release 1.0 are exposed that disallow the manipulation of values of witness type as required by the compiler interface. A new style of witness type checking that is suitable for use with the compiler interface is then described along with its implementation.

4.4.1 Napier88 Abstract Data Types

Napier88 abstract data types are the existentially quantified types of Mitchell and Plotkin [MP88]. Consider the abstract data type *count* shown in Figure 4.20. *count* is declared to be an abstract type with a single witness type *t* which is the type that is abstracted over. The abstract type interface consists of an identifier *val* with the type *t* and an identifier *inc* which is a procedure that takes a parameter of type *t* and returns a result of that type.

```
type count is abstype[ t ]( val : t ;
                            inc : proc( t → t ) )
```

Figure 4.20 The abstract data type *count*.

To create an instance of the abstract type the fields of the interface are initialised. This requires a value of type *t* and another of type *proc(t → t)* for some type *t*. For example, if type *t* is specialised to type *int*, the procedure value *incInt* created in Figure 4.21 may be used in the abstract type instance *counterOne* also created in Figure 4.21.

The value *counterOne* is of abstract type *count* with concrete witness type *int* in which the *val* field is initialised to *0* and the *inc* field to *incInt*.

```
let incInt = proc( x : int → int ) ; x + 1

let counterOne = count[ int ]( 0,incInt )
```

Figure 4.21 Creating an instance of *count*.

Once the abstract instance has been created the user of the interface can never again tell that the concrete witness type is *int*. The concrete witness type of the abstract instance *counterTwo*, shown in Figure 4.22, is *real*. *counterOne* and *counterTwo* are however type compatible since the type of the concrete witness is abstracted after creation.

```
let incReal = proc( x : real → real ) ; x + 1.0

let counterTwo = count[ real ]( 0.0,incReal )
```

Figure 4.22 Creating an instance of *count* using *real* as the concrete witness type.

Accessing the Fields of Abstract Instances

The focus of interest in this section is the type checking that is required to ensure the safe use of abstract types. Consider the procedure call and assignment of Figure 4.23. Since *counterOne* and *counterTwo* are type compatible this appears to be a type-safe operation. However the implementations used to construct the two abstract types are different and so the application of *counterOne(inc)* to *counterTwo(val)* is unsafe. The *inc* procedure expects a parameter of concrete type *int* but instead is supplied with one of type *real*. The compiler cannot find out the concrete types and so cannot detect the erroneous operation.

```
counterOne( val ) := counterOne( inc )( counterTwo( val ) )
```

Figure 4.23 Unsafe interaction between abstract instances.

A mechanism is required therefore to ensure that only operations and values from the same instance of an abstract type are mixed. It is desirable for this mechanism to operate statically. The abstract *use* clause of Napier88 [CDM+90] performs this function. Use of an abstract instance is restricted to a specified code section by the *use* clause. Consider the *use* clause of Figure 4.24. The use of *intCounter* is restricted to the block after the *in* which in this case is a single assignment operation. The compiler cannot detect statically which abstract instance is being used since the abstract value after the *use* can be expressed as any legal expression and therefore may be anonymous. However, the compiler can ensure that the same instance, whichever it is, is used throughout the *use* block by aliasing it to a constant identifier, in this case *aliasedCounter*. Inside the block the only access to the abstract instance being used is via this identifier.

As well as aliasing the abstract data type to a constant identifier, the witness types of the abstract type are aliased to types that are unique to the *use* clause. The user may specify identifiers for these types. In Figure 4.24 the witness type of *counterOne* has

been aliased to *uniqueT*. Name equivalence is used over these types, i.e. *uniqueT* is only equivalent to itself.

```
use counterOne as aliasedCounter[ uniqueT ] in
        aliasedCounter( val ) := aliasedCounter( inc )( aliasedCounter( val ) )
```

Figure 4.24 Abstract type *use* clause.

The unsafe interaction of the program in Figure 4.v can now be detected statically. The same program must be rewritten with *use* clauses as in Figure 4.25. Using the aliased types, the dereference *aliasedCounterOne(inc)* produces a procedure of type *proc(uniqueT1 → uniqueT1)*. The result of the dereference *aliasedCounterTwo(val)* to which this procedure is applied is of type *uniqueT2*. The compiler may therefore detect the erroneous operation.

```
use counterOne as aliasedCounterOne[ uniqueT1 ] in
        use counterTwo as aliasedCounterTwo[ uniqueT2 ] in
        aliasedCounterOne( val ) := aliasedCounterOne( inc )(
                                                aliasedCounterTwo( val ) )
```

Figure 4.25 Detecting an unsafe interaction.

Placing Values of Witness Type into Infinite Unions

The protection offered by the *use* clause mechanism depends on the assumption that values of witness type or values containing values of witness type can only originate from the abstract instance aliased to the constant identifier. In the presence of infinite unions this assumption is incorrect since such values may be introduced into the use block from an infinite union.

Storage and retrieval of values of witness type from infinite unions may be required during the use of the compiler interface abstract type of Figure 4.10. For example, in the program of Figure 4.26 the checker part of the compiler is used and the result is placed into the persistent root which is of the infinite union type *env*.

```
use PS() with compilerPackage : compiler in
begin
        use compilerPackage as comp[ intRep ] in
        begin
                let someSource = ....
                let someUnboundValues = ....

                let checkerProc = comp( checker )
                let checkerResult = checkerProc(  someSource,
                                                  someUnboundValues )

                project checkerResult as X onto
                    fail          : ....
                    stillUnbound  : !** Value of witness type placed
                                    !** into an infinite union here.
                                    in PS() let anIntermediateProg = X
                default           : ....
        end
end
```

Figure 4.26 Placing a value of witness type into an infinite union.

The program of Figure 4.27 then retrieves the intermediate result and uses the binder procedure to specialise it. The execution of these two programs is only safe if the same abstract instance is used in both programs. This cannot be ensured statically. A dynamic mechanism is required to determine whether values of witness type introduced via an infinite union originated from the same abstract instance that is being manipulated by the *use* clause. The mechanism described here hinges on the use of appropriate type representations for the dynamic type checking operations associated with infinite unions.

```
use PS() with compilerPackage : compiler in
begin
        use compilerPackage as comp[ intRep ] in
        begin
                !** Value of witness type retrieved from infinite union here.
                use PS() with anIntermediateProg : intRep in
                begin
                        let someUnboundValues = ....

                        let binderProc = comp( binder )
                        let binderResult = binderProc(  anIntermediateProg,
                                                        someUnboundValues )

                        !** Manipulation of binderResult.
                        ....
                end
        end
end
```

Figure 4.27 Retrieving a value of witness type from an infinite union.

4.4.2 Dynamic Witness Type Checking

To reiterate, a dynamic check is required when a value of witness type is introduced into a *use* block from an infinite union. The check ensures that the value originated from the abstract instance being used. A dynamic type check is already performed whenever a value is retrieved from an infinite union to ensure that the value is of the type expected by the program. A type representation to be used for this purpose is associated with a value in an infinite union. The compatibility check between a witness value and an instance may become part of the dynamic type check if the type representation associated with the witness value is dependent in some way on the abstract instance from which the value originated.

The required dependency may be set up if, for every abstract instance, a unique type is created for each witness type in the instance. Similarly to the unique witness types used during the compilation of *use* clauses, name equivalence is used over these types.

A value of witness type is placed into an infinite union along with the unique type created for the witness type of the abstract instance from which the value originated. This is shown in Figure 4.28 which contains a program to place the contents of the *val* field of *counterOne* into the persistent environment. The code in italics indicates how the program is transformed during compilation. On entry to the *use* block, the unique witness type is declared for use inside the block. It is the unique type that is associated with the witness value in the persistent environment.

```
use counterOne as aliasedCounter in
begin
        type uniqueWitness is the unique type for the witness type t in counterOne
        in PS() let aValue = aliasedCounter( val ) : uniqueWitness
end
```

Figure 4.28 Placing a witness value into an infinite union.

When a witness value is retrieved from an infinite union, the type associated with the value is checked against the expected type. The expected type is the unique witness type of the abstract instance being used. Figure 4.29 illustrates how the code to look up a value in the persistent environment is transformed.

```
use counterOne as aliasedCounter in
begin
        type uniqueWitness is the unique type for the witness type t in counterOne
        use PS() with aValue : uniqueWitness in
        begin
                ....
        end
end
```

Figure 4.29 Retrieving a witness value from an infinite union.

A further complication arises where values whose types contain components of witness type are placed into infinite unions. Values of these types must also not be mixed between abstract instances. Consider for example the two abstract *use* clauses of Figure 4.30. The first clause puts the *inc* procedure from *counterOne* into the

persistent environment whilst the second retrieves it. The type of the *inc* procedure contains components of witness type.

```
use counterOne as aliasedCounter[ t ] in
        in PS() let counterInc = aliasedCounter( inc )

use counterOne as aliasedCounter[ t ] in
        use PS() with counterInc : proc( t → t ) in
        begin
                ....
        end
```

Figure 4.30 Types with components of witness type.

The solution is to create a parameterised type for these types. Components of witness type in the original type are of parameter type in the new type. On entry to a *use* clause the parameterised type is specialised with the unique witness type associated with the particular abstract instance being used. For example, the compiler transforms the second *use* clause of Figure 4.30 into that shown in Figure 4.31.

```
use counterOne as aliasedCounter[ t ] in
begin
        type newType[ param1 ] is proc( param1 → param1 )
        type uniqueWitness is the unique type for the witness type t in counterOne
        type specialisedType is newType[ uniqueWitness ]
        use PS() with counterInc : specialisedType in
        begin
                ....
        end
end
```

Figure 4.31 Using parameterised type for types with components of witness type.

Implementation

Type representations for the unique witness types may be constructed when the abstract instance is constructed and then stored in hidden fields added to the abstract interface. For example the *count* abstract type of Figure 4.20 is converted by the compiler into the type shown in Figure 4.32, where *Type* is the type representation used in the system.

```
type count is abstype[ t ]( val    : t ;
                            inc    : proc( t → t ) ;
                            tRep   : Type )
```

Figure 4.32 Adding fields for unique witness type representations.

Whenever the type representation is required for type checking, the compiler can generate code to dereference it from the particular abstract instance.

4.5 Conclusions

There are a number of benefits associated with the caching of a compiler within the persistent environment. The benefits are two-fold in that the functionality and operation of the compiler are enhanced whilst at the same time the performance of the persistent environment is improved via the optimised code produced by the compiler.

To reap any of the benefits a compiler must first be cached within the persistent environment. It has been shown here how the persistent application architecture of Chapter 2 may be used to support the construction of a large application using the Napier88 compiler as an example.

The benefits to the compiler and the environment may be categorised in three groups. Firstly, executing programs may access the compiler in order to construct and execute new programs which manipulate the environment. The many uses of such a facility are discussed elsewhere [KCC+92,FDK+92,SSS+92,Kir92].

Secondly, structured persistent values may be associated with source code which is then passed to the compiler. New styles of binding between programs and data may be realised. An interface to the compiler permitting a wide range of binding styles from totally static to totally dynamic binding has been described here. The interface distinguishes between the new binding styles and the traditional compilation operations of lexical analysis, syntax analysis, type checking and code generation.

Thirdly, the compiler may manipulate values in the persistent environment. An example of the associated benefits may be found in Chapter 3 where the compiler's own operation is improved by caching type representations between invocations. Another example is given in Chapter 5 where the implementation of polymorphism is optimised using dynamically gathered information cached in the persistent environment.

5 Using Persistence to Optimise Execution

5.1 Introduction

A persistent environment provides a conceptually unbounded space for the storage of data. Information detailing aspects of the environment's operation may be recorded there by executing programs. This information may then be accessible to applications executing in the environment. An opportunity exists for optimisation within the environment based on data collected during its operation, as follows:

- Data is collected and stored in the environment during program execution. The data records details of the program's execution. For example interactions between the program and the environment or patterns of usage among code segments may be recorded. This information cannot be determined statically.
- At some later time, perhaps when the environment is quiescent, an environment enhancing program may traverse the environment to find the information recorded by executing programs. The enhancer program analyses the information in an attempt to find optimisations that will improve the performance of the environment. The analysis is performed by a cost function which depends on a trade-off between the cost of making an improvement and the benefit gained from it. The enhancer performs the optimisation if the analysis is favourable. Optimisations range from the reorganisation of stored data to the transformation of program code. There may be many enhancers for different optimisations, each operating over different collections of data.
- Subsequent execution of the environment will be improved by the actions of the enhancers. Optimisations made as a result of data recorded by one executing program may be of benefit to many other programs. It should be noted that the optimisations are made on the assumption that the patterns of use recorded by executing programs are a reasonable prediction of future behaviour. In addition, if two independent optimisations operate over the same feature of the system, care should be taken to avoid repeated readjustment of that feature according to each optimisation. This is analogous to thrashing [PS87] in a virtual memory system where the shared resource is the main memory and the optimisers are the paging mechanisms of each process running in the system.

Dynamic clustering [BD90] is an example of this technique. Language values are clustered into segments on secondary storage. When a value is to be retrieved from secondary storage the whole segment that contains the value is read into main memory. Many unwanted values will be retrieved if every value accessed by a program is in a separate segment. During execution, records may be taken of the data items that are accessed by a particular program. Execution of that program may be enhanced if those data items can be clustered onto the same segment in secondary storage.

A second example is the caching of the results of complex operations in the persistent environment. This has already been seen in Chapter 3 where the results of type system operations were cached. The results may benefit any program that has access to the cache. In this case the cost function is built in and always gives a positive result and the enhancement is performed automatically.

Both of the above examples, clustering and caching, are concerned with the efficient utilisation of data. Where programs are considered as data there is the possibility of using the same architecture to improve the execution sequences in accordance with the needs of the application. Code enhancement is possible since code generators make static trade-offs with regard to the space requirement and run-time execution speed of the code. Dynamic execution information can be used as a basis for changing these trade-offs. In this chapter the technique is used to enhance the implementation of polymorphism in Napier88 [MDC+91].

The concepts involved in the general optimisation architecture were developed during construction of the Napier88 integrated environment. Suggestions for optimisation of the implementation of polymorphism found in Napier88 Release 1.0 have been made in [MDC+91]. The optimisation architecture and the polymorphic optimisations were not combined until construction of the integrated environment was complete since a compiler available within the environment is a requirement of the optimisation technique.

5.2 Polymorphic Procedures

Polymorphism in a programming language is the ability to write programs that are independent of the form of the data values that they manipulate. Thus it provides an abstraction over the form of the data which is often categorised by type. A survey of different styles of polymorphism may be found in [MDC+91]. The style considered here is parametric polymorphism in which a polymorphic function has an explicit or implicit type parameter determining the type of an argument for each application of the function.

5.2.1 Napier88 Polymorphic Procedures

A description of the declaration and use of Napier88 polymorphic procedures will be given to indicate the features that an implementation must support. The polymorphic identity procedure shown in Figure 5.1 will be used as an example. The identifier *id* is declared to be a procedure which is quantified by *t*, written [t], that takes a parameter *x* of type *t* and yields a value of type *t*. The body of the procedure is the expression *x* which when evaluated yields the result.

```
let id = proc[ t ]( x : t → t ) ; x
```

Figure 5.1 Napier88 polymorphic identity procedure.

In the implementation of Napier88, a call of the identity procedure takes place in two stages. The procedure must first be specialised to a particular type and then applied to a value of that type. Figure 5.2 shows a call of the procedure where the quantifier *t* in procedure *id* has been specialised to integer, written **int**. The procedure is applied to the integer value *7* and the result is of course the integer *7*.

```
id[ int ]( 7 )
```

Figure 5.2 Calling *id* with an integer parameter.

The procedure may be specialised to any type. For example Figure 5.3 shows a specialisation and application for a real value yielding in the same manner the real value *7.0*.

```
id[ real ]( 7.0 )
```

Figure 5.3 Calling *id* with a real parameter.

The specialisation and application operations may take place independently. Figure 5.4 shows the specialisation of the *id* procedure to the string type without an application. The specialised procedure is assigned to the identifier *stringId*. Such a procedure will be referred to as partially specialised.

```
let stringId = id[ string ]
```

Figure 5.4 Specialising *id* to type string.

The same specialised procedure may now be applied many times to different string values. Figure 5.5 illustrates this point.

```
let hello = stringId( "hello" )
let bye = stringId( "bye" )
```

Figure 5.5 Applying a specialised procedure.

The procedure *stringId* is type compatible with monomorphic procedures taking a string parameter and returning a string. For example it is compatible with the procedure *monoStringId* in Figure 5.6.

```
let monoStringId = proc( a : string → string ) ; a
```

Figure 5.6 Non-polymorphic string identity function.

It is expected that the 'specialise once, use many times' mode of operation will occur frequently where polymorphic procedures may be placed into the persistent environment. This is where a polymorphic procedure stored in the environment is retrieved by a program, specialised once to the types associated with the program and then the specialised version used many times. For example consider the *length* procedure in Figure 5.7 which determines the length of a *list* value. *list* is a parameterised type used to model homogeneous lists of data values. Polymorphic procedures often operate over parameterised data types that are specialised by the quantified types of the procedure. This occurs in *length* which is parameterised by a value of the *list* type specialised to the quantifier type *t* of the procedure. ***in*** *PS()* at the start of the code indicates that the declaration of the procedure is to be made in the persistent environment and not in the local scope. Details of the procedure body are not important here.

```
type list[ s ] is .....

in PS() let length = proc[ t ]( theList : list[ t ] → int ) ; .....
```

Figure 5.7 Placing a polymorphic procedure into the persistent environment.

The procedure may be used in any program in which the length of a list is required. For example a program that organises information about people may need to know the length of lists of people. It is expected that the program would use the *length* procedure as shown in Figure 5.8 where the procedure has been retrieved from the store and specialised to the type *Person* at the start of the program. It is the specialised version *lengthPerson* that is used subsequently in the program.

```
type list[ s ] is ....
type Person is ....

use PS() with length : proc[ t ]( list[ t ] → int ) in
begin
      let lengthPerson = length[ Person ]

      !** The specialised procedure may be used many times in the program.
      .....
end
```

Figure 5.8 Specialising and using a persistent polymorphic procedure.

5.2.2 Implementing Polymorphism

Polymorphism may be expressed at many levels of abstraction depending on the style of implementation. The following three categories represent extremes in the possible range of implementation techniques [MDC+91]:

- Textual polymorphism. In this category polymorphism is only expressed at the source code level. Different executable code may be produced for each different specialising type. The execution efficiency of the equivalent monomorphic procedure will be achieved using this technique. This is the optimum efficiency. However textual polymorphism may be expensive in terms of the storage space required for the specialised forms. An example of this kind of polymorphism is found in the generics of Ada [DOD83].
- Uniform polymorphism. Both the source code and the executable code are independent of particular specialising types in this category and so only a single executable version of a polymorphic expression is required. This is achieved using a single representation for all data. The use of space for polymorphic code forms is optimal. However, the uniform data format has efficiency implications for non-polymorphic data values in terms of both space and time. This kind of polymorphism is found in ML [Mil78].
- Tagged polymorphism. In this category uniformity is expressed both at the source and executable levels but non-uniform data formats are used. Thus the executable code for a polymorphic expression is parameterised in some way by type information describing the representation of data

values being manipulated. Effectively every data item is tagged with its type. Code space is again optimised but all values have to pay the price of the tagging which is expensive if performed in software and is generally not available in hardware. Tagged polymorphism is used in the implementation of some object-oriented languages [GR83].

These implementations represent different trade-offs between run-time efficiency and the space required for polymorphic code forms. A particular trade-off is traditionally determined statically during the code generation phase of compilation and is therefore fixed for the lifetime of the procedure. The trade-off is affected by the various different styles of polymorphic procedures. For example, textual polymorphism is suitable for the compilation of the generics of Ada since the polymorphic code and the specialising types are known statically and so the number and type of required code forms is also known.

By contrast, the polymorphic procedures of Napier88 may be anonymous. It is therefore impossible to determine statically which specialisations will be performed during the lifetime of a polymorphic procedure. Moreover there may be a very large number of possible specialisations as shown in the following example.

Given that there are a fixed number of representations onto which a language's data types may be mapped there is a fixed upper limit to the number of different specialisations that may be required. This is the number of representations raised to the power of the number of type variables. For example the procedure in Figure 5.9 is quantified by three identifiers *d*, *e* and *f* and takes three parameters *D*, *E* and *F*. There is no result type and the body of the procedure is not of interest here. In Napier88 where there are seven possible representations onto which a data type may be mapped there are 7^3 or 343 possible specialisations of the polymorphic procedure.

```
let triples = proc[ d,e,f ]( D : d ; E : e ; F : f )
        .....
```

Figure 5.9 A polymorphic procedure with multiple quantifiers.

Where the trade-off between space and execution speed is made statically the use of textual polymorphism to implement the anonymous polymorphic procedures of Napier88 requires that all possible specialisations are constructed statically, a prohibitively expensive operation in terms of both time and space.

Both uniform and tagged polymorphism overcome the problems of excessive code space but suffer in terms of execution speed. In particular, the execution of all code is adversely affected by the inclusion of polymorphic constructs.

The implementation of the polymorphic procedures of Napier88 makes use of a general mechanism [MDC+91] which represents an optimisation over uniform and tagged polymorphism. It is a mix of the two schemes and will be referred to here as partly-tagged, semi-uniform polymorphism. A major advantage of the mechanism is that it does not affect the efficiency of non-polymorphic code in any way. However the poorer execution speeds of uniform and tagged polymorphism in comparison to textual polymorphism are inherited for code inside polymorphic contexts.

A Mixed Implementation Strategy

The design of partly-tagged, semi-uniform polymorphism is a result of the environment in which the construction of executable code for the procedures takes

place. This environment is traditionally separated from the execution environment of the procedures and is only reachable from the execution environment with considerable difficulty. Using such an environment, analysis of the use of polymorphic procedures takes place before generation of the executable code which in turn takes place before execution of the procedure.

In a persistent environment however, the construction and execution environments are the same and so the implementation decisions for partly-tagged, semi-uniform polymorphism may be reconsidered. Firstly, the persistent environment may be used to cache execution profiles of programs. This data may be analysed later. Secondly, the cached compiler described in Chapter 4 may be used at any time to construct code. Decisions of implementation methods may now be based on both static and dynamic information. For example, in the extreme case textual implementations of the polymorphic procedures may be constructed during execution when required. Although this may be efficient in terms of code space and execution speed of the procedures, the time required to construct the versions is likely to significantly reduce overall performance.

This chapter describes an extension to the partly-tagged semi-uniform implementation which makes use of textual polymorphism, thus providing execution speeds that are frequently optimal. During compilation, a version of the procedure is compiled using the partly-tagged, semi-uniform technique. Code is included in the procedure to cache execution profiles in the persistent environment concerning patterns of use such as the number of specialisations, calls and polymorphic operations.

After the partly-tagged, semi-uniform version of the procedure has been in use for some time, analysis of the cached execution profiles may indicate particular specialisations where the space-time trade-off favours a textual implementation. That is, the cost of constructing a specialised version of the procedure is small in comparison to the efficiency gained through using it. The analysis may be performed using a cost function operating over execution profiles. The cost function may restrict the potentially large number of specialised versions that could be constructed. The space-time trade-off determined by the cost function is based on static and dynamic information since the cost function itself may contain static characteristics of the procedure whilst the function's arguments record dynamic characteristics. There is another trade-off here between the amount of information coded into the cost function and the amount of information gathered during execution. The cost function may be complex since the expense incurred in its evaluation is offset by the fact that the analysis can be performed during quiescent points in system operation.

Specialised versions of a polymorphic procedure may be constructed using the compiler resident in the environment. The original procedure is accessible from the execution profiles. The source code is retained with the original procedure and is transformed to reflect the data format of the specialising type. Recompilation produces a form of the procedure containing no polymorphic operations and so optimum execution efficiency may be gained when using it. Such a specialised version will be referred to as a concrete specialisation.

Concrete specialisations may be associated with the partly-tagged, semi-uniform version of a procedure. During a specialise operation a search for an appropriate concrete specialisation is carried out. If one is found then it is returned as the result of the specialisation otherwise the partly-tagged, semi-uniform version is used.

The advantages of the extended implementation may be summarised as follows:

- Optimum execution speed may be achieved.
- Excessive use of code space may be limited using an appropriate cost function.

- Optimisation is based on both static and dynamic information about procedure execution.
- Analysis and construction of optimised forms may be carried out during quiescent periods so as not to affect the efficiency of system operation.

The extended implementation is described in the following four sections:

- The partly-tagged, semi-uniform implementation is described independently of the extended implementation.
- The association of concrete specialisations with the partly-tagged, semi-uniform version of a procedure is described along with the manner in which a particular concrete specialisation is chosen when the polymorphic procedure is specialised.
- The recording and analysis of execution profiles and the characteristics of cost functions are discussed.
- The use of a program known as a polymorphic code enhancer is described. An enhancer finds the execution profiles, performs the necessary analysis and constructs specialised versions as required.

5.3 Partly-tagged, Semi-uniform Polymorphism

Partly-tagged, semi-uniform polymorphism represents a mix of both the uniform and tagged styles. In non-polymorphic contexts values are contained in non-uniform representations and are manipulated by operations specialised to those representations in order to maximise efficiency. It is only the operations in polymorphic expressions that must function over values of any type. Values of the quantifier type are therefore coerced to a uniform polymorphic representation on entering a polymorphic procedure and coerced back on leaving it. Inside the procedure they are manipulated in the uniform polymorphic representation. This is the uniform part of the implementation.

The correct execution of some operations over values of quantifier type depends on the values' original representations. Information describing those representations must therefore be available during execution. For example, the coercion operations require knowledge of the original representation of the values to be coerced. This is the tagged part of the implementation. Note that it is only values of quantifier type that have an associated tag.

Tagging requires that the data types of a language's possibly infinite type system may be mapped onto a finite tag representation. Usually many different types share representations and an encoding of the representation can be used as a tag. For example, the infinite number of data types described by the Napier88 type system may be mapped onto seven distinct tags and so a tag is represented by an integer.

As described above, some polymorphic operations require access to tag information in order to execute correctly. Different mechanisms may be used to make this information available [Con91]. The Napier88 partly-tagged, semi-uniform implementation uses the block retention architecture of the language to support the storage of tag information.

Block Retention

The mixed implementation in Napier88 uses the block retention architecture of the language to support the storage of tag information. Block retention is necessary to support higher order procedures [Joh71]. The example in Figure 5.10 requires block retention to execute correctly. The value of the block is assigned to the identifier *counter*. Inside the block an integer variable *count* is declared and the value of the

block is a procedure. The space created for the block during the execution of the *counter* declaration must be retained since it contains the variable *count* which is in the closure of the procedure. Reclaiming the space would lead to incorrect operation of the procedure.

```
let counter = begin
                  let count := 0

                  proc( → int )
                  begin
                        count := count + 1
                        count
                  end
              end
```

Figure 5.10 Block retention.

Using Block Retention to Store Tag Information

A Napier88 polymorphic procedure is compiled into another procedure which takes the specialising tag information as a parameter. This procedure will be referred to as the *encapsulating* procedure. The encapsulating procedure returns an *encapsulated* procedure containing the executable code for the original procedure. For example the *id* procedure of Figure 5.1 is compiled as shown in Figure 5.11.

```
let compiledId = proc( tag : int → proc( α → α ) )      encapsulating procedure
                 proc( x : α → α ) ; x                  encapsulated procedure
```

Figure 5.11 Encapsulating and encapsulated procedures.

Specialisation of a polymorphic procedure is compiled into a call of the encapsulating procedure passing tags for the specialising types as parameters. For example the specialisation *id[**int**]* is compiled into *compiledId(intTag).* *intTag* is the tag for the integer type.

The result of a specialisation is the encapsulated procedure which contains the specialising tags in its closure. Tags are retained in the stack frame created on execution of the encapsulating procedure. They may be accessed by instructions in the encapsulated procedure that depend on the original representation of quantified values.

The specialised procedure may be applied immediately to parameter values of appropriate types or else stored for later use. The specialised procedure will be referred to as a polymorphic specialisation.

The code in the specialised procedure is the same for all specialisations. It is the retained stack frame from the execution of the encapsulating procedure that differs in each specialised version. The type α is statically unknown and depends on the concrete types of the quantifiers for a particular specialisation.

Converting to and from the Uniform Polymorphic Form

As stated earlier, parameters to the specialised procedure that are of quantified type must be converted to the uniform polymorphic form on entry to the procedure. In addition a result that is of quantified type must be reconverted from polymorphic form on procedure exit.

In the case of a specialised *id* procedure the parameter must be converted into the polymorphic form for use inside the procedure and then reconverted on exit as shown in Figure 5.12. The conversion operations use the tag information contained in the closure of the encapsulated procedure.

```
proc( x : α → α )
begin
        convert x to polymorphic form
        x
        convert x to concrete form
end
```

Figure 5.12 Converting values to and from polymorphic form.

Other operations also require access to tag information in order to function correctly, as follows:

- Equality is defined differently for the various data type of Napier88. In a polymorphic context, tag information is used to determine which equality function should be applied to polymorphic values.
- Values in data structures are always stored in their concrete format despite the fact that they may sometimes be viewed with a polymorphic type. This is because the same data structure may pass between monomorphic and polymorphic contexts [MDC+91]. The operations that store and retrieve values of polymorphic type from data structures must perform appropriate coercion operations and so require access to tag information.

5.4 Storing and Choosing Concrete Specialisations

Some explanation of the representation and execution of Napier88 procedures is required in order to describe the manner in which concrete specialisations are stored and chosen. Napier88 executes on the Persistent Abstract Machine (PAM) [CBC+90], a byte coded machine that operates over a single persistent heap of objects. A procedure value or closure consists of two pointers, one to the code vector object for the procedure and the other to the stack frames containing the environment of values in which the procedure was declared. The latter will be referred to as the environment pointer. Each stack frame is implemented as a separate object on the heap in order to implement the block retention architecture required for first-class procedures.

The code vector is formatted similarly for all kinds of procedures, i.e. for monomorphic procedures, the encapsulating and the encapsulated procedures of the mixed polymorphic implementation and the concrete specialisations of the textual extensions to be described here. The code vector associated with a concrete specialisation will be referred to as a concrete code vector. The format for code vectors is as shown in Figure 5.13. As with all PAM objects the code vector object

begins with a header and a size field followed by all the pointers in the object and then all the non-pointers. The first pointer is to the source code of the procedure and the second is used to determine the identity of procedures. They are followed by literal pointer values used during the execution of the procedure. This is followed by the non-pointer values made up of the code and other non-pointer values required for execution.

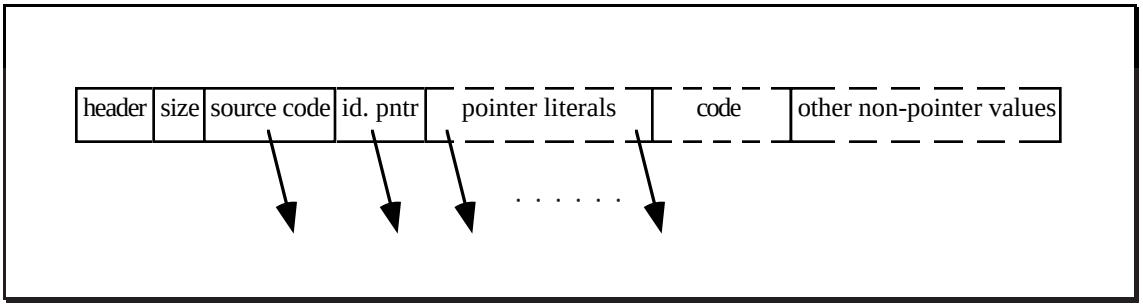

Figure 5.13 The format of a code vector.

Creation of a procedure value, or closure, consists of associating a pointer to a code vector constructed during compilation with a pointer to the list of stack frames making up the environment in which the procedure is declared.

Applying a procedure results in the creation of a new stack frame. The layout of a frame is shown in Figure 5.14. A frame contains a pointer to the code vector for the procedure (CV) and the static link or environment pointer of the applied procedure (SL). Each stack frame contains two stacks, one for pointer values and one for non-pointers. House-keeping information is kept at the top of the stack frame.

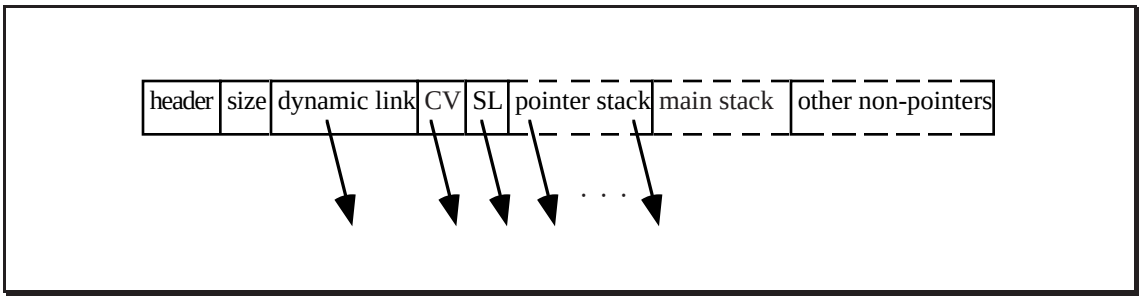

Figure 5.14 The format of a stack frame.

5.4.1 Storing Concrete Specialisations

The concrete specialisations of a polymorphic procedure are held in the recursive data structure denoted in Figure 5.15. A list has been used here for simplicity in explanation; a vector would be more sensible for a real implementation to allow fast lookup.

```
rec type listElement is structure( tags   : tagCombination ;
                                   cv     : codeVector ;
                                   next   : cVecList )
    & cVecList is variant( concrete : listElement;
                           tip      : null )
```

Figure 5.15 Data structure to hold specialised versions.

Each list element holds a single concrete code vector and key which is a combination of the particular type tags for which this code vector is specialised. The list initially consists of a single instance of *listElement* containing the code vector for the encapsulated procedure of the mixed implementation. As concrete specialisations are constructed they may be added to the end of this list. The list is associated with the encapsulating procedure by placing a pointer to it in the list of pointer literals as shown in Figure 5.16. Other pointers are not shown to avoid confusion.

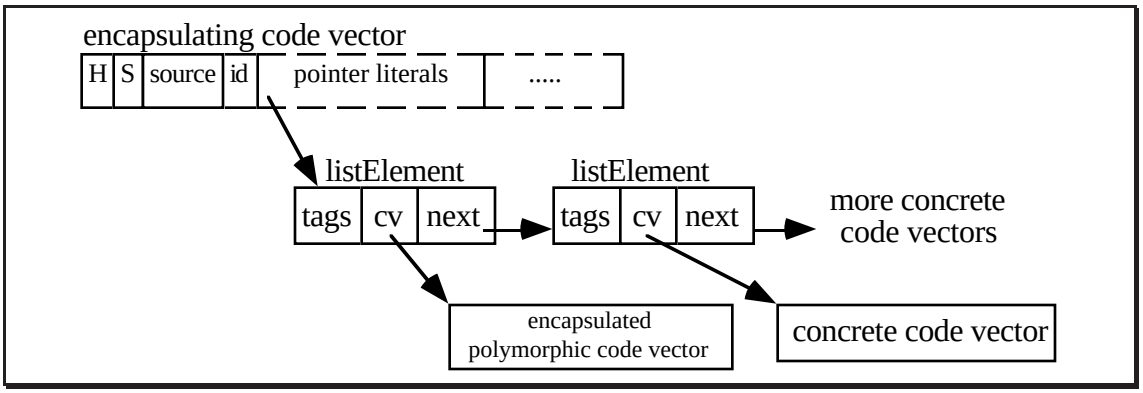

Figure 5.16 Storing the code vectors of concrete specialisations.

5.4.2 Retrieving Concrete Code Vectors

Once concrete code vectors have been constructed and associated with the polymorphic version of the procedure, access is required to them when suitable specialisations occur. This may be achieved by adjusting the encapsulating procedure used in the mixed implementation to perform the specialisation of a polymorphic procedure. Code is added to the encapsulating procedure to search the *cVecList* for a suitable code vector. This search is only performed during specialisation. Provided that the 'specialise once, use many' mode of use is prevalent then the cost of the search will not be significant in comparison to the subsequent calls. For the identity procedure of Figure 5.1 the new version of the encapsulating procedure is shown in Figure 5.17.

The body of the procedure is split into three sections. In the first section the list of specialised code vectors is accessed via the code vector for the encapsulating procedure. *polySpecCVs* is the field offset in the encapsulating code vector of the list of specialised code vectors. The code vector is found in the frame created on application of the procedure and accessed using the standard procedure *thisFrame*.

The second section uses the procedure *searchForConcrete* to determine whether the list of concrete code vectors contains a version suitable for the particular tag supplied to this application of the procedure. *searchForConcrete* chains down the list attempting

to match the tag passed as a parameter with the tag contained in the list element. If a match is found then the corresponding code vector is returned otherwise **nil** is returned.

```
let compiledId = proc( t : tag → proc( α → α ) )
begin
        !** Retrieve the list of code vectors.
        let currentFrame = thisFrame()
        let encapsulatingCVec = currentFrame( CV )
        let polyAndSpecCVecs = encapsulatingCVec( polySpecCVs )

        !** Attempt to find a concrete specialisation suitable for the tag t.
        !** searchForConcrete returns nil if no suitable version exists.
        let specialisedCVec = searchForConcrete( t,polyAndSpecCVecs )

        !** Return a concrete specialisation if available,
        !** otherwise a polymorphic version.
        if specialisedCVec ~= nil
           then  !** Concrete specialisation.
                 makeProcedure( specialisedCVec,currentFrame( StaticLink ) )
           else  !** Poly specialisation.
                 !** Code vector found at head of code vector list.
                 makeProcedure( polyAndSpecCVecs( cv ),currentFrame )
end
```

Figure 5.17 Extended encapsulating procedure.

The third section constructs a suitable specialised procedure value using the standard function *makeProcedure* which forms a closure by associating a code vector with a list of stack frames. Concrete and polymorphic code vectors are associated with slightly different environments. The polymorphic code vector is associated with the frame created on invocation of the encapsulating procedure since it is this frame that contains the tag information required by the polymorphic version. A concrete code vector however is associated with the same list of frames as was the encapsulating procedure. This list is the static link of the frame constructed on invocation of the encapsulating procedure. The two kinds of specialisation are constructed against different compilation environments to ensure they operate correctly during execution.

Concrete and polymorphic specialisations over the same type may be used entirely interchangeably. They both expect parameters in concrete form and will return results also in concrete form.

It should be noted that where polymorphic procedures are manifest values, such as in a hyper-programming environment [KCC+92], it may be possible during compilation to find an appropriate concrete version of the procedure that is being specialised. The concrete version may be bound directly into the compiled code.

5.5 Cost Functions and Execution Profiles

The decision on when to construct a concrete specialisation is made by a cost function on the basis of execution profiles recorded in the persistent environment during the execution of the polymorphic specialisations. The cost function may also access statically gathered information about procedures. It is of interest to determine which data should be recorded and in what manner it is stored.

5.5.1 Affordable vs. Exact Execution Profiles

Execution profiles and cost functions are used to determine which concrete specialisations are worth constructing, according to a particular trade-off. The trade-off concerns the following costs:

- The extra time spent executing polymorphic specialisations in comparison to their concrete counterparts.
- The extra space used to store concrete specialisations.
- The expense incurred in constructing a concrete specialisation.

A new concrete specialisation is constructed when the cost in execution speed incurred using a polymorphic specialisation is shown to be significant in comparison to the space and time required to construct and store a concrete specialisation.

The extra expense of executing a polymorphic specialisation in comparison to the equivalent concrete specialisation is caused by the execution of operations depending on the concrete type of values of quantifier type. In addition, operations that move polymorphic values around the system are also more expensive than their monomorphic counterparts. A record of the number of executions of these operations is required in order to determine the difference in efficiency between the two specialisations. Records of this kind may only be taken during execution of the program being measured and would significantly affect the performance of the program.

By making judicious choice of which execution profiles to collect and which static information to bind into the cost function, a good approximation to the exact cost of a polymorphic specialisation can be made without incurring a significant expense. The information bound into the cost function is determined statically and includes items such as:

- The number of conversions required on procedure entry and exit. This is a fixed cost that must be performed on every procedure invocation.
- The number of polymorphic operations in the code. This may be used to good effect when combined with the number of executions, but is heavily dependent on the flow of control within the program.

The execution profile consists of data that can only be collected accurately during execution, such as:

- The number of specialisations to a particular type.
- The number of calls of a particular specialisation.
- The number of executions of polymorphic instructions. This is likely to be prohibitively expensive to collect.

In the implementation described here, there is a single cost function for all procedures which takes both static and dynamic information as arguments. The static details are accessible in the encapsulating polymorphic code vector. Initially, the precision of the analysis has been limited to static details concerning the number of conversions and the size of the procedure and to dynamic details concerning the number of specialisations and number of calls of particular specialisations. The cost function determines threshold levels for combinations of the static and dynamic information above which optimisation is appropriate.

Since the cost function may be performed during quiescent periods it may be of arbitrary complexity. Approximations of expected dynamic details may be included in

the static information to reduce the size and hence the cost of recording and storing the execution profiles.

It should be noted that the validity of this analysis is based on the patterns of use up to the point of the analysis. An optimisation is only valuable if the patterns of use remain the same after optimisation.

5.5.2 Storing the Execution Profiles

Execution profiles may be held in an extended version of the data structure used to hold the specialised code vectors already given in Figure 5.15. The *cVecList* type may be extended as in Figure 5.18. A new *poly* branch may be constructed and added to the code vector list for each different specialisation of a polymorphic procedure that is encountered during execution. Execution profiles may therefore be recorded about each different specialisation. The structure associated with the *poly* branch, *profile*, contains a combination of the tags denoting the particular specialisation to which the corresponding execution profile relates. The version of the structure type shown here contains only the number of specialisations and calls. An arbitrary amount of data may be recorded here by extending the type.

```
rec type  listElement is structure(  tags   : tagCombination ;
                                     cv     : codeVector ;
                                     next   : cVecList )
&         profile is structure(      tags           : tagCombination ;
                                     specialisations : int ;
                                     calls          : int ;
                                     next           : cVecList )
&         cVecList is variant(       concrete : listElement;
                                     poly     : profile ;
                                     tip      : null )
```

Figure 5.18 Updated *cVecList* to hold execution profiles.

Execution profiles are recorded to determine when a particular concrete specialisation of a polymorphic procedure should be constructed. Once this has occurred the associated *poly* branch may be replaced by a *concrete* branch containing the new version. If profiling information is still to be recorded from the new concrete version then the *poly* branch may be associated with it.

The *searchForConcrete* procedure is adjusted to return one of the variant branches of the *cVecList* type. The procedure may now discover one of the following three situations when the *cVecList* is examined:

- A concrete version of the procedure exists for the combination of tags supplied. The variant branch containing the concrete version is returned from *searchForConcrete*.
- A *poly* branch exists for the combination of tags supplied which means that this specialisation has occurred before but a concrete version has not yet been constructed. The structure associated with this branch may be shared by all polymorphic specialisations to the corresponding tags by returning the branch from *searchForConcrete* and making it available to the result of this specialisation. The *specialisations* field may be incremented before returning the structure.

- Neither a *poly* nor a *concrete* branch exist for the supplied combination of tags and so a new *poly* branch is added to the end of the list and returned from the procedure. Again, the *specialisations* branch may be incremented.

The encapsulating procedure that is called on specialisation of a polymorphic procedure is also adjusted to accommodate the recording of execution profiles, as shown in Figure 5.19. If *searchForConcrete* returns a *concrete* branch then the associated structure is dereferenced to give the concrete code vector. Otherwise a polymorphic specialisation is constructed as before. The head of the list of frames making up the environment of the specialised closure is the frame constructed when the encapsulating procedure was executed, i.e. the *id* procedure of Figure 5.19. As can be seen from the figure, this frame must contain a location for the identifier *cVecOrStats*. For a polymorphic specialisation this location contains the *profile* structure for the particular specialisation, as shown in Figure 5.20. The polymorphic code vector may be constructed with this in mind to enable profiling information concerning its execution to be recorded.

```
let id = proc( t : tag → proc( α → α ) )
begin
    !** Retrieve the list of code vectors.
    let currentFrame = thisFrame()
    let encapsulatingCVec = currentFrame( CV )
    let polyAndSpecCVecs = encapsulatingCVec( polySpecCVs )

    !** Attempt to find a concrete specialisation suitable for the tag t.
    !** searchForConcrete returns a statistics structure
    !** if no suitable version exists.
    let cVecOrStats = searchForConcrete( t,polyAndSpecCVecs )

    !** Return a concrete specialisation if available,
    !** otherwise a polymorphic version.
    !** is tests a variant to determine which branch it is.
    if cVecOrStats is concrete
      then  !** Concrete specialisation.  ' projects a variant onto
            !** the named branch.
            makeProcedure(  cVecOrStats'concrete( cv ),
                            currentFrame( StaticLink ) )
      else  !** Poly specialisation code vector found at
            !** head of code vector list.
            makeProcedure( polyAndSpecCVecs( cv ),currentFrame )
end
```

Figure 5.19 Recording statistics in the encapsulating procedure.

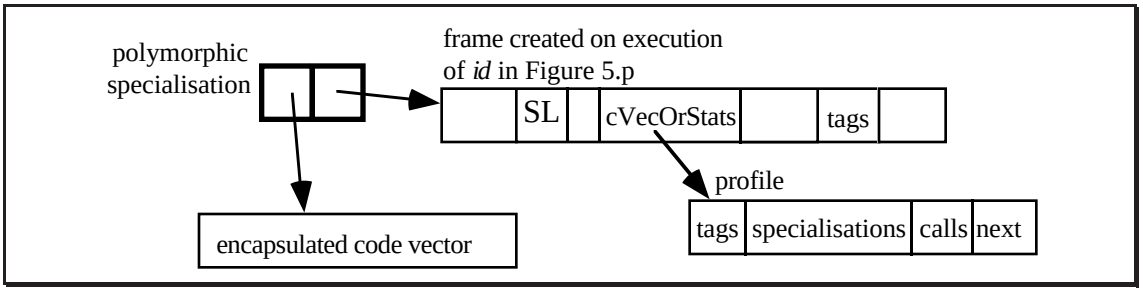

Figure 5.20 A polymorphic specialisation showing the associated profile information.

Making Execution Profiles Persistent

Changes made to the persistent environment do not become permanent until a *stabilise* operation is performed [Bro89]. This operation ensures that all changes to the environment are transferred from volatile to non-volatile storage. The Napier88 system performs a stabilise operation when the system shuts down normally and at other times during the execution of the system as required. The changes made to the statistics stored with polymorphic procedures will become permanent when these stabilise operations are performed provided that the procedures themselves are reachable in the persistent environment. The normal mode of operation is to terminate correctly in which case the execution profile will be correctly recorded in the persistent environment to be shared by all uses of the procedure.

5.6 Polymorphic Code Enhancer

The architecture described so far supports the storage and use of specialised forms of polymorphic procedures and the gathering of statistics concerning the use of polymorphic forms. The manner in which the statistics are analysed and the specialised forms constructed is now described.

A program known as the polymorphic code enhancer traverses the persistent environment in order to find procedure closures that can be optimised. The enhancer searches for two kinds of closure during its traversal. The first is the closure for an unspecialised polymorphic procedure and the second is for a polymorphic specialisation. The first contains the execution profiles of many different polymorphic specialisations, the second just a single profile for the particular specialisation.

The enhancer applies the cost function to both the static and the dynamic profiling information concerning a particular specialisation. The static information, which is the same for all specialisations, is recorded in the encapsulating code vector and is accessible via a pointer in the pointer literals vector. The information is therefore available from either of the closure types that the enhancer encounters since they both contain pointers to the encapsulating code vector.

Where the cost function returns a positive result, a new concrete specialisation is constructed. This is achieved by applying the compiler found in the persistent environment to a version of the procedure's source which has been transformed appropriately for the specialising type by the enhancer.

The new version is then included in the list of concrete specialisations attached to the encapsulating code vector. If the closure found originally by the enhancer was a

polymorphic specialisation then the new code vector overwrites the polymorphic version.

The various operations performed by the enhancer are now described in more detail.

5.6.1 Compiling Concrete Specialisations

The enhancer constructs a new concrete code vector when the cost function indicates that it is worthwhile. The enhancer has access to the following components:

- The source code of the encapsulated procedure. This is attached to the encapsulated code vector and consists of the lexemes that make up the procedure body and symbol tables for outer scope references.
- The tags for which the new concrete code vector is to be specialised. These are available in the *poly* branch for this specialisation.
- A compiler. As described in Chapter 4 a compiler is available in the persistent environment for use by any program.

A source code transformation is performed over the polymorphic procedure to reflect the particular specialising type tags. For example, consider the *id* procedure, reproduced in Figure 5.21

```
proc[ t ]( x : t → t ) ; x
```

Figure 5.21 Polymorphic *id* procedure.

If a version of *id* specialised to the *int* tag is to be constructed, the code is transformed as shown in Figure 5.22.

```
proc( x : int → int ) ; x
```

Figure 5.22 Code transformation of the *id* procedure.

The new concrete code vector is constructed by compiling the transformed source code using the persistent compiler available to the enhancer.

The enhancer only has access to the tag information for a particular specialisation, and not to a valid Napier88 type. However, in the source code transformation polymorphic types may be replaced with any type that is consistent with the specialising tags. For example, *int* and *bool* use the same tag and so either could have been used in the transformation of Figure 5.22.

5.6.2 Finding Procedures in the Persistent Environment

The enhancer must be able to identify procedure closures in order to operate. One method is for the compiler to record references to the code vectors of every encapsulating polymorphic procedure that it constructs in a data structure at a known location in the persistent environment. The enhancer may access these code vectors provided it has access to the data structure.

Alternatively, polymorphic specialisation closures can be found by traversing the persistent environment. The type information attached to values in the environment may be used to determine the locations of such closures. Type information is attached to values in two ways, as follows:

- Values may be placed into infinite unions. An infinite union contains the type of the associated value. The enhancer is able to determine the type of the value contained in an infinite union in order to traverse the value. Since the major structuring unit in the persistent environment is the infinite union *env*, this is the primary method by which the enhancer traverses the environment.
- Values may be contained in procedure closures. Provided that the source code is retained with all procedures, as is the case in the hyper-programming environment [KCC+92], the enhancer can find the appropriate closures using source code analysis.

5.6.3 Overwriting Polymorphic Specialisation Closures

When a polymorphic specialisation closure is encountered by the enhancer, enhancement may take place in one of the following two ways:

- Examination of the list of concrete specialisations accessible from the closure may show that a concrete version for this specialisation has already been constructed.
- Otherwise, analysis of the execution profile for the specialisation may indicate that a new concrete version should be constructed. If so, the enhancer constructs the new version.

Both cases produce a concrete code vector that is to be used in place of the polymorphic version encountered by the enhancer. The existing procedure value may be updated as shown in Figure 5.23. First, the location holding the polymorphic code vector is overwritten by the new concrete version. Secondly, the location holding the environment pointer of the polymorphic version is overwritten with the environment pointer of the encapsulating polymorphic procedure. The latter pointer is contained in the static link field of the frame pointed to by the former pointer as described in Section 5.3.3.

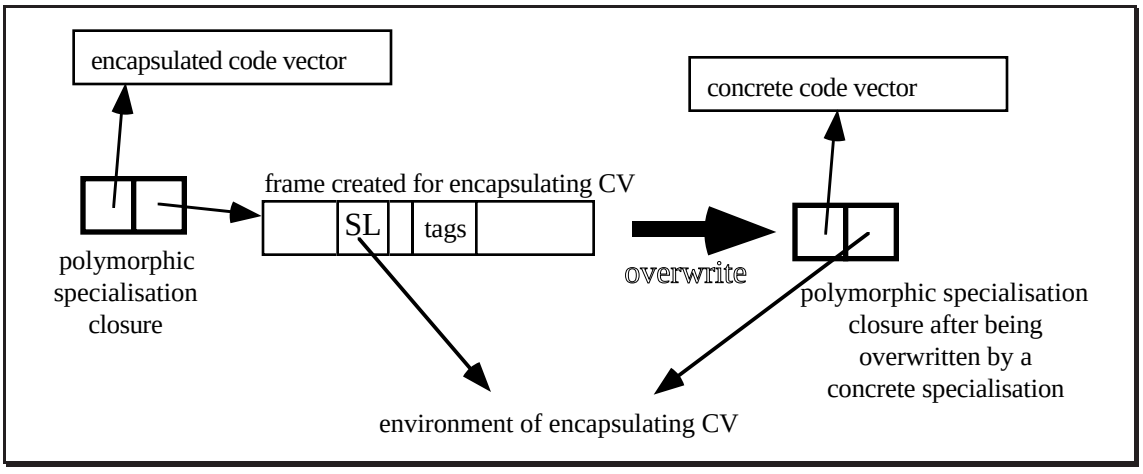

Figure 5.23 Overwriting a polymorphic specialisation closure in place.

5.7 Conclusions

The persistent environment simplifies the process of storing complex data in between program invocations. In this chapter it has been shown how data may be collected during execution that catalogues certain aspects of program behaviour. The data may be stored in locations accessible to programs that browse over the environment. Analysis of the data by the browsing programs may indicate optimisations for the programs that originally produced the data. These optimisations may also be of benefit to other programs.

The implementation of the polymorphic procedures of Napier88 has been described here as an example of the this optimisation technique. From a starting point of a complete but unoptimised implementation of the procedures it has been shown how data describing patterns of use over the procedures may be collected. This information cannot be determined statically. Enhancement programs traverse the persistent environment analysing this data and constructing optimised forms of the procedures where appropriate. The analysis is based on a trade-off between the cost of constructing optimised versions and the benefit gained from their use. Subsequent execution of the polymorphic procedures may access the optimised versions for improved efficiency.

6 Conclusions

The motivation for the research contained in this thesis is the simplification of the construction, maintenance and execution of large-scale, long-lived and data-intensive application systems. The complexity of these operations is increased by the plethora of independent sub-systems such as database systems, programming languages and operating systems used to make up a complete application management environment. Whilst the tasks performed by the environment are intrinsically complex the belief in this thesis is that the complexity caused by lack of integration between sub-components can be avoided.

The thesis assumes that the avoidable complexity may be removed by embedding all programming processes within a single strongly typed persistent environment. There are many accepted benefits of persistence to the software engineering process primarily realised in the areas of data modelling and protection resulting from simpler semantics and reduced complexity. The thesis demonstrates that many new benefits for software construction and execution may be achieved when these operations are performed within a single integrated persistent programming environment.

6.1 Delivering the Benefits of Persistence

A persistent system has been used to implement an integrated persistent programming environment. Such a system consists of a number of components as follows:

- An object store. The persistence abstraction hides the physical properties of data. As a consequence, a persistent object store has certain perceived attributes such as unbounded size, infinite speed and stability. The main technical problems encountered when constructing a persistent object store involve the simulation of these perceived properties [Bro89].
- A persistent language. It is desirable that the language used in a persistent system is suitable for the programming of large, long-lived, data-intensive applications [AB87]. Hence the type system of the language should support the modelling of data throughout its lifetime [ABM85]. Powerful abstraction mechanisms are required to reduce the complexity of large bodies of code [MBC+87]. Flexible binding mechanisms are necessary that in conjunction with the type system permit the safe and efficient construction and evolution of both program and data [MAD87].
- A compiler accessible within the persistent environment. Executing programs may use the compiler in the construction and execution of new programs that manipulate the persistent environment. This is a form of reflection [Mae87] and has been used to attain high levels of genericity [SFS+90], accommodate changes in systems [DB88], implement data models[Coo90a], optimise implementations[CAD+87] and validate specifications[FSS92].

A number of persistent languages and their associated object stores have already been implemented. These include PS-algol [PS88], Napier88 [MBC+89], Galileo [ACO85], DBPL[MS89], Staple [DM90] and Quest [Car89]. The compilers for these languages construct programs that can access and manipulate the persistent environment, however the compilers themselves are not available within the environment.

The starting point for the work required to support this thesis has therefore been the construction of a compiler accessible within a persistent environment. The language Napier88 has been chosen for this task. The compiler is the first major application to be constructed within the Napier88 persistent environment. Its construction has permitted an analysis to be undertaken of the manner in which persistence benefits the software construction process in general. The construction of a large application from a number of small components has also required a major reassessment of the type system implementation with respect to efficiency in terms of both space and time. The new implementation depends heavily on persistence. Having constructed an initial implementation of the compiler, the persistence of the environment has been exploited to gain new benefits in the areas of software construction and execution.

The major benefits derived from the construction and development of the integrated persistent programming environment may be briefly summarised as follows:

- Application construction. A methodology for the construction of applications from components has been developed which represents a new compromise between safety, flexibility and efficiency in comparison to existing techniques. Incremental component evolution is supported, static type safety is maintained and efficiency during execution is reduced marginally in comparison to the optimum efficiency.
- Optimisation. A methodology for optimisation based on dynamically gathered data has been developed. Both code and data may be optimised based on information gathered during their use and retained within the persistent environment. In addition the use of the persistent store as a cache for structured data has been used extensively throughout this thesis to avoid expensive recalculation or reconstruction.
- Extended compilation facilities. A compiler within the persistent environment can manipulate the structured values retained there. Development of this ability has lead to the construction of new interfaces to the compiler that offer increased functionality over traditional compilers. The interfaces increase the simplicity, flexibility and efficiency of software construction and execution. Again, the new facilities are entirely dependent on the persistent environment.

Both the methodologies and the new compilation facilities evolved together as the compiler was developed. As a consequence of this, the first application of the methodologies was in the implementation of the compiler and conversely some of the first applications of the new compilation facilities were seen in particular instances of the methodologies. It is these applications of the new technology that have been described in this thesis as examples of its validity. However, the methodologies and the compilation facilities need not be so closely inter-twined. The benefits derived from each of them are general and they all have applicability in many other areas of the persistent environment.

6.2 Methodologies for Persistent Software Engineering

6.2.1 Constructing Applications from Components

The construction of applications from separate components depends on a conflict between the following desires associated with the binding of components:

- Failure during execution caused by component binding should be avoided.
- Component binding should not affect execution efficiency.

- Incremental evolution of individual components within an application should be supported.

Existing application construction systems fall into one of the following categories:

- Applications execute safely and efficiently at the expense of flexible evolution[Mil84,Car89,ACO85].
- Components may evolve incrementally at the expense of safety and efficiency during application execution[MAE+62, GR83].
- The construction system supports a compromise between the three desires. Applications execute efficiently but with less static safety than those of the first category and the flexibility of evolution is between the extremes of the first two categories[Wir71,KR78, DOD83].

The application construction methodology that has been described in this thesis represents a new compromise between safety, flexibility and efficiency. Incremental component evolution is supported, static type safety is maintained and efficiency during execution is reduced marginally in comparison to the optimum efficiency.

Using the construction methodology, an application consists of a number of strongly typed persistent locations. The locations contain the executable components of the application which are first-class values of the language. Bindings between components take place between a component and the location of a component it uses. In so doing, incremental component evolution is supported since a single component may be updated by assigning the new version to its location. Safety is maintained since the locations are strongly typed and the binding is performed before execution. Update is slightly more complex when the component type changes. During execution a location dereference is required in order to access the component contained therein.

Where locations may be bound into source code as described in Chapter 4 the architecture may be used in a hyper-programming environment [KCC+92]. An example of this is given in Section 4.3.2 where the techniques have been applied to the construction of the compiler within the environment.

6.2.2 Optimisation techniques

Caching

The persistent environment is a conceptually infinite area for the storage of data. The environment is directly accessible to all executing programs and so may be used as a cache for data of any type to avoid expensive recalculation or reconstruction.

The use of the persistent environment as a cache for structured data is widespread throughout this thesis. In particular caching techniques have been used extensively in Chapter 3 to optimise type system operations.

Enhancement

The general optimisation methodology used in the example of Chapter 5 attempts to enhance the operation of some feature of a system. This is achieved by using the persistent environment as a cache for records of dynamic behaviour associated with the particular feature. The records are analysed by enhancement programs as the basis for a decision on possible optimisations. Generally, analysis and optimisation take place in three stages:

- Using the cached records the enhancer determines whether a potential optimisation exists. Optimisation involves altering trade-offs associated with the implementation of some characteristic of the feature
- If an optimisation exists then the enhancer determines whether the optimisation should be performed. The decision is made using a cost function that may operate over both static and dynamic information and determines whether the benefits of the optimisation outweigh the cost of making it.
- The optimisation is applied to the persistent environment if the cost function returns a positive result.

Subsequent execution of the program recording the profiles will benefit from the optimisation. In addition, other programs in the environment that are associated with the optimised feature may also benefit.

6.3 Enhancing the Functionality of the Compiler

Compilation is traditionally viewed as a process that translates unstructured source code into executable code. Usually it is performed in an environment independent of other software functions such as source code composition, component linking and execution.

In an integrated persistent environment, the source code may contain or be associated with structured values. Using such source representations, binding between code and data may take place at any time during the lifetime of a program, from composition through compilation to linking and execution. The interface to the compiler described in Chapter 4 supports all of these binding times. In particular hyper-programming is supported [KCC+92]. Implementation of the interface has required a new protection mechanism over intermediate code and type representations in order to maintain system integrity. This was described in Chapter 4.

6.4 Future Work

Future work will consist of new applications of the methodologies and techniques developed here to other aspects of the persistent environment. The use of the new compiler interface in traditionally reflective areas of computation will increase the availability of these computational models through a reduction in complexity. The wide range of binding techniques supported by the compilation interface and the construction methodology will be used in further fitting-out of the environment.

The optimisation techniques may be applied to many aspects of the environment. Any program or process that must make a trade-off between possible options may use the enhancement optimisation technique. Analysis of the chosen trade-off based on dynamic measurements may be used to change the trade-off to improve performance.

6.5 Finale

The work described in this thesis attempts to improve the functionality and efficiency of software construction and execution in a persistent system. This is an essential step towards the eventual goal of widespread use of persistent systems outside the research environment. If the work can help to avoid travel hold-ups, banking errors, reservation nightmares, dole cheque delays and bureaucratic *faux pas* then so much the better.

Let us hope that such improvements may be made to the world *without* having to "bring the system down".

Appendix 1 The Napier88 Type System

The Napier88 type system is based on the notion of types as sets of objects from the value space. These sets may be predefined, like integer, or they may be formed by using one of the predefined type constructors, like structure. The constructors obey the *Principle of Data Type Completeness* [Mor79]. That is, where a type may be used in a constructor, any type is legal without exception. This has two benefits. Firstly, since all the rules are very general and without exceptions, a very rich type system may be described using a small number of defining rules. This reduces the complexity of the defining rules. The second benefit is that the type constructors are as powerful as is possible since there are no restrictions to their domain. This increases the power of the language.

A1.1 Universe of Discourse

There are an infinite number of data types in Napier88 defined recursively by the following rules:

1. The scalar data types are integer, real, boolean, string, pixel, picture, file and null.

2. The type image is the type of an object consisting of a rectangular matrix of pixels.

3. For any data type t, *t is the type of a vector with elements of type t.

4. For identifiers $I_1,...,I_n$ and types $t_1,...,t_n$, structure (I_1: $t_1,...,I_n$: t_n) is the type of a structure with fields I_i and corresponding types t_i, for i = 1..n.

5. For identifiers $I_1,...,I_n$ and types $t_1,...,t_n$, variant (I_1: $t_1,...,I_n$: t_n) is the type of a variant with identifiers I_i and corresponding types t_i, for i = 1..n.

6. For any data types $t_1,...,t_n$ and t, proc ($t_1,...,t_n \rightarrow t$) is the type of a procedure with parameter types t_i, for i = 1..n and result type t. The type of a resultless procedure is proc ($t_1,...,t_n$).

7. For any procedure type, proc ($t_1,...,t_n \rightarrow t$) and type identifiers $T_1,...,T_m$, proc [$T_1,...,T_m$] ($t_1,...,t_n \rightarrow t$) is the type proc ($t_1,...,t_n \rightarrow t$) universally quantified by types $T_1,...,T_m$. These are polymorphic procedures.

8. env is the type of an environment.

9. For any type identifiers $W_1,...,W_m$, identifiers $I_1,...,I_n$ and types $t_1,...,t_n$, abstype [$W_1,...,W_m$] (I_1: $t_1,...,I_n$: t_n), is the type of an existentially quantified data type. These are abstract data types.

10. The type any is the infinite union of all types.

11. For any user-constructed data type t and type identifiers, $T_1,...,T_n$, t [$T_1,...,T_n$] is the type t parameterised by $T_1,...,T_n$.

In addition to the above data types, there are a number of other objects in Napier88 to which it is convenient to give a type in order that the compiler may check their use for consistency.

12. Clauses which yield no value are of type void, as are procedures with no result.

The world of data objects is defined by the closure of rules 1 and 2 under the recursive application of rules 3 to 12.

A1.2 Context Free Syntax and Type Rules

The type rules form a second set of rules to be used in conjunction with the context free syntax to define well-formed programs. The generic types that are required for the formal definition of Napier88 can be described by the following:

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

In the above, the generic type arith can be either an int or a real, representing the types integer and real in the language. In the type rules, the concrete types and generic types are written in shadow font to distinguish them from the reserved words. Each of the type categories given above corresponds to one of the type construction rules.

To check that a syntactic category is correctly typed, the context free syntax is used in conjunction with a type rule. For example, the type rule for the two-armed if clause is

t : type, **if** <clause> : bool **then** <clause> : t **else** <clause> : t **=>** t

This rule may be interpreted as follows: t is given as a type from the table above. It can be any type including void. Following the comma, the type rule states that the reserved word **if** may be followed by a clause which must be of type boolean. This is indicated by : bool. The **then** and **else** alternatives must have clauses of the same type t for any t. The resultant type, indicated by **=>**, of this production is also t, the same as the alternatives.

The full context free syntax and type rules are included at the end of this appendix.

A1.3 Type Equivalence Rule

Two data objects have the same type if they are structurally equivalent, that is the types represent the same set of values. Thus, even if a type identifier is aliased, for example,

type quintin **is int**

the fact that objects of type *quintin* are integers cannot be hidden. Abstract data types can be used for this purpose. The meaning of structural equivalence for two types is that they represent the same set. For scalar types, the construction of these sets is obvious. For constructed types, it is not so obvious and is defined as part of the semantics of the constructor.

vector Two vector types are equivalent if the type of the elements is equivalent.

structure Two structure types are equivalent if each structure contains the same set of (identifier,type) pairs. The ordering of the fields is not important.

variant Two variants are equivalent if each variant contains the same set of (label,type) pairs. The ordering of the branches is not important.

procedure Two procedures are equivalent if the ordering and types of the parameters are equivalent as are the result types.

universally quantified procedure Similar to a monomorphic procedure. In addition, the two procedures must have the same number of quantifiers. The names of the quantifiers is not important.

env All environments have the same type, **env**, and are hence type compatible.

any All **any**s have the same type and are hence type compatible.

existentially quantified abstract data type Similar to structures. In addition, the two types must have the same number of witness types. The names of the witnesses is not important.

For example, the types,

type man **is structure** (age : **int** ; size : **real**)

and

type house **is structure** (size : **real** ; age : **int**)

have the same type since they both represent the same set of objects.

A1.4 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

A1.5 Napier88 Type Rules

These should be interpreted in conjunction with the context free syntax as described in Section A1.2.

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 : type, **use**<clause> : env **with**<signature>**in**<clause> : t => t

use<clause> : abstype **as**<identifier>[<witness_decls>]
in<clause> : void => void

t : type ; t1 : nonvoid, **case** <clause> : t1 **of** <case_list>
default : <clause> : t => t

where <case_list> ::= <clause_list>:<clause> : t ; [<case_list>]

where <clause_list> ::= <clause> : t1 [,<clause_list>]

<raster_op><clause> : image **onto**<clause> : image => void

drop<identifier>**from**<clause> : env => void

t : type, **project**<clause> : any **as**<identifier>**onto**<any_project_list>
default : <clause> : t => t
where <any_project_list> ::= <type_id>:<clause> : t ; [<any_project_list>]

t : type, **project**<clause> : variant **as**<identifier>**onto**<variant_project_list>
default : <clause> : t => t
where <variant_project_list> ::=<identifier>:<clause> : t ;
<variant_project_list>]

t : nonvoid, <name> : t := <clause> : t => void

Expressions :

<expression> : bool **or** <expression> : bool => bool

<expression> : bool **and** <expression> : bool => bool

[~]<expression> : bool => bool

t : nonvoid, <expression> : t <eq.op> <expression> : t => bool
where <eq_op> ::= = | ≠

t : ordered, <expression> : t <co_op> <expression> : t => bool
where <co_op> ::= < | <= | > | >=

<expression> : variant<type_op><identifier> => bool
where <type_op> ::= **is** | **isnt**

t : nonvoid,**any** (<clause>) : t => any

<expression> : env **contains** [**constant**]<identifier>[:<type>] => bool

t : arith, <expression> : t <add_op><expression> : t => t

t : arith, <add_op> <expression> : t => t

t : int, <expression> : t <int_mult_op> <expression> : t => t
where <int_mult_op> ::= <star> | div | rem

t : real, <expression> : t <real_mult_op> <expression> : t => t
where <real_mult_op> ::= <star> | /

t : string, <expression> : t <string_mult_op> <expression> : t => t
where <string_mult_op> ::= ++

t : pic, <expression> : t <pic_mult_op> <expression> : t => t
where <pic_mult_op> ::= ^ | ++

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

References

[AB87] M.P. Atkinson and O.P. Buneman
"Types and Persistence in Database Programming Languages"
ACM Computing Surveys 19, 2 (1987) pp 105-190.

[ABC+83] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshott and R. Morrison
"An Approach to Persistent Programming"
Computer Journal 26, 4 (1983) pp 360-365.

[ABC+84] M.P. Atkinson, P.J. Bailey, W.P. Cockshott, K.J. Chisholm and R. Morrison
"Progress with Persistent Programming"
Universities of Glasgow and St Andrews Report PPRR-8-84 (1984).

[ABC83] M.P. Atkinson, P.J. Bailey, W.P. Cockshott, K.J. Chisholm and R. Morrison
"PS-algol Papers: a collection of related papers on PS-algol"
Universities of Glasgow and St Andrews Report PPRR-2-83 (1983).

[ABM85] M. Atkinson, P. Buneman and R. Morrison
"Data Types and Persistence"
(1985).

[ACC82] M.P. Atkinson, K.J. Chisholm and W.P. Cockshott
"PS-algol: An Algol with a Persistent Heap"
ACM SIGPLAN Notices 17, 7 (1982) 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.

[ACP+91] M. Abadi, L. Cardelli, B.C. Pierce and G. Plotkin
"Dynamic Typing in a Statically Typed Language"
ACM ToPLAS 13, 2 (1991) pp 237-268.

[AGO88] A. Albano, G. Ghelli and R. Orsini
"The Implementation of Galileo's Values Persistence"
In **Data Types and Persistence**, Springer-Verlag (1988) pp 253-263.

[AM85] M.P. Atkinson and R. Morrison
"Procedures as Persistent Data Objects"
ACM ToPLAS 7, 4 (1985) pp 539-559.

[AM88] M.P. Atkinson and R. Morrison
"Types, Bindings and Parameters in a Persistent Environment"
In **Data Types and Persistence**, Springer-Verlag (1988) pp 3-20.

[AMP86] M.P. Atkinson, R. Morrison and G.D. Pratten
"A Persistent Information Space Architecture"
Universities of Glasgow and St Andrews Report PPRR-21-85 (1985).

[atk78] M.P. Atkinson
"Programming Languages and Databases"
In Proc. Very Large Databases, (1978) pp 408 - 419.

[Atk92] M.P. Atkinson
"FIDE2 Technical Annex for Basic Research Action 6309"
(1992).

[BBB+88] F. Bancilhon, G. Barbedette, B. Benzaken, C. Delobel, S. Gamerman, C. Lecluse, P. Pfeffer, P. Richard and F. Valez
"The Design and Implementation of O2, an Object-Oriented Database System"
In **Lecture Notes in Computer Science 334**, Springer-Verlag (1988) pp 1-22.

[BCC+88] A.L. Brown, R. Carrick, R.C.H. Connor, A. Dearle and R. Morrison
"The Persistent Abstract Machine"
Universities of Glasgow and St Andrews Report PPRR-59-88 (1988).

[BD90] V. Benzaken and C. Delobel
"Enhancing Performance in a Persistent Object Store: Clustering Strategies in O_2"
In **Implementing Persistent Object Bases**, Morgan Kaufmann (1990) pp 403-412.

[BMM+92] A.L. Brown, G. Mainetto, F. Matthes, R. Müller and D.J. McNally
"An Open System Architecture for a Persistent Object Store"
In Proc. 25th International Conference on Systems Sciences, Hawaii (1992) pp 766-776.

[BOP+89] B. Bretl, A. Otis, J. Penney, B. Schuchardt, J. Stein, E.H. Williams, M. Williams and D. Maier
"The GemStone Data Management System"
In **Object-Oriented Concepts, Applications, and Databases**, Morgan-Kaufman (1989)

[Bro89] A.L. Brown
"Persistent Object Stores"
Ph.D. Thesis, University of St Andrews (1989).

[CAD+87] R.L. Cooper, M.P. Atkinson, A. Dearle and D. Abderrahmane
"Constructing Database Systems in a Persistent Environment"
In Proc. 13th International Conference on Very Large Data Bases, (1987) pp 117-125.

[Car83] L. Cardelli
"The Functional Abstract Machine"
AT & T Bell Laboratories Report TR-107 (1983).

[Care89] M. Carey
"The EXODUS Extensible DBMS Project: An Overview"
In **Readings in Object-Oriented Databases**, Morgan-Kaufmann (1989)

[Car89] L. Cardelli
"Typeful Programming"
DEC Report 45 (1989).

[CBC+90] R.C.H. Connor, A.B. Brown, Q.I. Cutts, A. Dearle, R. Morrison and J. Rosenberg
"Type Equivalence Checking in Persistent Object Systems"
In **Implementing Persistent Object Bases**, Morgan Kaufmann (1990) pp 151-164.

[CBC+90] R.C.H. Connor, A.L. Brown, R. Carrick, A. Dearle and R. Morrison
"The Persistent Abstract Machine"
In **Persistent Object Systems**, Springer-Verlag (1990) pp 353-366.

[CDM+90] R.C.H. Connor, A. Dearle, R. Morrison and A.L. Brown
"Existentially Quantified Types as a Database Viewing Mechanism"
In **Lecture Notes in Computer Science 416**, Springer-Verlag (1990) pp 301-315.

[Con88] R.C.H. Connor
"The Napier Type-Checking Module"
Universities of Glasgow and St Andrews Report PPRR-58-88 (1988).

[Con91] R.C.H. Connor
"Types and Polymorphism in Persistent Programming Systems"
Ph.D. Thesis, University of St Andrews (1991).

[Con92] R. Connor
"Panel on Persistent Type Systems"
In Proc. 5th International Workshop on Persistent Object Systems, San Miniato, Italy (1992)

[Coo90a] R.L. Cooper
"On The Utilisation of Persistent Programming Environments"
Ph.D. Thesis, University of Glasgow (1990).

[Coo90b] R.L. Cooper
"Configurable Data Modelling Systems"
In Proc. 9th International Conference on the Entity Relationship Approach, Lausanne, Switzerland (1990) pp 35-52.

[CW85] L. Cardelli and P. Wegner
"On Understanding Types, Data Abstraction and Polymorphism"
ACM Computing Surveys 17, 4 (1985) pp 471 - 523.

[DB88] A. Dearle and A.L. Brown
"Safe Browsing in a Strongly Typed Persistent Environment"
Computer Journal 31, 6 (1988) pp 540-544.

[DCK89] A. Dearle, Q.I. Cutts and G.N.C. Kirby
"Browsing, Grazing and Nibbling Persistent Data Structures"
In **Persistent Object Systems**, Springer-Verlag (1990) pp 56-69.

[Dea87] A. Dearle
"A Persistent Architecture Intermediate Language"
Universities of Glasgow and St Andrews Report PPRR-35-87 (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"
In Proc. 22nd International Conference on Systems Sciences, Hawaii (1989) pp 46-55.

[DM81] A.J.T. Davie and R. Morrison
Recursive Descent Compiling
Ellis Horwood (1981)

[DM90] A.J.T. Davie and D.J. McNally
"Statically Typed Applicative Persistent Language Environment (STAPLE) Reference Manual"
University of St Andrews Report CS/90/14 (1990).

[DOD83] "Reference Manual for the Ada Programming Language"
U.S. Department of Defense Report ANSI/MIL-STD-1815A (1983).

[Far91] A.M. Farkas
"ABERDEEN: A Browser allowing intERactive DEclarations and Expressions in Napier88"
University of Adelaide Report Honours Project (1991).

[FDK+92] A.M. Farkas, A. Dearle, G.N.C. Kirby, Q.I. Cutts, R. Morrison and R.C.H. Connor
"Persistent Program Construction through Browsing and User Gesture with some Typing"
In Proc. 5th International Workshop on Persistent Object Systems, San Miniato, Italy (1992) pp 375-394.

[FID90] "The FIDE Project"
Esprit II Basic Research Action 3070 (1990).

[FID91] "A Proposal for Basic Research Action - FIDE Phase 2"
Esprit III (1991).

[FS91] L. Fegaras and D. Stemple
"Using Type Transformation in Database System Implementation"
In Proc. 3rd International Conference on Database Programming Languages, Nafplion, Greece (1991) pp 289-305.

[FSS92] L. Fegaras, T. Sheard and D. Stemple
"Uniform Traversal Combinators: Definition, Use and Properties"
In Proc. 11th International Conference on Automated Deduction (CADE-11), Saratoga Springs, New York (1992)

[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 **Data Types and Persistence**, Springer Verlag (1988) pp 353-368.

[IBM78] "IBM Report on the Contents of a Sample of Programs Surveyed"
IBM, San Jose, California (1978).

[Joh71] J.B. Johnston
"The Contour Model of Block Structure Processes"
ACM SIGPLAN Notices 6, 2 (1971) pp 56-82.

[KCC+92] G.N.C. Kirby, Q.I. Cutts, R.C.H. Connor, A. Dearle and R. Morrison
"Programmers' Guide to the Napier88 Standard Library, Edition 2.1"
In preparation (1992).

[KCC+92] G.N.C. Kirby, R.C.H. Connor, Q.I. Cutts, A. Dearle, A.M. Farkas and R. Morrison
"Persistent Hyper-Programs"
In Proc. 5th International Workshop on Persistent Object Systems, San Miniato, Italy (1992) pp 73-95.

[Kir92] G.N.C. Kirby
"Reflection and Hyper-programming in Persistent Programming Systems"
Ph.D. Thesis, University of St Andrews (1992)

[KR78] B.W. Kernighan and D.M. Ritchie
The C programming language
Prentice-Hall (1978)

[Leh80] M.M. Lehman
"Programs, life cycles and the laws of software evolution"
Proc. IEEE 15, 3 (1980) pp 225-252.

[LR89] C. Lécluse and P. Richard
"The O2 Database Programming Language"
In Proc. 15th VLDB Conference, Amsterdam (1989)

[MAD87] R. Morrison, M.P. Atkinson and A. Dearle
"Flexible Incremental Bindings in a Persistent Object Store"
Universities of Glasgow and St Andrews Report PPRR-38-87 (1987).

[MAE+62] J. McCarthy, P.W. Abrahams, D.J. Edwards, T.P. Hart and M.I. Levin
The Lisp Programmers' Manual
M.I.T. Press (1962)

[Mae87] P. Maes
"Concepts and Experiments in Computational Reflection"
In Proc. OOPSLA'87, (1987) pp 147-155.

[MBC+87] R. Morrison, A.L. Brown, R.C.H. Connor and A. Dearle
"Polymorphism, Persistence and Software Reuse in a Strongly Typed Object Oriented Environment"
Universities of Glasgow and St Andrews Report PPRR-32-87 (1987).

[MBC+89] R. Morrison, A.L. Brown, R.C.H. Connor and A. Dearle
"The Napier88 Reference Manual"
University of St Andrews Report PPRR-77-89 (1989).

[MBC+90] R. Morrison, A.L. Brown, R.C.H. Connor, Q.I. Cutts, G.N.C. Kirby, A. Dearle, J. Rosenberg and D. Stemple
"Protection in Persistent Object Systems"
In **Security and Persistence**, Springer-Verlag (1990) pp 48-66.

[MDC+91] R. Morrison, A. Dearle, R.C.H. Connor and A.L. Brown
"An Ad-Hoc Approach to the Implementation of Polymorphism"
ACM ToPLAS 13, 3 (1991) pp 342-371.

[Mil78] R. Milner
"A Theory of Type Polymorphism in Programming"
Journal of Computer and System Sciences 17, 3 (1978) pp 348-375.

[Mil84] R. Milner
"A Proposal for Standard ML"
In Proc. ACM Symposium on LISP and Functional Programming, Austin, Texas (1984) pp 184-197.

[Mor 79] R. Morrison
"On the Development of Algol"
Ph.D. Thesis, University of St Andrews (1979).

[MP88] J.C. Mitchell and G.D. Plotkin
"Abstract Types have Existential Type"
ACM ToPLAS 10, 3 (1988) pp 470 - 502.

[MMS92] F. Matthes, R. Müller and J.W. Schmidt
"Object Stores as Servers in Persistent Programming Environments—The P-Quest Experience"
ESPRIT BRA Project 3070 FIDE Report FIDE/92/? (1992).

[MS89] F. Matthes and J.W. Schmidt
"The Type System of DBPL"
In R. Hull, R. Morrison and D. Stemple (Editors)
Proc. 2nd International Conference on Database Programming Languages
Morgan Kaufmann (1989) pp 219-225.

[Org72] E.I. Organick
The Multics System: An Examination of its Structure
M.I.T. Press (1972)

[PS87] J.L. Peterson and A. Silberschatz
Operating System Concepts
Addison Wesley (1987)

[PS88] "PS-algol Reference Manual, 4th edition"
Universities of Glasgow and St Andrews Report PPRR-12-88 (1988).

[QL91] R.W. Quong and M.A. Linton
"Linking Programs Incrementally"
ACM ToPLAS 13, 1 (1991) pp 1-20.

[RCS89] J. Richardson, M. Carey and D. Schuh
"The Design of the E Programming Language"
Computer Sciences Dept., University of Wisconsin Report 824 (1989).

[RT78] D.M. Ritchie and K. Thompson
"The UNIX Time-Sharing System"
The Bell System Technical Journal 63, 6 (1978) pp 1905-1930.

[SFS+90] D. Stemple, L. Fegaras, T. Sheard and A. Socorro
"Exceeding the Limits of Polymorphism in Database Programming Languages"
In **Lecture Notes in Computer Science 416**, Springer-Verlag (1990) pp 269-285.

[SSF92] D. Stemple, T. Sheard and L. Fegaras
"Linguistic Reflection: A Bridge from Programming to Database Languages"
In Proc. 25th International Conference on Systems Sciences, Hawaii (1992) pp 844-855.

[SSS+92] D. Stemple, R.B. Stanton, T. Sheard, P. Philbrow, R. Morrison, G.N.C. Kirby, L. Fegaras, R.L. Cooper, R.C.H. Connor, M.P. Atkinson and S. Alagic
"Type-Safe Linguistic Reflection: A Generator Technology"
University of St Andrews Report CS/92/6 (1992).

[Str67] C. Strachey
Fundamental Concepts in Programming Languages
Oxford University Press (1967)

[Str86] B. Stroustrup
The C++ Programming Language
Addison-Wesley (1986)

[Sun87] Sun Microsystems Inc.
The SPARC Architecture Manual, Version 7
(1987)

[Amm73] U. Ammann
"The Method of Structured Programming applied to the Development of a Compiler"
In Proc. International Computing Symposium, (1973)

[Wai87] F. Wai
"Distribution and Persistence"
In Proc. 2nd International Workshop on Persistent Object Systems, Appin, Scotland (1987) pp 207-225.

[Weg90] P. Wegner
"Concepts and Paradigms of Object-oriented Programming"
ACM SIGPLAN OOPS Mess. 1,1 (Aug 1990) pp 7-87.

[Wir71] N. Wirth
"The Programming Language Pascal"
Acta Informatica 1, (1971) pp 35-63.